SYBASE®

Client-Library™/C Reference Manual

Open Client™

15.0

DOCUMENT ID: DC32840-01-1500-04

LAST REVISED: December 2008

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

Contents

About This Book

The Open Client™ *Client-Library/C Reference Manual*, contains reference information for the C version of Open Client Client-Library.

Audience

This manual is a reference manual for programmers who are writing Client-Library™ applications. It is written for application programmers who are familiar with the C programming language.

How to use this book

Use this manual as a source of reference information, when you are writing a Client-Library application.

- Chapter 1, “Introducing Client-Library,” contains a brief introduction to Client-Library.
- Chapter 2, “Client-Library Topics,” contains information on how to accomplish specific programming tasks, such as using Client-Library routines to read a text or image value from the server. This chapter also contains information on Client-Library structures, options, error messages, and conventions.
- Chapter 3, “Routines,” contains specific information about each Client-Library routine, such as what parameters the routine takes and what it returns.

Related documents

- The Open Server and SDK *New Features* bulletin for Microsoft Windows, Linux, and UNIX, which describes new features available for Open Server and the Software Developer’s Kit. This document is revised to include new features as they become available.
- The SDK and Open Server *Installation Guide* explains how to install Client-Library.
- The Open Client *Client-Library/C Programmers Guide* contains information on how to design and implement Client-Library programs.
- The Open Client and Open Server™ *Common Libraries Reference Manual* contains reference information for CS-Library and Bulk-Library.

- The Open Client and Open Server *Programmers Supplement* for Microsoft Windows and Open Client and Open Server *Programmers Supplement* for UNIX contain platform-specific material for Open Client and Open Server developers, including:
  - How to compile and link an application
  - The sample programs that are included with Open Client and Open Server products
  - Documentation for the routines that have platform-specific behavior
- The Open Client and Open Server *Configuration Guide* for Microsoft Windows and Open Client and Open Server *Configuration Guide* for UNIX contains information for System Administrators who configure the Open Client and Open Server installation environment:
  - Platform-specific localization mechanisms
  - Configuring Sybase® drivers for network services
  - The *interfaces* file
- The Open Client and Open Server *International Developer's Guide* contains information for programmers who use Client-Library to develop international applications:
  - A description of the localization mechanism used by the Open Client and Open Server™ libraries
  - Guidelines for developing international applications with the Open Client and Open Server libraries

In addition, the following manuals will prove to be particularly useful:

- The Adaptive Server® Enterprise *Reference Manual* describes the Transact-SQL® database language, which an application uses to create and manipulate Sybase Adaptive Server database objects.
- The *Transact-SQL User's Guide* serves as a textbook on Transact-SQL for new SQL programmers or programmers who are experienced with another Structured Query Language (SQL).
- The Open Client *DB-Library/C Reference Manual* describes DB-Library™. Like Client-Library, DB-Library is a collection of routines for use in writing client applications.

- The Open Client *Client-Library Migration Guide* contains information on how DB-Library applications can be migrated to Client-Library. For DB-Library programmers, this book is also a useful comparison of the DB-Library and Client-Library interfaces.
- The Open Server *Server-Library/C Reference Manual* contains reference information for Open Server Server-Library, a collection of routines for use in writing Open Server applications.

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 Document Types under Technical Documents from the navigation bar on the left. Then, click Certification Report.

3 In the Certification Report filter, select Product, Platform, and Timeframe. Then, click Go.

4 Click a Certification Report title to display the report.

❖ **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 braces in your option.
[ ]	Brackets mean choosing one or more of the enclosed items is optional. Do not include brackets in your option.
( )	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.

Code fragments

Most code fragments in this book are taken from the Client-Library sample programs. See the Open Client and Open Server *Programmers Supplement* for Microsoft Windows or Open Client and Open Server *Programmers Supplement* for UNIX, for a description of these samples and their location in your Sybase installation directory.

Many code fragments in these books reference routines and symbols defined in the sample programs, for example:

```
if (ct_close(connection, CS_UNUSED) != CS_SUCCEED)
 {
    ex_error("ct_close failed");
 }
```

All *ex_* and *EX_* symbols used in this book's code samples are defined in the sample programs. They are not part of the Client-Library programming interface.

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 Client-Library

This chapter provides an overview of client/server architecture and Open Server applications:

This chapter does not contain any introductory information on developing Client-Library applications. For that information, see Chapter 1, “Getting Started With Client-Library,” in the Open Client *Client-Library/C Programmer's Guide*.

Sybase client/server architecture

Client/server architecture divides the work of computing between “clients” and “servers.”

Clients make requests of servers and process the results of those requests. For example, a client application might request data from a database server. Another client application might send a request to an environmental control server to lower the temperature in a room.

Servers respond to requests by returning data or other information to clients, or by taking some action. For example, a database server returns tabular data and information about that data to clients, and an electronic mail server directs incoming mail toward its final destination.

Client/server architecture has several advantages over traditional program architectures:

- Application size and complexity can be significantly reduced, because common services are handled in a single location, a server. This simplifies client applications, reduces duplicate code, and makes application maintenance easier.
- Client/server architecture facilitates communication between various applications. Client applications that use dissimilar communications protocols cannot communicate directly, but can communicate through a server that “speaks” both protocols.
- Client/server architecture enables applications to be developed with distinct components, which can be modified or replaced without affecting other parts of the application.

Types of clients

A client is any application that makes requests of a server. Clients include:

- Sybase middleware products such as OmniConnect™ and OpenSwitch™
- Standalone utilities provided with Adaptive Server, such as isql and bcp
- Applications written using Open Client libraries
- Java applets and applications written using jConnect™ for JDBC™
- Applications written using Embedded SQL™

Types of servers

The Sybase product line includes servers and tools for building servers:

- Adaptive Server is a database server. Adaptive Servers manage information stored in one or more databases.
- Open Server provides the tools and interfaces needed to create a custom server application.

  An Open Server application can be any type of server. For example, it can perform specialized calculations, provide access to real-time data or interface with services such as electronic mail. An Open Server application is created individually, using the building blocks provided by Open Server Server-Library.

Adaptive Server and Open Server applications are similar in some ways:

- Both servers respond to client requests.
- Clients communicate with both Adaptive Server and Open Server applications through Open Client products.

But they also differ:

- An application programmer must create an Open Server application, using Server-Library's building blocks and supplying custom code. Adaptive Server is complete and does not require custom code.
- An Open Server application can be any kind of server and can be written to understand any language. Adaptive Server is a database server and understands only Transact-SQL.
- An Open Server application can communicate with non-Sybase protocols, as well as with Sybase applications and servers. Adaptive Server can communicate directly only with Sybase applications and servers; however, Adaptive Server can communicate with non-Sybase applications and servers by using an Open Server gateway application as an intermediary.

Open Client and Open Server products

Sybase provides two families of products to enable customers to write client and server application programs:

- Open Client
- Open Server

Open Client

Open Client provides customer applications, third-party products, and other Sybase products with the interfaces needed to communicate with Adaptive Server and Open Server.

Open Client can be thought of as comprising two components, programming interfaces and network services.

Open Client provides two core programming interfaces for writing client applications: Client-Library and DB-Library.

- Open Client Client-Library/C is described in this book. The Client-Library interface supports server-managed cursors and other features in System 10 and later versions of the product line.
- Open Client DB-Library is a separate API that supports earlier Open Client applications. DB-Library is documented in the Open Client *DB-Library/C Reference Manual*.

Client-Library programs also depend on CS-Library, which provides routines that are used in both Client-Library and Server-Library applications. Client-Library applications can also use Bulk-Library routines to facilitate high-speed data transfer.

CS-Library and Bulk-Library are both included in the Open Client product. These libraries are described further in the section titled “Shared common libraries.”

Open Client network services include Sybase Net-Library, which provides support for specific network protocols such as TCP/IP and DECnet. The Net-Library interface is invisible to application programmers. However, on some platforms an application may need a different Net-Library driver for different system network configurations. Depending on your host platform, the Net-Library driver is specified either by the system’s Sybase configuration, or when you compile and link your programs.

Instructions for driver configuration are in the Open Client and Open Server *Configuration Guide* for Microsoft Windows, and Open Client and Open Server *Configuration Guide* for UNIX. Instructions for building Client-Library programs are in the Open Client and Open Server *Programmer's Supplement* for Microsoft Windows and the Open Client and Open Server *Programmer's Supplement* for UNIX.

Open Server

Open Server provides the tools and interfaces needed to create custom servers. Like Open Client, Open Server consists of an interfaces component and a network services component.

The core programming interface for creating Open Server applications is Server-Library. Server-Library is documented in the Open Server *Server-Library/C Reference Manual*. Server-Library programs depend on Client/Server-Library (CS-Library for short). Gateway Server-Library applications can also use routines from Client-Library and Bulk-Library. Client-Library, CS-Library, and Bulk-Library are all included in the Open Server product.

Open Server network services are transparent.

Shared common libraries

The Open Client and Open Server products both include Bulk-Library and CS-Library. These libraries provide routines useful to both client applications and server applications. CS-Library and Bulk-Library are both documented in the Open Client and Open Server *Common Libraries Reference Manual*.

CS-Library

CS-Library provides utility routines for Client-Library and Open Server programs. CS-Library allocates the core data structure (the CS_CONTEXT) for Client-Library programs. CS-Library also provides facilities for data conversion and localizing the client character set and language. The type definitions for data sent between the client and server are the same for CS-Library, Client-Library, and Server-Library.

DB-Library is not integrated with CS-Library. DB-Library and CS-Library share no common data structures, and their datatype definitions differ.

Bulk-Library

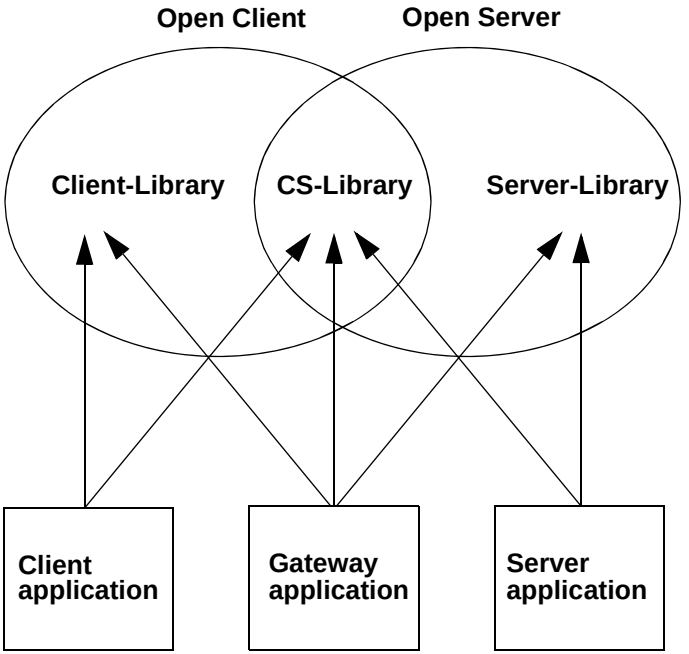
Bulk-Library/C provides routines that allow Client-Library and Server-Library applications to use Adaptive Server's **bulk copy** interface for high-speed data transfer. Client-Library programmers do not need to know Bulk-Library unless they want their applications to transfer data with the bulk copy interface. Bulk-Library, Client-Library, and Server-Library share common type definitions for data exchanged between client and server.

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

DB-Library has its own bulk copy interface and cannot be used with Bulk-Library.

The following diagram illustrates the relationship between the libraries included with Open Client and Open Server:

Figure 1-1: Open Client and Open Server library relationships



As an example, a client application might include calls to Client-Library and CS-Library, while an application that acts as both client and server might include calls to Client-Library, CS-Library, and Server-Library.

Although DB-Library is a completely separate interface from Client-Library, CS-Library, and Bulk-Library, it can be used in an Open Server gateway. It does not share Client-Library's advantages of sharing common data structures and type definitions with Server-Library.

Client-Library is a generic interface

Client-Library is a generic interface. Through Open Server and gateway applications, Client-Library applications can run against non-Sybase applications and servers as well as Adaptive Server.

Because it is generic, Client-Library does not enforce or reflect any particular server's restrictions. For example, Client-Library allows text and image stored procedure parameters, but Adaptive Server does not.

When writing a Client-Library application, keep the application's ultimate target server in mind. If you are unsure about what is legal on a server and what is not, consult your server documentation.

An application can call ct_capability to find out what capabilities a particular client/server connection supports.

Comparing the library approach to Embedded SQL

Either an Open Client library application or an Embedded SQL application can be used to send SQL commands to Adaptive Server.

An Embedded SQL application includes SQL commands in-line. The Embedded SQL precompiler processes the commands into calls to Client-Library routines. All Sybase precompilers 11.0 and later use a runtime library composed solely of documented Client-Library and CS-Library calls. Basically, the precompiler transforms an Embedded SQL application into a Client-Library application, which is then compiled using the host-language compiler.

An Open Client library application sends SQL commands through library routines and does not require a precompiler.

Generally, an Embedded SQL application is easier to write and debug, but a library application can take fuller advantage of the flexibility and power of Open Client routines.

What an application developer needs to know

The following describes the required programming interfaces and a brief description of the Client-Library functionality.

Programming interfaces

New Client-Library programmers will need to learn some or all of following programming interfaces:

- Client-Library, a collection of routines for use in writing client applications. Client-Library routines begin with “ct_”, as in ct_init. These are documented in Chapter 3, “Routines”.
- CS-Library, a collection of utility routines that are useful to both client and server applications. All Client-Library applications will include at least one call to CS-Library, because Client-Library routines use a structure that is allocated in CS-Library. CS-Library routines begin with “cs_”, as in cs_ctx_alloc. These routines are documented in the CS-Library chapters of the Open Client and Open Server *Common Libraries Reference Manual*.
- Bulk-Library, a collection of routines that allow Client-Library and Server-Library applications to use the Adaptive Server’s **bulk copy** interface for high-speed data transfer. Bulk copy of encrypted columns is supported if Adaptive Server supports encrypted columns. Client-Library programmers do not need to know Bulk-Library unless they want their program to transfer data using the bulk copy interface. Bulk-Library routines begin with “blk_”, as in blk_alloc. These routines are documented in the Bulk-Library chapters of the Open Client and Open Server *Common Libraries Reference Manual*.

Client-Library programmers must also know something about the server to which their client program connects.

- For connections to Adaptive Server, a client application developer should know the Transact-SQL language, Sybase’s implementation of Structured Query Language that allows access to Adaptive Server databases. Client application programmers must also be familiar with the tables and stored procedures that are in the Adaptive Server databases used by the application.
- For connections to Open Server gateways or other Open Server applications, the client application developer should know the feature set supported by the server. For example, not all Open Servers support language commands. Some only provide a collection of available registered procedures for RPC commands. When the server does support language commands, the client programmer must know the supported query language.

Getting started

For a quick tour of Client-Library functionality, including a simple sample program, see Chapter 1, “Getting Started With Client-Library,” in the Open Client *Client-Library/C Programmer's Guide*.

Summary of changes for version 15.0

This section contains information on changes to this manual in this version. The changes are:

- Sybase library name change: Naming conventions for Open Server and SDK libraries have changed, with the addition of *syb* to Sybase libraries. Names for non-Sybase libraries remain the same.
- BCP partitions: You can now copy ASE partitions with added support for BLKLIB and BCP programs.
- BCP computed columns: Two new Client-Library options have been added to support BCP computed columns.

  CS_OPT_HIDE_VCC instructs the Adaptive Server to hide Virtual Computed Columns (VCC), while CS_OPT_SHOW_FI adds columns for each Functional Index (FI).
- Large identifiers: Limits on lengths of identifiers have been reduced. This is now 255 bytes for identifiers.
- Unilib® support: Unicode Infrastructure Library (Unilib), an independent library of Unicode-based routines, has been included to facilitate character-set conversion.
- ASE default packet size support: You can now configure packet size centrally on the server, with the default set to 8192 bytes.
- Clusters support: A cluster of servers can now perform load balancing for all client connections coming into the cluster.
- Scrollable cursors: You can now set the position of a cursor anywhere in the cursor result set.
- Table 1-1 lists the new datatypes introduced in this version:

Table 1-1: New datatypes

Type category	Open Client and Open Server type constant	Description	Corresponding C datatype	Corresponding server datatype
XML type	CS_XML_TYPE	Variable-length character type	CS_XML	xml
Numeric types	CS_BIGINT_TYPE	8-byte integer type	CS_BIGINT	bigint
	CS_USMALLINT_TYPE	2-byte unsigned integer type	CS_USMALLINT	usmallint
	CS_UINT_TYPE	4-byte unsigned integer type	CS_UINT	uint
	CS_UBIGINT_TYPE	8-byte unsigned integer type	CS_UBIGINT	ubigint
Text and image types	CS_UNITEXT_TYPE	Variable-length character type	CS_UNITEXT	unitext

CHAPTER 2

Client-Library Topics

This chapter contains information about:

- Client-Library programming topics, such as asynchronous programming, browse mode, and text and image support
- How to use routines to accomplish specific programming tasks, such as declaring and opening a cursor
- Client-Library properties, datatypes, options, parameter conventions, and structures

The following topics are included in this chapter:

Asynchronous programming

Asynchronous applications are designed to make constructive use of time that would otherwise be spent waiting for certain operations to complete. Typically, reading from and writing to a network or external device is much slower than straightforward program execution. Also, it takes time for a server to process commands and send results back to the client application.

Some applications execute several tasks that involve idle time. For example, an interactive application might:

1 Wait for user input

2 Execute commands on a connection to Server X

3 Execute commands on a connection to Server Y

Client-Library's asynchronous modes help such an application to execute these tasks concurrently. When executing server commands, routines that send commands or read results return immediately. This means the application calls another routine to start command operations on another connection, and the application responds more quickly to user input.

Client-Library's asynchronous mode is one method for achieving concurrent task execution. The other is to use multiple threads. For information on using Client-Library in a multithreaded environment, see "Multithreaded programming" on page 158.

Asynchronous applications

By default, Client-Library applications are synchronous. Routines that read from or write to the network do not return control to the caller until all the necessary I/O requests are complete.

When writing an asynchronous application, the application programmer must enable asynchronous Client-Library behavior at the context or connection level by setting the Client-Library property CS_NETIO. The possible network I/O modes are:

- Fully asynchronous (CS_ASYNC_IO) – asynchronous routines return CS_PENDING immediately. When the requested operation completes, the connection's completion callback is invoked automatically.
- Deferred asynchronous (CS_DEFER_IO) – asynchronous routines return CS_PENDING immediately. The application must periodically call ct_poll to check whether the operation has completed. If the operation is finished, ct_poll invokes the connection's completion callback. ct_poll also indicates which operation (if any) completed, so a deferred-asynchronous application can operate without a completion callback if desired.
- Synchronous (CS_SYNC_IO) – all Client-Library routines do not return until the requested operation is complete. This mode is the default.

When fully asynchronous or deferred-asynchronous mode is enabled, all Client-Library routines that read from or write to the network either:

- Initiate the requested operation and return CS_PENDING immediately, or
- Return CS_BUSY to indicate that an asynchronous operation is already pending for this connection. Non-asynchronous routines also return CS_BUSY if they are called when an asynchronous operation is pending for a connection.

By returning CS_PENDING, a routine indicates that the requested operation has begun and will complete asynchronously. The application receives the completion status from the call either by polling (that is, calling ct_poll periodically) or when Client-Library invokes the application's completion callback. Both methods are described under "Completions" on page 15.

Asynchronous routines

The following Client-Library routines behave asynchronously:

- ct_cancel

- ct_close
- ct_connect
- ct_ds_lookup
- ct_fetch
- ct_get_data
- ct_options
- ct_recvpassthru
- ct_results
- ct_send
- ct_send_data
- ct_sendpassthru

The CS_BUSY return code

Any Client-Library routine that takes a command or connection structure as a parameter returns CS_BUSY. The CS_BUSY response indicates that a routine cannot perform because the relevant connection is currently busy, waiting for an asynchronous operation to complete.

An application calls the following routines while an asynchronous operation is pending:

- Any routine that takes a CS_CONTEXT structure as a parameter. If the CS_CONTEXT structure is an optional parameter, it must be non-NULL.
- ct_cancel(CS_CANCEL_ATTN)
- ct_cmd_props(CS_USERDATA)
- ct_con_props(CS_USERDATA)
- ct_poll

Completions

Every asynchronous mode Client-Library call that returns CS_PENDING produces a completion status. This value corresponds to the return code that would have been returned by a synchronous-mode call to the routine (for example, CS_SUCCEED or CS_FAIL).

An application determines when an asynchronous routine completes either by polling or through completion callbacks. For polling, the application periodically calls ct_poll to determine if the asynchronous call has completed. With completion callbacks, the application is automatically notified when asynchronous calls complete.

To properly exit Client-Library, wait until all asynchronous operations are complete, then call ct_exit.

If an asynchronous operation is in progress when ct_exit is called, the routine returns CS_FAIL and does not exit Client-Library properly, even when CS_FORCE_EXIT is used.

Deferred asynchronous completions

The application polls for the completion status by calling ct_poll periodically until ct_poll indicates that the operation is complete. This mode of operation is called deferred-asynchronous and corresponds to setting the CS_NETIO property to CS_DEFER_IO.

If the application installs a completion callback, the callback routine is called by ct_poll when ct_poll detects the completion. The application itself must call ct_poll.

The application learns the completion status of the asynchronous call from ct_poll. If a completion callback is installed, it also receives the completion status as an input parameter.

Note For a description of the supported asynchronous modes, see the Client-Library chapter in the Open Client and Open Server *Programmers Supplement* for Microsoft Windows and Open Client and Open Server *Programmers Supplement* for UNIX.

Fully asynchronous completions

On platforms where Client-Library uses signal-driven or thread-driven I/O, Client-Library automatically calls the application's completion callback routine when an asynchronous routine completes. This mode of operation is called fully asynchronous and corresponds to setting the CS_NETIO property to CS_ASYNC_IO.

When a connection is fully asynchronous, the application does not have to poll for the completion status. Client-Library automatically invokes the application's completion callback, which receives the completion status as an input parameter. Completion callbacks are described under "Defining a completion callback" on page 35.

Note When Client-Library is used within an Open Server, the CS_NETIO property cannot be set to CS_ASYNC_IO. The Open Server thread scheduler allows multitasking in an Open Server application.

On asynchronous connections, it is possible for Client-Library to complete an asynchronous operation and call the callback routine before the initiating routine returns. When this happens, the initiating routine still returns CS_PENDING, and the application's completion callback receives the completion status.

Client-Library's fully asynchronous operation is either thread-driven or signal-driven. On platforms that do not support either multiple threads or signal-driven I/O, Client-Library cannot be fully asynchronous.

Signal-driven completion handling

On some platforms such as UNIX, Client-Library uses operating system signals (also called interrupts) to read results and send commands over the network. Internally, Client-Library interacts with the network using non-blocking system calls and installs its own internal signal handler to receive the completion status for these system calls.

Note that on signal-driven I/O platforms, Client-Library may be signal-driven even when the CS_NETIO property is not CS_ASYNC_IO. On signal-driven I/O platforms, Client-Library uses signal-driven I/O if any of the following is true:

- The value of the CS_NETIO connection property is CS_ASYNC_IO.

- The value of the CS_ASYNC_NOTIFS connection property is CS_TRUE. The default is CS_FALSE. See “Asynchronous notifications” on page 203 for a description of this property.
- The application has specified a finite timeout limit for the CS_TIMEOUT or CS_LOGIN_TIMEOUT context properties. The default is CS_NO_LIMIT, which is equivalent to infinity. See “Login timeout” on page 216 and “Timeout” on page 230 for a description of these properties.

Warning! When Client-Library uses signal-driven I/O, a signal can interrupt the processing of system calls made by the application. If an error code indicates that a system call was interrupted, reissue the call.

On platforms where signal-driven I/O is used to implement Client-Library’s fully asynchronous mode, fully asynchronous applications have the following restrictions:

- Any signal handlers required by the application must be installed using ct_callback. See “Signal callbacks” on page 57 for more information.
- The application must provide a safe way for Client-Library to obtain memory at the interrupt level. See “Client-Library’s interrupt-level memory requirements” on page 18 for more information.
- On systems where Client-Library uses signal-driven I/O in fully asynchronous mode (UNIX), be sure to check the return value and error code after each system call to make sure that it completed properly. Some system calls fail when interrupted by a signal. If an error code indicates that the call was interrupted, issue the call again. This restriction is not an issue on platforms such as Windows where Client-Library does not use signals.

Thread-driven completion handling

On some platforms, such as Windows, Client-Library uses thread-driven I/O to operate in fully asynchronous mode.

When this I/O strategy is used, Client-Library spawns internal worker threads to interact with the network. When the application calls a routine that requires network I/O, the I/O request is passed to the worker thread. The asynchronous routine then returns CS_PENDING and the worker thread waits for the completion. When the I/O request completes, the worker thread calls the application’s completion callback.

On platforms where thread-driven I/O is used, fully asynchronous applications have the following restrictions:

- All of the application's callback functions installed for each fully asynchronous connection must be thread-safe.
- Because the application's completion callback is invoked by a Client-Library worker thread, the application logic must be designed so that the completion callback communicates with mainline code in a thread-safe manner.

On thread-driven I/O platforms such as Windows, a fully asynchronous program is multithreaded in its callback execution even if the mainline code is single-threaded. For thread-driven I/O, a Client-Library worker thread interacts with the network for each fully asynchronous connection. The worker thread invokes the connection's callbacks for any callback events that it discovers. See "Fully asynchronous completions" on page 16.

Note When fully asynchronous I/O is in effect on platforms where Client-Library uses thread-driven I/O, the application's callbacks are invoked by a Client-Library worker thread. On these platforms, a fully asynchronous application's callbacks are multithreaded even if the application itself uses a single-threaded design.

Issues affecting multithreaded application design are discussed in "Multithreaded programming" on page 158.

Client-Library's interrupt-level memory requirements

On operating systems where Client-Library uses signal-driven I/O, such as UNIX-based systems, fully asynchronous applications must provide a way for Client-Library to satisfy its interrupt-level memory requirements.

Ordinarily, Client-Library routines satisfy their memory requirements by calling malloc. However, not all implementations of malloc are safely called at the interrupt level. For this reason, fully asynchronous applications are required to provide an alternate way for Client-Library to satisfy its memory requirements.

Client-Library provides two mechanisms by which an asynchronous application satisfies Client-Library's memory requirements:

- The application uses the CS_MEM_POOL property to provide Client-Library with a memory pool.
- The application uses the CS_USER_ALLOC and CS_USER_FREE properties to install memory allocation routines that Client-Library safely calls at the interrupt level.

On platforms that use signal-driven I/O, Client-Library's behavior is undefined if a fully asynchronous application fails to provide a safe way for Client-Library to satisfy memory requirements.

Client-Library attempts to satisfy memory requirements from the following sources in the following order:

1. Memory pool
2. User-supplied allocation and free routines
3. System routines

Layered applications

Asynchronous applications are often layered. In these types of applications, the lower layer protects the higher layer from low-level asynchronous detail.

The higher-level layer typically consists of:

- Mainline code
- Routines that asynchronously perform *large* operations.

  In this discussion, a "large" operation is a task that requires several Client-Library calls to complete. For example, updating a database table is a large operation because an application calls ct_command, ct_send, and ct_results to perform the update.

The lower-level layer typically consists of:

- The Client-Library routines required to perform a large operation
- Code to handle low-level asynchronous operation completions

Using ct_wakeup and CS_DISABLE_POLL

ct_wakeup and the CS_DISABLE_POLL property are used in layered asynchronous applications as follows:

- A layered application uses CS_DISABLE_POLL to prevent ct_poll from reporting asynchronous Client-Library routine completions.
- A layered application uses ct_wakeup to let the higher layer know when a large asynchronous operation is complete.

A layered application that is using a routine to perform a large operation typically uses ct_wakeup and CS_DISABLE_POLL as follows:

1. The application performs any necessary initialization, installs callback routines, opens connections, and so on.
2. The application calls the routine that is performing the large operation.
3. If the application uses ct_poll to check for asynchronous completions, then the routine must disable polling. This prevents ct_poll from reporting lower-level asynchronous completions to the higher-level layer. To disable polling, the routine sets CS_DISABLE_POLL to CS_TRUE.

   If the application does not call ct_poll, the routine does not need to disable polling.
4. The routine calls ct_callback to replace the higher-level layer's completion callback with its own completion callback.
5. The routine performs its work.
6. The routine reinstalls the higher-level layer's completion callback.
7. If polling has been disabled, the routine enables it again by setting the CS_DISABLE_POLL property to CS_FALSE.
8. The routine calls ct_wakeup to trigger the higher-level layer's completion callback routine.

An example

An application that performs asynchronous database updates might include the routine do_update, where do_update calls all of the Client-Library routines that are necessary to perform a database update.

The main application calls do_update asynchronously and goes on with its other work.

When called,do_update replaces the main application's completion callback routine with its own callback (so that the main application's callback routine is not triggered by low-level asynchronous completions). Then, it proceeds with the work of the update. To perform the update, do_update calls several Client-Library routines, including ct_send and ct_results, which behave asynchronously. When each asynchronous routine completes, it triggers do_update's completion callback.

When do_update has finished the update operation, it reinstalls the main application's completion callback and calls ct_wakeup with function as its own function ID. This triggers the main application's completion callback, letting the main application know that do_update has completed.

Browse mode

Note Browse mode is included in Client-Library to provide compatibility with Open Server applications and older Open Client libraries. Sybase discourages its use in new Open Client Client-Library applications because cursors provide the same functionality in a more portable and flexible manner. Further, browse mode is Sybase-specific and is not suited for use in a heterogeneous environment.

Browse mode provides a means for browsing through database rows and updating their values one row at a time. From the standpoint of an application program, the process involves several steps, because each row must be transferred from the database into program variables before it can be browsed and updated.

Because a row being browsed is not the actual row residing in the database but is a copy residing in program variables, the program must be able to ensure that changes to the variables' values are reliably used to update the original database row. In particular, in multiuser situations, the program needs to ensure that updates made to the database by one user do not unwittingly overwrite updates made by another user between the time the program selected the row and sent the command to update it. A timestamp column in browsable tables provides the information necessary to regulate this type of multiuser updating.

Because some applications permit users to enter ad hoc browse mode queries, Client-Library provides two routines, ct_br_table and ct_br_column, that allow an application to retrieve information about the tables and columns underlying a browse-mode result set. This information is useful when an application is constructing commands to perform browse-mode updates.

A browse-mode application requires two connections, one for selecting the data and one for performing the updates.

For more information on browse mode, see the Adaptive Server Enterprise *Reference Manual*.

Using Browse mode

Conceptually, using Browse mode involves two steps:

1. Select rows containing columns derived from one or more database tables.
2. Where appropriate, change values in columns of the result rows (not the actual database rows), one row at a time, and use the new values to update the original database tables.

These steps are implemented in a program as follows:

1. Set a connection's CS_HIDDEN_KEYS property to CS_TRUE. This ensures that Client-Library returns a table's *timestamp* column as part of a result set. In browse-mode updates, the *timestamp* column is used to regulate multiuser updates.
2. Execute a select...for browse language command. This command generates a regular row result set. This result set contains hidden key columns (one of which is the *timestamp* column) in addition to explicitly selected columns.
3. After ct_results indicates regular row results, call ct_describe to get CS_DATAFMT descriptions of the result columns:
   - To indicate the *timestamp* column, ct_describe sets the CS_TIMESTAMP and CS_HIDDEN bits in the *datafmt–>status field.
   - To indicate an ordinary hidden key column, ct_describe sets the CS_HIDDEN bit in the *datafmt–>status field. If the CS_HIDDEN bit is not set, the column is an explicitly selected column.

4. Call ct_bind to bind the result columns of interest. An application must bind all hidden columns because it requires these column values to build a where clause at update time.
5. Call ct_br_table, if necessary, to retrieve information about the database tables that underlie the result set. Call ct_br_column, if necessary, to retrieve information about a specific result set column. Both of these types of information are useful when building a language command to update the database.
6. Call ct_fetch in a loop to fetch rows. When a row is fetched that contains values that need to be changed, update the database table(s) with the new values. To do this:
   - Construct a language command containing a Transact-SQL update statement with a where clause that uses the row's hidden columns (including the *timestamp* column).
   - Send the language command to the server and process the results of the command.

     A language command containing a browse-mode update statement generates a result set of type CS_PARAM_RESULT. This result set contains a single result item, the new timestamp for the row.

     If the application plans to update this same row again, it must save the new timestamp for later use.

After one browse-mode row has been updated, the application fetches and process the next row.

The Browse mode *where* clause

To perform browse-mode updates, the application sends an update language command with the where clause formatted as follows:

```
where key1 = value_1
 and key2 = value_2 ...
 and tsequal(timestamp, ts_value)
```

where:

- *key1*, *value_1*, *key2*, *value_2* and so forth are the key columns and their values, obtained by calls to ct_br_table and ct_br_column.
- *ts_value* is the binary timestamp column value converted to a character string.

Browse mode conditions

The following conditions must be true to use browse mode:

- The select command that generated the result set must end with the keywords for browse.
- The table(s) to be updated must be browsable (each must have a unique index and a timestamp column).
- The result columns to be updated cannot be the result of SQL expressions, such as colname + 1.

Callbacks

Callbacks are user-supplied routines that are automatically called by Client-Library when certain triggering events, known as *callback events*, occur.

Some callback events are the result of a server response arriving for an application. For example, a notification callback event occurs when a registered procedure notification arrives from an Open Server.

Other callback events occur at the internal Client-Library level. For example, a client message callback event occurs when Client-Library generates an error message.

When Client-Library recognizes a callback event, it calls the appropriate callback routine.

Client-Library must be actively engaged in reading from the network to recognize some callback events. Most callback events of this type are raised automatically when Client-Library is reading results from the network.

However, for applications that use Client-Library's asynchronous modes, or that use Open Server registered procedure notifications, two types of callback events may require special handling:

- The completion callback event, which occurs in asynchronous mode applications when an asynchronous Client-Library routine completes. Depending on the operating system, applications either receive completions automatically or by polling. See "Completions" on page 15.

- The notification callback event, which occurs when an Open Server notification arrives for an application. Applications must take special steps to ensure that they receive notification events. See “Receiving notifications asynchronously” on page 240.

Note Because some types of callback routines are executed from within a system interrupt handler or from a Client-Library worker thread, you must code applications so that data accessed by both the application’s mainline code and the callbacks is safely shared.

Callback types

Table 2-1 lists the types of callbacks and when they are called:

Table 2-1: Types of callbacks

Callback type	When called	How called
Client message	In response to a Client-Library error or informational message	When Client-Library generates an error or informational message, Client-Library automatically triggers the client message callback. See “Client message callbacks” on page 31.
Completion	When an asynchronous Client-Library routine completes	An asynchronous routine completion can occur at any time. On platforms that support signal- or thread-driven I/O, the completion callback is called automatically when the completion occurs. On platforms that do not support signal- or thread-driven I/O, an application must use ct_poll to find out if any routines have completed. See “Completion callbacks” on page 34.
Directory	During a directory search that began when the application called ct_ds_lookup	Called automatically by Client-Library to pass the application the directory objects that were found in the search. On an asynchronous connection, called before the completion callback. On a synchronous connection, called before ct_ds_lookup returns. Client-Library invokes the callback repeatedly until: • The callback has received all directory objects found in the lookup operation, or • The callback returns CS_SUCCEED. See “Directory callbacks” on page 39.
Encryption	During the connection process, in response to a server request for an encrypted password	If password encryption is enabled and an encryption callback is installed, then Client-Library automatically triggers the encryption callback when a server requests an encrypted password during a connection attempt. If encryption is enabled and an encryption callback is not installed, then Client-Library performs the default password encryption. For details, see “Encryption callbacks” on page 41.

Callback type	When called	How called
Negotiation	During the connection process: • In response to a server request for login security labels • In response to a server challenge	If a connection's CS_SEC_NEGOTIATE property is CS_TRUE, then Client-Library automatically triggers the negotiation callback when a server requests login security labels during a connection attempt. If a connection's CS_SEC_CHALLENGE property is CS_TRUE, then Client-Library automatically triggers the negotiation callback when a server issues a challenge during a connection attempt. For details, see "Negotiation callbacks" on page 45.
Notification	When an Open Server notification arrives	An Open Server notification can arrive at any time. Client-Library reads the notification information and calls the applications's notification callback. The CS_ASYNC_NOTIFS property determines how the notification callback is triggered. See the description of this property under "Asynchronous notifications" on page 203 and "Notification callbacks" on page 48.
Security session	During the connection process, when the connection uses network-based security services	Invoked automatically by ct_connect in response to a security session challenge from the target server. For details, see "Security session callbacks" on page 50. **Note** Security session callbacks are required only in gateway applications that set up direct security sessions between their own client and a remote server.
Server message	In response to a server error or informational message	Server messages occur as the result of specific commands. When an application processes the results of a command, Client-Library reads any error or informational messages related to the command, automatically triggering the server message callback. For details, see "Server message callbacks" on page 53.

Callback type	When called	How called
Signal	In response to an operating-system signal	When a signal handler has been installed with ct_callback and a signal arrives, Client-Library's own signal handler automatically calls the signal callback. On platforms that support signals, applications must call ct_callback to install any needed signal handlers. For details on signal callbacks, see "Signal callbacks" on page 57.
SSL validation	During the connection process, when the connection uses SSL session-based security services.	Invoked automatically by ct_connect during the SSL handshake. SSL validation callback is installed with ct_callback using CS_SSLVALIDATE_CB. For details on SSL validation callbacks, see "SSL validation callbacks" on page 59.

Callbacks are not always supported

Callbacks cannot be implemented for programming language and platform combinations that do not support function calls by pointer reference. If this is the case, an application:

- Must handle Client-Library and server messages inline, using ct_diag.
- Still uses ct_poll to check for a completion or notification callback event, but must directly call any routine handling the event.

To determine whether callbacks are supported for a programming language and platform version of Client-Library, use the Open Client and Open Server *Programmers Supplement* for Microsoft Windows or Open Client and Open Server *Programmers Supplement* for UNIX.

Installing a callback routine

Applications must be coded to install any needed runtime callbacks. An application installs a callback routine by calling ct_callback, passing a pointer to the callback routine, and indicating its type using the type parameter.

A callback of a particular type can be installed at the context or connection level. When a connection is allocated, it picks up default callbacks from its parent context. An application overrides these default callbacks by calling ct_callback to install new callbacks at the connection level.

When a callback event occurs

For most types of callbacks, when a callback event occurs:

- If a callback of the proper type exists at the proper level, it is called.
- If a callback of the proper type does not exist at the proper level then the callback event information is discarded.

The client message callback is an exception to this rule. When an error or informational message is generated for a connection that has no client message callback installed, Client-Library calls the connection's parent context's client message callback (if any) rather than discarding the message. If the context has no client message callback installed, then the message is discarded.

Retrieving and replacing callback routines

To retrieve a pointer to a currently installed callback, call ct_callback with the parameter action as CS_GET. ct_callback sets *func to the address of the current callback. An application saves this address for reuse at a later time.

To deinstall a callback, call ct_callback with the parameter action as CS_SET and func as NULL.

To replace an existing callback routine with a new one, call ct_callback to install the new routine. ct_callback replaces the existing callback with the new callback.

Restrictions on Client-Library calls in callbacks

All callback routines are limited as to which Client-Library routines they can call, as indicated in Table 2-2:

Table 2-2: Callbacks can call these Client-Library routines

Callback type	Callable routines	Permitted use
All callback routines	ct_config	To retrieve information only.
	ct_con_props	To retrieve information or to set the CS_USERDATA property only.
	ct_cmd_props	To retrieve information only. The CS_USERDATA property can be set on command structures allocated with ct_cmd_alloc. The CS_USERDATA property cannot be set on command structures obtained by the call's to ct_con_props(CS_EED_CMD) or ct_con_props(CS_NOTIF_CMD).
	ct_cancel (CS_CANCEL_ATTN)	
Server message	ct_describe	The routines must be called with the command structure returned by the callback's ct_con_props(CS_EED_CMD) call. See "Extended error data" on page 129.
Notification	ct_bind, ct_describe, ct_fetch, ct_get_data, ct_res_info(CS_NUMDATA)	The routines must be called with the command structure returned by the callback's ct_con_props(CS_NOTIF_CMD) call. This command structure allows the application to retrieve parameter values associated with the notification event. See "Registered procedures" on page 238.
Completion	Any Client-Library or CS-Library routine except cs_objects(CS_SET), ct_init, ct_exit, ct_poll, ct_setloginfo, and ct_getloginfo.	**Note** cs_objects(CS_SET) is not asynchronous-safe, and ct_init, ct_exit, and ct_getloginfo perform system-level memory allocation or deallocation, and should not be used.
Directory	ct_ds_dropobj, ct_ds_objinfo.	To drop or inspect a directory object.

Declaring callbacks with CS_PUBLIC

All of an application's Client-Library and CS-Library callbacks must be declared with CS_PUBLIC. On some platforms (such as Windows), a compiler may use one of many calling conventions for functions in generated code. A function's calling convention determines how the machine registers and the machine stack are manipulated when the function is called. The compiler generates different machine instructions for different calling conventions. CS_PUBLIC (along with any required compiler options) ensures that the application's callbacks are compiled with the same calling convention with which Client-Library invokes them.

Note Compiler options are described in the Open Client and Open Server *Programmers Supplement* for Microsoft Windows or the Open Client and Open Server *Programmers Supplement* for UNIX.

On many platforms, CS_PUBLIC is defined such that it adds nothing to a function declaration. On these platforms, applications that declare callbacks with CS_PUBLIC behave no differently than those that omit CS_PUBLIC. However, for portability, CS_PUBLIC should be used to declare callbacks on any platform.

Client message callbacks

An application handles Client-Library error and informational messages inline or through a client message callback routine.

When a connection is allocated, it picks up a default client message callback from its parent context. If the parent context has no client message callback installed, then the connection is created without a default client message callback.

After allocating a connection, an application:

- Installs a different client message callback for the connection.
- Calls ct_diag to initialize inline message handling for the connection. Note that ct_diag automatically de-installs all message callbacks for the connection.

If a client message callback is not installed for a connection or its parent context and inline message handling is not enabled, Client-Library discards message information.

If callbacks are not implemented for a particular programming language or platform version of Client-Library, an application must handle Client-Library messages inline, using ct_diag.

If a connection is handling Client-Library messages through a client message callback, then the callback is called whenever Client-Library generates an error or informational message.

Note The exception to this rule is that Client-Library does not call the client message callback when a message is generated from within most types of callback routines. Client-Library does call the client message callback when a message is generated within a completion callback. That is, if a Client-Library routine fails within a callback routine other than the completion callback, the routine returns CS_FAIL but does not trigger the client message callback.

Defining a client message callback

A client message callback is defined as follows:

```
CS_RETCODE CS_PUBLIC clientmsg_cb(context, connection,
                                  message)

CS_CONTEXT            *context;
CS_CONNECTION         *connection;
CS_CLIENTMSG          *msg;
```

where:

- *context* is a pointer to the CS_CONTEXT structure for which the message occurred.
- *connection* is a pointer to the CS_CONNECTION structure for which the message occurred. *connection* can be NULL.
- *message* is a pointer to a CS_CLIENTMSG structure containing Client-Library message information. For information about this structure, see the section, “Client-Library and SQL Structures” on page 74.

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

A client message callback must return either CS_SUCCEED or CS_FAIL:

- CS_SUCCEED instructs Client-Library to continue any processing that is occurring on this connection.

If the callback was invoked due to a timeout error, returning CS_SUCCEED causes Client-Library to wait for the duration of a full timeout period before calling the client message callback again. It continues this behavior until either the command succeeds without timing out or until the server cancels the current command in response to a ct_cancel(CS_CANCEL_ATTN) call from the client message callback.

Note In some cases a server may be unable to respond to a client's ct_cancel command. Such a situation can occur, for example, if the server is processing a very complex query and is not in an interruptible state.

- CS_FAIL instructs Client-Library to terminate any processing that is currently occurring on this connection. A return of CS_FAIL results in the connection being marked as "dead", or unusable. To continue using the connection, the application must close the connection and reopen it.

Table 2-3 lists the Client-Library routines that a client message callback can call:

Table 2-3: Routines that a client-message callback can call

Callable routine	Permitted use
ct_config	To retrieve information only
ct_con_props	To retrieve information or to set the CS_USERDATA property only
ct_cmd_props	To retrieve information or to set the CS_USERDATA property only
ct_cancel (CS_CANCEL_ATTN)	Any circumstances

Most applications use a client message callback that simply displays the error details or logs them to a file. However, some applications may require a callback that recognizes certain errors and takes specific action. See "Handling specific Client-Library messages" on page 84 for more information on how this is done.

Client message callback example

This is an example of a client message callback:

```
/*
 ** ex_clientmsg_cb()
 **
 ** Type of function:
 **      Example program client message handler
```

```
 **
 ** Purpose:
 **      Installed as a callback into Open Client.
 **
 ** Returns:
 **      CS_SUCCEED
 **
 ** Side Effects:
 **      None
 */

CS_RETCODE CS_PUBLIC
 ex_clientmsg_cb(context, connection, errmsg)
 CS_CONTEXT        *context
 CS_CONNECTION     *connection;
 CS_CLIENTMSG      *errmsg;
 {
     fprintf(EX_ERROR_OUT, "\nOpen Client Message:\n");
     fprintf(EX_ERROR_OUT, "Message number:
         LAYER = (%ld) ORIGIN = (%ld) ",
         CS_LAYER(errmsg->msgnumber),
         CS_ORIGIN(errmsg->msgnumber));
     fprintf(EX_ERROR_OUT, "SEVERITY = (%ld)
         NUMBER = (%ld)\n",
         CS_SEVERITY(errmsg->msgnumber),
         CS_NUMBER(errmsg->msgnumber));
     fprintf(EX_ERROR_OUT, "Message String: %s\n",
         errmsg->msgstring);
     if (errmsg->osstringlen > 0)
     {
         fprintf(EX_ERROR_OUT, "Operating System \
             Error: %s\n", errmsg->osstring);
     }

    return CS_SUCCEED;
 }
```

Completion callbacks

A completion callback signals an application that an asynchronous routine has completed.

A context or a connection is defined to be asynchronous so that routines that read to or write from the network return immediately rather than blocking until the necessary I/O operations have completed. The value of a connection structure's CS_NETIO property determines whether Client-Library routines behave asynchronously. See "Network I/O" on page 219 for details.

When a connection is asynchronous, Client-Library routines that perform network I/O return CS_PENDING immediately rather than completing the requested operation before returning. In a fully asynchronous application (CS_NETIO is CS_ASYNC_IO), a completion callback is needed to notify the mainline code of the asynchronous operation's completion.

For more information on Client-Library's asynchronous programming modes, see "Asynchronous programming" on page 12.

Defining a completion callback

A completion callback is defined as follows:

```
CS_RETCODE CS_PUBLIC completion_cb(connection, cmd,
                                   function,status)

CS_CONNECTION       *connection;
CS_COMMAND          *cmd;
CS_INT              function;
CS_RETCODE          status;
```

where:

- *connection* is a pointer to the CS_CONNECTION structure representing the connection that performed the I/O for the routine.
- *cmd* is a pointer to the CS_COMMAND structure for the routine. *cmd* can be NULL.
- *function* indicates which routine has completed. Table 2-4 on page 36 lists the symbolic values possible for *function*:

Table 2-4: Values for the completion callback function parameter

Value	Meaning
BLK_DONE	blk_done has completed.
BLK_INIT	blk_init has completed.
BLK_ROWXFER	blk_rowxfer has completed.
BLK_SENDROW	blk_sendrow has completed.
BLK_SENDTEXT	blk_sendtext has completed.
BLK_TEXTXFER	blk_textxfer has completed
CT_CANCEL	ct_cancel has completed.
CT_CLOSE	ct_close has completed.
CT_CONNECT	ct_connect has completed.
CT_DS_LOOKUP	ct_ds_lookup has completed.
CT_FETCH	ct_fetch has completed.
CT_GET_DATA	ct_get_data has completed.
CT_OPTIONS	ct_options has completed.
CT_RECVPASSTHRU	ct_recvpassthru has completed.
CT_RESULTS	ct_results has completed.
CT_SEND	ct_send has completed.
CT_SEND_DATA	ct_send_data has completed.
CT_SENDPASSTHRU	ct_sendpassthru has completed.
A user-defined value. This value must be greater than or equal to CT_USER_FUNC.	A user-defined function has completed.

- *status* is the completion status of the completed routine. This value corresponds to the value that would be returned by a synchronous call under the same conditions. To find out what values *status* can have, see “Returns” on the reference page for the routine that corresponds to the value of the *function* parameter.

  If the application calls ct_wakeup to invoke the completion callback, the call to ct_wakeup specifies the status value received by the completion callback.

A completion callback routine calls any Client-Library or CS-Library routine except cs_objects (CS_SET), ct_init, ct_exit, ct_setloginfo, and ct_getloginfo. cs_objects(CS_SET) is not asynchronous-safe, and ct_init, ct_exit, ct_setloginfo, and ct_getloginfo perform system-level memory allocation and deallocation.

If a completion callback calls an asynchronous Client-Library routine, it should return the value returned by the routine itself. Otherwise, there are no restrictions on what a completion callback can return. Sybase recommends, however, that the completion callback return CS_SUCCEED, if the completion callback succeeded, or CS_FAIL, if an error occurred.

Completion callback example

The following is an example of a completion callback. This code is from the Client-Library sample programs (file *ex_alib.c*):

```
/*
** ex_acompletion_cb()
**
** Type of function:
**       example async lib
**
** Purpose:
**       Installed as a callback into Open Client. It
**       will dispatch to the appropriate completion
**       processing routine based on async state.
**
**       Another approach to callback processing is to
**       have each completion routine install the
**       completion callback for the next step in
**       processing. We use one dispatch point to aid
**       in debugging the async processing (only need
**       to set one breakpoint).
**
** Returns:
**       Return of completion processing routine.
**
** Side Effects:
**       None
*/

CS_STATIC CS_RETCODE CS_PUBLIC
 ex_acompletion_cb(connection, cmd, function, status)
 CS_CONNECTION    *connection;
 CS_COMMAND       *cmd;
 CS_INT           function;
 CS_RETCODE       status;
 {
      CS_RETCODE  retstat;
      ExAsync     *ex_async;

     /*
```

```
        ** Extract the user area out of the command
        ** handle.
        */
        retstat = ct_cmd_props(cmd, CS_GET, CS_USERDATA,
             &ex_async, CS_SIZEOF(ex_async), NULL);
        if (retstat != CS_SUCCEED)
        {
             return retstat;
        }

       fprintf(stdout, "\nex_acompletion_cb: function \
             %ld Completed", function);

       /* Based on async state, do the right thing */
        switch ((int)ex_async->state)
        {
             case EX_ASEND:
             case EX_ACANCEL_CURRENT:
             retstat = ex_asend_comp(ex_async, connection,
                  cmd, function, status);
             break;

            case EX_ARESULTS:
             retstat = ex_aresults_comp(ex_async,
                  connection, cmd, function, status);
             break;

            case EX_AFETCH:
             retstat = ex_afetch_comp(ex_async,
                  connection, cmd, function, status);
             break;

            case EX_ACANCEL_ALL:
             retstat = ex_adone_comp(ex_async, connection,
                  cmd, function, status);
             break;

            default:
             ex_apanic("ex_acompletion_cb: unexpected \
                  async state");
             break;
        }

       return retstat;
}
```

Directory callbacks

The ct_ds_lookup routine and the application's directory callback provide the mechanism which an application uses to examine the contents of directory entries.

When an application calls ct_ds_lookup to begin a directory search, Client-Library retrieves the appropriate entries from the directory and then calls the directory callback once for each entry. Each time the callback is invoked, it receives a pointer to one directory object structure. Each directory object structure contains a copy of information read from a directory entry.

Client-Library calls the directory callback once for each entry retrieved, as long as the callback returns CS_CONTINUE. When the callback returns CS_SUCCEED, Client-Library discards any remaining objects that the callback has not received.

The directory callback calls only the Client-Library routines ct_con_props, ct_config, ct_ds_objinfo, and ct_ds_dropobj. On an asynchronous connection, the application uses the completion callback to call other Client-Library routines (see Table 2-2 on page 30).

Defining a directory callback

A directory callback is defined as follows:

```
CS_RETCODE CS_PUBLIC
   directory_cb (connection, reqid, status, numentries,
                 ds_object, userdata)

CS_CONNECTION      *connection;
CS_INT             reqid;
CS_RETCODE         status;
CS_INT             numentries;
CS_DS_OBJECT       *ds_object;
CS_VOID            *userdata;
```

where:

- *connection* is a pointer to the CS_CONNECTION structure used for the directory lookup.
- *reqid* is the request identifier returned by the ct_ds_lookup call that began the directory lookup.
- *status* is the status of the directory lookup request. *status* can be one of the following values:

Status value	Meaning
CS_SUCCEED	Search was successful.
CS_FAIL	Search failed.
CS_CANCELED	Search was canceled with ct_ds_lookup(CS_CLEAR).

- *numentries* is the count of directory objects remaining to be examined. If entries are found, *numentries* includes the current object. If no entries are found, *numentries* is 0.
- *ds_object* is a pointer to information about one directory object. *ds_object* is (CS_DS_OBJECT *)NULL if either of the following is true:
  - The directory lookup failed (indicated by a *status* value that is not equal to CS_SUCCEED), or
  - No matching objects were found (indicated by a *numentries* value that is 0 or less).
- *userdata* is a pointer to a user-supplied data area. If the application passes a pointer as ct_ds_lookup's *userdata* parameter, then the directory callback receives the same pointer when it is invoked. *userdata* provides a way for the callback to communicate with mainline code.

Directory search results processing

A directory callback typically performs the following to collect and optionally process the results of a directory search:

1. Checks the values of *status* and *numentries* to determine whether the search was successful and whether entries were returned.
   - A *status* value of CS_SUCCEED indicates that the search was successful.
   - *A numentries* value greater than 0 indicates that entries were found.
2. Either saves the pointer to the directory object; or copies any information that it wants to keep (using ct_ds_objinfo to extract the information), then frees the directory object's memory with ct_ds_dropobj.
3. Returns control to Client-Library in one of the following ways:
   - Returns CS_SUCCEED to drop all remaining unexamined entries
   - Returns CS_CONTINUE so that Client-Library calls the callback routine again to process the next object returned by the directory search

Callback invocation sequence

If a search is successful, Client-Library invokes the directory callback with *numentries* as the total number of entries to be examined. If the search finds no entries, *numentries* is 0. If the search finds one or more entries, *numentries* gives the number of unexamined entries including the current entry.

The application examines all the entries simply by returning CS_CONTINUE from the callback each time Client-Library invokes the callback. ct_ds_lookup invokes the callback repeatedly until one of the following conditions is satisfied:

- The callback returns CS_SUCCEED.
- The callback has received every directory object in the search results. If the callback returns CS_CONTINUE when *numentries* is 0 or 1, it is not invoked again before ct_ds_lookup completes.
- If the callback returns a value other than CS_CONTINUE or CS_SUCCEED, the current Client-Library response is the same as for CS_SUCCEED. However, this behavior may change in future versions. To ensure compatibility with future versions, applications should return only CS_CONTINUE or CS_SUCCEED from directory callbacks.

If asynchronous network I/O is in effect for the connection, all invocations of the directory callback occur before Client-Library invokes the application's completion callback.

If synchronous network I/O is in effect for the connection, all invocations of the directory callback occur before ct_ds_lookup returns.

Directory callback example

Directory callbacks are used with ct_ds_lookup. See the ct_ds_lookup reference page for an example directory callback.

Encryption callbacks

Adaptive Server version 10.0 and later uses an encrypted password handshake when the client requests it. Servers based on Open Server version 10.0 or later may also use this feature.

The client application must enable password encryption by calling ct_con_props and setting the CS_SEC_EXTENDED_ENCRYPTION or CS_SEC_ENCRYPTION property. If an Open Client application logs onto a server with both CS_SEC_EXTENDED_ENCRYPTION and CS_SEC_ENCRYPTION set to CS_TRUE, it uses extended password encryption as the first preference.

The Client-Library default encryption handler performs the password encryption required by Adaptive Server. Simple client applications that connect to either of these servers do not need an encryption callback. However, Client-Library applications that act as gateways to Adaptive Server need to handle password encryption explicitly. These applications must install an encryption callback routine that passes the server's encryption key to the client and returns the encrypted password back to the server. See "Password encryption in gateway applications" on page 44.

In addition, Client-Library applications that connect to an Open Server using a customized password encryption technique must install an encryption callback routine to perform the required password encryption.

For an explanation of the handshaking process for password encryption, see "Security handshaking: encrypted password" on page 274.

Note Do not confuse password encryption with data encryption. An encryption callback encrypts only passwords. Data encryption encrypts all commands and results sent over the connection and is performed by an external security service provider. See "Security features" on page 251 for more information.

Defining an encryption callback

The encryption callback prototype for extended and normal password encryption are defined below.

Normal password encryption

```
CS_RETCODE CS_PUBLIC encrypt_cb(connection, pwd,
                pwdlen, key, keylen, buf, buflen, outlen)


CS_CONNECTION        *connection;
CS_BYTE              *pwd;
CS_INT               pwdlen;
CS_BYTE              *key;
CS_INT               keylen;
```

```
CS_BYTE              *buffer;
CS_INT               buflen;
CS_INT                *outlen;
```

where:

- *connection* is a pointer to the CS_CONNECTION structure representing the connection that is logging in to the server.
- *pwd* is a user password or a remote-server password to be encrypted. A user password matches the value of the CS_PASSWORD connection property. A remote-server password matches the string passed to ct_remote_pwd. The *pwd* string is not always null-terminated.
- *pwdlen* is the length, in bytes, of the password.
- *key* is the key that the encryption callback uses to encrypt the password. The encryption key is supplied by the remote server.
- *keylen* is the length, in bytes, of the encryption key.
- *buffer* is a pointer to a buffer. The encryption callback should place the encrypted password in this buffer. This buffer is allocated and freed by Client-Library. Its length is described by *buflen*.
- *buflen* is the length, in bytes, of the **buffer* data space.
- *outlen* is a pointer to a CS_INT. The encryption callback must set **outlen* to the length of the encrypted password in **buffer*.

Extended password encryption

```
CS_RETCODE extended_encrypt_cb(
      CS_CONNECTION *connection,
      CS_BYTE       *pwd,
      CS_INT        pwdlen,
      CS_INT        *ciphersuite,
      CS_BYTE       *pubkey,
      CS_INT        pubkeylen,
      CS_VOID       *buffer,
      CS_INT        buflen,
      CS_INT         *outlen)
```

where:

- *connection* is a pointer to the CS_CONNECTION structure representing the connection that is logging in to the server.

- *pwd* is a user password or a remote-server password to be encrypted. A user password matches the value of the CS_PASSWORD connection property. A remote-server password matches the string passed to ct_remote_pwd. The *pwd* string is not always null-terminated.
- *pwdlen* is the length, in bytes, of the password.
- *ciphersuite* is a pointer to the ciphersuite used to encrypt the password. This parameter is not used by the default encryption.
- *pubkey* is a pointer to the public key used to encrypt the password.
- *pubkeylen* is the length, in bytes, of the public key.
- *buffer* is a pointer to a buffer. The encryption callback should place the encrypted password in this buffer. This buffer is allocated and freed by Client-Library. Its length is described by *buflen*.
- *buflen* is the length, in bytes, of the **buffer* data space.
- *outlen* is a pointer to a CS_INT to store the length of the newly accepted password. The encryption callback must set **outlen* to the length of the encrypted password in **buffer*.

An encryption callback should return CS_SUCCEED to indicate that the password has been successfully encrypted. If the encryption callback returns a value other than CS_SUCCEED, Client-Library aborts the connection attempt, causing ct_connect to return CS_FAIL.

Password encryption in gateway applications

To handle encrypted passwords, a gateway application must:

- Supply an encryption callback routine.
- Call ct_callback to install the encryption callback either at the context level or for a specific connection.
- Call ct_con_props to set the CS_SEC_EXTENDED_ENCRYPTION or CS_SEC_ENCRYPTION property to CS_TRUE.

When the gateway calls ct_connect to connect to the remote server:

1. The remote server responds with an encryption key, causing Client-Library to trigger the encryption callback.
2. The encryption callback passes the key on to the gateway's client.
3. The gateway's client encrypts the password and returns it to the encryption callback.

4 The encryption callback places the encrypted password into **buffer*, sets **outlen*, and returns a status code to Client-Library.

- If the callback returns CS_SUCCEED, Client-Library sends the encrypted password to the remote server.
- If the callback returns CS_FAIL, Client-Library aborts the connection process, causing ct_connect to return CS_FAIL.

Client-Library calls the encryption once to encrypt the password defined by CS_PASSWORD, and one additional time for each remote server password defined by ct_remote_pwd.

A gateway to Adaptive Server must take special steps to make sure that encrypted remote passwords are handled correctly. The first time the encryption callback is called for a connect attempt, the gateway must perform the following actions:

1 Clear the default remote password with ct_remote_pwd (CS_CLEAR).

ct_connect creates a default remote password if the gateway has defined no remote passwords before calling ct_connect. The gateway must clear this default.

2 Challenge the gateway's client for encrypted local and remote passwords with srv_negotiate.

3 Call ct_remote_pwd once for each encrypted remote password.

4 Place the encrypted local password into **buffer* and set **outlen* to its length.

5 Return CS_SUCCEED if no error occurred.

Each subsequent invocation of the callback should return one of the encrypted remote passwords read from the gateway's client in response to the challenge.

A gateway forwards the encryption key and reads the client's response with Server-Library calls. See srv_negotiate in the Open Server *Server-Library/C Reference Manual*.

See "Choosing a network security mechanism" on page 253 for more information.

Negotiation callbacks

Client-Library uses the negotiation callback to handle both trusted-user security handshakes and challenge/response security handshakes.

For more information on these types of handshakes, see the “Security features” on page 251.

Challenge/response security handshakes

During server login, a challenge/response security handshake occurs when the server issues a challenge, to which the client must respond.

A connection uses a negotiation callback to provide its response to the challenge. To do this, the connection installs a negotiation callback routine. At connection time, when Client-Library receives the server challenge, Client-Library triggers the negotiation callback.

A connection that participates in challenge/response security handshakes must have its CS_SEC_CHALLENGE property or its CS_SEC_APPDEFINED property set to CS_TRUE.

When the application calls ct_connect to connect to the server:

1. If the server replies with a challenge, then Client-Library calls the connection’s negotiation callback routine.
2. The negotiation callback routine generates the response and returns either CS_CONTINUE, CS_SUCCEED, or CS_FAIL.
   - If the callback routine returns CS_CONTINUE, Client-Library calls the negotiation callback again to get an additional response.
   - If the callback returns CS_SUCCEED, Client-Library sends the response(s) to the server.
   - If the callback returns CS_FAIL, Client-Library aborts the connection process, causing ct_connect to return CS_FAIL.

Defining a negotiation callback

A negotiation callback is defined as follows:

```
CS_RETCODE CS_PUBLIC
     negotiation_cb(connection, inmsgid, outmsgid,
                    inbuffmt, inbuf,outbuffmt,
                    outbuf, outbufoutlen)

CS_CONNECTION      *connection;
CS_INT             inmsgid;
CS_INT             *outmsgid;
CS_DATAFMT         *inbuffmt;
```

```
CS_BYTE             *inbuf;
CS_DATAFMT          *outbuffmt;
CS_BYTE             *outbuf;
CS_INT              *outbufoutlen;
```

where:

- *connection* is a pointer to the CS_CONNECTION structure representing the connection that is logging into the server.
- *inmsgid* is the type of information that the server is requesting. *inmsgid* can be any of the following values:

Value of inmsgid	Meaning
CS_MSG_GETLABELS	The server is requesting security labels.
A value < CS_USER_MSGID	The server is requesting a Sybase-defined value.
A user-defined value >= CS_USER_MSGID and <= CS_USER_MAX_MSGID	The Open Server application is requesting an application-defined value. The negotiation callback's must interpret *inmsgid*.

- *outmsgid* is the type of information that the negotiation callback is returning. This table lists the values that are legal for *outmsgid*:

Value of outmsgid	Indicates:
CS_MSG_LABELS	The negotiation callback is returning security labels.
A value < CS_USER_MSGID	The callback is returning a Sybase-defined value.
A user-defined value >= CS_USER_MSGID and <= CS_USER_MAX_MSGID	The callback is returning an application-defined value.

- *inbuffmt* is a pointer to a CS_DATAFMT structure. If the negotiation callback is handling a trusted-user handshake, *inbuffmt* is NULL. If the negotiation callback is handling a challenge/response handshake, **inbuffmt* describes the *inbuf* challenge key.
- *inbuf* is a pointer to data space. If the negotiation callback is handling a trusted-user handshake, *inbuf* is NULL. If the negotiation callback is handling a challenge/response handshake, *inbuf* points to the challenge key.

- *inbuffmt* is a pointer to a CS_DATAFMT structure. The negotiation callback should fill this CS_DATAFMT with a description of the security label or response that it is returning.

  Client-Library does not define which fields in the CS_DATAFMT need to be set.

- *outbuf* is a pointer to a buffer. The negotiation callback should place the security label or response in this buffer. This buffer is allocated and freed by Client-Library. Its length is described by *outbuffmt–>maxlength*.
- *outbufoutlen* is the length, in bytes, of the data placed in **outbuf*.

A negotiation callback must return CS_SUCCEED, CS_FAIL, or CS_CONTINUE:

- If the callback returns CS_CONTINUE, Client-Library calls the negotiation callback again to generate an additional security label or response.
- If the callback returns CS_SUCCEED, Client-Library sends the security label(s) or response(s) to the server.
- If the callback returns CS_FAIL, Client-Library aborts the connection process, causing ct_connect to return CS_FAIL.

Notification callbacks

A registered procedure is a procedure that is defined and installed in a running Open Server. A Client-Library application uses a remote procedure call command to execute a registered procedure, and also “watches” for a registered procedure to be executed by another application or by the application itself.

To watch for the execution of a registered procedure, a Client-Library application must be connected to the host Open Server. The client application remotely calls the Open Server sp_regwatch system registered procedure.

When a registered procedure executes, applications watching for it receive a notification that includes the procedure’s name and the arguments it was called with. Client-Library receives the notification (through the connection to the Open Server) and calls the application’s notification callback routine.

The CS_ASYNC_NOTIFS property determines how the notification callback is triggered. See the description of this property under “Asynchronous notifications” on page 203.

The arguments with which the registered procedure was called are available inside the notification callback as a parameter result set. To retrieve these arguments, an application:

- Calls ct_con_props(CS_NOTIF_CMD) to retrieve a pointer to the command structure containing the parameter result set
- Calls ct_res_info(CS_NUMDATA), ct_describe, ct_bind, ct_fetch, and ct_get_data to describe, bind, and fetch the parameters

For more information on registered procedures and notifications, see the "Registered procedures" on page 238.

Defining a notification callback

A notification callback is defined as follows:

```
CS_RETCODE CS_PUBLIC notification_cb(conn, proc_name,
                                     namelen)

CS_CONNECTION     *conn;
CS_CHAR           *proc_name;
CS_INT            namelen;
```

where:

- *connection* is a pointer to the CS_CONNECTION structure receiving the notification. This CS_CONNECTION is the parent connection of the CS_COMMAND that sent the request to be notified.
- *proc_name* is a pointer to the name of the registered procedure that has been executed.
- *namelen* is the length, in bytes, of **proc_name*.

A notification callback must return CS_SUCCEED.

Table 2-5 on page 50 lists the Client-Library routines that a notification callback calls:

Table 2-5: Routines that a notification callback can call

Callable routine	Permitted use
ct_config	To retrieve information only.
ct_con_props	To retrieve information or to set the CS_USERDATA property only.
ct_cmd_props	To retrieve information only. The CS_USERDATA property can be set on command structures allocated with ct_cmd_alloc. **Note** The CS_USERDATA property cannot be set on the command structure obtained by the callback's ct_con_props(CS_NOTIF_CMD) call.
ct_cancel (CS_CANCEL_ATTN)	Any circumstances.
ct_bind, ct_describe, ct_fetch, ct_get_data, ct_res_info(CS_NUMDATA)	To retrieve the notification parameter values. The routines must be called with the command structure returned by the callback's ct_con_props(CS_NOTIF_CMD) call.

Retrieving notification parameters

The parameter values with which a registered procedure was invoked are available in the notification callback. To get the values, the application retrieves the command structure stored as the CS_NOTIF_CMD connection property. Using this command structure, the application retrieves the parameter values with the usual calls to ct_res_info(CS_NUMDATA), ct_describe, and ct_fetch.

See "Registered procedures" on page 238.

Security session callbacks

An Open Server gateway needs a security session callback only if all of the following statements are true:

- The Open Server is a gateway.
- The gateway allows clients to connect using network-based user authentication.
- The gateway wants to establish a direct security session between the gateway's client and the remote server.

If not all of the above conditions apply, Client-Library provides a default callback that is adequate.

See “Requesting login authentication services” on page 256.

Establishing a direct security session

A *security session* is a client/server connection where the client and the server have agreed to use an external security mechanism (such as DCE) and a set of security services (such as data encryption).

In a gateway application, a *direct security session* is established between a gateway’s client and a remote server. The gateway acts as an intermediary while the session is established, but afterwards, the gateway is not part of the security session. Direct security sessions are useful in the following circumstances:

- Full-passthrough gateways that support per-packet security services

  A full-passthrough gateway establishes a direct security session to support per-packet security services such as data integrity and data confidentiality while eliminating some of the associated overhead. For example, if the gateway supports data confidentiality without a direct security session, the contents of each TDS packet that passes through the gateway must be decrypted upon receipt and re-encrypted upon sending. If the gateway does not inspect the packet contents, this overhead is unnecessary. With a direct security session, no per-packet services are performed within the gateway.

- Gateways where delegated client credentials are not available

  A gateway’s clients may not delegate their security credentials to a gateway (using the CS_SEC_DELEGATION connection property), or a security mechanism may not support credential delegation. In these cases, the gateway must set up a direct security session to connect to the remote server using the same user name as the gateway’s client.

A security session callback allows the gateway to set up a direct security session. When the connection to the remote server is made, the callback routine acts as an intermediary for the handshaking required between the remote server and the gateway’s client. The handshaking process is outlined below:

1. When the gateway calls ct_connect, the remote server issues one or more security session messages.

2 For each security session message sent by the remote server, Client-Library invokes the callback, passing the security session information sent by the remote server as the callback's input parameters.

3 The callback forwards the information to the gateway's client by calling the Server-Library routine srv_negotiate(CS_SET, SRV_NEG_SECSESSION).

4 The callback then reads the client's response and returns it to Client-Library using the callback's output parameters.

5 Client-Library forwards the response to the remote server.

If the remote server sends another security session message, the process is repeated.

Defining a security session callback

A security session callback is defined as follows:

```
CS_RETCODE CS_PUBLIC
     secsession_cb (conn, numinputs, infmt, inbuf,
                    numoutputs, outfmt, outbuf, outlen)

CS_CONNECTION        *conn;
CS_INT               numinputs;
CS_DATAFMT           *infmt;
CS_BYTE              **inbuf;
CS_INT               *numoutputs;
CS_DATAFMT           *outfmt;
CS_BYTE              **outbuf;
CS_INT               *outlen;
```

where:

- *connection* is a pointer to the connection structure that controls the connection to the gateway's remote server.
- *numinputs* is the number of input parameters sent by the remote server with the security session message.
- *infmt* is the address of an array of CS_DATAFMT structures that describe each input parameter sent by the remote server.
- *inbuf* is the address of an array of CS_BYTE * pointers that point to buffers containing the data for each input parameter. The length of each buffer *inbuf*[*i*] is given as *infmt*[*i*] ->*.maxlength*

- *numoutputs* is the address of a CS_INT. The callback must return the number of items sent by the client in **numoutputs*. On input, **numoutputs* specifies the length of the *outfmt*, *outbuf*, and *outlen* arrays.
- *outfmt* is the address of an array of CS_DATAFMT structures. The callback must place a description of each item in the client's response into the corresponding CS_DATAFMT structure. The input value of **numoutput*s specifies the length of this array.
- *outbuf* is the address of an array of CS_BYTE * buffers. The callback must copy the data items from the client's response into the corresponding buffer. The input value of **numoutputs* specifies the length of this array, and for each buffer *i*, the input value of *outfmt*[*i*]–>*maxlength* specifies the allocated length of the buffer pointed at by *outbuf*[*i*].
- *outlen* is the address of an array of CS_INT. The callback places the number of bytes written to each buffer into *outlen*[*i*].

The callback forwards the security session message data and reads the client's response with Server-Library calls. See the reference page for srv_negotiate in the Open Server *Server-Library/C Reference Manual*.

A security session callback returns CS_SUCCEED or CS_FAIL. If the callback returns CS_FAIL, Client-Library aborts the connection attempt. Other return values are illegal: Client-Library responds by raising an error and aborting the connection attempt.

Server message callbacks

An application handles server errors and informational messages inline or through a server message callback routine.

When a connection is allocated, it picks up a default server message callback from its parent context. If the parent context has no server message callback installed, then the connection is created without a default server message callback.

After allocating a connection, an application:

- Installs a different server message callback for the connection.
- Calls ct_diag to initialize inline message handling for the connection. Note that ct_diag automatically deinstalls all message callbacks for the connection.

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

If callbacks are not implemented for a particular programming language and platform version of Client-Library, an application must handle server messages inline, using ct_diag.

If a connection is handling server messages through a server message callback, then the callback is called whenever a server message arrives.

Defining a server message callback

A server message callback is defined as follows:

```
CS_RETCODE CS_PUBLIC
 servermsg_cb(context, connection, message)

 CS_CONTEXT        *context;
 CS_CONNECTION     *connection;
 CS_SERVERMSG       *message;
```

where:

- *context* is a pointer to the CS_CONTEXT structure for which the message occurred.
- *connection* is a pointer to the CS_CONNECTION structure for which the message occurred.
- *message* is a pointer to a CS_SERVERMSG structure containing server message information. For information on this structure, see the "CS_SERVERMSG structure" on page 95.

  Note that *message* can have a new value each time the server message callback is called.
- A server message callback must return CS_SUCCEED.

Table 2-6: Routines that a server message callback can call

Callable routines	Permitted use
ct_config	To retrieve information only.
ct_con_props	To retrieve information or to set the CS_USERDATA property only.
ct_cmd_props	To retrieve information only. The CS_USERDATA property can be set on command structures allocated with ct_cmd_alloc. The CS_USERDATA property cannot be set on the command structure obtained by the callback's ct_con_props(CS_EED_CMD).
ct_cancel (CS_CANCEL_ATTN)	Any circumstances.
ct_bind, ct_describe, ct_fetch, ct_get_data, ct_res_info	The routines must be called with the command structure returned by the callback's ct_con_props(CS_EED_CMD) LAN. A server message callback calls these routines only while extended error data is available; that is, until ct_fetch returns CS_END_DATA. See "Extended error data" on page 129.

Warning! Do not call ct_poll from within any Client-Library callback function or from within any other function that can execute at the system-interrupt level. Calling ct_poll at the system-interrupt level can corrupt Open Client and Open Server internal resources and cause recursion in the application.

Server message callback example

Following is an example of a server message callback:

```
/*
 ** ex_servermsg_cb()
 **
 ** Type of function:
 **      Example program server message handler
 **
 ** Purpose:
 **      Installed as a callback into Open Client.
 **
 ** Returns:
 **      CS_SUCCEED
 **
```

```
** Side Effects:
**      None
*/
CS_RETCODE CS_PUBLIC
ex_servermsg_cb(context, connection, srvmsg)
CS_CONTEXT        *connection;
CS_CONNECTION     *cmd;
CS_SERVERMSG      *srvmsg;
{
     fprintf(EX_ERROR_OUT, "\nServer message:\n");
     fprintf(EX_ERROR_OUT, "Message number: %ld, \
          Severity %ld, ", srvmsg->msgnumber,
          srvmsg->severity);
     fprintf(EX_ERROR_OUT, "State %ld, Line %ld",
          srvmsg->state, srvmsg->line);

    if (srvmsg->svrnlen > 0)
     {
          fprintf(EX_ERROR_OUT, "\nServer '%s'",
               srvmsg->svrname);
     }

    if (srvmsg->proclen > 0)
     {
          fprintf(EX_ERROR_OUT, " Procedure '%s'",
               srvmsg->proc);
     }

    fprintf(EX_ERROR_OUT, "\nMessage String: %s",
          srvmsg->text);

    return CS_SUCCEED;
}
```

Handling specific messages

In some applications, the programmer may want to code special handling for certain message numbers.

For example, if a message is known to be informational and not an error message, you may not want the application to display the message to the end user. The example below shows a fragment from a server message callback that does not display messages 5701, 5703, or 5704. Adaptive Server always sends a 5701 message when a connection is opened and may also send the other two. Adaptive Server also sends a 5701 message after every successful use database command. Some end users may not want to see such messages. If the code shown below is placed at the top of the server message callback, these message numbers are ignored:

```
/*
 ** Ignore these Server messages:
 **  5701 (changed database),
 **  5703 (changed language),
 **  or 5704 (changed client character set)
 */
 if (srvmsg->msgnumber == 5701
     || srvmsg->msgnumber == 5703
     || srvmsg->msgnumber == 5704)
 {
   return CS_SUCCEED;
 }
```

This code is specific to Adaptive Server. These message numbers may mean something else entirely when connected to another type of server, such as an Open Server gateway or a custom Open Server application.

Signal callbacks

A signal callback is called whenever a process receives a signal on a UNIX platform.

On UNIX platforms, Client-Library uses signal-driven I/O to interact with the network. On these platforms, if an application handles signals, it must install the signal handler through Client-Library, even if the signals relate to non-Client-Library work. To install a signal handler, call ct_callback instead of using a system call. A system call to install a signal handler overwrites Client-Library's signal handler. If this occurs, Client-Library behavior is undefined.

When Client-Library is used in an Open Server gateway, signal handlers should be installed using Server-Library routines.

When Client-Library receives the Client-Library signal handler:

- Performs any internal Client-Library processing that is required

- Calls the appropriate user-defined signal callback, if any

Defining a signal callback

A signal callback must be defined according to operating system specifications.

An application that defines and installs a signal callback must include the appropriate operating system header file (*sys/signal.h* on most UNIX platforms).

Installing a signal callback

A signal callback is installed only at the context level. Signal callbacks are identified by adding the signal number on to the defined constant CS_SIGNAL_CB.

The following routine demonstrates how to install a signal callback:

```
/*
 ** INSTALLSIGNALCB
 **
 ** This routine installs a signal callback for the
 ** specified signal
 **
 ** Parameters:
 **      cp   Context handle
 **      signo   Signal number
 **      signalhandler   Signal handler to install
 **
 ** Returns:
 **      CS_SUCCEED   Signal handler was installed
 **                   successfully
 **      CS_FAIL      An error was detected while
 **                   installing the signal handler
 */
 CS_RETCODE  installsignalcb(cp, signo, signalhandler)
 CS_CONTEXT  *cp;
 CS_INT  signo;
 CS_VOID *signalhandler;
 {
      CS_INT       adjustedsigno;
      CS_RETCODE   ret;

     /*
      ** Add the signal number to the CS_SIGNAL_CB
```

```
        ** define to indicate the signal number that this
        ** handler is being installed for.
        */
        adjustedsigno = CS_SIGNAL_CB + signo;

    ret = ct_callback(cp, (CS_CONNECTION *)NULL,
          CS_SET, adjustedsigno, signalhandler);

    return(ret);
}
```

SSL validation callbacks

The Secure Socket Layer (SSL) validation callback intercepts SSL handshakes, overriding SSL validation checks. SSL validation callbacks are only required when a Client-Library application wants to override SSL validation checks.

You may, for example, attempt a SSL connection using ct_con_props (CS_SET, CS_SERVERADDR), with the server address set to *hostname port ssl*.

If the *server_name* parameter passed to ct_connect does not match the common name in the server's certificate, SSL validation fails. Use the SSL validation callback to override this check.

Defining an SSL validation callback

An SSL validation callback is defined as follows:

```
CS_RETCODE CS_PUBLIC
validate_srvname_cb(CS_VOID *userdata, CS_SSLCERT *certptr,
        CS_INT certcount, CS_INT valid)
```

where:

- *userdata* refers to the CS_USERDATA of the connection structure
- *certptr* is a pointer to an array of CS_SSLCERT structures
- *certcount* indicates the number of entries in the array
- *valid* is the value determined by the SSL validation check. *valid* can be any of the following values:

Value of valid	Indicates
CS_SSL_VALID_CERT	Valid certificate

Value of valid	Indicates
CS_SSL_INVALID_ BADCHAIN	Certificate chain is invalid
CS_SSL_INVALID_ EXPCERT	A certificate in the chain has expired
CS_SSL_INVALID_ INCOMPLETE	Certificate chain is not terminated with self-signed root certificate
CS_SSL_INVALID_ UNKNOWN	SSL validation check failed because of unknown reasons
CS_SSL_INVALID_ UNTRUSTED	Certificate chain does not include a trusted certificate
CS_SSL_INVALID_ MISSINGNAME	Common name missing in the certificate
CS_SSL_INVALID_ MISMATCHNAME	Common name does not match the server name

SSL validation callback example

Following is an example of a SSL validation callback:

```
CS_RETCODE CS_PUBLIC
validate_srvname_cb(CS_VOID *userdata, CS_SSLCERT *certptr,
    CS_INT certcount, CS_INT valid)
{
    if (valid == CS_SSL_INVALID_MISMATCHNAME)
    {
          return CS_SSL_VALID_CERT;
    }
    else
    {
          return valid;
    }
}
```

Capabilities

Capabilities describe features that a client/server connection supports. In particular, capabilities describe the types of requests that an application sends on a specific connection and the types of server responses that a server returns on a specific connection.

Wide tables and larger page size

Open Client and Open Server 12.5 and later allow client applications to send and receive wide data and data for larger numbers of columns that are supported in Adaptive Server; that is, columns in excess of 255 bytes, and more than 255 columns per table.

Client-Library applications compiled with versions earlier than 12.5 must be recompiled with a higher version to enable the byte limits.

Page size

Adaptive Server Enterprise 12.5 and later supports logical page sizes of 2K, 4K, 8K, and 16K. Open Client and Open Server uses the Bulk-Library (blklib) routines to populate these pages. Until the release of 12.5, blklib only supported the Adaptive Server page size of 2K.

Table 2-7 lists bulk library constants and their values.

Table 2-7: Page size values

blk_pagesize	blk_maxdatarow	blk_maxcolsize	blk_maxcolno	blk_boundary
2K	1962	1960	1962	1960
4K	4010	4008	4010	4008
8K	8106	8104	8106	8104
16K	16298	16296	16298	16298

Increased page size limits allow for increased number of columns, depending upon the type of table. The limits are:

- 1024 for fixed-length columns in both all-pages locking (APL) and data-only locking (DOL) tables
- 254 for variable-length columns in an APL table
- 1024 for variable-length columns in an DOL table

No changes have been made to the existing blklib APIs, nor have any new APIs been added to accommodate the larger page size support in Adaptive Server.

Compatibility

Support for wide data and a larger number of columns is automatically enabled if:

- The client is set to CS_VERSION_150,

- It is linked with Open Client Server 15.0 library, and
- The Adaptive Server to which it is connected has the capabilities to handle wide tables. To determine the version of Adaptive Server:

```
1> select @@version
2> go
```

If Open Client and Open Server 15.0 blklib is linked to a version 15.0 bcp application that communicates with a pre-12.5 Adaptive Server, the bcp utility assumes that Adaptive Server has the 2K page size.

If the blklib is linked to a bcp application that was built with a version of the utility earlier than 12.5, it cannot support the copy of large pages.

Wide tables

Adaptive Server Enterprise 12.5 and later supports tables with more than 255 columns and column sizes in excess of 255 bytes or 255-byte binary data. To accommodate the expanded table limits in Adaptive Server, Open Client and Open Server 15.0 can send and receive wider data and column information for tables with more columns.

Capability

To support wide tables, the client sends a login packet to the server along with a capability packet. Possible ct_capability parameters include:

- CS_WIDETABLE – a request capability that a client sends to the server indicating the client has the capability to receive larger data table formats.
- CS_NOWIDETABLE – a response capability that a client sends to the server to have the server disable wide table support for this particular connection.

If the version of the application is set to CS_VERSION_150, Client-Library always sends CS_WIDETABLE capability to the server; the application does not have control of the request capability. However, the application can set CS_NOWIDETABLE response capability before the connection is established to specifically request the server not to enable wide table capabilities.

The syntax of ct_capability is:

```
CS_RETCODE ct_capability (connection, action, type,
         capability, value)
CS_CONNECTION   *connection;
CS_INT          action;
CS_INT          type;
```

```
CS_INT          capability;
CS_VOID         *value;
```

where the values of *type* are CS_WIDETABLE or CS_NOWIDETABLE.

If you do not want to enable wide table support, you can send the server a CS_NOWIDETABLE command to disable this feature.

```
...
CS_BOOL     boolv = CS_TRUE
...
retcode = ct_capability (*conn_ptr, CS_SET,
            CS_CAP_RESPONSE, CS_NOWIDETABLES, &boolv);
...
```

ct_dynamic() with CS_CURSOR_DECLARE supports the flags CS_PREPARE, CS_EXECUTE, and CS_EXEC_IMMEDIATE to prepare and execute dynamic SQL statements that reference the 1024-column limit of Adaptive Server 12.5.

ct_param() can be used to pass as many as 1024 arguments to a dynamic SQL statement.

Changes in application program

If the column data you are retrieving is in excess of CS_MAX_CHAR (256 characters or 256 binary data), you must edit the CS_DATAFMT structure field datafmt.maxlength definition to the maximum length, in bytes, of the data that you are retrieving. Otherwise, you get a truncation error.

If you expect wider columns in the client program, change the column array size in the application program.

For example, if the application expects a column that is 300 bytes wide, then the column should mention CS_CHAR col1[300] at an appropriate place. Assign an appropriate length-of-character datatype, to the maxlength parameter of the CS_DATAFMT structure for RPC applications if the column is more than 255 bytes. The following is recommended for the CS_DATAFMT parameter:

```
datafmt.datatype = CS_LONGCHAR_TYPE
datafmt.maxlength = sizeof(col1)
```

The following example is a small ctlib program using the pubs2 database.

1 Alter the *authors* table and add a column “comment” declare as a *varchar(500)*:

```
1>alter table authors add comment varchar(500) null
```

```
2>go
```

2. Update the new column within the table:

```
1>update authors set comment = replicate (substring(state,1,1), 500)
2>go
/* This SQL command will update the comment column with a replicate of
500 times the first letter of the state for each row. */
```

3. Modify the *example.h* file to set the “new limits” capabilities:

   ```
   #define EX_CTLIB_VERSION CS_VERSION_150
   ```

4. Update the *exutils.h* file and reset the MAX_CHAR_BUF to 16384 (16K).
5. Recompile and link ctlib using 15.0 headers and libraries.
6. Execute and test on a Adaptive Server version 15.0 *X* k page size server.

   If you set CS_VERSION_150, you see the following (only displays the last 2 rows):

```
Heather McBadden 95688 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Anne Ringer 84152 UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
```

7. Update the *example.h* file and reset ctlib to CS_VERSION_150. Recompile and link using *OCS-15_0* headers and libraries.

   Note If you execute the same program without setting CS_VERSION_150 first, you retrieve only the first 255 bytes of the comment column and cannot retrieve wide columns, even if you are using version 15.0 of Adaptive Server and OCS 15.0 libraries.

   Open Client message:

```
Message number: LAYER = (1) ORIGIN = (4) SEVERITY = (1) NUMBER = (132)
```

```
Message String: ct_fetch(): user api layer: internal common library
error: The bind of result set item 4 resulted in truncation.
Error on row 21.
Heather McBadden 95688 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Open Client message:

```
Message number: LAYER = (1) ORIGIN = (4) SEVERITY = (1) NUMBER = (132)
Message String: ct_fetch(): user api layer: internal common library
error: The bind of result set item 4 resulted in truncation.
Error on row 22.
Anne Ringer 84152 UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
```

Wide-table compatibility

Wide-table support is activated automatically if:

- The client is set to CS_VERSION_150,
- It is linked with Open Client Server 15.0 library, and
- The Adaptive Server to which it is connected has the capabilities to handle wide tables.

If the Client-Library application's version string is not set to CS_VERSION_150, and it is linked to an Open Client and Open Server 15.0, the application does not support the extended limits and there is no behavioral change.

If the Open Client and Open Server version 15.0 connects to a pre-12.5 Adaptive Server, the server returns a capability bit of 0, indicating that it does not support wide tables; the connection is still made but there are no behavioral changes.

If a pre-12.5 version of Open Client and Open Server connects to an Adaptive Server 15.0, the new limits are not enabled. However, if the Adaptive Server determines that it must send a wide-table format to an older client, the data is truncated and sent.

Note Adaptive Server version 11.0 and later returns a mask length of 0 for any mask length in excess of 7 bytes. If the connection request receives a capability mask of 0, you see this error message:

```
ct_connect(): protocol specific layer: external error:
“This server does not accept new larger cpability mask,
the original cap mask will be used.”
```

and the extended limits are not enabled.

unichar datatype

Open Client and Open Server 12.5 and later unichar supports 2-byte characters, supporting multilingual client applications, and reducing the overhead associated with character-set conversions.

Designed the same as the Open Client and Open Server CS_CHAR datatype, CS_UNICHAR is a shared, C-programming datatype that can be used anywhere the CS_CHAR datatype is used. The CS_UNICHAR datatype stores character data in Unicode UCS Transformational Format 16-bit (UTF-16), which is 2-byte characters.

The Open Client and Open Server CS_UNICHAR datatype corresponds to the Adaptive Server 15.0 UNICHAR fixed-width and UNIVARCHAR variable-width datatypes, which store 2-byte characters in the Adaptive Server database.

As a standalone, Open Client 15.0 applications can use this new functionality to convert other datatypes to and from CS_UNICHAR at the client site, even if the server does not have the capability to process 2-byte characters.

Datatypes and capabilities

To send and receive 2-byte characters, the client specifies its preferred byte order during the login phase of the connection. Any necessary byte-swapping is performed on the server site.

The Open Client ct_capability() parameters are as follows:

- CS_DATA_UCHAR – is a request sent to the server to determine whether the server supports 2-byte characters.
- CS_DATA_NOUCHAR – is a parameter sent from the client to tell the server not to support unichar for this specific connection.

To access 2-byte character data, Open Client and Open Server implements the following:

- CS_UNICHAR – a datatype.

- CS_UNICHAR_TYPE – a datatype constant to identify the data's datatype.

Setting the CS_DATAFMT parameter's datatype to CS_UNICHAR_TYPE allows you to use existing API calls, such as ct_bind, ct_describe, ct_param, and so on.

CS_UNICHAR uses the format bitmask field of CS_DATAFMT to describe the destination format.

For example, in the Client-Library sample program, rpc.c, the BuildRpcCommand() function contains the section of code that describes the datatype:

```
...
strcpy (datafmt.name, "@charparam");
datafmt.namelen =CS_NULLTERM;
datafmt.datatype = CS_CHAR_TYPE;
datafmt.maxlength = CS_MAX_CHAR;
datafmt.status = CS_RETURN;
datafmt.locale = NULL;
...
```

In this example from the uni_rpc.c sample program, the character type is defined as `datafmt.datatype = CS_CHAR_TYPE`. Use an ASCII text editor to edit the datafmt.datatype field:

```
...
strcpy (datafmt.name, "@charparam");
datafmt.namelen =CS_NULLTERM;
datafmt.datatype = CS_UNICHAR_TYPE;
datafmt.maxlength = CS_MAX_CHAR;
datafmt.status = CS_RETURN;
datafmt.locale = NULL;
...
```

Samples are found in *%SYBASE%\%SYBASE_OCS%\sample* for Windows, and *$SYBASE/$SYBASE_OCS/sample* for UNIX.

Since CS_UNICHAR is a UTF-16 encoded Unicode character datatype that is stored in 2 bytes, the maximum length of CS_UNICHAR string parameter sent to the server is restricted to one-half the length of CS_CHAR, which is stored in 1-byte format.

Table 2-8 lists the CS_DATAFMT bitmask fields.

Table 2-8: CS_DATAFMT structure

Bitmask field	Description
CS_FMT_NULLTERM	The data is 2-byte Unicode null-terminated (0x0000).
CS_FMT_PADBLANK	The data is padded with 2-byte Unicode blanks to the full length of the destination variable (0x0020).
CS_FMT_PADNULL	The data is padded with 2-byte Unicode nulls to the full length of the destination variable (0x0000).
CS_FMT_UNUSED	No format information is provided.

isql and *bcp* utilities

Both the isql and the bcp utilities automatically support unichar data if the server supports 2-byte character data. bcp now supports 4K, 8K and 16K page sizes.

If the client's default character set is UTF-8, isql displays 2-byte character data, and bcp saves 2-byte character data in the UTF-8 format. Otherwise, the data is displayed or saved, respectively, in 2-byte Unicode data in binary format.

Use `isql -Jutf8` to set the client character set for isql. Use `bcp -Jutf8` to set the client character set for the bcp utility.

Limitations

The sever to which the Open Client and Open Server is connecting must support 2-byte Unicode datatypes, and use UTF-8 as the default character set. If the server does not support 2-byte Unicode datatypes, the server returns an error message: `"Type not found. Unichar/univarchar is not supported."`

CS_UNICHAR does not support the conversion from UTF-8 to UTF-16 byte format for CS_BOUNDARY and CS_SENSITIVITY. All other datatype formats are convertible.

CS_UNICHAR does not provide C programming operations on UTF-16 encoded Unicode data such as Unicode character strings. For full support for Unicode character strings, you must use the Sybase product, Unilib. See the *Unilib Reference Manual* for information.

unitext datatype

CS_UNITEXT is an Open Client and Open Server C Programming datatype that corresponds directly to the server UNITEXT datatype. CS_UNITEXT also exhibits identical syntax and semantics to CS_TEXT. The difference is that CS_UNITEXT encodes character data in the Unicode UTF-16 format.

Datatypes and capabilities

To send and receive 2-byte characters, the client specifies its preferred byte order during the login phase of the connection. Any necessary byte-swapping is performed on the server side.

The Open Client ct_capability() parameters are as follows:

- CS_DATA_UNITEXT – is a request sent to the server to determine whether the server supports 2-byte Unicode datatypes.
- CS_DATA_NOUNITEXT – is a parameter sent from the client to tell the server not to send unitext for this specific connection.

To access 2-byte character data, Open Client and Open Server implements the following:

- CS_UNITEXT – a datatype.
- CS_UNITEXT_TYPE – a datatype constant to identify the data's datatype.

Setting the CS_DATAFMT parameter's datatype to CS_UNITEXT_TYPE allows you to use existing API calls, such as ct_bind, ct_describe, ct_param, ct_setparam, cs_convert and so on.

Since CS_UNITEXT is encoded as a UTF-16 Unicode datatype and stored in the 2-byte format, it can be used anywhere CS_TEXT is used. The maximum length of the CS_UNITEXT string parameter is half of the maximum length of CS_TEXT.

Like CS_TEXT, CS_UNITEXT uses CS_DATAFMT to describe the destination format. The symbols and meanings of the format field values are as follows:

Table 2-9: CS_DATAFMT structure

Bitmask field	Description
CS_FMT_NULLTERM	The data is 2-byte Unicode null-terminated (0x0000).
CS_FMT_PADBLANK	The data is padded with 2-byte Unicode blanks to the full length of the destination variable (0x0020).
CS_FMT_PADNULL	The data is padded with 2-byte Unicode nulls to the full length of the destination variable (0x0000).
CS_FMT_UNUSED	No format information is provided.

isql and *bcp* utilities

In an Open Client application, UNITEXT is always activated, with no configuration parameter required. UNITEXT is part of the Open Client and Open Server libraries and the utilities (isql & bcp) that are shipped with them. isql displays and bcp saves the server's UNITEXT in binary format.

Limitations

The server to which the Open Client and Open Server is connecting must support 2-byte Unicode datatypes.

If the server does not support 2-byte Unicode datatypes, the server returns an error message.

However, the client can convert other datatypes to or from CS_UNITEXT.

CS_UNITEXT does not provide C programming operations on UTF-16 encoded Unicode data such as Unicode character strings. For full support for Unicode character strings, you must use the Sybase product, Unilib. See the *Unilib Reference Manual*. The reference manual is part of the Sybase Unicode Developers Kit 2.0.

xml datatype

CS_XML is a variable-width Open Client and Open Server C Programming datatype. CS_XML corresponds directly to CS_TEXT and CS_IMAGE datatypes. CS_XML can be used anywhere CS_TEXT and CS_IMAGE are used to represent XML documents and contents.

Datatypes and capabilities

The Open Client ct_capability() parameters:

- CS_DATA_XML – is a request sent to the server to determine whether the server supports XML.
- CS_DATA_NOXML – is a parameter sent from the client to tell the server not to support xml for this specific connection.

To access XML datatypes, Open Client and Open Server implements:

- CS_XML – a datatype.
- CS_XML_TYPE – a datatype constant to identify the data's datatype.

Setting the CS_DATAFMT parameter's datatype to CS_XML_TYPE allows you to use existing API calls, such as ct_bind, ct_describe, ct_param, ct_setparam, cs_convert and so on.

isql and *bcp* utilities

In an Open Client application, XML is always activated, with no configuration parameter required. XML is part of the Open Client and Open Server libraries and the utilities (isql & bcp) that are shipped with them. isql displays and bcp saves the server's XML in binary format.

Limitations

XML data can only be transmitted between client and server if the server supports XML. If there is no support, the server returns an error message. cs_capability is used to test if the server supports XML. A client can convert other possible datatypes to or from the CS_XML datatype.

Note the following syntax rules of XML:

- Closing XML tags cannot be omitted.
- XML tags are case sensitive.
- XML elements must be properly nested.
- XML documents must have a root element.
- XML attribute values must always be quoted.

With XML, white space is preserved. Also, with XML, CR/LF is converted to LF.

The Open Client and Open Server does not check or validate CS_XML documents or contents.

Capabilities and the connection's TDS level

Sybase clients and servers communicate using the Tabular Data Stream™ (TDS) protocol. Different TDS versions support different features. For example, 4.0 TDS is the earliest version that supports Remote Procedure Calls (RPCs).

The TDS version level is determined when a connection is established. When an application calls ct_connect to connect to a server, Client-Library presents the server with a preferred TDS level. If the server cannot support this TDS level, it negotiates with Client-Library to find a TDS level that is acceptable.

For example, when the 15.0 Client-Library application connects to Adaptive Server version 11.0, the default TDS level that the application requests is 5.0 TDS. Because Adaptive Server 11.0 does not support 5.0 TDS, it replies with a request to use version 4.x of TDS instead. Because 4.x TDS is acceptable to 15.0 Client-Library, the connection is established with a TDS version level of CS_TDS_4x.

Note jConnect does not negotiate TDS version; if the server doesn't support TDS 5.0 jConnect will terminate the connection.

Capabilities describe which client requests and which server responses are sent over a connection. By default, capabilities are based on the TDS version level, but a client application can be coded to further limit response capabilities and a server can be coded to further limit request capabilities.

When a Client-Library calls ct_capability:

1. Before opening a connection, it sets up the connection structure to tell a server not to send a particular type of response on a connection.
2. After opening a connection, it determines whether the connection supports a particular type of request or response.

For information on how an Open Server application sets or retrieves capabilities, see the Open Server *Server-Library/C Reference Manual*.

There are two types of capabilities:

- CS_CAP_REQUEST capabilities, or *request capabilities*, describe the types of client requests that can be sent on a server connection.
- CS_CAP_RESPONSE capabilities, or *response capabilities*, describe the types of server responses that a connection does not wish to receive.

For a list of capabilities, see the reference page for ct_capability.

Setting and retrieving capabilities

Before calling ct_connect, an application:

- Retrieves request or response capabilities to determine what request and response features are normally supported at the connection's current TDS version level. A connection's TDS level defaults to the version level that the application requested in its call to ct_init. An application can change a connection's TDS level by calling ct_con_props with *property* as CS_TDS_VERSION (see "TDS version" on page 228).
- Sets response capabilities to indicate that a connection does not wish to receive particular types of responses. For example, an application sets a connection's TDS_RES_NOEED capability to CS_TRUE to indicate that the connection does not wish to receive extended error data.

After a connection is open, an application:

- Retrieves request capabilities to find out what types of requests the connection will support
- Retrieves response capabilities to find out whether the server has agreed to withhold the previously indicated response types from the connection

Setting and retrieving multiple capabilities

Gateway applications often need to set or retrieve all capabilities of a type category with a single call to ct_capability. To do this, an application calls ct_capability with:

- *type* as the type category of interest
- *capability* as CS_ALL_CAPS
- *value* as a CS_CAP_TYPE structure

Client-Library provides the following macros to enable an application to set, clear, and test bits in a CS_CAP_TYPE structure:

- CS_CLR_CAPMASK(*mask*, *capability*) – modifies the CS_CAP_TYPE structure *mask* by clearing the bits specified in *capability*.
- CS_SET_CAPMASK(*mask*, *capability*) – modifies the CS_CAP_TYPE structure *mask* by setting the bits specified in *capability*.
- CS_TST_CAPMASK(*mask*, *capability*) – determines whether the CS_CAP_TYPE *mask* includes the bits specified in *capability*.

Client-Library and SQL Structures

This section provides an overview of Client-Library structures and the SQL structures.

Exposed and hidden structures

Client-Library structures fall into two categories: a hidden structure is a structure whose internals are not documented, and an exposed structure is a structure whose internals are documented.

Exposed structures

Exposed structures provide a way for Client-Library to exchange information with an application. Typically, applications set fields in an exposed structure before passing the structure as a parameter to a Client-Library routine, and retrieve the values of fields in an exposed structure after calling a Client-Library routine.

Exposed structures include:

- CS_BROWSEDESC, the browse descriptor structure
- CS_CLIENTMSG, the Client-Library message structure
- CS_DATAFMT, the data format structure
- CS_IODESC, the I/O descriptor structure
- CS_SERVERMSG, the server message structure
- SQLCA, the SQL Communications Area structure
- SQLCODE, the SQL Code structure

- SQLSTATE, the SQL State structure

These exposed structures are documented in the following sections.

Hidden structures

Client-Library uses hidden structures to manage a variety of internal tasks.

A Client-Library application cannot directly access hidden structure internals. Instead, the application must call Client-Library routines to allocate, manipulate, and deallocate hidden structures.

Hidden structures include:

- CS_BLKDESC, a control structure used by Client-Library's and Server-Library's bulk copy routines.
- CS_CAP_TYPE, which is used to store capability information.
- CS_COMMAND, which is used to send commands and process results.
- CS_CONNECTION, which defines an individual client/server connection.
- CS_CONTEXT, which defines a Client-Library programming context.
- CS_LOCALE, which is used to store localization information.
- CS_LOGINFO, the server login information structure. This structure, which is associated with a CS_CONNECTION, contains server login information such as user name and password.

Table 2-10 lists the routines and macros that allocate, manipulate, and deallocate hidden structures:.

Table 2-10: Routines that manipulate hidden structures

Structure	Routines	For more information
CS_BLKDESC	blk_alloc, blk_drop	Open Client and Open Server *Common Libraries Reference Manual*.
CS_CAP_TYPE	CS_CLR_CAPMASK, CS_SET_CAPMASK, CS_TST_CAPMASK	"Setting and retrieving multiple capabilities" on page 73.
CS_COMMAND	ct_cmd_alloc, ct_cmd_props, ct_cmd_drop	Reference pages for these routines.
CS_CONNECTION	ct_con_alloc, ct_con_props, ct_con_drop	Reference pages for these routines.
CS_CONTEXT	cs_ctx_alloc, ct_config, cs_config, cs_ctx_drop	Reference pages for these routines. CS-Library routines are documented in the Open Client and Open Server *Common Libraries Reference Manual*.
CS_LOCALE	cs_loc_alloc, cs_locale, cs_loc_drop	"International Support" on page 146. Open Client and Open Server *Common Libraries Reference Manual*.
CS_LOGINFO	ct_getloginfo, ct_setloginfo	Reference pages for these routines.

CS_BROWSEDESC structure

ct_br_column uses a CS_BROWSEDESC structure to return information about a column returned as the result of a browse-mode select. This information is useful when an application needs to construct a language command to update browse-mode tables.

A CS_BROWSEDESC structure is defined as follows:

```
/*
 ** CS_BROWSEDESC
 ** The Client-Library browse column description
 ** structure.
 */
 typedef struct _cs_browsedesc
```

```
{
    CS_INT      status;
    CS_BOOL     isbrowse;
    CS_CHAR     origname[CS_MAX_CHAR];
    CS_INT      orignlen;
    CS_INT      tablenum;
    CS_CHAR     tablename[CS_OBJ_NAME];
    CS_INT      tabnlen;
} CS_BROWSEDESC;
```

where:

- *status* is a bitmask of the following symbols, on which a bitwise OR operation is performed:
  - CS_EXPRESSION indicates the column is the result of an expression, for example, “sum*2” in the query “select sum*2 from areas”.
  - CS_HIDDEN indicates that the column is a *hidden column* that has been exposed. See “Hidden keys” on page 214.
  - CS_KEY indicates that the column is a key column. See the ct_keydata reference page.
  - CS_RENAMED indicates that the column’s heading is not the original name of the column. Columns will have a different heading from the column name in the data base if they are the result of a query of the form:

    ```
    select Author = au_lname from authors
    ```

- *isbrowse* indicates whether or not the column can be browse-mode updated.

  A column may be updated if it is not the result of an expression and if it belongs to a browsable table. A table is browsable if it has a unique index and a timestamp column.

  isbrowse is set to CS_TRUE if the column can be updated and CS_FALSE if it cannot.

- *origname* is the original name of the column in the database. *origname* is a null-terminated string.

  Any updates to a column must refer to it by its original name, not the heading that may have been given the column in a select statement.

- *orignlen* is the length, in bytes, of *origname*.

- *tablenum* is the number of the table to which the column belongs. The first table in a select statement's from list is table number 1, the second is number 2, and so forth.
- *tablename* is the name of the table to which the column belongs. *tablename* is a null-terminated string.
- *tabnlen* is the length, in bytes, of *tablename*.

CS_CLIENTMSG structure

A CS_CLIENTMSG structure contains information about a Client-Library error or informational message.

Client-Library uses a CS_CLIENTMSG structure in two ways:

- For connections using the callback method to handle messages, a CS_CLIENTMSG is the third parameter that Client-Library passes to an application's client message callback routine.
- For connections handling messages inline, ct_diag returns information in a CS_CLIENTMSG.

For information on how to handle Client-Library error handling and server message handling, see "Error handling" on page 123.

A CS_CLIENTMSG structure is defined as follows:

```
/*
 ** CS_CLIENTMSG
 ** The Client-Library client message structure.
 */

 typedef struct _cs_clientmsg
 {
     CS_INT      severity;
     CS_MSGNUM   msgnumber;
     CS_CHAR     msgstring[CS_MAX_MSG];
     CS_INT      msgstringlen;

     /*
     ** If the error involved the operating
     ** system, the following fields contain
     ** operating-system-specific information:
     */
     CS_INT      osnumber;
     CS_CHAR     osstring[CS_MAX_MSG];
```

```
        CS_INT      osstringlen;

        /*
        ** Other information:
        */
        CS_INT      status;
        CS_BYTE     sqlstate[CS_SQLSTATE_SIZE];
        CS_INT      sqlstatelen;

    } CS_CLIENTMSG;
```

where:

- *severity* is a symbolic value representing the severity of the message. Table 2-11 shows the legal values for severity:

Table 2-11: CS_CLIENTMSG severity field values

Severity	Explanation
CS_SV_INFORM	No error has occurred. The message is informational.
CS_SV_CONFIG_FAIL	A Sybase configuration error has been detected. Configuration errors include missing localization files, a missing *interfaces* file, and an unknown server name in the *interfaces* file.
CS_SV_RETRY_FAIL	An operation has failed, but can be retried. An example of this type of operation is a network read that times out.
CS_SV_API_FAIL	A Client-Library routine generated an error. This error is typically caused by a bad parameter or calling sequence. The server connection is probably usable.
CS_SV_RESOURCE_FAIL	A resource error has occurred. This error is typically caused by a malloc failure or lack of file descriptors. The server connection is probably not usable.
CS_SV_COMM_FAIL	An unrecoverable error in the server. The server connection is not usable.
CS_SV_INTERNAL_FAIL	An internal Client-Library error has occurred.
CS_SV_FATAL	A serious error has occurred. All server connections are unusable.

- *msgnumber* is the Client-Library message number. See “Client-Library message numbers” on page 81.
- *msgstring* is the null-terminated Client-Library message string.

If an application is not sequencing messages, *msgstring* is guaranteed to be null-terminated, even if it has been truncated.

If an application is sequencing messages, *msgstring* is null-terminated only if it is the last chunk of a sequenced message.

See “Sequencing long messages” on page 127.

- *msgstringlen* is the length, in bytes, of *msgstring*. This is always the actual length, never the symbolic value CS_NULLTERM.
- *osnumber* is the operating system error number, if any. Client-Library sets *osnumber* to 0 if no operating system error has occurred.
- *osstring* is the null-terminated operating system error string, if any.
- *osstringlen* is the length of *osstring*. This is always the actual length, never the symbolic value CS_NULLTERM.
- *status* is a bitmask that indicates various types of information, such as whether or not this is the first, a middle, or the last chunk of an error message. The values that can be present in *status* include:

Table 2-12: CS_CLIENTMSG status field values

Symbolic value	Meaning
CS_FIRST_CHUNK	The message text contained in *msgstring* is the first chunk of the message. If CS_FIRST_CHUNK and CS_LAST_CHUNK are both on, then *msgstring* contains the entire message. If neither CS_FIRST_CHUNK nor CS_LAST_CHUNK is on, then *msgstring* contains a middle chunk of the message. See “Sequencing long messages” on page 127.
CS_LAST_CHUNK	The message text contained in *msgstring* is the last chunk of the message. If CS_FIRST_CHUNK and CS_LAST_CHUNK are both on, then *msgstring* contains the entire message. If neither CS_FIRST_CHUNK nor CS_LAST_CHUNK is on, then *msgstring* contains a middle chunk of the message. See “Sequencing long messages” on page 127.

- *sqlstate* is a byte string describing the error.

Not all client messages have SQL state values associated with them. If no SQL state value is associated with a message, *sqlstate* has the value "ZZZZZ".

- *sqlstatelen* is the length, in bytes, of the *sqlstate* string.

Client-Library message numbers

Client-Library message numbers are represented by a CS_INT value that encodes four byte-size components.

Decoding a message number

Client-Library provides the following macros for decoding a message number so that each component is displayed separately:

- CS_LAYER – unpacks the layer number that identifies the Client-Library layer that generated the message.
- CS_ORIGIN – unpacks the message's origin, which indicates whether the error occurred internal or external to Client-Library.
- CS_SEVERITY – unpacks the severity of the message. See "Client-Library message severities" on page 82 for a list of severity codes and their meanings.
- CS_NUMBER – unpacks the layer-specific message number that (together with severity, layer, and origin) identifies the message.

These macros are defined in the header file *cstypes.h* (which is included in *ctpublic.h*).

A typical application uses these macros to split a message number into four parts, which it then displays separately. For examples that demonstrates the use of these macros, see "Client message callback example" on page 33 and "Handling timeout errors" on page 230.

Client-Library and CS-Library use the message number components layer, origin, and number as keys for building a localized message string from text retrieved from the library's locales file. The localized message strings are then passed to the application as the *msgstring* field of the CS_CLIENTMSG structure.

Note See the Open Client and Open Server *Configuration Guide* for Microsoft Windows or the Open Client and Open Server *Configuration Guide* for UNIX, to view the Sybase localization file structure on your platform.

The error message text is composed from the components as follows:

```
routine: layer: origin: description
```

where:

- *routine* is the name of the library routine where the error occurred.
- *layer* is a layer description retrieved from either the [cslayer] section of *cslib.loc* (for CS-Library errors) or the [ctlayer] section of *ctlib.loc* (for Client-Library errors).
- *origin* is a phrase retrieved from either the [csorigin] section of *cslib.loc* (for CS-Library errors) or the [ctorigin] section of *ctlib.loc* (for Client-Library errors).
- *description* is an error description retrieved from the appropriate layer-specific section of the file.

The following is a U.S. English error string as it might be printed by a typical client message callback routine:

```
Client Library error(16843066):
 severity(1) number(58) origin(1) layer(1)
 ct_bind(): user api layer: external error: The format
field of the CS_DATAFMT structure must be CS_FMT_UNUSED
if the datatype field is int.
```

Client-Library message severities

Table 2-13 lists Client-Library message severities:

Table 2-13: Client-Library message severities

Severity	Explanation	User action
CS_SV_INFORM	No error has occurred. The message is informational.	No action is required.

Severity	Explanation	User action
CS_SV_CONFIG_FAIL	A Sybase configuration error has been detected. Configuration errors include missing localization files, a missing *interfaces* file, and an unknown server name in the *interfaces* file.	Raise an error so that the application's end user can correct the problem.
CS_SV_RETRY_FAIL	An operation has failed, but the operation may be retried. An example of this type of operation is a network read that times out.	The return value from an application's client message callback determines whether or not Client-Library retries the operation. If the client message callback returns CS_SUCCEED, Client-Library retries the operation. If the client message callback returns CS_FAIL, Client-Library does not retry the operation and marks the connection as dead. In this case, call ct_close(CS_FORCE_CLOSE) to close the connection and then reopen it by calling ct_connect.
CS_SV_API_FAIL	A Client-Library routine generated an error. This error is typically caused by a bad parameter or calling sequence. The server connection is probably usable.	Call ct_cancel(CS_CANCEL_ALL) to clean up the connection. If ct_cancel(CS_CANCEL_ALL) returns CS_SUCCEED, the server connection is unharmed. It is illegal to perform this type of cancel from within a client message callback routine.
CS_SV_RESOURCE_FAIL	A resource error has occurred. This error is typically caused by a malloc failure or lack of file descriptors. The server connection is probably not usable.	Call ct_close(CS_FORCE_CLOSE) to close the server connection and then reopen it, if desired, by calling ct_connect. It is illegal to make these calls from within a client message callback routine.
CS_SV_COMM_FAIL	An unrecoverable error in the server communication channel has occurred. The server connection is not usable.	Call ct_close(CS_FORCE_CLOSE) to close the server connection and then re-open it, if desired, by calling ct_connect. It is illegal to make these calls from within a client message callback routine.
CS_SV_INTERNAL_FAIL	An internal Client-Library error has occurred.	Call ct_exit(CS_FORCE_EXIT) to exit Client-Library, and then exit the application. It is illegal to call ct_exit from within a client message callback routine.

Severity	Explanation	User action
CS_SV_FATAL	A serious error has occurred. All server connections are unusable.	Call ct_exit(CS_FORCE_EXIT) to exit Client-Library, and then exit the application. It is illegal to call ct_exit from within a client message callback routine.

Handling specific Client-Library messages

Most Client-Library messages represent a coding error in the program, and the error description tells you the problem. These errors are best handled by either displaying the message or logging it to an application error file.

In other cases, the program may want to recognize the error and take specific action. For example:

- If a read from the server times out, then the program may decide to cancel the command that is being processed.
- For configuration errors, the program may want to recognize the specific problem and display an application-defined message that gives specific instructions to the application end user.

Errors are uniquely described by the four components of the error. A macro such as the ERROR_SNOL example below is useful for recognizing message numbers:

```
/*
 ** ERROR_SNOL(error_numb, severity, number, origin, layer)
 **
 **   Error comparison for Client-Library or CS-Library errors.
 **   Breaks down a message number and compares it to the given
 **   constants for severity, number, origin, and layer. Returns
 **   non-zero if the error number matches the 4 components.
 */
 #define ERROR_SNOL (e, s, n, o, l) \
   ( (CS_SEVERITY(e) == s) && (CS_NUMBER(e) == n) \
    && (CS_ORIGIN(e) == o) && (CS_LAYER(e) == l ) )
```

Table 2-14 lists the error codes for some Client-Library messages. These errors are either recoverable, or they represent a configuration problem either on the client machine or the remote server machine.

Table 2-14: Client-Library errors

Severity	Number	Origin	Layer	Cause
CS_SV_RETRY_FAIL	63	2	1	A read from the server timed out. See “Handling timeout errors” on page 230.

Severity	Number	Origin	Layer	Cause
CS_SV_CONFIG_FAIL	8	3	5	The *interfaces* file (or platform equivalent) was not found. See “Location of the interfaces file” on page 215.
CS_SV_CONFIG_FAIL	3	3	5	The server name is not found in the *interfaces* file or the connection’s directory source.
CS_SV_COMM_FAIL	4	3	4	The connection attempt failed because a login dialog could not be established with the remote server. This error occurs when the remote server is down.
CS_SV_COMM_FAIL	131	3	5	ct_init failed because Net-Library drivers could not be initialized. The client message callback is not called for this error—Client-Library prints a message to the *stderr* device. The most likely cause of this error is a misconfigured [DRIVER] section in the *libtcl.cfg* file. See the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Server *Configuration Guide* for UNIX, to view details on how Client-Library loads Net-Library drivers.
CS_SV_API_FAIL	132	4	1	The bind of a result item resulted in truncation while fetching the data. This error occurs when calling ct_fetch if a destination variable (bound with ct_bind) is too small for the data to be received. If column indicators are used, the application checks the indicator values to see which column(s) were truncated.

CS_DATAFMT structure

A CS_DATAFMT structure is used to describe data values and program variables. For example:

- ct_bind requires a CS_DATAFMT structure to describe a destination variable.
- ct_describe returns a CS_DATAFMT structure to describe a result data item.
- ct_param and ct_setparam both require a CS_DATAFMT to describe an input parameter.

- cs_convert requires CS_DATAFMT structures to describe source and destination data. cs_convert is documented in the Open Client and Open Server *Common Libraries Reference Manual*.

Most routines use only a subset of the fields in a CS_DATAFMT. For example, ct_bind does not use the *name*, *status*, and *usertype* fields, and ct_describe does not use the *format* field. For information on which fields in the CS_DATAFMT a routine uses, see the reference page for the routine.

```
typedef struct _cs_datafmt
 {
     CS_CHAR name[CS_MAX_CHAR];  /* Name of data */
     CS_INT  namelen;            /* Length of name */
     CS_INT  datatype;           /* Datatype */
     CS_INT  format;             /* Format symbols */
     CS_INT  maxlength;          /* Data max length */
     CS_INT  scale;              /* Scale of data */
     CS_INT  precision;          /* Data precision */
     CS_INT  status;             /* Status symbols */

   /*
     ** The following field indicates the number of
     ** rows to copy, per ct_fetch call, to a bound
     ** program variable. ct_describe sets this field
     ** to a default value of 1. ct_bind is the only
     ** routine that reads this field.
     */
     CS_INT  count;

   /*
     ** These fields are used to support Adaptive Server
     ** user-defined datatypes and international
     ** datatypes:
     */
    CS_INT      usertype;   /* Svr user-def'd type */
    CS_LOCALE   *locale;    /* Locale information */

} CS_DATAFMT;
```

where:

- *name* is the name of the data. *name* is often a column or parameter name.
- *namelen* is the length, in bytes, of *name*. Set *namelen* to CS_NULLTERM to indicate a null-terminated name. Set *namelen* to 0 to if *name* is NULL.

- *datatype* is a type constant representing the datatype of the data. This is either one of the Open Client datatypes or an Open Client user-defined datatype. For information about datatypes, see “Datatypes support” on page 294.

  Do not confuse the *datatype* field with the *usertype* field. *datatype* is always used to describe the Open Client datatype of the data. *usertype* is used only if the data has an Adaptive Server user-defined datatype in addition to an Open Client datatype.

  For example, the following Adaptive Server command creates the server user-defined type *birthday*:

  ```
  sp_addtype birthday, datetime
  ```

  and this command creates a table containing a column of the type:

  ```
  create table birthdays
   (
       name        varchar(30),
       happyday    birthday
   )
  ```

  If a Client-Library application executes a select against this table and calls ct_describe to get a description of the *birthday* column in the result set, the *datatype* and *usertype* fields in the CS_DATAFMT structure are set as follows:

  datatype is set to CS_DATETIME_TYPE.
  usertype is set to the Adaptive Server ID for the type *birthday*.

- *format* describes the destination format of character or binary data. *format* is a bitmask of the following symbols, combined with the OR operator:

Table 2-15: CS_DATAFMT format field values

Symbol	Meaning	Notes
CS_FMT_NULLTERM	The data should be null-terminated.	For character or text data
CS_FMT_PADBLANK	The data should be padded with blanks to the full length of the destination variable.	For character or text data
CS_FMT_PADNULL	The data should be padded with NULLs to the full length of the destination variable.	For character, text, binary or image data
CS_FMT_UNUSED	No format information is being provided.	For all data types

- *maxlength* represents various lengths, depending on which Open Client routine is using the CS_DATAFMT. Table 2-16 lists the meanings of *maxlength*:

Table 2-16: CS_DATAFMT maxlength field values

Open Client routine	Value of maxlength
ct_bind	The length of the bind variable.
ct_describe	The maximum possible length of the column or parameter being described.
ct_dyndesc	The maximum possible length of the column or parameter being described.
ct_dynsqlda	The maximum possible length of the column or parameter being described.
ct_param	The maximum desired length of return parameter data.
ct_setparam	The maximum desired length of return parameter data. If ct_setparam's *datalen* parameter is passed as NULL, *maxlength* specifies the length of all input values for the parameter.
cs_convert	The length of the source data and the length of the destination buffer space.

- *scale* is the maximum number of digits to the right of the decimal point in the data. *scale* is used only with decimal or numeric datatypes.

  Permitted values for *scale* are from 0 to 77. The default is 0. CS_MIN_SCALE, CS_MAX_SCALE, and CS_DEF_PREC define the minimum, maximum, and default scale values, respectively.

  To indicate that destination data should use the same scale as the source data, set *scale* to CS_SRC_VALUE.

scale must be less than or equal to *precision*.

- *precision* is the maximum number of decimal digits that can be represented in the data. *precision* is used only with decimal or numeric datatypes.

  Values for *precision* are from 1 to 77. The default is 18. CS_MIN_PREC, CS_MAX_PREC, and CS_DEF_PREC define the minimum, maximum, and default precision values, respectively.

  To indicate that destination data should use the same precision as the source data, set *precision* to CS_SRC_VALUE.

  precision must be greater than or equal to *scale*.

- *status* is a bitmask that indicates various types of information. Table 2-17 lists the values that can make up *status*:

Table 2-17: CS_DATAFMT status field values

Symbolic value	Meaning	Legal for
CS_CANBENULL	The column can contain NULL values.	ct_describe, ct_dyndesc, ct_dynsqlda
CS_HIDDEN	The column is a hidden column that has been exposed. See “Hidden keys” on page 214.	ct_describe, ct_dyndesc, ct_dynsqlda
CS_IDENTITY	The column is an identity column.	ct_describe, ct_dyndesc, ct_dynsqlda
CS_KEY	The column is a key column. See the reference page for ct_keydata.	ct_describe, ct_dyndesc, ct_dynsqlda
CS_UPDATABLE	The column is an updatable cursor column.	ct_describe, ct_dyndesc, ct_dynsqlda
CS_VERSION_KEY	The column is part of the version key for the row. Adaptive Server uses version keys for positioning cursors. See the reference page for ct_keydata.	ct_describe, ct_dyndesc, ct_dynsqlda
CS_TIMESTAMP	The column is a *timestamp* column. An application uses *timestamp* columns when performing browse-mode updates.	ct_describe
CS_UPDATECOL	The parameter is the name of a column in the update clause of a cursor declare command.	ct_param, ct_setparam, ct_dyndesc, ct_dynsqlda
CS_INPUTVALUE	The parameter is an input parameter value for a Client-Library command.	ct_param, ct_setparam, ct_dyndesc, ct_dynsqlda
CS_RETURN	The parameter is a return parameter to an RPC command.	ct_param, ct_setparam, ct_dyndesc, ct_dynsqlda

- *count* is the number of rows to copy to program variables per ct_fetch call. *count* is used only by ct_bind.

- *usertype* is the server user-defined datatype, if any, of data returned by the server. *usertype* is used only for server user-defined types, not for Client-Library user-defined types. For a discussion of Client-Library user-defined types, see "Datatypes support" on page 294.
- *locale* is a pointer to a CS_LOCALE structure containing localization information. Set *locale* to NULL if localization information is not required.

  Before using a CS_DATAFMT structure, make sure that *locale* is valid either by setting it to NULL or to the address of a valid CS_LOCALE structure.

CS_IODESC structure

A CS_IODESC structure, or *I/O descriptor structure*, describes *text* or *image* data.

An application calls ct_data_info to retrieve a CS_IODESC structure after retrieving a *text* or *image* value that it plans to update at a later time. After it has a valid CS_IODESC, a typical application changes only the values of the *locale*, *total_txtlen*, and *log_on_update* fields before using the CS_IODESC to update the *text* or *image* value.

An application calls ct_data_info to define a CS_IODESC structure after calling ct_command to initiate a send-data operation to update a *text* or *image* value.

A CS_IODESC is defined as follows:

```
typedef struct _cs_iodesc
 {
     CS_INT        iotype;                  /* CS_IODATA.          */
     CS_INT        datatype;                /* Text or image.      */
     CS_LOCALE     *locale;                 /* Locale information. */
     CS_INT        usertype;                /* User-defined type.  */
     CS_INT        total_txtlen;            /* Total data length.  */
     CS_INT        offset;                  /* Reserved.           */
     CS_BOOL       log_on_update            /* Log the insert?     */
     CS_CHAR       name[CS_OBJ_NAME];       /* Name of data object.*/
     CS_INT        namelen                  /* Length of name.     */
     CS_BYTE       timestamp[CS_TS_SIZE];   /* Adaptive Server id. */
     CS_INT        timestamplen;            /* Length of timestamp.*/
     CS_BYTE       textptr[CS_TP_SIZE];     /* Adaptive Server ptr.*/
     CS_INT        textptrlen;              /* Length of textptr.  */
```

```
    CS_INT        delete_length;           /* Number of bytes to  */
                                           /* delete/overwrite for*/
                                           /* partial updates.    */
} CS_IODESC;
```

where:

- *iotype* indicates the type of I/O to perform. For text and image operations, *iotype* can have the value CS_IODATA or CS_IOPARTIAL. The CS_IOPARTIAL setting specifies that a partial update is to be performed on the text or image column.
- *datatype* is the datatype of the data object. The values for *datatype* are CS_TEXT_TYPE and CS_IMAGE_TYPE.
- *locale* is a pointer to a CS_LOCALE structure containing localization information for the text or image value. Set *locale* to NULL if localization information is not required.

  Before using a CS_IODESC structure, make sure that *locale* is valid by setting it either to NULL or to the address of a valid CS_LOCALE structure.
- *usertype* is the Adaptive Server user-defined datatype of the data object, if any. On send-data operations, *usertype* is ignored. On get-data operations, Client-Library sets *usertype* in addition to (not instead of) *datatype*.
- *total_txtlen* is the total length, in bytes, of the text or image value.

  Unicode and partial updates
  If your client application performs partial updates on 2-byte Unicode datatypes, the application must make sure that it sends an even number of bytes to avoid a character split. You can use the *buflen* parameter of ct_send_data() and the *total_txtlen* field of CS_IODESC to specify the length, in bytes, of the Unicode data. For partial updates to Unitext data, the *offset* and *delete_length* values must be specified as a character count while *total_txtlen* must be specified in bytes.

- *offset* indicates the first byte in the column that is affected by a partial update.
- *log_on_update* describes whether the server should log the update to this text or image value.
- *name* is the name of the text or image column. *name* is a null-terminated string of the form *table.column*.

- *namelen* is the length, in bytes, of *name* (not including the null terminator). When filling in a CS_IODESC, an application sets *namelen* to CS_NULLTERM to indicate a null-terminated name.
- *timestamp* is the text timestamp of the column. A text timestamp marks the time of a text or image column's last modification.
- *timestamplen* is the length, in bytes, of *timestamp*.
- *textptr* is the text pointer for the column. A text pointer is an internal server pointer that points to the data for a text or image column. *textptr* identifies the target column in a send-data operation.
- *textptrlen* is the length, in bytes, of *textptr*.
- *delete_length* indicates the number of bytes that are to be overwritten or deleted from column a text or image column for which a partial update has been specified.

CS_OID structure

CS_OID structures store object identifiers.

An **Object Identifier (OID)** is an encoded character string that provides a machine- and network-independent method of uniquely identifying objects in a distributed environment. An OID functions as a symbolic **global name** that means the same to all applications in a distributed environment.

Sybase uses OIDs to represent the following:

- Directory objects.
- Attribute types within a directory object.
- Security mechanisms that assure secure client/server connections. A security mechanism may have a different **local name** on the client machine than on the server machine. To avoid confusion, an OID is used as a global name that identifies the security mechanism for both the client and the server. See "Choosing a network security mechanism" on page 253 for a description of how a security mechanism is associated with a connection.

Encoding of object identifiers

OIDs are encoded as a sequence of decimal integers separated by dots. OIDs are defined according to ISO standards and organized in a hierarchy that avoids duplication among different vendors. In the hierarchy, unique prefixes are assigned to different vendors. For example, the prefix “1.3.1.4.1.897” belongs to Sybase, and all Sybase OIDs have this prefix.

Definition of the CS_OID structure

A CS_OID structure is required to exchange an OID between Client-Library routines and application code.

The CS_OID structure is used with calls to ct_ds_lookup or ct_ds_objinfo.

The CS_OID structure is defined as follows:

```
typedef struct _cs_oid
 {
         CS_INT  oid_length;
         CS_CHAR oid_buffer[CS_MAX_DS_STRING];
 } CS_OID;
```

where:

- *oid_length* is the length of the OID string. If the OID string is null-terminated, the length does not include the null terminator.
- *oid_buffer* is an array of bytes that holds the OID string. This string is not always null-terminated.

Using predefined OID strings

The Client-Library header files define OID strings for applications to use in initializing or comparing OIDs. Predefined OID strings are used for the following purposes:

- Identifying directory object. Sybase directory object is Server, and the OID is CS_OID_OBJSERVER.

  For more information, see “Server directory object” on page 276
- Identifying the attributes of a given directory object. See the definition of the directory object for the predefined OID strings that identify each attribute.

CS_SERVERMSG structure

A CS_SERVERMSG structure contains information about a server error or informational message.

Client-Library uses a CS_SERVERMSG structure in two ways:

- For connections using the callback method to handle messages, a CS_SERVERMSG is the third parameter that Client-Library passes to the connection's server message callback.
- For connections handling messages inline, ct_diag returns information in a CS_SERVERMSG.

For information on error and message handling, see "Error handling" on page 123.

A CS_SERVERMSG structure is defined as follows:

```
/*
 ** CS_SERVERMSG
 ** The Client-Library server message structure.
 */

typedef struct _cs_servermsg
 {
     CS_MSGNUM   msgnumber;
     CS_INT      state;
     CS_INT      severity;
     CS_CHAR     text[CS_MAX_MSG];
     CS_INT      textlen;
     CS_CHAR     svrname[CS_MAX_CHAR];
     CS_INT      svrnlen;

    /*
     ** If the error involved a stored procedure,
     ** the following fields contain information
     ** about the procedure:
     */
     CS_CHAR     proc[CS_MAX_CHAR];
     CS_INT      proclen;
     CS_INT      line;

    /*
     ** Other information.
     */
     CS_INT      status;
     CS_BYTE     sqlstate[CS_SQLSTATE_SIZE];
     CS_INT      sqlstatelen;
```

```
} CS_SERVERMSG;
```

where:

- *msgnumber* is the server message number. For a list of Adaptive Server messages, execute the Transact-SQL command:

  ```
  select * from sysmessages
  ```

- *state* is the server error state.
- *severity* is the severity of the message. For a list of Adaptive Server message severities, execute the Transact-SQL command:

  ```
  select distinct severity from sysmessages
  ```

- text is the text of the server message.

  If an application is not sequencing messages, text is guaranteed to be null-terminated, even if it has been truncated.

  If an application is sequencing messages, text is null-terminated only if it is the last chunk of a sequenced message.

  For more information on sequenced messages, see “Sequencing long messages” on page 127.

- *textlen* is the length, in bytes, of text. This is always the actual length, never the symbolic value CS_NULLTERM.
- *svrname* is the name of the server that generated the message. This is the name of the server as it appears in the *interfaces* file. *svrname* is a null-terminated string.
- *svrnlen* is the length, in bytes, of *svrname*.
- *proc* is the name of the stored procedure that caused the message, if any. *proc* is a null-terminated string.
- *proclen* is the length, in bytes, of *proc*.
- *line* is the line number, if any, of the line that caused the message. *line* may be a line number in a stored procedure or a line number in a command batch.
- *status* is a bitmask used to indicate various types of information, such as whether or not extended error data is included with the message. Table 2-18 lists the values that can be present in *status*:

Table 2-18: CS_SERVERMSG status field values

Symbolic value	Meaning
CS_HASEED	Extended error data is included with the message. For more information on extended error data, see "Extended error data" on page 129.
CS_FIRST_CHUNK	The message text contained in text is the first chunk of the message. If CS_FIRST_CHUNK and CS_LAST_CHUNK are both on, then text contains the entire message. If neither CS_FIRST_CHUNK nor CS_LAST_CHUNK is on, then text contains a middle chunk of the message. See "Sequencing long messages" on page 127.
CS_LAST_CHUNK	The message text contained in text is the last chunk of the message. If CS_FIRST_CHUNK and CS_LAST_CHUNK are both on, then text contains the entire message. If neither CS_FIRST_CHUNK nor CS_LAST_CHUNK is on, then text contains a middle chunk of the message. See "Sequencing long messages" on page 127.

- *sqlstate* is a byte string describing the error.

  Not all server messages have SQL state values associated with them. If no SQL state value is associated with a message, *sqlstate* has the value "ZZZZZ".

- *sqlstatelen* is the length, in bytes, of the *sqlstate* string.

SQLCA structure

A SQLCA structure is used in conjunction with ct_diag to retrieve Client-Library and server error and informational messages.

A SQLCA structure is defined as follows:

```
/*
 ** SQLCA
 ** The SQL Communications Area structure.
 */

 typedef struct _sqlca
```

```
{
    char    sqlcaid[8];
    long    sqlcabc;
    long    sqlcode;

    struct
    {
        long    sqlerrml;
        char    sqlerrmc[256];
    } sqlerrm;

    char    sqlerrp[8];
    long    sqlerrd[6];
    char    sqlwarn[8];
    char    sqlext[8];

} SQLCA;
```

where:

- *sqlcaid* is “SQLCA”.
- *sqlcabc* is ignored.
- *sqlcode* is the server or Client-Library message number. For information about how Client-Library maps message numbers to *sqlcode*, see “SQLCODE structure” on page 99.
- *sqlerrml* is the length of the actual message text (not the length of the text placed in *sqlerrmc*).
- *sqlerrmc* is the null-terminated text of the message. If the message is too long for the array, Client-Library truncates it before appending the null terminator.
- *sqlerrp* is the null-terminated name of the stored procedure, if any, being executed at the time of the error. If the name is too long for the array, Client-Library truncates it before appending the null terminator.
- *sqlerrd[2]* is the number of rows affected by the current command. This field is set only if the current message is a “number of rows affected” message. Otherwise, *sqlerrd[2]* has a value of CS_NO_COUNT.
- *sqlwarn* is an array of warnings:
    - If *sqlwarn[0]* is blank, then all other *sqlwarn* variables are blank. If *sqlwarn[0]* is not blank, then at least one other *sqlwarn* variable is set to “W”.

- If *sqlwarn[1]* is “W”, then Client-Library truncated at least one column’s value when copying it into a host variable.
- If *sqlwarn[2]* is “W”, then at least one null value was eliminated from the argument set of a function.
- If *sqlwarn[3]* is “W”, then some but not all items in a result set have been bound. This field is set only if the CS_ANSI_BINDS property is set to CS_TRUE (see “ANSI-style binds” on page 202).
- If *sqlwarn[4]* is “W”, then a dynamic SQL update or delete statement did not include a where clause.
- If *sqlwarn[5]* is “W”, then a server conversion or truncation error has occurred.

- *sqlext* is ignored.

SQLCODE structure

A SQLCODE structure is used in conjunction with ct_diag to retrieve Client-Library and server error and informational message codes.

An application must declare a SQLCODE structure as a long integer.

Client-Library always sets SQLCODE and the *sqlcode* field of the SQLCA structure identically.

Mapping server messages to SQLCODE

A server message number is mapped to a SQLCODE of 0 if it has a severity of 0. Other server messages may also be mapped to a SQLCODE of 0.

Server message numbers are inverted before being placed into SQLCODE. This ensures that SQLCODE is negative if an error has occurred.

For a list of server messages, execute the Transact-SQL statement:

```
select * from sysmessages
```

Mapping Client-Library messages to SQLCODE

The Client-Library message “No rows affected” is mapped to a SQLCODE of 100. Client-Library messages with CS_SV_INFORM severities are mapped to a SQLCODE of 0. Other Client-Library messages may also be mapped to a SQLCODE of 0.

Client-Library message numbers are inverted before being placed into SQLCODE. This ensures that SQLCODE is negative if an error has occurred.

See “Client-Library message numbers” on page 81.

SQLSTATE structure

A SQLSTATE structure is used in conjunction with ct_diag to retrieve SQL state information, if any, associated with a Client-Library or server message.

An application must declare a SQLSTATE structure as an array of 6 characters.

The *sqlstate* fields of the CS_CLIENTMSG and CS_SERVERMSG structures are treated identically to SQLSTATE, except that they are defined as 8 bytes. The last 2 bytes are ignored.

Commands

In the client/server model, a server accepts commands from multiple clients and responds by returning data and other information to the clients. Open Client applications use Client-Library routines to communicate commands to servers.

Table 2-19 summarizes the Client-Library command types:

Table 2-19: Command types

Command type	Initiated by	Summary
Language	ct_command	Defines the text of a query that the server will parse, interpret, and execute.
RPC, Package	ct_command	Specifies the name of a server procedure (Adaptive Server stored procedure or Open Server registered procedure) to be executed by the server. The procedure must already exist on the server. Package commands are only supported by mainframe Open Server servers. They are otherwise identical to RPC commands.
Cursor	ct_cursor	Initiates one of several commands to manage a Client-Library cursor.
Dynamic SQL	ct_dynamic	Initiates a command to execute a literal SQL statement (with restrictions on statement content) or to manage a prepared dynamic SQL statement.
Message	ct_command	Initiates a message command and specifies the message-command ID number.
Send-Data	ct_command	Initiates a command to upload a large text/image column value to the server.

Sending commands

All commands are defined and sent in three steps:

1. Initiate the command. This identifies the command type and what it executes.
2. Define parameter values, if necessary.
3. Send the command. ct_send writes the command symbols and data to the network. The server then reads the command, interprets it, and executes it.

Initiating a command

An application sends several types of commands to a server:

- An application calls ct_command to initiate a language, message, package, remote procedure call (RPC), or send-data command.

- An application calls ct_cursor to initiate a cursor command.
- An application calls ct_dynamic to initiate a dynamic SQL command.

Defining parameters for a command

The following types of commands take parameters:

- A language command, when the command text contains variables
- An RPC command, when the stored procedure takes parameters
- A cursor declare command, when the body of the cursor contains host variables or when some of the cursor's columns are for update
- A cursor open command, when the body of the cursor contains host language parameters
- A cursor update command, if the text of the update statement contains variables
- A message command
- A dynamic SQL execute command

An application calls ct_param or ct_setparam once for each parameter a command requires. These routines perform the same function, except that ct_param copies a parameter value, while ct_setparam copies the address of a variable that contains the value. If ct_setparam is used, Client-Library reads the parameter value when the command is sent. This allows the application to change the parameter values that were specified with ct_setparam before resending the command.

Sending a command

After a command has been initiated and its parameters have been defined, an application calls ct_send to send the command to the server. The server then interprets the command, executes it, and returns the results to the client application.

Resending a command

For most command types, Client-Library allows an application to resend the command after the results of previous execution have been processed. Enhancements to ct_send, ct_cursor, and ct_bind, and the addition of ct_setparam routine in version 11.1 allow batch-processing applications to resend commands and reuse binds when repeatedly executing the same server command. This feature can eliminate redundant calls to ct_bind, ct_command, ct_cursor, and ct_param.

The application resends commands as follows:

- If necessary, the application changes values in parameter source variables.

  If the command requires parameters, the application should define parameter source variables with ct_setparam instead of passing values with ct_param. Input parameter values passed with ct_param can not be changed when a command is resent.

- The application calls ct_send to resend the command after the results of the previous command execution have been processed and before a new command is initiated on the command structure.

An application can resend all types of commands except:

- Send-data commands initiated by ct_command(CS_SEND_DATA_CMD)
- Send-bulk commands initiated by ct_command(CS_SEND_BULK_CMD)
- ct_cursor cursor commands other than cursor-update or cursor-delete
- ct_dynamic commands other than execute-immediate commands or a command to execute a prepared statement

Deciding which type of command to use

See Chapter 5, “Choosing Command Types,” in the Open Client *Client-Library/C Programmers Guide* for guidance on which command type is right for your application.

Connection migration

Open Client supports connection migration for connections made to a server that understands the connection migration protocol and can move client connections to another server after login has completed. Connection migration is enabled by the CS_PROP_MIGRATABLE property. The default for this property is CS_TRUE, and the property is valid for both ct_config and ct_con_props.

Note DB-Library does not support connection migration.

Directory services

A **directory** stores system information as **directory entries** and associates a logical name with each entry. Each directory entry contains information about a network entity such as a user, a server, or a printer. A directory organizes this information and removes the requirement to modify applications when the location of a network entity changes.

A **directory service** (sometimes called a naming service) manages creation, modification, and retrieval of directory entries.

Directory service providers and drivers

Directory driver configuration determines the default directory source for a Client-Application. The directory is provided by either:

- The Sybase **interfaces file**, which is simply an operating system file on the local host machine. If not explicitly set, the *interfaces* file is the default.
- Network-based directory service software, such as Distributed Computing Environment Cell Directory Services (DCE/CDS), Lightweight Directory Access Protocol (LDAP), or the Windows Registry.

See “Directory service provider” on page 120 for more information.

For information on configuring directory drivers, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

Network-based directory services

A distributed directory service allows Client-Library and Server-Library to use a network-based directory rather than the Sybase *interfaces* file as the source for server address information. Using a network directory service can simplify the administration of an environment that contains many client machines.

A network-based directory use requires a Sybase directory driver that interacts with the network directory service. For Client-Library applications, the CS_DS_PROVIDER connection property specifies the directory source to be used by calls to ct_connect and ct_ds_lookup.

Client-Library routines ct_ds_lookup, ct_ds_objinfo, and ct_ds_dropobj allow directory browsing. Using these routines, an application can search for available servers in the directory or *interfaces* file.

LDAP

Lightweight Directory Access Protocol (LDAP) is used to access directory listings. A directory listing, or service, provides a directory of names, profile information, and machine addresses for every user and resource on the network. It can be used to manage user accounts and network permissions.

LDAP servers are typically hierarchical in design and provide fast lookups of resources. LDAP can be used as a replacement to the traditional Sybase *interfaces* file (*sql.ini* on Windows) to store and retrieve information about Sybase servers.

Any type of LDAP service, whether it is an actual server or a gateway to other LDAP services, is called an LDAP server. An LDAP driver calls LDAP client libraries to establish connections to an LDAP server. The LDAP driver and client libraries define the communication protocol, such as whether encryption is enabled, and the contents of messages exchanged between clients and servers. Messages are operators, such as client requests for read, write, and queries, and server responses, including data-format information.

When the LDAP driver connects to the LDAP server, the server establishes the connection based on two authentication methods—anonymous access, and user name and password authentication.

- Anonymous access – does not require any authentication information; therefore, you do not have to set any properties. Anonymous access is typically used for read-only privileges.

- User name and password – can be specified in the *libtcl.cfg* file *(libtcl64.cfg* file for 64-bit platforms) as an extension to the LDAP URL or set with property calls to Client-Library. The user name and password that are passed to the LDAP server through Ct-Lib are separate and distinct from the user name and password used to log in to Adaptive Server. Sybase strongly recommends that you use user name and password authentication.

OpenLDAP

OpenLDAP is the open-source version of LDAP.

From Open Client and Open Server version 15.0 ESD#2 onwards, OpenLDAP libraries are used on the following platforms:

- HP-UX 11.11 32-bit and 64-bit
- HP Itanium 32-bit and 64-bit
- IBM RS/6000 (AIX version 5.2 or later) 32-bit and 64-bit
- Linux x86 32-bit
- Linux AMD64 (Opteron)/EM64T
- Linux on POWER 32-bit and 64-bit
- Sun Solaris 8 (SPARC 32-bit and 64-bit)
- Sun Solaris 10 x64 32-bit and 64-bit
- Microsoft Windows

More information on configuration can be found in the Open Client and Open Server *Configuration Guide* for your platform.

SSL/TLS

In Open Client and Open Server 15.0 ESD #7 and later, you can establish an encrypted (SSL) connection between the application and the LDAP server. This encrypted connection is set up in one of two ways:

- LDAPS – connects to the secure port, typically port 636, of the LDAP directory server. This method, also known as LDAP over SSL, is non-standard, but widely supported.

- StartTLS – upgrades an existing standard connection, typically using port 389, to a secure connection using transport layer security. This method is only possible if the connection uses LDAPv3.

During the SSL/TLS negotiation, the LDAP server sends its certificate to prove its identity. The client verifies that this certificate was signed by a trusted Certificate Authority (CA). A list of trusted CAs is maintained in the trusted roots file *trusted.txt*. This file is located in *$SYBASE/config* or in an alternate file location stored in the CS_PROP_SSL_CA property.

Once the LDAP server is successfully authenticated, the client and the LDAP server continue their SSL handshake to establish the encrypted connection. Once initiated, there is no difference between the connections established with LDAPS and StartTLS, except that LDAPS requires a separate listener for the LDAP server.

For more information on certificates and the trusted roots file, see the Open Client and Open Server *Configuration Guide* for Unix.

LDAP Directory Server lookup time limit

Connections or directory lookups to the LDAP Directory Server may be unable to complete due to a hanging or otherwise unavailable LDAP Server.

By setting CS_DS_TIMELIMIT, you can specify time limits on failed connections or lookups to the LDAP Directory Server. If CS_DS_TIMELIMIT is not set, the login timeout value is used as the default time limit for LDAP Directory Server lookups. For information on CS_DS_TIMELIMIT, refer to Table 2-29.

Retry and delay options

The retry option specifies the number of times to retry a search connection to the LDAP Directory Server after the initial attempt fails or times out. The delay option is the number of seconds to wait between a failed and new retry. Both options are set in *libtcl.cfg* and apply to the designated LDAP Directory Server only, for example:

```
[DIRECTORY]
myldap=libsybdldap.so retry=3 delay=5
ldap://nlnognix/dc=sybase,dc=com????bind…
```

By default, both options are 0.

LDAP for Microsoft Active Directory

Sybase supports LDAP for Microsoft Active Directory, which is a directory service used to centralize networks and store information about network resources. See “Schema and name syntax for Microsoft Active Directory” on page 112 for information on importing directory schema and creating a container for the Sybase server entries.

Use of the directory by applications

Client-Library applications require a directory to connect to a server. When an application calls ct_connect, Client-Library looks up the server name in the directory and reads the necessary information to establish a connection to the server.

Applications can also search the directory for Sybase-defined entries by calling ct_ds_lookup. For example, an application calls ct_ds_lookup to search for an available server or to check the status of a particular server.

Directory organization

Since directory services are provided by different vendors, each directory may have a different way of organizing and storing entries.

A directory has either a flat structure or a hierarchical structure. A hierarchical structure allows related entries to be combined into distinct logical groupings that descend from a parent entry. In a flat structure, all entries in the directory are in one logical grouping.

A hierarchical structure can be thought of as an inverted tree. The “root” entry is at the top and is the “ancestor” of all other entries. “Parent” entries represent logical groupings of related entries. If an entry is the parent of no other entry, it is called a “leaf” entry.

In any directory structure, each entry has a **fully qualified name** that uniquely identifies the entry. Entries also have a **common name** that is unique only among entries that have the same parent node.

In a hierarchical directory structure, names must contain navigation information. Only at the root node are the common name and the fully qualified name the same. For any other entry, the fully qualified name is constructed by combining the entry’s common name with the fully qualified name of the entry’s parent node.

In a flat directory structure, there is no root node, and every entry's fully qualified name is the same as its common name.

The Sybase *interfaces* file is an example of a flat directory. Most network-based directory services provide a hierarchical directory.

Directory entry name formats

Entry names must be recognized by the directory provider software. Each provider requires a different name syntax. Table 2-20 illustrates some examples of fully qualified names.

Note These examples are for discussion purposes only. For name syntax information on directories other than the *interfaces* file, please see the documentation for the network directory provider software used on your system. All example entry names in this book are fictional.

Table 2-20: Fully qualified name syntax examples

Directory service provider	Fully qualified name example
OSF DCE Cell Directory Services (DCE CDS)	*/.:/dataservers/sybase/license_data* (cell-relative) */.../sales.fictional.com/dataservers/sybase/license_data* (global)
Windows Registry	*SOFTWARE\SYBASE\SERVER\the_server*
LDAP directory services	UNIX: *ldap=libsybdldap.so ldap://host:port/ditbase??scope?? bindname=username?password* Windows: *ldap=libsybdldap.dll ldap://host:port/ditbase??scope?? bindname=username?password* **Note** The LDAP URL must be on a single line.
Sybase *interfaces* file	*my_server*

Table 2-20 does not provide an exhaustive list of supported directory service providers, and the providers listed may not be supported on all platforms. See the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX to view information about the directory providers supported on your platform.

Name syntax for DCE CDS

Sybase applications access a DCE directory by using DCE Cell Directory Services (CDS) as the directory provider.

In DCE CDS, the directory name space is divided into cells. Each cell acts as an administrative domain for managing network resources and their users. In CDS, a fully qualified name can be cell-relative or globally qualified:

- Cell-relative qualified names begin with the special token "/.:". The common name of descendant nodes are listed in order (from left to right) and each common name is separated from its parent with a forward slash (/). The following example illustrates a cell-relative qualified name:

  ```
  /.:/eng/license_data
  ```

- Globally qualified names begin with the special token "/...". Following "/..." is a Domain Name Services (DNS) name for the DCE cell. The rest of the name consists of the descendant nodes from the cell root, in left-to-right order and separated by slashes. The following example illustrates a globally qualified name. In this example, "sales.fictional.com" identifies the cell that contains the entry:

  ```
  /.../sales.fictional.com/dataservers/license_data
  ```

Name syntax for Windows Registry

The Windows Registry comprises a hierarchical structure in which nodes are called "keys." The common name of descendant nodes are listed in order (from left to right) and each common name is separated from its parent by a backslash ("\"). Registry storage is local to each machine, but entries may be read from another machine's Registry by including the machine name in the fully qualified name.

The following example shows a fully qualified name for an entry in the local Registry:

```
SOFTWARE\SYBASE\SERVER\the_server
```

The example below names an entry in the machine `queenbee`'s Registry:

```
queenbee:SOFTWARE\SYBASE\SERVER\the_server
```

All entry names for Sybase directory entries are located relative to the key "\HKEY_LOCAL_MACHINE\".

Registry entries are not case sensitive.

Name syntax for LDAP directory services

The *libtcl.cfg* and the *libtcl64.cfg* files (collectively *libtcl*.cfg* files) determine whether the *interfaces* file or LDAP directory services should be used. If LDAP is specified in the *libtcl*.cfg* file, the *interfaces* file is ignored unless the application specifically overrides the *libtcl*.cfg* file by passing the `-I` parameter while connecting to a server.

You use the *libtcl*.cfg* to specify the LDAP server name, port number, DIT base, user name, and password to authenticate the connection to an LDAP server. In the *libtcl*.cfg* file, LDAP directory services are specified with a URL in the DIRECTORY section.

For example:

```
[DIRECTORY]
ldap=libsybdldap.so
ldap://huey:11389/dc=sybase,dc=com??
     one????bindname=cn=Manager,dc=sybase,dc=com secret
```

Table 2-21 defines the keywords for the *ldapurl* variables.

Table 2-21: ldapurl variables

Keyword	Description	Default	CS_* property
host (required)	The host name or IP address of the machine running the LDAP server	None	
port	The port number on which the LDAP server is listening	389	
ditbase (required)	The default DIT base	None	CS_DS_DITBASE
username	Distinguished name (DN) of the user to authenticate	NULL (anonymous authentication)	CS_DS_PRINCIPAL
password	Password of the user to be authenticated	NULL (anonymous authentication)	CS_DS_PASSWORD

You can find a complete list of Sybase's LDAP directory schema in:

- UNIX – *$SYBASE/$SYBASE_OCS/config*
- Windows – *%SYBASE%\%SYBASE_OCS%\ini*
  In the same directory, there is also a file called *sybase-schema.conf*, which contains the same schema but in a Netscape-specific syntax.

To create an encrypted connection with the LDAP server using LDAPS, use:

```
ldap=libsybdldap.so
ldaps://huey:636/dc=sybase,dc=com????
bindname=cn=Manager,dc=Sybase,dc=com?secret
```

If no port number is specified using ldaps://, port 636 will be used by default.

To create an encrypted connection by using and upgrading the standard LDAP listener, use:

```
ldap=libsybdldap.so starttls
ldap://huey:389/dc=sybase,dc=com????
bindname=cn=Manager,dc=Sybase,dc=com?secret
```

If no port number is specified using ldap://, port 389 will be used by default.

Schema and name syntax for Microsoft Active Directory

The directory schema for use with Microsoft Active Directory is *sybase.ldf*. You can import *sybase.ldf* into the Active Directory (AD) or into an Active Directory Application Mode (ADAM) instance using the ldifde.exe command provided in the ADAM installation. To import the directory schema, execute the ldifde.exe command from the ADAM installation using this syntax:

```
ldifde -i -u -f sybase.ldf -s server:port -b username
domain password -j . -c "cn=Configuration,dc=X"
#configurationNamingContext
```

After the schema has been successfully imported into the Active Directory, you can create a container for the Sybase server entries and set appropriate read and write permissions for the container and its child objects.

For example, a container with a relative distinguished name (RDN) “CN=SybaseServers” is created in the root of the Active Directory for domain “mycompany.com” to store and retrieve Sybase server entries. The root distinguished name (rootDN) for this container is reflected in the *libtcl.cfg* file as:

```
ldap=libsybdldap.dll ldap://localhost:389/
cn=SybaseServers,dc=mycompany,dc=com??...
```

If you create a dedicated user account name “Manager” with password “secret” in the Active Directory to add and modify Sybase server entries, the complete entry in the *libtcl.cfg* file is:

- For Windows:

```
ldap=libsybdldap.dll
ldap://localhost:389/cn=SybaseServers,dc=mycompany,
dc=com????bindname=cn=Manager,cn=Users,dc=mycompay,
```

```
dc=com?secret
```

- For UNIX:

```
ldap=libsybdldap.so
ldap://myADhost:389/cn=SybaseServers,dc=mycompany,
dc=com????bindname=cn=Manager,cn=Users,dc=mycompay,
dc=com?secret
```

After setting the appropriate read and write permissions, you will be able to use the Sybase utility programs such as dscp or dsedit to store, view, and modify Sybase server entries in the Active Directory.

Name syntax for the Interfaces file

The *interfaces* file is a flat directory. The fully qualified name for an *interfaces* file entry is the same as the common name. See “Interfaces file” on page 142.

Locating entries with a DIT base

A Directory Information Tree base, or **DIT base**, is an intermediate node in a directory tree used to qualify partial entry names. An application’s DIT base setting is similar in concept to an application’s current working directory in a hierarchical file system.

For any directory source other than the *interfaces* file, an application can specify a DIT base by setting the CS_DS_DITBASE connection property (see “Base for directory searches” on page 117).

ct_connect uses the DIT base to resolve partial server names. An application specifies a server name for ct_connect in one of two ways:

- By specifying the fully qualified name, or
- By setting the CS_DS_DITBASE connection property and specifying a name relative to the CS_DS_DITBASE node.

Some directory service providers provide a special name syntax to indicate that an entry is fully qualified. When using these directory providers, the application overrides the current DIT base.

The sections below give examples of how the DIT base is combined with partial names. The rules vary by directory service provider. If your directory service provider is not listed, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or the Open Client and Open Server *Configuration Guide* for UNIX.

DIT base for DCE CDS

With DCE CDS as the directory provider, the DIT base may be a cell-relative name or a global name. If a global name is used, it must contain enough information to completely identify a cell.

The following two examples illustrate DIT-base settings for DCE CDS. The first example identifies a DIT base within the current DCE cell:

```
/.:/dataservers
```

The second example identifies a DIT base in the cell `sales.fictional.com` by specifying a global name:

```
/.../sales.fictional.com/dataservers
```

The following example shows a partial name that is passed to ct_connect (as the *server_name* parameter):

```
sybase/test_server
```

ct_connect combines the DIT base and the value of the *server_name* as follows:

```
dit_base_value/server_name
```

For example:

```
/.:/dataservers/sybase/test_server
```

or

```
/.../sales.fictional.com/dataservers/sybase/test_server
```

Client-Library appends a slash (`/`) and the *server_name* value to the DIT base. The DIT base cannot end with a slash, and the *server_name* value cannot begin with a slash.

Client-Library ignores the DIT base when *server_name* contains special syntax that indicates a fully qualified name. This syntax is:

- A cell-relative qualified name (*server_name* begins with "/.:"), or
- A globally qualified name (*server_name* begins with "/...").

In either of these cases, *server_name* is considered to be a fully qualified name, and ct_connect ignores the DIT base.

The default DIT base for the DCE CDS directory driver is:

```
/.:/subsys/sybase/dataservers
```

This default may be overridden by the directory driver configuration. To override the configured default, call ct_con_props to set the CS_DS_DITBASE property.

DIT base for Windows Registry

With the Registry as the connection's directory service provider, ct_connect appends a backslash "\" and the *server_name* value to the DIT base value. The DIT base cannot end with a backslash, and a *server_name* value that represents a partial name cannot begin with a backslash.

This is an example of a DIT base for the Windows Registry:

```
SOFTWARE\SYBASE\SERVER
```

This an example of a partial name that is given to ct_connect:

```
dataserver\fin_data
```

These are combined to yield:

```
SOFTWARE\SYBASE\SERVER\dataserver\fin_data
```

The default DIT base for the Registry directory driver is

```
SOFTWARE\SYBASE\SERVER
```

This default may be overridden by the directory driver configuration. To override the configured default, call ct_con_props to set the CS_DS_DITBASE property.

Names are considered fully qualified when they begin with the DIT base value. For example, if the DIT base is "SOFTWARE\SYBASE\SERVER", then the following is a fully qualified name:

```
SOFTWARE\SYBASE\SERVER\debug\fin_data
```

All DIT base nodes are located relative to the "\HKEY_LOCAL_MACHINE\" key.

To specify a DIT base node from another machine's Registry, include the machine name and a colon (:) in the DIT base value. For example, the following DIT base value refers to the machine *queenbee*'s registry:

```
queenbee:SOFTWARE\SYBASE\SERVER
```

DIT base for the Interfaces file

The CS_DS_DITBASE property is not supported when the connection's directory source is the *interfaces* file.

Viewing directory entries

Using Client-Library, you may view directory entries by coding an application that installs a directory callback routine, then calls ct_ds_lookup and ct_ds_objinfo. See Chapter 9, “Using Directory Services,” in the Open Client *Client-Library/C Programmers Guide* for details.

Directory objects

The attributes of a directory object are determined by what kind of directory object it is. Sybase directory object is Server and the OID is CS_OID_OBJSERVER. See “Server directory object” on page 276.

Properties for directory services

The following properties control an application’s use of directory services:

Directory service cache use

CS_DS_COPY determines whether the connection’s directory service provider is allowed to use cached information to satisfy requests for information in the directory. For directory drivers that support the property, the default is CS_TRUE, which allows the use of cached information.

Not all directory service providers support caching. An application calls ct_con_props(CS_SUPPORTED) to determine if the current directory driver supports caching.

Note CS_DS_COPY cannot be set, cleared, or retrieved unless Client-Library is using a directory service provider that supports caching.

Some directory service providers support a distributed model with *directory server agents* (DSAs) and *directory user agents* (DUAs). Directory server agents are programs that manage the directory and respond to requests from the directory user agents. The DUAs run on each machine and transmit application requests to directory server agents and forward the responses to the application.

Directory caching allows the directory user agent to provide a cached copy of recently read information rather than sending the request to the directory server agent. This can speed up directory request handling.

Using the local copy may be faster, but querying the actual directory ensures that the application receives the most recent changes to directory entries.

Base for directory searches

CS_DS_DITBASE specifies a directory node where directory searches start. This node is called the DIT base.

Note CS_DS_DITBASE cannot be set, cleared, or retrieved unless Client-Library is using a network-based directory service rather than the *interfaces* file.

The default DIT base value is specified as follows:

- In the configuration of the directory driver, or
- By the driver-specific default, if the configuration specifies no default DIT base value.

Directory driver configuration is described in the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

DIT base values must be fully qualified names in the name syntax of the directory service that Client-Library is using. In addition, each driver/provider combination has different rules for combining fully and partially qualified names. See “Directory entry name formats” on page 109 for details.

Directory service expansion of aliases

CS_DS_EXPANDALIAS determines whether the connection’s directory service provider expands alias entries when searching the directory. For directory drivers that support this property, the default is CS_TRUE, which means alias entries are expanded.

Not all directory service providers support aliases. An application calls ct_con_props(CS_SUPPORTED) to determine if the current directory driver supports this property.

Note CS_DS_EXPANDALIAS cannot be set, cleared, or retrieved unless Client-Library is using a directory driver that supports alias entries.

Some directory service providers allow directory alias entries to be created. An alias entry contains a link to a primary entry. Aliases allow the primary entry to appear as one or more entries in different locations.

If CS_DS_EXPANDALIAS is CS_TRUE, the directory service provider is permitted to follow alias links when searching the directory. If the value is CS_FALSE, the links in alias entries are not followed.

Warning! Directories that contain alias entries may contain cyclic search paths as a result of the alias links. If CS_DS_EXPANDALIAS is enabled, it is possible for searches begun by ct_ds_lookup to go on indefinitely if the directory tree contains a cyclic search path.

Directory service failover

CS_DS_FAILOVER determines whether Client-Library fails over to the next directory driver entry in *libtcl.cfg* file and eventually to the *interfaces* file when the current directory driver does not load or when the current directory server is not available. The default CS_DS_FAILOVER value is CS_TRUE, which means Client-Library silently fails over when the current directory driver cannot be loaded.

Client-Library requires a directory for (among other things) mapping logical server names to network addresses. The directory can either be the Sybase *interfaces* file or a network-based directory service such as DCE Cell Directory Services (CDS).

To use a directory source other than the *interfaces* file, Client-Library requires a directory driver.

Failover occurs when an application requests (or defaults) to use a network-based directory service rather than the *interfaces* file. If Client-Library cannot load the directory driver, by default, failover occurs to the next directory driver entry in the *libtcl.cfg* file and, eventually, to the *interfaces* file. An application may set CS_DS_FAILOVER to CS_FALSE to prevent failover.

If directory service failover is not permitted, and Client-Library loads a specified directory driver, then the connection's directory source is undefined. In this case, any subsequent action that requires directory access will fail. These actions are:

- A call to ct_con_props to get, set, or clear any CS_DS_ property besides CS_DS_FAILOVER or CS_DS_PROVIDER
- A call to ct_con_props to get or clear the CS_DS_PROVIDER property
- A call to ct_connect or ct_ds_lookup

For a description of when Client-Library loads a directory driver, see “Directory service provider” on page 120.

For information about Sybase failover options see “High-availability failover” on page 139.

Directory service password

CS_DS_PASSWORD specifies a directory service password to go with the principal (user) name specified as CS_DS_PRINCIPAL. Some directory providers require an authenticated principal (user) name to control the application’s access to directory entries.

For details on CS_DS_PRINCIPAL, see “Directory service principal name” on page 119.

Not all directory service providers support passwords. An application calls ct_con_props(CS_SUPPORTED) to determine if the current directory driver supports this property.

Note CS_DS_PASSWORD cannot be set, cleared, or retrieved unless Client-Library is using a directory service provider that supports the property.

Directory service principal name

CS_DS_PRINCIPAL specifies a directory service principal (user) name to go with the password specified as CS_DS_PASSWORD. Some directory providers require an authenticated principal (user) ID to control the application’s access to directory entries. For drivers that support the property, the default is NULL.

For details on CS_DS_PASSWORD, see “Directory service password” on page 119.

Not all directory service providers support CS_DS_PRINCIPAL. An application calls ct_con_props(CS_SUPPORTED) to determine if the current directory driver supports this property.

Note CS_DS_PRINCIPAL cannot be set, cleared, or retrieved unless Client-Library is using a directory service provider that supports the property.

Directory service random offset

By default, CS_DS_RAND_OFFSET is set to true to avoid disrupting current installations. When set to true, CS_DS_RAND_OFFSET starts from a random offset and scans the network address list until a successful connect occurs. The random offset is determined when the network address list is retrieved from the directory service.

If CS_DS_RAND_OFFSET is set to false, connection attempts start from the initial entry in the network address list.

CS_DS_RAND_OFFSET can be set via ct_con_props, ct_config or *ocs.cfg*.

Directory service provider

CS_DS_PROVIDER contains the name of the current directory service provider as a null-terminated string.

Client-Library uses a driver configuration file to map directory service provider names to directory driver file names. On most platforms, this file is named *libtcl.cfg*. See the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX to view a full description of this file.

Loading the default directory driver

The default provider name corresponds to the first entry in the [DIRECTORY] section of the *libtcl.cfg* driver configuration file. This section has entries of the form:

```
[DIRECTORY]
 provider_name = driver_file_name init_string
 provider_name = driver_file_name init_string
```

where:

- *provider_name* specifies a possible value for the CS_DS_PROVIDER property
- *driver_name* is a file name for the driver
- *init_string* specifies start-up settings for the driver.

If no driver configuration file is present on the system, or the file lacks a [DIRECTORY] section, then the default provider name is “InterfacesDriver” to indicate that Client-Library uses the *interfaces* file as the directory source.

See the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX to view a detailed description of driver configuration on your platform.

For each connection structure, Client-Library loads the default directory driver in any of the following circumstances:

- A call to ct_con_props to get, set, or clear any CS_DS_ property besides CS_DS_FAILOVER or CS_DS_PROVIDER loads the default directory driver if a driver is not already loaded.
- A call to ct_con_props to get the CS_DS_PROVIDER property loads the default directory driver if a driver is not already loaded. A call to clear CS_DS_PROVIDER always unloads the existing driver and reloads the default driver.
- A call to ct_connect or ct_ds_lookup loads the default directory driver if a driver is not already loaded.

When Client-Library cannot load a directory driver, Client-Library silently fails over to the *interfaces* file by default. An application may change this behavior by setting the CS_DS_FAILOVER property before performing any of the actions listed above. For details, see "Directory service failover" on page 118.

Changing to a different directory service provider

Applications change a connection's directory service provider by calling ct_con_props(CS_SET, CS_DS_PROVIDER).

When setting CS_DS_PROVIDER, the new property value must be mapped to a valid directory driver. If this is the case, then Client-Library loads the new driver and initializes it.

If Client-Library cannot load the requested driver, then the connection's state depends on the value of the CS_DS_FAILOVER property and whether a driver was loaded before.

- CS_DS_FAILOVER determines whether Client-Library will fail over to the next directory driver entry in *libtcl.cfg* when a driver does not load or when the current directory server is not available. Client-Library will fail over to the *interfaces* file when the last entry of *libtcl.cfg* is reached. For details, see "Directory service failover" on page 118.
- A connection will have a previously loaded driver if the application previously set the CS_DS_PROVIDER property or if the application previously issued one of the calls that requires a driver. See "Loading the default directory driver" on page 120 for a list of calls that load the default directory driver.

The following table describes the directory source after a call to ct_con_props(CS_SET, CS_DS_PROVIDER) fails.

Driver previously loaded	Value of CS_DS_FAILOVER	New directory provider
Yes	CS_TRUE	Next *libtcl.cfg* entry or *interfaces* file
Yes	CS_FALSE	Revert to previous driver
No	CS_TRUE	Next *libtcl.cfg* entry or *interfaces* file
No	CS_FALSE	Undefined

Directory driver initialization

When a directory driver is loaded, Client-Library assigns a default value for the DIT-base property based on the associated configuration file entry.

For driver configuration instructions, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

Directory service search depth

CS_DS_SEARCH restricts the depth to which a directory search descends from the starting point.

Note CS_DS_SEARCH cannot be set, cleared, or retrieved unless Client-Library is using a directory driver that supports the property.

The following values are legal for CS_DS_SEARCH:

Value	Meaning
CS_SEARCH_ONE_LEVEL (default)	Search includes only the leaf entries that are immediately descendants of the node specified by CS_DS_DITBASE.
CS_SEARCH_SUBTREE	Search the entire subtree whose root is specified by CS_DS_DITBASE.

Not all directory service providers support the search-depth property. An application calls ct_con_props(CS_SUPPORTED) to determine if the current directory driver supports this property.

Note The DCE directory driver does not allow CS_DS_SEARCH to be set to a value other than the default, CS_SEARCH_ONE_LEVEL.

Searches start at the directory node indicated by the value of the CS_DS_DITBASE property (see “Base for directory searches” on page 117).

Directory search size limit

CS_DS_SIZELIMIT limits the number of entries returned by a directory search started with ct_ds_lookup. The default is 0, which indicates there is no size limit.

Not all directory service providers support search-results size limits. An application calls ct_con_props(CS_SUPPORTED) to determine if the current directory driver supports this property.

Note CS_DS_SIZELIMIT cannot be set, cleared, or retrieved unless Client-Library is using a directory driver that supports the property.

Directory search time limit

CS_DS_TIMELIMIT specifies an absolute time limit for completion of a directory search, expressed in seconds. The default is 0, which indicates there is no time limit.

Not all directory service providers support search time limits. An application calls ct_con_props(CS_SUPPORTED) to determine if the current directory driver supports this property.

Note CS_DS_TIMELIMIT cannot be set, cleared, or retrieved unless Client-Library is using a directory driver that supports the property.

Error handling

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

Error reporting during initialization

This section describes how error information is returned during the initialization of a Client-Library application.

cs_ctx_alloc and *cs_ctx_global*

When an application call to either cs_ctx_alloc or cs_ctx_global returns CS_FAIL, extended error information is sent to standard error (STDERR) and to the file *sybinit.err*. The *sybinit.err* file is created in the current working directory.

ct_init

When an application call to ct_init returns CS_FAIL due to a Net-Library error, extended error information is sent to standard error (STDERR) and to the file *sybinit.err*. The *sybinit.err* file is created in the current working directory.

Error and message handling

After initialization, Client-Library applications must handle two types of error and informational messages:

- Client-Library messages, or **client messages**, are generated by Client-Library. They range in severity from informational messages to fatal errors.
- Server messages are generated by the server. They range in severity from informational messages to fatal errors.

Adaptive Server stores the text of its messages in the sysmessages system table. See the Adaptive Server Enterprise *Reference Manual* for a description of this table.

See the Open Server *Server-Library/C Reference Manual* for a list of Open Server messages.

Note Do not confuse Client-Library and server messages with a result set of type CS_MSG_RESULT. Client-Library and server messages are the means through which Client-Library and the server communicate error and informational conditions to an application. An application accesses Client-Library and server messages either through message callback routines or inline, using ct_diag. A message result set, on the other hand, is one of several types of result sets that a server may return to an application. An application processes a result set of type CS_MSG_RESULT by calling ct_res_info to get the message's ID.

Two methods of handling messages

An application handles Client-Library and server messages in one of two ways:

- By installing callback routines to handle messages
- Inline, using the Client-Library routine ct_diag

The callback method has the advantages of:

- Centralizing message handling code.
- Providing a method to gracefully handle unexpected errors. Client-Library automatically calls the appropriate message callback whenever a message is generated, so an application will not fail to trap unexpected errors. An application using only mainline error-handling logic may not successfully trap errors that have not been anticipated.

Inline message handling has the advantage of allowing an application to check for messages at particular times. For example, an application that is creating a connection might choose to wait until all connection-related commands are issued before checking for messages.

Most applications use the callback method to handle messages. However, an application that is running on a platform and language combination that does not support callbacks must use the inline method.

An application indicates which method it will use by calling ct_callback to install message callbacks or by calling ct_diag to initialize inline message handling.

An application uses different methods on different connections. For example, an application installs message callbacks at the context level, allocates two connections, and then calls ct_diag to initialize inline message handling for one of the connections. The other connection will use the default message callbacks that it picked up from its parent context.

An application may switch back and forth between the inline and callback methods:

- Installing either a client message callback or a server message callback turns off inline message handling. Any saved messages are discarded.
- Likewise, calling ct_diag to initialize inline message handling de-installs a connection's message callbacks. If this occurs, the connection's first CS_GET call to ct_diag will retrieve a warning message to this effect.

If a callback of the proper type is not installed and inline message handling is not enabled, Client-Library discards message information.

Using callbacks to handle messages

An application calls ct_callback to install message callbacks.

Client-Library stores callbacks in the CS_CONNECTION and CS_CONTEXT structures. Because of this, when a Client-Library error occurs that makes a CS_CONNECTION or CS_CONTEXT structure unusable, Client-Library cannot call the client message callback. However, the routine that caused the error still returns CS_FAIL.

For more information about using callbacks to handle Client-Library and server messages, see “Callbacks” on page 24 and the ct_callback on page 338 reference page.

Inline message handling

An application calls ct_diag to initialize inline message handling for a connection. A typical application calls ct_diag immediately after calling ct_con_alloc to allocate the connection structure.

An application cannot use ct_diag at the context level. That is, an application cannot use ct_diag to retrieve messages generated by routines that take a CS_CONTEXT (and no CS_CONNECTION) as a parameter. These messages are unavailable to an application that is using inline error handling.

An application that is retrieving messages into a SQLCA, SQLCODE, or SQLSTATE should set the Client-Library property CS_EXTRA_INF to CS_TRUE. See “The CS_EXTRA_INF property” on page 127 for more information.

The CS_DIAG_TIMEOUT property controls whether Client-Library fails or retries when a Client-Library routine generates a timeout error.

If a Client-Library error occurs that makes a CS_CONNECTION structure unusable, ct_diag returns CS_FAIL when called to retrieve information about the original error.

For more information about the inline method of handling Client-Library and server messages, see ct_diag on page 453.

Client-Library message structures

Client-Library uses the following structures to return message information:

- CS_CLIENTMSG – described in the section, “Client-Library and SQL Structures” on page 74.

- CS_SERVERMSG – described in the section, “Client-Library and SQL Structures” on page 74.
- SQLCA – described in the section, “Client-Library and SQL Structures” on page 74.
- SQLCODE – described in the section, “Client-Library and SQL Structures” on page 74.
- SQLSTATE – described in the section, “Client-Library and SQL Structures” on page 74 .

The CS_EXTRA_INF property

The CS_EXTRA_INF property determines whether or not Client-Library returns certain kinds of informational messages.

An application that is retrieving messages into a SQLCA, SQLCODE, or SQLSTATE should set the Client-Library property CS_EXTRA_INF to CS_TRUE. This is because the SQL structures require information that Client-Library does not customarily return. If CS_EXTRA_INF is not set, you may lose information.

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 Client-Library messages.

The additional information returned includes the number of rows affected by the most recent command.

Sequencing long messages

Message callback routines and ct_diag return Client-Library and server messages in CS_CLIENTMSG and CS_SERVERMSG structures. In the CS_CLIENTMSG structure, the message text is stored in the *msgstring* field. In the CS_SERVERMSG structure, the message text is stored in the text field. Both *msgstring* and text are CS_MAX_MSG bytes long.

If a message longer than CS_MAX_MSG, minus 1 bytes is generated, Client-Library’s default behavior is to truncate the message. However, an application can use the CS_NO_TRUNCATE property to tell Client-Library to sequence long messages instead of truncating them.

When Client-Library is sequencing long messages, it uses as many CS_CLIENTMSG or CS_SERVERMSG structures as necessary to return the full text of a message. The message's first CS_MAX_MSG bytes are returned in one structure, its second CS_MAX_MSG bytes are returned in a second structure, and so forth.

Client-Library null-terminates only the last chunk of a message. If a message is exactly CS_MAX_MSG bytes long, the message is returned in two chunks: The first contains CS_MAX_MSG bytes of the message, and the second contains a null terminator.

If an application is using callback routines to handle messages, Client-Library calls the callback routine once for each message chunk.

If an application is using ct_diag to handle messages, it must call ct_diag once for each message chunk.

Note The SQLCA, SQLCODE, and SQLSTATE structures do not support sequenced messages. An application cannot use these structures to retrieve sequenced messages. Messages that are too long for these structures are truncated.

Operating system messages are reported through the *osstring* field of the CS_CLIENTMSG structure. Client-Library does not sequence operating system messages.

Message structure fields for sequenced messages

The *status* field in the CS_CLIENTMSG and CS_SERVERMSG structures indicates whether the structure contains a whole message or a chunk of a message.

- The following *status* values are related to sequenced messages:

Symbolic value	Meaning
CS_FIRST_CHUNK	The message text is the first chunk of the message.
CS_LAST_CHUNK	The message text is the last chunk of the message.

  - If CS_FIRST_CHUNK and CS_LAST_CHUNK are both on, then the message text in the structure is the entire message.
  - If neither CS_FIRST_CHUNK nor CS_LAST_CHUNK is on, then the message text in the structure is a middle chunk.

- The *msgstringlen* field in the CS_CLIENTMSG structure and the *textlen* field in the CS_SERVERMSG structure reflect the length of the current message chunk.
- All other fields in the CS_CLIENTMSG and CS_SERVERMSG are repeated with each message chunk.

Sequenced messages and extended error data

If a sequenced server message has extended error data associated with it, an application can retrieve the extended error data while processing any single chunk of the sequenced message. Once the application has retrieved the extended error data, however, it is no longer available. For more information about extended error data, see “Extended error data” on page 129.

Sequenced messages and ct_diag

If an application is using sequenced error messages, ct_diag acts on message chunks instead of messages. This has the following effects:

- A ct_diag(CS_GET, *index*) call returns the message chunk that has number *index*.
- A ct_diag(CS_MSGLIMIT) call limits the number of chunks, not the number of messages, that Client-Library will store.
- A ct_diag(CS_STATUS) call returns the number of currently stored chunks, not the number of currently stored messages.

Extended error data

Some server messages have extended error data associated with them. Extended error data is simply additional information about the error.

For Adaptive Server messages, the additional information is usually which column or columns provoked the error.

Client-Library makes extended error data available to an application in the form of a parameter result set, where each result item is a piece of extended error data. A piece of extended error data may be named and can be of any datatype.

An application can retrieve extended error data but is not required to do so.

Uses for extended error data

Applications that allow end users to enter or edit data often need to report errors to their users at the column level. The standard server message mechanism, however, makes column-level information available only within the text of the server message. Extended error data provides a means for applications to conveniently access column-level information.

For example, imagine an application that allows end users to enter and edit data in the *titleauthor* table in the *pubs2* database. *titleauthor* uses a key composed of two columns, *au_id* and *title_id*. Any attempt to enter a row with an *au_id* and *title_id* that match an existing row will cause a “duplicate key” message to be sent to the application.

On receiving this message, the application needs to identify the problem column or columns to the end user, so that the user can correct them. This information is not available in the duplicate key message, except in the message text. The information is available, however, as extended error data.

Retrieving extended error data

Not all server messages provide extended error data. When Client-Library returns standard server message information to an application in a CS_SERVERMSG structure, it sets the CS_HASEED bit of the *status* field of the CS_SERVERMSG structure if extended error data is available for the message.

Extended error data is returned to an application in the form of a parameter result set that is available on a special CS_COMMAND structure that Client-Library provides.

To retrieve extended error data, an application processes the parameter result set.

Server message callbacks and extended error data

Within a server message callback routine, an application retrieves the CS_COMMAND with the extended error data by calling ct_con_props with *property* as CS_EED_CMD:

```
CS_RETCODE      ret;
 CS_COMMAND      *eed_cmd;
 CS_INT          outlen;

ret = ct_con_props(connection, CS_GET, CS_EED_CMD,
     &eed_cmd, CS_UNUSED, &outlen);
```

ct_con_props sets *eed_cmd* to point to the CS_COMMAND on which the extended error data is available.

After it has the CS_COMMAND, the callback routine processes the extended error data as a normal parameter result set, calling ct_res_info,ct_describe, ct_bind, ct_fetch, and ct_get_data to describe, bind, and fetch the parameters. It is not necessary for the callback routine to call ct_results.

Inline error handling and extended error data

An application that is handling server messages inline retrieves the CS_COMMAND with the extended error data by calling ct_diag with *operation* as CS_EED_CMD:

```
CS_RETCODE      ret;
 CS_COMMAND      *eed_cmd;
 CS_INT          index;

ret = ct_diag (connection, CS_EED_CMD,
     CS_SERVERMSG_TYPE, index, &eed_cmd);
```

In this call, *type* must be CS_SERVERMSG_TYPE and *index* must be the index of the message for which extended error data is available. ct_diag sets *eed_cmd* to point to the CS_COMMAND on which the extended error data is available.

After it has the CS_COMMAND, the application processes the extended error data as a normal parameter result set, calling ct_res_info,ct_describe, ct_bind, ct_fetch, and ct_get_data to describe, bind, and fetch the parameters. It is not necessary for the application to call ct_results.

Server transaction states

Server transaction state information is useful when an application needs to determine the outcome of a transaction.

The following table lists the symbolic values that represent transaction states:

Table 2-22: Transaction states

Symbolic value	Meaning
CS_TRAN_IN_PROGRESS	A transaction is in progress.
CS_TRAN_COMPLETED	The most recent transaction completed successfully.
CS_TRAN_STMT_FAIL	The most recently executed statement in the current transaction failed.
CS_TRAN_FAIL	The most recent transaction failed.
CS_TRAN_UNDEFINED	A transaction state is not currently defined.

Retrieving transaction states in mainline code

In mainline code, an application retrieves a transaction state by calling ct_res_info with *type* as CS_TRANS_STATE:

```
CS_RETCODE      ret;
 CS_INT          outlen;
 CS_INT          trans_state;

ret = ct_res_info (cmd, CS_TRANS_STATE,
     &trans_state, CS_UNUSED, &outlen)
```

ct_res_info sets *trans_state* to one of the symbolic values listed in Table 2-22 on page 132.

Transaction state information is available only for CS_COMMAND structures with pending results or an open cursor. That is, transaction state information is available if an application's last call to ct_results returned CS_SUCCEED.

Transaction state information is guaranteed to be correct only after ct_results sets **result_type* to CS_CMD_DONE, CS_CMD_SUCCEED, or CS_CMD_FAIL.

Retrieving transaction states in a server message callback

An application retrieves transaction states inside a server message callback only if extended error data is available.

Within a server message callback, Client-Library indicates that extended error data is available by setting the CS_HASEED bit of the *status* field of the CS_SERVERMSG structure describing the message.

If extended error data is available, the application retrieves the current transaction state as follows:

1 Retrieves the CS_COMMAND with the extended error data by calling ct_con_props with *property* as CS_EED_CMD.

2 Calls ct_res_info with *type* as CS_TRANS_STATE. ct_res_info sets its **buffer* parameter to one of the symbolic values listed in Table 2-22 on page 132.

Sample programs

The following sample programs and header files are installed with Client-Library. Each file contains a header describing the file's contents and purpose. See the readme file for a complete description of each sample program.

Sample program	Description
arraybind.c	Demonstrates use of array binding in conjunction with a CS_LANG_CMD initiated by ct_command.
blktxt.c	Uses the bulk copy routines to copy static data to a table.
compute.c	Shows how to send a Transact-SQL command and process compute and regular results.
csr_disp.c	Demonstrates the use of a read-only cursor.
csr_disp_ implicit.c	Uses a scrollable cursor to retrieve data from the author table in pubs2 database. Also uses a single pre-fetch buffer and regular program variables.
csr_disp_ scrollcurs.c	Uses a scrollable cursor to retrieve data from the author table in pubs2 database. Also uses a single pre-fetch buffer and regular program variables.
csr_disp_ scrollcurs2.c	Uses a scrollable cursor with arrays as program variables; array binding is used. A single ct_scroll_fetch call displays results in an array.
ctexact.c	A two-phase commit sample program.
ctpr.c	Provides maximum printing lengths for fixed-length data.
ex_alib.c *ex_amain.c*	A collection of routines that form an example of how to write an asynchronous layer on top of Client-Library.
example.h	A header file for the Client-Library sample programs.
exasync.h	Sends a language command and processes the results asynchronously. A header file for the constants and data structures in *ex_alib.c* and *ex_amain.c*.

Sample program	Description
exconfig.c	Shows how to set CS_SERVERNAME property value through the default external configuration file: *$SYBASE/$SYBASE_OCS/config/ocs.cfg*.
exutils.c	Contains utility routines used by all of the other sample programs, and demonstrates how an application can hide some of the implementation details of Client-Library from higher-level programs.
exutils2.c	Contains utility routines used by scrollable cursor sample programs. Used with the csr_disp_scrollable and csr_disp_scrollable2 examples.
exutils.h	A header file for the utility functions in *exutils.c* and *exutils2.c*.
firstapp.c	Connects to a server, sends a select query, and prints the rows.
getsend.c	Shows how to retrieve and update text data.
id_update.c	Demonstrates use of identity_update option.
i18n.c	Demonstrates some of the international features available in Client-Library.
multithrd.c	With *thrdfunc.c*, demonstrates techniques for coding a multithreaded client application with Client-Library. **Note** This sample is coded for use with Solaris native threads package, and DCE pthread APIs.
rpc.c	Illustrates sending an RPC command to a server and then processing the row, parameter, and status results returned from the remote procedure.
secct.c	Demonstrates how to use network-based security features in a Client-Library application. To use the program, DCE or CyberSafe Kerberos must be installed and running, and you must connect to a server that supports network-based security.
secct_dec *secct_krb*	Demonstrates how to use networked-based security with DCE or CyberSafe Kerberos.
thrdfunc.c	With *multithrd.c*, demonstrates techniques for coding a multithreaded client application with Client-Library.
thrdutil.c	Contains utility routines used by multithreaded sample programs. Demonstrates how applications can hide implementation details of Client-Library from higher level programs.
twophase.c	A two-phase commit sample program that performs a simple update on two different servers. Once you run the example, use isql on each server to determine whether the update took place.
uni_blktxt.c	Uses the bulk-copy routines to copy static data to a server table.

Sample program	Description
uni_compute.c	Demonstrates processing compute results. It is a modification of *compute.c*.
uni_csr_disp.c	Demonstrates using a read-only cursor. It is a modification of *uni_csr_disp.c* and requires the unipubs database.
uni_firstapp.c	Modification of *firstapp.c* for use with unichar and univarchar datatypes, and is an introductory example that connects to the server, sends a select query, and prints the rows.
uni_rpc.c	Sends an RPC command to a server and processes the results. This is a modification of *rpc.c* for use with unichar and univarchar datatypes, and requires the unipubs database.
usedir.c	Illustrates searching for servers that are available to connect to.
wide_compute.c	Demonstrates processing compute results with wide tables and larger column sizes.
wide_curupd.c, *wide_dynamic.c*	Uses a cursor to retrieve data from the publishers table in the pubs2 database.
wide_rpc.c	Sends an RPC command to a server and processes the results. Same as *wide_rpc.c*, but uses wide tables and larger column sizes.
wide_util.c	Contains generic routines used by the *wide_** sample programs. They include init_db, cleanup_db, connect_db, handle_returns and fetch_n_print.

Before building and running an example, you must make sure the server and the client application environment are set up properly. In addition, you may want to change the user name with which the example is connecting to the server. For instructions, see the Open Client and Open Server *Programmers Supplement* for Microsoft Windows or Open Client and Open Server *Programmers Supplement* for UNIX.

Client-Library routines in sample programs

The table below lists Client-Library and CS-Library routines along with sample programs that demonstrate their use:

Routine	Sample program(s)
blk_alloc	*blktxt.c*, *uni_blktxt.c*
blk_bind	*blktxt.c*, *uni_blktxt.c*, *wide_compute.c*
blk_done	*blktxt.c*, *uni_blktxt.c*
blk_drop	*blktxt.c*, uni_blktxt.c

Routine	Sample program(s)
blk_init	*blktxt.c, uni_blktxt.c, uni_compute.c*
blk_props	*blktxt.c*
blk_rowxfer	*blktxt.c, uni_blktxt.c*
blk_textxfer	*blktxt.c, uni_blktxt.c*
cs_config	*i18n.c, firstapp.c, thrdutil.c, uni_compute.c*
cs_convert	*exutils.c, i18n.c, rpc.c, thrdutil.c, uni_rpc.c, wide_rpc.c*
cs_ctx_alloc	*ex_amain.c, exutils.c, firstapp.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
cs_ctx_drop	*ex_amain.c, exutils.c, firstapp.c, secct.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c*
cs_loc_alloc	*i18n.c*
cs_loc_drop	*i18n.c*
cs_locale	*i18n.c*
cs_set_convert	*i18n.c*
cs_setnull	*i18n.c, rpc.c*
cs_will_convert	*exutils.c, thrdutil.c*
ct_bind	*compute.c, ex_alib.c, exutils.c, firstapp.c, getsend.c, i18n.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, uni_rpc.c, wide_rpc.c*
ct_callback	*ex_alib.c, ex_amain.c, exutils.c, firstapp.c, thrdutil.c, usedir.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_cancel	*ex_alib.c, ex_amain.c, exutils.c, getsend.c, thrdutil.c*
ct_close	*ex_amain.c, exutils.c, firstapp.c, secct.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c,* wide_compute.c
ct_cmd_alloc	*compute.c, csr_disp.c, ex_alib.c, exutils.c, firstapp.c, getsend.c, i18n.c, multthrd.c, rpc.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_cmd_drop	*compute.c, csr_disp.c, ex_alib.c, exutils.c, firstapp.c, i18n.c, multthrd.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_cmd_props	*ex_alib.c, rpc.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c*
ct_command	*compute.c, ex_alib.c, exutils.c, firstapp.c, getsend.c, i18n.c, multthrd.c, rpc.c, thrdutil.c, arraybind.c, uni_rpc.c, wide_rpc.c*

Routine	Sample program(s)
ct_compute_info	*compute.c, uni_compute.c, wide_compute.c*
ct_con_alloc	*blktxt.c, ex_amain.c, exconfig.c, exutils.c, firstapp.c, secct.c, thrdutil.c, usedir.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_con_drop	*blktxt.c, ex_amain.c, exutils.c, firstapp.c, secct.c, thrdutil.c, usedir.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c*, wide_compute.c
ct_con_props	*blktxt.c, ex_alib.c, ex_amain.c, exconfig.c, exutils.c, firstapp.c, rpc.c, secct.c, thrdutil.c, usedir.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_config	*exutils.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_csr_disp.c, wide_compute.c*
ct_connect	*blktxt.c, ex_amain.c, exconfig.c, exutils.c, firstapp.c, secct.c. thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_ctx_drop	*uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_cursor	*csr_disp.c, multithrd.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_csr_disp.c*
ct_data_info	*getsend.c*
ct_debug	*ex_alib.c, ex_amain.c, exutils.c, thrdutil.c*
ct_describe	*compute.c, ex_alib.c, exutils.c, getsend.c, i18n.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_ds_dropobj	*usedir.c*
ct_ds_lookup	*usedir.c*
ct_ds_objinfo	*usedir.c*
ct_exit	*ex_amain.c, exutils.c, firstapp.c, secct.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c*, wide_compute.c
ct_fetch	*compute.c, ex_alib.c, exutils.c, firstapp.c, getsend.c, i18n.c, thrdutil.c, arraybind.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_get_data	*getsend.c*
ct_init	*ex_amain.c, exutils.c, firstapp.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_csr_disp.c, wide_compute.c*
ct_param	*rpc.c, uni_rpc.c, wide_rpc.c*
ct_poll	*ex_amain.c*

Routine	Sample program(s)
ct_res_info	*compute.c, ex_alib.c, exutils.c, i18n.c, rpc.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, uni_rpc.c, wide_compute.c, wide_rpc.c*
ct_results	*compute.c, csr_disp.c, ex_alib.c, exutils.c, getsend.c, i18n.c, multthrd.c, rpc.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, arraybind.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_send	*compute.c, csr_disp.c, ex_alib.c, exutils.c, firstapp.c, getsend.c, i18n.c, multthrd.c, rpc.c, thrdutil.c, csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, uni_compute.c, uni_csr_disp.c, wide_compute.c*
ct_scroll_fetch	*csr_disp_scrollcurs.c, csr_disp_scrollcurs2.c, ex_utils2.c*
ct_send_data	*getsend.c*
ct_wakeup	*ex_alib.c*

Header files

The header file *ctpublic.h* is required in all application source files that contain calls to Client-Library.

ctpublic.h includes:

- Definitions of symbolic constants used by Client-Library routines
- Declarations for Client-Library routines
- *cspublic.h*, the CS-Library header file

cspublic.h includes:

- Definitions of common client/server symbolic constants
- Type declarations for common client/server structures
- Declarations for CS-Library routines
- *cstypes.h*, which contains type declarations for Client-Library datatypes
- *sqlca.h*, which contains a type declarations for the SQLCA structure
- *csconfig.h*, which contains platform-dependent datatypes and definitions

High-availability failover

A high availability cluster includes two machines that are configured so that, if one machine (or application) is brought down, the second machine assumes the workload of both machines. Each of these machines is called a **node** of the high-availability cluster. A high-availability cluster is typically used in an environment that must always be continuously available.

Sybase's Failover feature is documented in the Adaptive Server Enterprise *Using Sybase Failover in a High Availability System* manual. This section contains information necessary to configure your Open Client applications to connect to the secondary companion during failover.

Add *hafailover* line to *interfaces* file

Clients with the failover property automatically reconnect to the secondary companion when the primary companion crashes or you issue shutdown or shutdown with nowait, triggering failover. To give a client the failover property, you must add a line labeled "hafailover" to the *interfaces* file to provide the information necessary for the client to connect to the secondary companion. You can add this line using either a file editor or the dsedit utility.

UNIX platforms

The following UNIX *interfaces* file entry is for an asymmetric configuration between the primary companion PERSONNEL1 and its secondary companion MONEY1. It includes an hafailover entry that enables clients connected to PERSONNEL1 to reconnect to MONEY1 during failover:

```
PERSONNEL1
   master tcp ether host port
   query tcp ether host port
```

Windows

The following is a Windows *sql.ini* entry for a symmetric configuration between the MONEY1 and PERSONNEL1 companions:

```
[MONEY1]
 query=TCP,FN1,9835
 master=TCP,FN1,9835
 hafailover=PERSONNEL1
[PERSONNEL1]
 query=TCP,HUM1,7586
 master=TCP,HUM1,7586
 hafailover=MONEY1
```

For more information about adding this information to the *interfaces* file, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

Note Client applications must resend any queries that were interrupted by failover. Other information specific to the connection, such as cursor declarations, will also need to be restored.

Client-Library application changes

Note An application installed in a cluster must be able to run on both the primary and secondary companions. If you install an application that requires a parallel configuration, the secondary companion must also be configured for parallel processing so it can run the application during failover.

You must modify any application written with Client-Library calls before it can work with Sybase's Failover software. The following steps describe the modifications:

1. Set the CS_HAFAILOVER property using the ct_config and ct_con_props Client-Library API calls. Legal values for the property are CS_TRUE and CS_FALSE. The default value is CS_FALSE. You can set this property at either the context or the connection level using code similar to:

```
CS_INT TRUE = CS_TRUE;
CS_INT FALSE = CS_FALSE;
retcode = ct_config(context, CS_SET, CS_HAFAILOVER,
&true, CS_UNUSED, NULL);
retcode = ct_con_props(connection, CS_SET,
CS_HAFAILOVER, &false, CS_UNUSED, NULL);
```

2. Handle failover messages. As soon as the companion begins to go down, clients receive an informational message that failover is about to occur. Treat this as an informational message in the client error handlers.

3. Confirm failover configuration. Once you have set the failover property and the *interfaces* file has a valid entry for the secondary companion server, the connection becomes a failover connection, and the client reconnects appropriately.

However, if the failover property is set but the *interfaces* file does not have an entry for the hafailover server (or vice-versa), it does not become a failover connection. Instead, it is a normal non-high availability connection with the failover property turned off. You must check the failover property to know whether or not the connection is a failover connection. You can do this by calling ct_con_props with an *action* of CS_GET.

4 Check return codes. When a successful failover occurs, calls to ct_results and ct_send return CS_RET_HAFAILOVER.

   On a synchronous connection, the API call returns CS_RET_HAFAILOVER directly. On an asynchronous connection, the API returns CS_PENDING and the callback function returns CS_RET_HAFAILOVER. Depending on the return code, the application can do the required processing, such as sending the next command to be executed.

5 Restore option values. Any set options that you have configured for this client connection (for example, set role) were lost when the client disconnected from the primary companion. Reset these options in the failed over connection.

6 Rebuild your applications, linking them with the libraries included with the failover software.

Note You cannot connect clients with the failover property (for example, isql -Q) until you issue sp_companion resume. If you do try to reconnect them after issuing sp_companion prepare_failback, the client hangs until you issue sp_companion resume.

Using *isql* with Sybase Failover

To use isql to connect to a primary server with failover capability, you must:

- Choose a primary server that has a secondary companion server specified in its *interfaces* file entry.
- Use the -Q command-line option.

If your *interfaces_file_name* file contained the example entry given in “Add hafailover line to interfaces file,” you could use isql with Failover by entering `isql -S PERSONNEL1 -Q`.

Interfaces file

The **interfaces file** is a dictionary of connection information for Adaptive Servers and Open Server applications. For every server to which a client might connect, the *interfaces* file contains an entry that includes the server name and the necessary information to connect to that server.

The *interfaces* file is the default directory for Client-Library. However, applications may be configured to use a Sybase directory driver so that Client-Library uses a network-based directory service provider. For information on configuring Sybase directory drivers, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX. For information on network-based directory services, see “Directory services” on page 104.

For an Open Server or Open Client application that uses an *interfaces* file, a linked list containing all the *interfaces* file entries is loaded into memory when the application builds an outgoing connection. This linked list is then referenced by subsequent outgoing connections. If the application uses more than one *interfaces* file, linked lists for all of these files are loaded into memory. If an *interfaces* file is updated while the application is running, a linked list corresponding to the updated *interfaces* file is loaded into memory when a new connection is built. The linked list corresponding to the old, unupdated copy of the *interfaces* file is not released from memory until all connections built on the old *interfaces* file have been closed. Consequently, there may be more than one linked list for a given *interfaces* file in memory simultaneously.

On most platforms, the *interfaces* file is an operating system file in text format. On these systems, the default name, default location, and internal format of the *interfaces* file differs by platform. Other platforms use an alternate form of storage. Table 2-23 summarizes *interfaces* files for some common platforms.

Table 2-23: Name of the interfaces file by platform

Platform or platform family	File name
UNIX (all)	*interfaces* Path is specified by the setting of the SYBASE environment variable.
Windows 2000, Windows 2003, and Windows XP	*sql.ini* Path is specified by the setting of the SYBASE environment variable.

Applications can set the CS_IFILE context property to specify a file name and location that are different from the defaults. (See "Location of the interfaces file" on page 215). An alternate default file name and path for the *interfaces* file can be specified by the CS_DEFAULT_IFILE property. For detailed information about the CS_DEFAULT_IFILE property, see the Open Client and Open Server *Common Libraries Reference Manual*.

Overview of Interfaces file entries

The *interfaces* file format varies by platform. To edit your *interfaces* file, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX. The guides have a complete description of the *interfaces* file and how it is used by ct_connect and ct_ds_lookup on your platform.

The discussion here is a platform-independent overview of *interfaces* file entries and how they are used by Client-Library.

Table 2-24 summarizes the common components of an *interfaces* file entry.

Table 2-24: Components of an Interfaces file entry

Component	Description
Transport Address Values	One or more addresses associated with the server name, in a platform-specific format.
Retry Count Value	UNIX platforms provide this component as an alternative to setting the CS_RETRY_COUNT connection property. **Note** Use of CS_RETRY_COUNT is recommended instead.
Loop Delay Value	UNIX platforms provide this component as an alterative to setting the CS_LOOP_DELAY connection property. **Note** Use of CS_LOOP_DELAY is recommended instead.
Security Mechanisms	A list of **object identifier** strings, each of which represents the global name of a security mechanism supported by the server.

Server objects from the Interfaces file

ct_ds_lookup searches for server directory objects in the *interfaces* file when either of the following occurs:

- The application chooses or defaults to use the *interfaces* file as the directory source for a CS_CONNECTION structure. A connection's directory source is specified with the CS_DS_PROVIDER connection property (see "Directory service provider" on page 120).
- Client-Library could not load the directory driver specified by the driver configuration, and failover to the *interfaces* file occurred. Directory service failover occurs only when it has been enabled with the CS_DS_FAILOVER connection property (see "Directory service failover" on page 118).

In these situations, Client-Library maps the contents of each *interfaces* file entry onto an instance of the server directory object that may be viewed with ct_ds_objinfo. Table 2-25 describes the mapping.

Table 2-25: Mapping of server directory object attributes to interfaces file entries

Attribute	Corresponding interfaces file component
Server Entry Version	None. This value is always 1250 when the *interfaces* file is searched.
Server Name Attribute	The server's name in the *interfaces* file. When the *interfaces* file is searched, the value of the name attribute and the directory entry name are the same.
Service Type	None. This value is always "Adaptive Server" when the *interfaces* file is searched.
Server Status	None. The status is always CS_STATUS_UNKNOWN when the *interfaces* file is searched.
Transport Address	The contents of each "query" line in the *interfaces* file entry, returned to the application within a CS_TRANADDR structure. If multiple "query" lines are present in the *interfaces* file, then the CS_ATTRVALUE array which contains the values for this attribute has the same order as the *interfaces* file. "Master" lines are ignored. Clients use only "query" lines when establishing a connection; therefore, "master" lines are ignored when ct_ds_lookup reads the *interfaces* file. For information about the format of transport addresses, see "Transport address values" on page 474.
Security Mechanisms	The OIDs listed on the "secmech" line of the entry, each within a CS_OID structure.
Retry Count	The "retry_count" option can be included in *interfaces* file entries for some platforms. It controls the number of times Client-Library attempts to connect each server address. Applications may set the CS_RETRY_COUNT property for the connection to get equivalent behavior—see "Retry count" on page 227. If present in the entry, this value is returned as an attribute with OID string CS_OID_ATTRRETRYCOUNT and integer syntax.

Attribute	Corresponding interfaces file component
Loop Delay	The “loop_delay” option can be specified in the *interfaces* file for some platforms. Applications can set the CS_LOOP_DELAY connection property to get equivalent behavior—see “Loop delay” on page 217. If present in the entry, this value is returned as an attribute with OID string CS_OID_ATTRLOOPDELAY and integer syntax.

International Support

Client-Library provides support for international applications through **localization**. Typically, an application that is localized:

- Uses a native language for Client-Library and Adaptive Server messages
- Uses localized datetime formats
- Uses a specific character set and collating sequence (also called **sort order**) when converting or comparing strings

On most platforms, Client-Library uses environment variables to determine the default localization values that an application will use. If these default values meet an application’s needs, the application does not have to localize further.

If the default values do not meet an application’s needs, the application may use a CS_LOCALE structure to set custom localization values at the context, connection, or data-element levels. For information about using a CS_LOCALE structure, see “Using a CS_LOCALE structure” on page 147.

The Open Client and Open Server *International Developer’s Guide* contains detailed guidelines for coding international Open Client and Open Server applications. This topics page summarizes the information that is specific to Client-Library application development.

When an application needs to use a CS_LOCALE structure

Warning! Platform-specific mechanisms for determining a default locale are discussed in the localization chapter of the Open Client and Open Server *Configuration Guide* for your platform. Client-Library's localization mechanism is platform-specific, and you must read that chapter to understand how the default locale is determined on your platform.

Typically, an application's default locale reflects the language or character set of the local environment. The default locale is determined by the value of the CS_LOC_PROP CS-Library context property. A typical application uses a CS_LOCALE structure only if it is working in a language or character set that is different from the context structure's locale.

For example:

- A German application might need to associate a CS_LOCALE structure with a connection structure to receive Client-Library error messages in French.
- An application that performs its own character set conversion must initialize a CS_LOCALE structure for use with cs_convert.

Using a CS_LOCALE structure

A CS_LOCALE structure defines localization values. An application uses the CS_LOCALE structure to define custom localization values at the context, connection, and data element levels.

To do this, an application:

1. Calls cs_loc_alloc to allocate a CS_LOCALE structure.
2. Calls cs_locale to load the CS_LOCALE with custom localization values. Depending on what parameters it is called with, cs_locale may search for the LC_ALL, LC_CTYPE, LC_COLLATE, LC_MESSAGE, LC_TIME or LANG environment variables.
3. Uses the CS_LOCALE. An application:
   - Calls cs_config with *property* as CS_LOC_PROP to copy the custom localization values into a context structure.

- Calls ct_con_props with *property* as CS_LOC_PROP to copy the custom localization values into a connection structure. Note that because CS_LOC_PROP is a login property, an application cannot change its value after a connection is open.
- Supplies the CS_LOCALE as a parameter to a routine that accepts custom localization values (cs_strcmp, cs_time).
- Includes the CS_LOCALE in a CS_DATAFMT structure describing a destination program variable (cs_convert, ct_bind).

4 Calls cs_loc_drop to deallocate the CS_LOCALE.

Context-level localization

Context-level localization values define the localization for an Open Client context.

When an application allocates a CS_CONTEXT structure, CS-Library assigns default localization values to the new context. On most platforms, environment variables determine the default values. For specific information about how default localization values are assigned on your platform, see the Open Client and Open Server *Configuration Guide*.

Because default localization values are always defined, an application needs to define new context-level localization values only if the default values are not acceptable.

Connection-level localization

Connection-level localization values define the localization for a specific client-server connection.

A new connection inherits default localization values from its parent context, so an application needs to define new localization values for a connection only if the parent context's values are not acceptable.

When an application calls ct_connect to open a connection, the server determines whether or not it can support the connection's language and character set. If it cannot, the connection attempt fails.

Note This functionality is different from that of DB-Library, where a connection uses the Server default native language unless the application calls DBSETLNATLANG to set the native language name.

Data-element level localization

At the data element level, CS_LOCALE defines localization values for a specific data element, for example, a bind variable.

An application needs to define localization values at the data element level only if the existing connection's values are not acceptable.

For example, suppose a connection is using a U.S. English locale (U.S. English language, iso_1 character set, and appropriate datetime formats), but the connection needs to display a *datetime* result column using French day and month names.

The application:

- Calls cs_loc_alloc to allocate a CS_LOCALE structure.
- Calls cs_locale to load the CS_LOCALE structure with French datetime formats.
- Calls cs_dt_info to customize the CS_LOCALE structure's datetime conversion format.
- Calls ct_bind to bind the result column to a character variable. The CS_DATAFMT structure that describes the bind variable must reference the French CS_LOCALE.

When the application calls ct_fetch, the datetime value in the result column is automatically converted to a character string containing French days and months and copied into the bound variable.

Locating localization information

When determining which localization values to use, Client-Library starts at the data-element level and proceeds upward. The order of precedence is:

1. Data element localization values:
   - The CS_LOCALE associated with the CS_DATAFMT structure that describes a data element, or
   - The CS_LOCALE passed to a routine as a parameter.
2. Connection structure localization values.
3. Context structure localization values.

Context-level localization values are always defined because when an application allocates a context structure, CS-Library provides the new context with default localization values.

After allocating a context, an application may change its localization values by calling cs_loc_alloc, cs_locale, and cs_config.

The locales file

The Sybase locales file associates locale names with languages, character sets, and sort orders. Open Client and Open Server products use the locales file when loading localization information.

The locales file directs Open Client and Open Server products to language, character set, and sort order names, but does not contain actual localized messages or character set information.

For more information about the locales file, see the Open Client and Open Server *Configuration Guide*.

Locales file entries

The locales file has platform-specific sections, each of which contains entries of the form:

```
locale = locale_name, language, charset, sortorder
```

sortorder is an optional field. If not specified, the sort order for the specified locale defaults to binary.

Each entry defines a locale name by associating it with a language, character set, and sort order.

For example, a section of the locales file might contain the following entries:

```
locale = default, us_english, iso_1, dictionary
 locale = fr, french, iso_1, noaccents
 locale = japanese.sjis, japanese, sjis
```

These entries indicate that:

- When a locale name of “default” is specified, a language of “us_english,” a character set of “iso_1,” and a sort order of “dictionary” should be used.
- When a locale name of “fr” is specified, a language of “french,” a character set of “iso_1,” and a sort order of “noaccents” should be used.

- When a locale name of “japanese.sjis” is specified, a language of “japanese,” a character set of “sjis,” and a sort order of “binary” (the default sort order) should be used.

Note Sybase predefines some locale names by including entries for them in the locales file. If these entries do not meet your needs, you may either modify them or add entries that define new locale names.

---

cs_locale and the locales file

Before using a CS_LOCALE structure to set custom localization values for a context, connection, or data element, a Client-Library application must call cs_locale to load the CS_LOCALE with the desired localization values.

In loading the CS_LOCALE structure, cs_locale:

1. Determines what to use as a locale name:
   - If cs_locale’s *buffer* parameter is supplied, this is the locale name.
   - If cs_locale’s *buffer* parameter is NULL, cs_locale performs a platform-specific operating system search for a locale name. For information about this search, see the Open Client and Open Server *Configuration Guide* for your platform.
2. Looks up the locale name in the locales file to determine which language, character set, and sort order are associated with it.
3. Loads the type of information specified by the *type* parameter into CS_LOCALE. For example, if *type* is CS_LC_CTYPE, cs_locale loads character set information. If *type* is CS_LC_MESSAGE, cs_locale loads message information.

Macros

Macros are C language definitions that typically take one or more arguments and expand into inline C code when the source file is preprocessed. The following sections introduce the Open Client macros by presenting them in their functional contexts.

Decoding a message number

Client-Library and CS-Library message numbers are CS_INT sized integers that consist of four components: layer, origin, severity, and number. The macros CS_LAYER, CS_ORIGIN, CS_SEVERITY, and CS_NUMBER unpack the components from a message number. See “Client-Library message numbers” on page 81 for more information.

Manipulating bits in a CS_CAP_TYPE structure

Capabilities describe features that a client/server connection supports. Each connection’s capability information is stored in a CS_CAP_TYPE structure.

The macros CS_CLR_CAPMASK, CS_SET_CAPMASK, and CS_TST_CAPMASK manipulate the bits in a CS_CAP_TYPE structure. See “Setting and retrieving multiple capabilities” on page 73 for descriptions of these macros.

Using the *sizeof* operator

The C sizeof operator returns the size of a specified item in bytes. Because the datatype of its return value varies from platform to platform, specifying sizeof in place of a CS_INT argument to a Client-Library routine may result in a compiler error or warning if the type returned is not the same base type as CS_INT.

Client-Library provides the following macro to enable an application to use the sizeof function when calling Client-Library routines

CS_SIZEOF(*variable*) – casts a sizeof return value to CS_INT.

This macro is defined in the header file *cstypes.h*.

Prototyping functions

Some C compilers require each function to be declared with an ANSI-style prototype that indicates the placement and datatype of each argument received by the function. Other compilers do not recognize ANSI-style prototypes.

The PROTOTYPE macro allows forward declarations that are agreeable to both ANSI and non-ANSI compilers. This macro is used in forward declarations of C functions as:

```
PROTOTYPE (( argument_list ));
```

where *argument_list* is the ANSI-style argument list. PROTOTYPE is conditionally defined. If the compiler supports ANSI-style prototypes, then PROTOTYPE echos the argument list into the compiled code. Otherwise, PROTOTYPE echos nothing.

The following example illustrates the use of PROTOTYPE:

```
extern CS_RETCODE CS_PUBLIC ex_clientmsg_cb PROTOTYPE((
         CS_CONTEXT *context,
         CS_CONNECTION *connection,
         CS_CLIENTMSG *errmsg
         ));

CS_RETCODE CS_PUBLIC
 ex_clientmsg_cb(context, connection, errmsg)
 CS_CONTEXT      *context;
 CS_CONNECTION   *connection;
 CS_CLIENTMSG    *errmsg;
 {
    ... function body goes here ...
 }
```

CS_PUBLIC is used in callback function prototypes to make sure that machine-specific declaration requirements are satisfied. See “Declaring callbacks with CS_PUBLIC” on page 31 for more information.

Multithreaded applications: signal handling

This section provides information about signal handling for multithreaded applications on UNIX platforms. It supplements the Open Client and Open Server documentation that explains how to use the reentrant versions of the Sybase libraries to build multithreaded applications.

Basic concepts

UNIX operating systems use a signal to report an exceptional situation to a process. Some signals report synchronous events, such as references to an invalid address. Other signals report asynchronous events, such as the disconnection of a phone line.

You can install a handler function to specify an action to be taken when a signal occurs. When the signal occurs, the operating system executes the handler function.

Use Sybase-provided calls to install signal handlers. If you use operating system calls to install signal handlers, this interferes with the internal workings of the Sybase libraries.

Signal handling in nonthreaded environments

Signal handling is straightforward in a traditional, nonthreaded UNIX environment that uses pre-12.0 or version 12.0 and later nonthreaded Sybase libraries. Each process has a single thread of control. You register a handler for a given signal with Open Client and Open Server library calls. Use ct_callback in Client-Library and srv_signal in Server-Library.

When a signal occurs, the Sybase library traps the signal and calls the designated signal handler. To mask a signal, blocking it from delivery to a process, use the sigprocmask UNIX system call.

Types of signals

Signals fall into two categories that correspond to the events by which they are generated.

Type of event	Type of signal
Exception	Synchronous signal
External event	Asynchronous signal

Exceptions and synchronous signals

Synchronous signals are generated by exceptions, or errors, which are caused by invalid operations in a program. Examples of exceptions include attempts to access invalid memory addresses and attempts to divide by zero.

Examples of synchronous signals include SIGILL, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, and SIGPIPE.

External events and asynchronous signals

Asynchronous signals are generated by events outside the control of the process that receives them, and arrive at unpredictable times. Asynchronous signals are delivered to the process without regard to the instruction that is executing.

Typically, the asynchronous signals are SIGHUP, SIGINT, SIGQUIT, SIGALRM, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGPWR, SIGVTALRM, SIGPROF, SIGIO, SIGWINCH, SIGTSTP, SIGCONT, SIGTTIN, SIGTTOU, and SIGURG.

The Sybase libraries treat the asynchronous signal SIGTRAP as a synchronous signal. See “SIGTRAP signal” on page 158 for more information.

Signal handlers

For all UNIX platforms, signal handlers are installed on a per-process basis. In a multithreaded environment, there is only one signal handler for each signal in a process. The last signal handler installed for any thread is valid for all threads in the process. The handler is called when the signal is delivered.

Signal masking

Signal masking lets you specify that a signal will not be delivered until some condition is met.

Nonthreaded environments have only one thread of control. Each signal is masked or unmasked for the entire process.

In multithreaded environments, signal masking is handled differently on different platforms:

- On platforms that do not support native threads, such as HP 9000/800 HP-UX 10.x, signals are masked on a per-process basis. Changing the signal mask on one thread affects the entire process.
- On platforms that support native threads, such as Sun Solaris 2.x (SPARC), IBM RS/6000 AIX, HP 9000/800 HP-UX 11.x, and others, signals are masked on a per-thread basis. Masking a signal on one thread affects that thread only. To mask a signal for the entire process, you must mask the signal for each of its threads.

Threads spawned by a parent thread inherit the signal mask of the parent thread. You can build applications to take advantage of signal-mask inheritance. If a signal must be masked for an entire process, mask it for the main or initial thread. Any thread created thereafter inherits this thread's signal mask.

Signal delivery

A nonthreaded environment has only one thread of control. Synchronous and asynchronous signals are delivered to the process.

In a multithreaded environment, multiple threads represent multiple threads of control. A synchronous signal is always delivered for the thread that caused the exception. An asynchronous signal is delivered for the first executing thread for which delivery of the signal is enabled.

You can specify that an asynchronous signal will be delivered for a thread or set of threads. Unmask the signal for a set of threads to enable signal delivery for these threads. Mask the signal for all other threads to disable signal delivery. The kernel holds a signal until it executes a thread for which delivery of the signal is enabled.

Using sigwait to handle asynchronous signals

Client-Library and Server-Library support the installation of signal handlers using the ct_callback and srv_signal routines. To allow the use of functions that are not asynchronous-signal-safe, multithreaded applications should, instead of installing a regular signal handler, install a thread that calls sigwait to obtain any pending signals. To allow the ct_callback and srv_signal routines to correctly install signal handler functions this way, the first call to cs_ctx_alloc or cs_ctx_global blocks the signals in all but one thread. This thread can be instructed to start or stop blocking any thread signal and is known as the catcher thread. When a signal handler is installed using the ct_callback or srv_signal routines, the catcher thread blocks the corresponding signal. A separate thread is then spawned to invoke sigwait for this signal and to execute the appropriate user-provided signal handler function when the signal is received.

Note This functionality is possible only for applications that invoke the cs_ctx_alloc and cs_ctx_global routines before creating any threads and that install signal handlers only with ct_callback or srv_signal.

You may not want Open Client/Open Server libraries to intervene and handle thread signals. To override the Open Client/Open Server thread-handling and allow your application to handle signals itself, use the following procedure.

❖ **Allowing your application to handle a thread signal by itself**

1. In the main process thread, block the signal you wish to handle before your first call to cs_ctx_alloc or cs_ctx_global.
2. Install a dummy signal handler to avoid the signal handler being set to SIG_IGN.
3. To create the process threads, including the catcher thread, invoke cs_ctx_global. The catcher thread will have the signal blocked.
4. Do one of the following:
   - Install your own thread, calling sigwait, or
   - Unblock the signal, and install a regular signal handler (using, for example, sigaction). Make sure that the signal handler is asynchronous-signal-safe.
5. From the main process thread, unblock the signal that was blocked in step 1. Now all Open Client/Open Server threads will have the signal blocked. The main process thread will not have the signal blocked, nor will any threads created directly in the main thread. Do not install a signal handler with ct_callback or srv_signal.

Special Sybase signal handlers

In nonthreaded environments, you can mask or unmask signals using UNIX system calls.

In multithreaded environments, using versions of Open Client and Open Server earlier than 12.0, you could not change masking for threads used internally by Sybase libraries. Using version 12.0 or later of the Sybase libraries, however, two special signal handlers are available for masking or unmasking signals:

- CS_SIGNAL_BLOCK – to mask a signal, install this signal handler using the Sybase-provided signal installation routine. When the signal occurs, it is held until you unmask it.
- CS_SIGNAL_UNBLOCK – to unmask a signal, install this signal handler using the Sybase-provided signal installation routine.

Other special signal handlers for multithreaded environments include:

- CS_SIGNAL_IGNORE – this signal handler ignores a signal. CS_SIGNAL_IGNORE works the same way as the UNIX special signal handler SIG_IGN.
- CS_SIGNAL_DEFAULT – this signal handler takes a default action when a signal occurs. CS_SIGNAL_DEFAULT works the same way as the UNIX special signal handler SIG_DFL.

SIGTRAP signal

Sybase libraries treat the asynchronous signal SIGTRAP as a synchronous signal.

If it were treated as an asynchronous signal, the signal would be masked on the calling thread when an application called srv_init or ct_init. That would disable debugging, because many debuggers use SIGTRAP to communicate with the application being debugged. To avoid interfering with debugging, SIGTRAP is treated as a synchronous signal.

Note UNIX does not allow asynchronous signals to be handled like synchronous signals. You cannot install signal handlers for SIGTRAP using Sybase-provided calls.

Using Sun's ALARM and SETITIMER routines

If you use the Sun routines ALARM or SETITIMER on Solaris 2.8, review the bug that is documented in each routine's man page.

To work around the bug, use alarm and link in *pthread* ahead of *thread*, as shown.

```
-lpthread -lthread
```

Multithreaded programming

This version of Client-Library supports multithreaded programming. Multithreaded applications need to be linked with the multithreaded libraries included with Client-Library.

Not all operating systems provide threads, and Client-Library may not support every thread interface available on a system. For a list of thread *interfaces* that Client-Library supports, see the Open Client and Open Server *Programmers Supplement* for Microsoft Windows or Open Client and Open Server *Programmers Supplement* for UNIX.

On platforms where no thread support is available, the application may be able to use Client-Library's asynchronous interface to achieve the same effect. See "Asynchronous programming" for more information.

What is a thread

A path of execution through a program in memory. With traditional systems, each process on the system has only one thread of execution. On a multithreaded system, several threads can be started within one process. Threads within a process share the same access to the process resources such as memory and open file descriptors.

Multithreaded systems typically provide the following features:

- Thread-management routines to create and destroy threads.
- A thread scheduler that manages the simultaneous execution of multiple threads within the same process.
- Thread serialization primitives for ensuring that access to shared resources from different threads is mutually exclusive. That is, if one thread has begun to access a shared resource, then no other thread must access the resource until the first one is finished.

  For example, if a linked list is shared by multiple threads, then each traversal, insertion, and deletion operation is a critical section that must be serialized with other traversals, insertions, or deletions. All such critical sections must be serialized so that the execution a critical section in one thread is not interleaved with the execution of a related critical section in another thread.

  A serialization primitive consists of a lockable object (for instance, a mutex) and routines to lock and unlock the object.
- Thread synchronization primitives for synchronizing dependent actions performed by different threads. A synchronization primitive consists of a synchronization object (for instance, a condition variable), a routine to wait on the condition, and a routine to signal that the condition is satisfied.

Benefits of multiple threads

As an application designer, you can use multiple threads to allow different parts of a program to execute concurrently.

For example, an interactive Client-Library application can use one thread to query a server and another thread to manage the user interface. Such an application seems more responsive to the user because the user-interface thread is able to respond to user actions while the query thread is waiting for results.

As another example, consider an application that uses several connections to one or more servers. In this situation, each connection can be run within a dedicated thread. Then, while one thread is waiting for command results, the other threads can be processing received results or sending new commands. Such an approach may increase throughput because the application spends less idle time while waiting for results.

Another reason to use multiple threads is that on a multiple-processor system, the system's thread library may schedule an application's threads to run on different processors.

Threads provide one method of achieving concurrency in a Client-Library program. The other method is to use Client-Library's asynchronous programming interface. Asynchronous programming allows limited concurrency. For more information on this alternative, see "Asynchronous programming" on page 12.

Types of threads

As an application designer, you can use multiple threads to allow different parts of a program to execute concurrently.

- A native thread is a thread that the application creates via direct calls to operating system routines and is scheduled by the operating system.
- An Open Server thread is a thread that is created and scheduled by Server-Library. Gateway applications use Open Server threads.

In some cases, Open Server threads may actually be implemented using native threads. However, an Open Server application always manages thread operations by calling Server-Library routines, even when Open Server is using a native-thread implementation. In this document, the term native thread always refers to a thread created directly by an application call to a system routine.

Note that native threads are not available on all platforms. In particular, threads are not available in Windows 3.1 or MS-DOS. In addition, some platforms may be able to use the DCE pthread library even though the operating system does not supply system-level threads. For these platforms, a version of Client-Library library files may be provided for use with DCE threads. For more information on what thread environments are supported by a target platform, see the Open Client and Open Server *Programmers Supplement* for your platform .

The Open Client and Open Server *Programmers Supplement* for your platform contains important platform-specific information on using Client-Library with the thread interface available on your system.

Write thread-safe code

While threads allow an application to execute different tasks concurrently, they can also complicate the program logic. You must code you multithreaded programs so that they are thread-safe. A thread-safe program satisfies the following conditions:

1. Access to shared data (such as global variables) must be serialized so that data reads and writes are consistent and atomic. For more information, see “Serializing access to shared data and shared resources.”
2. Access to shared resources (such as file descriptors) must be serialized so that the resource maintains a consistent state. For more information, see “Serializing access to shared data and shared resources.”
3. Dependent actions in different threads must be synchronized so that they are performed in the required order. For more information, see “Synchronizing dependent actions.”
4. Calls to thread-unsafe system routines must be serialized so that only one call is active at one time. For more information, see “Calling thread-unsafe system routines.”
5. Thread serialization primitives must be used in a way that avoids deadlock. For more information, see “Avoiding deadlock.”
6. Calls to CS-Library, Client-Library, and Bulk-Library routines must satisfy the restrictions explained in “Client-Library restrictions for multithreaded programs.”

Program code that does not meet these restrictions is *thread-unsafe*. In general, *thread-unsafe* code does not yield predictable behavior when executed in a multithreaded program. Restrictions 1-5 are the general rules for making any application thread safe. Restriction 6 is specific to Client-Library applications. The following sections explain each restriction in more detail.

Note This explanation is not intended to replace the documentation for your system's thread interface. Please read and understand your system documentation before attempting to use Client-Library in a multithreaded environment.

Note When using multithreaded libraries included with Client-Library, fork() without exec() is not supported. This is because PC Net Library threads required for I/O operations, are not duplicated in the child by the fork() system call.

Serializing access to shared data and shared resources

Because all threads share the same memory and other process resources, data or resources modified by different threads can become inconsistent. This problem is avoided by proper use of serialization primitives to guarantee that data access is atomic.

For example, if multiple threads read and increment a global counter variable, then you must design the application to serialize access to the counter. You can associate a mutex with the counter, and add code that locks the mutex before reading or incrementing the counter to guarantee that each data access is atomic.

As a general rule, avoid resource sharing except when absolutely necessary. The use of serialization primitives can complicate your program, and an overabundance of locking calls can adversely affect performance on some systems.

Read your thread system documentation to understand what serialization primitives are provided and how they are used.

Synchronizing dependent actions

Because threads run concurrently, dependent actions that execute in different threads require synchronization to ensure that they execute in the correct, intended order. You must design the application to use synchronization primitives such as condition variables to ensure the desired order of execution.

For example, when several threads share a queue, some threads may read from the queue (the consumer threads) while others write to the same queue (the producer threads). In this case, access to the queue must be both serialized (to keep the queue data consistent) and synchronized (so that consumers do not read from an empty queue and producers do not write to a full queue).

Both conditions can be satisfied by associating a mutex queue_mutex and condition variables *queue_notempty* and *queue_notfull* with the queue. If the POSIX pthread interface is used, then each producer thread performs the steps below to insert to the queue:

```
pthread_mutex_lock(queue_mutex)
while queue is full
pthread_cond_wait(queue_notfull, queue_mutex)
end while
insert an item
pthread_cond_signal(queue_notempty, queue_mutex)
pthread_mutex_unlock(queue_mutex)
```

Meanwhile, the consumer thread performs the steps below to read from the queue:

```
pthread_mutex_lock(queue_mutex)
while queue is empty
pthread_cond_wait(queue_notempty, queue_mutex)
end while
remove an item
pthread_cond_signal(queue_notfull, queue_mutex)
pthread_mutex_unlock(queue_mutex)
```

If the consumer thread finds the queue empty, it calls the pthread_cond_wait routine on the *queue_notempty* condition. This call will not return until a producer thread calls pthread_cond_signal with *queue_notempty*. When a producer thread inserts an item, it calls pthread_cond_signal to signal that the *queue_notempty* condition is satisfied.

Read your thread system documentation to understand what synchronization primitives are provided and how they are used.

Calling thread-unsafe system routines

If any thread-unsafe routines are called from multithreaded code, then each call must be serialized so that calls to the unsafe routines are not simultaneously active. You can use a serialization primitive such as a *global lock* for this purpose.

In some systems, some of the C standard library routines are not *thread-safe*. If a routine is to be called in multithreaded code, then consult the documentation for that routine to find out whether it is *thread-safe* and what the routine's *thread-safe* usage requirements are.

For Client-Library routines, the section “Client-Library restrictions for multithreaded programs” summarizes *thread-safe* usage.

Avoiding deadlock

In multithreaded code, deadlock can occur when two threads each request a lock held by the other. For example, suppose that there are two threads (thread 1 and thread 2) and two mutexes (A and B). The following scenario is a deadlock:

```
thread 1 locks B
thread 2 locks A
thread 2 requests a lock on B
thread 1 requests a lock on A
```

In this situation, both thread 1 and thread 2 wait forever for the requested locks.

You can typically avoid deadlock by designing *locking protocols* for the application. These specify the order in which simultaneously held locks must be requested. In the scenario above, such a protocol might be stated: “If both mutex A and mutex B are taken, then A must be acquired first.”

On some systems, a thread can deadlock with itself by requesting a lock that it already holds. Read your thread system documentation for the recommended practices to avoid deadlock.

Client-Library restrictions for multithreaded programs

Client-Library applications must satisfy the general restrictions listed above and the specific Client-Library usage restrictions listed here to be thread-safe.

Client-Library's restrictions on thread-safe usage are categorized as follows:

- Context-Level - Thread-safe restrictions on accessing a CS_CONTEXT structure. For details, see “Calling context-level routines.”
- Connection-Level - Thread-safe restrictions on using a CS_CONNECTION structure or subordinate structures (CS_COMMAND, CS_BLKDESC). For details, see “Calling connection-level routines.”
- CS_LOCALE Usage - Thread-safe restrictions on using CS_LOCALE structures. For details, see “Using CS_LOCALE structures.”
- Context-Level - Thread-safe restrictions on accessing a CS_CONTEXT structure. For details, see “Calling context-level routines.”

Calling context-level routines

Client-Library and CS-Library context-level routines are listed in Table 2-26.

Thread-safe calls to context-level routines abide by the following restrictions:

- Calls to cs_ctx_alloc and cs_ctx_drop must not occur simultaneously with any other call to cs_ctx_alloc or cs_ctx_drop.
- Calls to ct_init and ct_exit must not occur simultaneously with any other call to ct_init or ct_exit.
- If a CS_CONTEXT structure is shared by different threads, and thread-unsafe calls are made on that CS_CONTEXT structure, then all calls to context-level routines for that CS_CONTEXT must be serialized. The thread-unsafe context-level calls are indicated in Table 2-26.

Table 2-26: Thread-safe use of CS-Library and Client-Library context-level routines

Routine name	Thread-safe calls	Thread-unsafe calls	Notes
cs_calc	All.		
cs_cmp	All.		
cs_config	When action is CS_GET.	When action is CS_SET or CS_XCLEAR.	
cs_convert	All.		If CS_LOCALE pointers are used within srcfmt or destfmt, access to the CS_LOCALE structures must be thread-safe.

Routine name	Thread-safe calls	Thread-unsafe calls	Notes
cs_ctx_alloc		All	Thread-unsafe for any context. For more information, see "Context initialization and cleanup."
cs_ctx_drop		All, for any context.	Thread-unsafe for any context. For more information, see "Context initialization and cleanup."
cs_ctx_global	All calls after the first call has completed.	First executed call only.	
cs_dt_crack	All.		
cs_dt_info	All.		Access to the CS_LOCALE structure must be *thread-safe*.
cs_diag	When action is not CS_INIT.	When action is CS_INIT.	Only messages generated by the calling thread are visible. For more information, see "CS-Library error handling."
cs_loc_alloc	All.		Access to the CS_LOCALE structure must be thread-safe. See "Using CS_LOCALE structures."
cs_loc_drop	All calls.		
cs_locale	All calls.		
cs_objects	All calls.		
cs_set_convert	When action CS_GET.	When action is CS_SET or CS_CLEAR.	
cs_setnull		All calls that share the same context.	
cs_strbuild	All calls.		
cs_strcmp	All calls.		
cs_time	All calls.		
cs_will_convert	All calls.		
ct_callback	When context is not NULL.		Thread-unsafe at the connection level (all calls where context is NULL). See "Calling connection-level routines" for more information.
ct_con_alloc	All calls.		

Routine name	Thread-safe calls	Thread-unsafe calls	Notes
ct_con_drop	All calls.		
ct_config	When action is CS_GET.	When action is CS_SET or CS_CLEAR.	
ct_debug		When context is not NULL.	See “Calling connection-level routines” for the case when context is NULL.
ct_exit		All calls, for any context.	Thread-unsafe for any context. For more information, see “Context initialization and cleanup.”
ct_init		All calls, for any context.	Thread-unsafe for any context. For more information, see “Context initialization and cleanup.”.
ct_poll		Only when context is not NULL.	

Context initialization and cleanup

The routines cs_ctx_alloc, cs_ctx_drop, ct_init, and ct_exit are thread-unsafe and must be serialized as follows:

- Any call to cs_ctx_alloc or cs_ctx_drop must be serialized with other calls to cs_ctx_alloc or cs_ctx_drop.
- Additionally, any call to ct_init or ct_exit must be serialized with other calls to ct_init or ct_exit.

You need not worry about this issue if your program allocates and initializes all necessary CS_CONTEXT structures in single-threaded initialization code and performs all context-level cleanup operations in single-threaded cleanup code. An alternative strategy limits the use of a given context structure to a single thread to eliminate the need for serialization.

CS-Library error handling

If multiple threads share a context, all of them must use the same error handling method: all threads sharing the context must use the inline (cs_diag) method or all threads must use the callback method.

If errors are handled inline with cs_diag, then each thread must call cs_diag to retrieve its own CS-Library error messages. cs_diag only shows messages generated by the calling thread.

Calling connection-level routines

Connection-level routines are routines in Client-Library and Bulk-Library that take a pointer to any of the following structures as an argument:

- A connection structure (CS_CONNECTION *)
- A command structure (CS_COMMAND *)
- A directory object structure (CS_DS_OBJECT *)
- A Bulk-Library bulk descriptor structure (CS_BLKDESC *)

The following routines are also considered connection-level routines:

- ct_callback (only with a non-NULL connection argument).
- ct_debug (only with a NULL context argument). Note that ct_debug(CS_DBG_ALL) and other calls that require a non-NULL context are considered thread-unsafe context-level calls.
- ct_poll (only with a non-NULL connection argument).

Calls to routines using a command structure, a directory object structure, or a bulk descriptor structure should be treated as connection-level calls on the parent connection. Thread-safe calls to connection-level routines abide by the following restrictions:

- Threads that share the same connection must synchronize connection-level calls so that:
  - Calls are not simultaneously active, and
  - Calls occur in the intended order.
- Calls that reference a given connection structure cannot be simultaneously active with any thread-unsafe context-level call that access the connection's parent context. The thread-unsafe context-level calls are listed in Table 2-26.

Using CS_LOCALE structures

Client-Library, CS-Library, and Bulk-Library routines can receive a pointer to a CS_LOCALE structure directly or indirectly within an exposed structure. The exposed structures CS_DATAFMT and CS_IODESC each contain a locale field that can hold a CS_LOCALE pointer.

Any call to a Client-Library, CS-Library, or Bulk-Library routine that receives a non-NULL CS_LOCALE pointer is thread safe as long as:

- The routine does not modify the CS_LOCALE structure. The table below lists the routines that modify a CS_LOCALE structure.
- No call listed below is simultaneously active with any other call that references the same CS_LOCALE structure:
  - cs_dt_info(CS_SET)
  - cs_loc_drop
  - cs_locale(CS_SET)

Coding thread-safe callback routines

If a callback function can be called from multiple threads, then the callback must be *thread-safe*. Generally, callbacks are called from the thread that provoked the callback event, but some callbacks can be invoked by an internal Client-Library worker thread and execute in the context of the worker thread. Table 2-27 on page 171 summarizes which thread invokes each callback type.

Callbacks for use in a multithreaded environment should be coded according to the following rules:

- Callbacks should follow the general implementation rules described in the “Callbacks” topics page.
- Callback code must be thread-safe and follow the restrictions noted under “Write thread-safe code.”
- Callbacks that can be invoked by a Client-Library worker thread require an appropriate design. The callback can run concurrently with mainline code, and access to any data structures shared between the callback and mainline code (including data stored as a CS_USERDATA property) must be *thread-safe*.

Threads and fully asynchronous mode

On some platforms such as Windows, Client-Library implements fully asynchronous network I/O by spawning internal worker threads to handle network I/O. When a fully asynchronous I/O is in effect on these platforms, an internal Client-Library thread waits for the completion of each I/O request, and invokes the application's completion callback. Before the operation completes, the thread may call other application callbacks. For example, if the server sends server messages, the internal Client-Library thread reads the messages and calls the application's server-message callback.

In these situations, the callback code and the application's mainline code run in different threads. When coding a fully asynchronous application on a platform where Client-Library uses thread-driven I/O, you must make sure that the callbacks communicate properly with the mainline code. Table 2-27 summarizes which thread invokes each callback type.

Table 2-27: Callback types and the thread they are invoked from

Callback type	Invocation thread
CS-Library error handler	The thread that provoked the error event.
Client message	When the CS_NETIO connection property is CS_DEFER_IO or CS_SYNC_IO, the callback is invoked from the thread that provoked the error. On thread-driven I/O platforms, when the CS_NETIO connection property is CS_ASYNC_IO, the callback can be invoked from a Client-Library worker thread or the thread that provoked the error event, depending on when the error is discovered.
Completion	When the CS_NETIO connection property is CS_DEFER_IO, the callback is invoked from the thread that calls ct_poll. On thread-driven I/O platforms, when the CS_NETIO connection property is CS_ASYNC_IO, the callback is invoked from a Client-Library worker thread.
Directory	When the CS_NETIO connection property is CS_SYNC_IO or CS_DEFER_IO, the callback is invoked from the thread that called ct_ds_lookup. On thread-driven I/O platforms, when the CS_NETIO connection property is CS_ASYNC_IO, the callback is invoked from an internal Client-Library thread.
Encryption, negotiation, or security session	When the CS_NETIO connection property is CS_SYNC_IO or CS_DEFER_IO, the callback is invoked from the thread that called ct_connect. On thread-driven I/O platforms, when the CS_NETIO connection property is CS_ASYNC_IO, the callback is invoked from an internal Client-Library thread.

Callback type	Invocation thread
Notification	When the CS_ASYNC_NOTIFS property is CS_FALSE (the default), the callback is invoked from the thread that was reading from the network at the time the notification arrived. On thread-driven I/O platforms, when the CS_ASYNC_NOTIFS property is CS_TRUE, the callback is invoked from an internal Client-Library thread.
Server message	When the CS_NETIO connection property is CS_DEFER_IO or CS_SYNC_IO, the callback is invoked from the thread that is processing results or sending commands on the connection when the message arrives. On thread-driven I/O platforms, when the CS_NETIO connection property is CS_ASYNC_IO, the callback is invoked by an internal Client-Library thread.
Signal	On platforms that support signals and where Client-Library uses thread-driven I/O, signal callbacks must be installed with ct_callback. The signal callback is invoked by an internal Client-Library thread, and not at interrupt level.

Multithreaded programming models for Client-Library

This section outlines some programming strategies that can simplify the design of multithreaded applications.

One-thread, one-connection model

In this model, your program:

- Performs all needed library initialization and cleanup in single-threaded initialization and cleanup code.
- Creates a dedicated thread for each connection and limits all use of a particular connection to its dedicated thread.

The one-thread, one-connection model is the simplest and requires the least amount of inter-thread synchronization. It is also the most natural model for an Open Server gateway.

The basic steps are as follows:

- Initialization - Any thread-unsafe context-level calls (listed in Table 2-26 on page 165) are called in single-threaded initialization code. If the application is a multithreaded library that calls Client-Library routines, the library's public initialization routine can call the POSIX pthread_once() routine (or your system's equivalent) to safely invoke an internal, single-threaded routine that initializes Client-Library. Typically, the start-up thread will wait for some event that indicates the program (or library) should terminate.
- Processing - After all initialization has been performed, the application spawns one thread for each connection to be created. The thread then allocates its own connection with ct_con_alloc, connects to a server, and performs the processing for that connection.
- Shutdown - When the program or library determines that it should terminate, each thread that is bound to a connection closes its connection (and terminates itself if necessary). The application then performs any cleanup (such as calling ct_exit and cs_ctx_drop) in single-threaded code.

Worker-thread model

In this model, you design the application to maintain a pool of available Client-Library connection structures. Connections can be shared by multiple threads, but a given connection is used in only one thread at any given time. When a thread needs to perform a Client-Library operation, it takes an available connection and marks it “unavailable,” then performs connection-level operations. When the connection-level operations are complete, the thread marks the connection as “available.” The application design can use the CS_USERDATA connection property to associate state information (such as availability) with a connection structure.

This model is similar to the one-thread, one-connection model, except that the binding between connections and threads is dynamic rather than static. The application code and data structures used to manage the *connection pool* must be thread-safe.

Other thread models

Using other programming models, shared connections may be active on different threads, or thread-unsafe context-level calls may be made in multithreaded code. In these situations, synchronization is more complex. The program must follow all restrictions described in this“Multithreaded programming” section.

Options

Options affect how Sybase Servers respond to commands.

An application sets options to customize a server’s query-processing behavior. For example, an application sets the CS_OPT_FIPSFLAG option to tell a server to flag any nonstandard SQL commands that it receives.

A Client-Library application sets and clears Adaptive Server query-processing options in one of two ways:

- By using a Transact-SQL language command (set)
- By calling ct_options

An application may use only one of these methods; otherwise, Client-Library/server communications may become confused.

The ct_options method is recommended because it allows an application to check the status of an option, which is not allowed by the Transact-SQL set command.

For more information about Adaptive Server query-processing options, see the set command in the *Adaptive Server Reference Manual*.

Setting options externally

The Client-Library routine ct_connect optionally reads a section from the Open Client and Open Server runtime configuration file to set server options for a newly-opened connection.

For a description of this feature, see “Using the runtime configuration file” on page 305.

Table 2-28 lists the symbolic constants used with ct_options:

Table 2-28: Symbolic constants for server options

Symbolic constant	Meaning	Default value
CS_OPT_ANSINULL	Determines whether evaluation of NULL-valued operands in SQL equality (=) or inequality (!=) comparisons is ANSI-compliant. If CS_TRUE, Adaptive Server enforces the ANSI behavior that “= NULL” and “is NULL” are not equivalent. In standard Transact SQL, “= NULL” and “is NULL” are considered to be equivalent. This option affects “<> NULL” and “is not NULL” behavior in a similar fashion.	CS_FALSE.
CS_OPT_ANSIPERM	Determines whether Adaptive Server is ANSI-compliant with respect to permissions checks on update and delete statements. If CS_TRUE, Adaptive Server is ANSI-compliant.	CS_FALSE.
CS_OPT_ARITHABORT	Determines how Adaptive Server behaves when an arithmetic error occurs. If CS_TRUE, both the arith_overflow and numeric_truncation options are set to on. An entire transaction or batch in which an error occurred is rolled back when a divide-by-zero error or a loss of precision occurs during either an explicit or implicit datatype conversion. If a loss of scale by an exact numeric type occurs during an implicit datatype conversion, the statement that caused the error is aborted, but the other statements in the transaction or batch continue to be processed. If CS_FALSE, both the arith_overflow and numeric_truncation options are set to off. The statement that caused a divide-by-zero error or a loss of precision during either an explicit or implicit datatype conversion is aborted, but the other statements in the transaction or batch continue to be processed. If a loss of scale by an exact numeric type during an implicit datatype conversion occurs, the query results are truncated and other statements in the transaction or batch continue to be processed.	CS_FALSE.
CS_OPT_ARITHIGNORE	Determines whether Adaptive Server returns a message after a divide-by-zero error or a loss of precision. If CS_TRUE, warning messages are suppressed after these errors. If CS_FALSE, warning messages are displayed after these errors.	CS_FALSE.

Symbolic constant	Meaning	Default value
CS_OPT_AUTHOFF	Turns the specified authorization level off for the current server session. When a user logs in, all authorizations granted to that user are automatically turned off.	Not applicable.
CS_OPT_AUTHON	Turns the specified authorization level on for the current server session. When a user logs in, all authorizations granted to that user are automatically turned on.	Not applicable.
CS_OPT_CHAINXACTS	If CS_TRUE, Adaptive Server uses chained transaction behavior. Chained transaction behavior means that each server command is considered to be a distinct transaction. Adaptive Server implicitly executes a begin transaction before any of the following statements: delete, fetch, insert, open, select, and update. If CS_FALSE, an application must specify an explicit commit transaction statement to end a transaction and begin a new one.	CS_FALSE.
CS_OPT_ CURCLOSEONXACT	If CS_TRUE, all cursors opened within a transaction are closed when the transaction completes.	CS_FALSE.
CS_OPT_DATEFIRST	Sets the first day of the week.	For us_english, the default is CS_OPT_SUNDAY.
CS_OPT_ DATEFORMAT	Sets the order of the date parts month/day/year for entering *date, datetime* or *smalldatetime* data.	For us_english, the default is CS_OPT_FMTMDY.
CS_OPT_FIPSFLAG	Determines whether Adaptive Server displays a warning message when SQL extensions are used. If CS_TRUE, Adaptive Server flags any non-standard SQL commands that are sent. If CS_FALSE, Adaptive Server does not flag non-ANSI SQL.	CS_FALSE.
CS_OPT_FORCEPLAN	If CS_TRUE, Adaptive Server joins tables in the order in which the tables are listed in the from clause of the query.	CS_FALSE.
CS_OPT_ FORMATONLY	If CS_TRUE, Adaptive Server sends back a description of the data, rather than the data itself, in response to a select query. If CS_FALSE, Adaptive Server sends back data in response to a select query.	CS_FALSE.
CS_OPT_HIDE_VCC	When set to CS_TRUE, CS_OPT_HIDE_VCC hides Virtual Computed Columns (VCC) in tables. Column numbers passed to, for example, blk_bind would therefore not include VCC columns. When set to CS_FALSE, VCC's are included in tables.	CS_FALSE.

Symbolic constant	Meaning	Default value
CS_OPT_IDENTITYOFF	Disables inserts into a table's IDENTITY column. See the set command (identity_insert option) in the Adaptive Server documentation.	Not applicable.
CS_OPT_IDENTITYON	Enables inserts into a table's IDENTITY column. See the set command (identity_insert option) in the Adaptive Server documentation.	Not applicable.
CS_OPT_IDENTITYUPD_OFF	Disables the identity update option.	Null
CS_OPT_IDENTITYUPD_ON	Enables the identity update option. This option allows you to update the "high", out-of-range identity column values by writing a single SQL update statement that specifies the required range of rows and replaces them with the correct values.	Null
CS_OPT_ISOLATION	Specifies a transaction isolation level. Possible levels are CS_OPT_LEVEL0, CS_OPT_LEVEL1, and CS_OPT_LEVEL3. These correspond to the three levels for Adaptive Server's set transaction isolation level command. CS_OPT_LEVEL0 requires Adaptive Server.	CS_OPT_LEVEL1.
CS_OPT_NOCOUNT	Turns off the return of the number of rows affected by each SQL statement. An application obtains this information by calling ct_res_info.	CS_FALSE.
CS_OPT_NOEXEC	If CS_TRUE, Adaptive Server compiles each query but does not execute it. Use this option in conjunction with CS_OPT_SHOWPLAN.	CS_FALSE.
CS_OPT_PARSEONLY	If CS_TRUE, Adaptive Server checks the syntax of each query and returns any error messages as necessary, but does not execute the query.	CS_FALSE.
CS_OPT_QUOTED_IDENT	If CS_TRUE, Adaptive Server treats all strings enclosed in double quotes (") as identifiers.	CS_FALSE.
CS_OPT_RESTREES	If CS_TRUE, Adaptive Server checks the syntax of each query and returns parse resolution trees (in the form of image columns in a regular row result set) and error messages as necessary, but does not execute the query.	CS_FALSE.

Symbolic constant	Meaning	Default value
CS_OPT_ROWCOUNT	Sets a limit for the number of rows that are affected by a query: limits the number of regular rows returned by a select statement and the number of rows affected by an update or delete statement. If set to 0, the number of rows returned or affected by a command is not limited. If set to a value greater than 0, Adaptive Server stops processing a command when the specified number of rows has been affected. This option does not limit the number of compute rows returned.	0
CS_OPT_SHOW_FI	When set to CS_TRUE, CS_OPT_SHOW_FI adds a column to the table for each Functional Index (FI). Correspondingly, when CS_OPT_SHOW_FI is set to CS_FALSE, FI's are hidden.	CS_FALSE.
CS_OPT_SHOWPLAN	Determines whether a description of each query's processing plan is returned between its compilation and execution. If CS_TRUE, Adaptive Server compiles a query, generates a description of its processing plan, and then executes the query. Client-Library receives the description as a sequence of informational server messages. Application programs access them through the user-supplied server message handler.	CS_FALSE.
CS_OPT_SORTMERGE	Determines whether the use of sort-merge joins during a session are enabled or disabled. See the Adaptive Server Enterprise *Performance and Tuning Guide*.	CS_FALSE.
CS_OPT_STATS_IO	Determines whether Adaptive Server internal I/O statistics (the number of scans, logical reads, physical reads, and pages written) are returned for each query. If CS_TRUE, statistics are returned. Client-Library receives these statistics as informational server messages. Application programs access them through the user-supplied server message handler.	CS_FALSE.

Symbolic constant	Meaning	Default value
CS_OPT_STATS_TIME	Determines whether Adaptive Server parsing, compilation, and execution time statistics are returned for each query. If CS_TRUE, statistics are returned. Client-Library receives these statistics as informational server messages. Application programs access them through the user-supplied server message handler.	CS_FALSE.
CS_OPT_STR_ RTRUNC	Determines whether Adaptive Server is ANSI-compliant with respect to right truncation of character data. When this option is set to CS_TRUE, Adaptive Server raises an error when an insert or an update operation truncates a char or varchar column value and the truncated characters are not all blank. This behavior is ANSI- compliant. When this option is set to CS_FALSE, the Adaptive Server silently truncates char or varchar values that are too long for the column definition.	CS_FALSE.
CS_OPT_TEXTSIZE	Specifies the value of the Adaptive Server global variable *@@textsize*, which limits the size of text or image values that Adaptive Server returns. When setting this option, supply a parameter that specifies length, in bytes, of the longest text or image value that Adaptive Server should return. The Client-Library property CS_TEXTLIMIT has a similar effect. The CS_TEXTLIMIT property sets the size of the largest text/image value that Client-Library returns to the application. CS_TEXTLIMIT does not affect server processing: Client-Library truncates text/image values as they are read from the network. On the other hand, CS_OPT_TEXTSIZE causes the server to truncate values before sending them. In programs that allow application users to run ad hoc queries, the user may override the CS_OPT_TEXTSIZE option with the Transact-SQL set textsize command. To set a text limit that the user cannot override, use the CS_TEXTLIMIT property instead.	32,768 bytes.
CS_OPT_ TRUNCIGNORE	If CS_TRUE, Adaptive Server ignores truncation errors. This is standard ANSI behavior. If CS_FALSE, Adaptive Server raises an error when conversion results in truncation.	CS_FALSE.

Properties

Properties are named values that are stored in a CS_CONTEXT, CS_CONNECTION, or CS_COMMAND hidden structure.

Properties define aspects of Client-Library behavior. For example, a connection structure's CS_NETIO property determines whether the connection is synchronous or asynchronous, and a command structure's CS_HIDDEN_KEYS property determines whether or not hidden keys returned as part of a result set are exposed.

Comparing properties, options, and capabilities

Do not confuse properties with server options or a connection's capabilities. Server options control the server's behavior while executing commands sent to a client. A connection's capabilities determine which types of client requests or server responses can be sent over a connection.

In general, properties control Client-Library's behavior, while server options control the server's response to commands. At a lower level, capabilities constrain the protocol that the client and the server use to communicate.

For example, consider the problem of limiting the size of a text or image datatype value that is selected by an application. To solve this problem, a programmer might code the application to do any of the following:

- Set the CS_OPT_TEXTSIZE option to limit how much of a large text/image value the server sends over the network. (The best solution.
- Set the CS_TEXTLIMIT connection property so that Client-Library truncates CS_TEXT or CS_IMAGE data values that are retrieved from the server. An inefficient solution, since the entire value must be sent over the network before truncation.
- Before opening the connection, call ct_capability to inhibit the connection's CS_DATA_TEXT and CS_DATA_IMAGE response capabilities. A poor solution, since no text or image values are sent by the server in this case.

See "Options" on page 174 and "Capabilities" on page 60.

Login properties

Login properties define values used when logging in to a server. Login properties include CS_USERNAME, CS_PASSWORD, and CS_PACKETSIZE.

A server changes the values of some login properties during the login process. For example, if an application sets CS_PACKETSIZE to 2048 bytes and then logs in, the server will use the specified packet size or can choose a smaller or larger packet size.

Setting and retrieving properties

An application calls ct_config, ct_con_props, and ct_cmd_props to set and retrieve Client-Library properties at the context, connection, and command structure levels, respectively. An application calls cs_config to set and retrieve CS-Library context properties.

When a connection structure is allocated, it picks up default property values from its parent context. For example, if CS_TEXTLIMIT is set to 16,000 at the context level, then any connection created within this context will have a default text limit value of 16,000. Likewise, when a command structure is allocated, it picks up default property values from its parent connection.

An application overrides a default property value by calling cs_config, ct_config, ct_con_props, or ct_cmd_props to change the value of the property.

Most properties' values are set or retrieved by an application, but some properties are "retrieve only."

Three kinds of context properties

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-specific context properties. With the exception of CS_LOC_PROP, properties set through cs_config affect only CS-Library. CS-Library-specific context properties are listed on the reference page for cs_config in the Open Client and Open Server *Common Libraries Reference Manual*.

ct_config sets and retrieves the values of Client-Library-specific context properties. Properties set through ct_config affect only Client-Library. Client-Library-specific context properties are listed in Table 2-29 on page 184.

srv_props sets and retrieves the values of Server-Library-specific context properties. Properties set through srv_props affect only Server-Library.

Checking whether a property is supported

Properties are not always supported. A property may be unsupported in the following instances.

- The property is associated with external directory provider software.

  Some of the CS_DS properties control the behavior of external directory provider software (indicated by the connection's CS_DS_PROVIDER property). Sybase's directory driver maps the Client-Library property to an equivalent service-provider setting. However, if the provider has no equivalent setting, the property is not supported. Applications cannot call ct_con_props to get, set, or clear the value of unsupported directory properties.

- The property is associated with external security provider software.

  Some of the CS_SEC properties enable security services, such as data encryption, that are performed by external security software (indicated by the CS_SEC_MECHANISM property). Sybase's security driver maps the Client-Library property to an equivalent service-provider setting. However, a security mechanism may not support every service. Applications cannot call ct_con_props or ct_config to enable a security service that is not supported by the current security mechanism for the connection or context.

Applications check to determine whether a property is supported by calling ct_config or ct_con_props with the *action* parameter as CS_SUPPORTED and the buffer parameter as the address of a CS_BOOL variable.

For example, the following code checks to see if the CS_DS_SEARCH property is supported. You can use this sample code to check support for other properties by replacing the CS_DS_SEARCH parameter with the parameter you are interested in.

```
/* Is CS_DS_SEARCH supported? */
 ret = ct_con_props(conn, CS_SUPPORTED,
                    CS_DS_SEARCH, &boolval,
                    CS_UNUSED,NULL);
 if (ret != CS_SUCCEED)
   ... handle the error ...
 printf("CS_DS_SEARCH %s supported",
        boolval == CS_TRUE ? "is" : "is not");
```

Note The CS_SUPPORTED *action* is allowed only for properties associated with a directory or security driver.

Copying login properties

A login property is a connection property that specifies a value needed to connect to a server. For example, CS_USERNAME and CS_PASSWORD are login properties.

An application copies login properties from an established connection to a new connection structure. To do this, an application:

1. Allocates a connection structure (ct_con_alloc).
2. Customizes the connection (ct_con_props).
3. Opens the connection (ct_connect).
4. Calls ct_getloginfo to allocate a CS_LOGINFO structure and copy the connection‘s login properties into it.
5. Allocates a second connection structure (ct_con_alloc).
6. Calls ct_setloginfo to copy login properties from the CS_LOGINFO structure to the second connection structure. After copying the properties, ct_setloginfo deallocates the CS_LOGINFO structure.
7. Customizes any properties which should be different in the second connection (ct_con_props).
8. Opens the second connection (ct_connect).

Setting properties externally

The Client-Library routines ct_init and ct_connect optionally read a section from the Open Client and Open Server runtime configuration file to set property values for a context or connection. For a description of this feature, see “Using the runtime configuration file” on page 305.

Properties quick reference table

Table 2-29 lists Client-Library properties. The context properties in this table are set through ct_config. For a list of context properties set through cs_config, see the reference page for cs_config in the Open Client and Open Server *Common Libraries Reference Manual*.

Table 2-29: Client-Library properties

Property name	Meaning	Possible values	Applicable level	Notes
CS_ANSI_BINDS	Whether or not to use ANSI-style binds. See “ANSI-style binds” on page 202 for more information.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	
CS_APPNAME	At the context level, the name the application calls itself. At the connection level, the application name used when logging into the server. See “Application name” on page 202.	A character string. The default is NULL.	Connection. To set at the context level, call cs_config.	Login property. At connection level, cannot be set after connection is established.
CS_ASYNC_ NOTIFS	Whether or not a connection receives registered procedure notifications asynchronously. See “Asynchronous notifications” on page 203.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Must be set to CS_TRUE to receive notifications on an idle connection.
CS_BULK_LOGIN	Whether or not a connection is enabled to perform bulk-copy-in operations. See “Bulk copy operations” on page 205	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Login property. Cannot be set after connection is established.

Property name	Meaning	Possible values	Applicable level	Notes
CS_CHARSETCNV	Whether or not character set conversion is taking place. See “Character set conversion” on page 205.	CS_TRUE or CS_FALSE. A default is not applicable.	Connection.	Retrieve only, after connection is established.
CS_COMMBLOCK	A pointer to a communication sessions block. This property is specific to IBM-370 systems and is ignored by all other platforms. See “Communications session block” on page 206.	A pointer value. The default is NULL.	Connection.	Cannot be set after connection is established.
CS_CONNECTED_ ADDR	The transport address of the server to which the current connection is established.	A valid transport address.	Connection.	This property cannot be set. It requires a pointer to a CS_ TRANADDR structure that will be filled in with the server's address.
CS_CON_ KEEPALIVE	Whether for use the KEEPALIVE option.	CS_TRUE or CS_FALSE. The default is CS_TRUE	Context or connection	Some Net-Library protocol drivers do not support this property. See “Checking whether a property is supported” on page 182.
CS_CON_STATUS	The connection’s status. See “Connection status” on page 206.	A CS_INT-sized bitmask.	Connection.	Retrieve only.

Property name	Meaning	Possible values	Applicable level	Notes
CS_CON_TCP_ NODELAY	Whether to use the TCP_NODELAY property.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context or connection	Some Net-Library protocol drivers do not support this property. See "Checking whether a property is supported" on page 182.
CS_CONFIG_ BY_SERVERNAME	Whether ct_connect uses its *server_name* parameter or the value of the CS_APPNAME property as the section name to read external configuration data from. See "Using the runtime configuration file" on page 305.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Meaningful only when external configuration has been enabled by setting CS_EXTERNAL _CONFIG. Requires initialization with CS_VERSION_ 110 or later.
CS_CONFIG_FILE	The name and location of the Open Client/ Server runtime configuration file. See "Using the runtime configuration file" on page 305.	A character string. The default is NULL, which means a platform-specific default is used.	Connection.	Meaningful only when external configuration has been enabled by setting CS_EXTERNAL _CONFIG. Requires initialization with CS_VERSION_ 110 or later.
CS_CUR_ID	The cursor's identification number. See "Cursor ID" on page 207.	An integer value. A default is not applicable.	Command.	Retrieve only, after CS_CUR_ STATUS indicates an existing cursor.

Property name	Meaning	Possible values	Applicable level	Notes
CS_CUR_NAME	The cursor's name, as defined in an application's ct_cursor(CS_CURSOR_DECLARE) call. See "Cursor name" on page 207.	A character string. A default is not applicable.	Command.	Retrieve only, after ct_cursor (CS_CURSOR_DECLARE) returns CS_SUCCEED.
CS_CUR_ROWCOUNT	The current value of cursor rows. Cursor rows is the number of rows returned to Client-Library per internal fetch request. See "Cursor rowcount" on page 208.	An integer value. A default is not applicable.	Command.	Retrieve only, after CS_CUR_STATUS indicates an existing cursor.
CS_CUR_STATUS	The cursor's status. See "Cursor status" on page 208.	A CS_INT-sized value.	Command.	Retrieve only.
CS_DIAG_TIMEOUT	Whether Client-Library should fail or retry on timeout errors when inline error handling is in effect. See "Diagnostic timeout fail" on page 210.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means Client-Library should retry.	Connection.	
CS_DISABLE_POLL	Whether or not to disable polling. If polling is disabled, ct_poll does not report asynchronous operation completions. See "Disable poll" on page 210.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means that polling is not disabled.	Context, connection.	Useful in layered asynchronous applications.
CS_DS_COPY	Whether the directory service is allowed to satisfy an application's request with cached copies of directory entries. See "Directory service cache use" on page 116.	CS_TRUE or CS_FALSE. The default is CS_TRUE, which allows cache use.	Connection.	Not supported by all directory providers.

Property name	Meaning	Possible values	Applicable level	Notes
CS_DS_DITBASE	Fully qualified name of directory node where directory searches begin. See “Base for directory searches” on page 117.	A character string. The default is directory-provider specific.	Connection.	Not supported by all directory providers.
CS_DS_EXPAND ALIAS	Whether the directory service expands directory alias entries. See “Directory service expansion of aliases” on page 117.	CS_TRUE or CS_FALSE. The default is CS_TRUE, which allows alias expansion.	Connection.	Not supported by all directory providers.
CS_DS_FAILOVER	Whether to allow failover to the next *libtcl.cfg* entry or the *interfaces* file when a directory service driver cannot be initialized. See “Directory service failover” on page 118.	CS_TRUE or CS_FALSE The default is CS_TRUE.	Connection.	
CS_DS_PASSWORD	Password to go with the directory user ID specified as CS_DS_PRINCIPAL. See “Directory service password” on page 119.	A character string. The default is NULL.	Connection.	Not supported by all directory providers. The user name and password that are passed to the LDAP server for user authentication purposes are distinct and different from those used to access Adaptive Server.

Property name	Meaning	Possible values	Applicable level	Notes
CS_DS_PRINCIPAL	A directory user id for use of the directory service to go with the password specified as CS_DS_PASSWORD. See “Directory service principal name” on page 119.	A character string. The default is NULL.	Connection.	Not supported by all directory providers. The user name and password that are passed to the LDAP server for user authentication purposes are distinct and different from those used to access Adaptive Server.
CS_DS_PROVIDER	The name of the directory provider for the connection. See “Directory service provider” on page 120.	A character string. The default depends on directory driver configuration.	Connection.	
CS_DS_RAND_ OFFSET	Enables or disables random offset in connection lists. See “Directory service random offset” on page 120.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Determined when the network address list is retrieved from the directory service.
CS_DS_SEARCH	Restricts the depth of a directory search. See “Directory service search depth” on page 122.	A CS_INT sized symbolic value. For a list of possible values, see “Directory service search depth” on page 122.	Connection.	Not supported by all directory providers.
CS_DS_SIZELIMIT	Restricts the number of directory entries that are returned by a search started with ct_ds_lookup. See “Directory search size limit” on page 123.	A CS_INT value greater than or equal to 0. The default is 0, which indicates there is no size limit.	Connection.	Not supported by all directory providers.

Property name	Meaning	Possible values	Applicable level	Notes
CS_DS_TIMELIMIT	Sets an absolute time limit, in seconds, for completion of directory searches begun with ct_ds_lookup. See “Directory search time limit” on page 123.	A CS_INT value greater than or equal to 0. The default is 0, which indicates there is no time limit.	Connection.	Not supported by all directory providers.
CS_EED_CMD	A pointer to a command structure containing extended error data. See “Extended error data command structure” on page 211.	A pointer value. A default is not applicable.	Connection.	Retrieve only.
CS_ENDPOINT	The file descriptor for a connection. See “Endpoint polling” on page 211.	An integer value. A default is not applicable. Value is -1 on platforms that do not support endpoint handles.	Connection.	Retrieve only, after connection is established.
CS_EXPOSE_FMTS	Whether to expose results of type CS_ROWFMT_RESULT and CS_COMPUTEFMT_RESULT. See “Expose formats” on page 211.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established.
CS_EXTERNAL_CONFIG	Whether ct_init or ct_connect reads the Open Client and Open Server runtime configuration file to set properties and options for the connection to be opened. See “Using the runtime configuration file” on page 305.	CS_TRUE or CS_FALSE. The default is inherited from the CS-Library context property of the same name.	Context, connection.	Requires initialization with CS_VERSION_110 or later.

Property name	Meaning	Possible values	Applicable level	Notes
CS_EXTRA_INF	Whether to return the extra information that's required when processing Client-Library messages inline using a SQLCA, SQLCODE, or SQLSTATE. See "Extra information" on page 212.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	
CS_HAFAILOVER	See "High-availability failover" on page 139.	CS_TRUE or CS_FALSE.	Context, connection.	Requires initialization with CS_VERSION_120 or later.
CS_HAVE_BINDS	Whether any saved result bindings are present for the current result set. See "Have bindings" on page 213.	CS_TRUE or CS_FALSE. A default is not applicable.	Command.	Retrieve only.
CS_HAVE_CMD	Whether a resendable command exists for the command structure. See "Have resendable command" on page 213.	CS_TRUE or CS_FALSE.	Command.	Retrieve only.
CS_HAVE_CUROPEN	Whether a restorable cursor-open command exists for the command structure. See "Have restorable cursor-open command" on page 214.	CS_TRUE or CS_FALSE.	Command.	Retrieve only.
CS_HIDDEN_KEYS	Whether to expose hidden keys. See "Hidden keys" on page 214.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection, command.	Cannot be set at the command level if results are pending or a cursor is open.
CS_HOSTNAME	The host machine name. See "Host name" on page 215.	A character string. The default is NULL.	Connection.	Login property. Cannot be set after connection is established.

Property name	Meaning	Possible values	Applicable level	Notes
CS_IFILE	The path and name of the *interfaces* file. See “Location of the interfaces file” on page 215.	A character string.	Context.	
CS_LOC_PROP	A CS_LOCALE structure that defines localization information. See “Locale information” on page 216.	A CS_LOCALE structure. A connection picks up default localization information from its parent context.	Connection. To set CS_LOC_PROP at the context level, call cs_config.	Login property. Cannot be set after connection is established.
CS_LOGIN_STATUS	Whether the connection is open. See “Login status” on page 216.	CS_TRUE or CS_FALSE. A default is not applicable.	Connection.	Retrieve only.
CS_LOGIN_TIMEOUT	The login timeout value. See “Login timeout” on page 216.	An integer value. The default is 60 seconds. A value of CS_NO_LIMIT represents an infinite timeout period.	Context, connection.	
CS_LOOP_DELAY	The delay, in seconds, that ct_connect waits before retrying the sequence of addresses associated with a server name. See “Loop delay” on page 217.	A CS_INT >= 0. The default is 0.	Connection.	CS_RETRY_COUNT specifies the number of times to retry.
CS_MAX_CONNECT	The maximum number of connections for this context. See “Maximum number of connections” on page 218.	An integer value. The default is 25.	Context.	
CS_MEM_POOL	A memory pool that Client-Library will use to satisfy interrupt-level memory requirements. See “Memory pool” on page 218.	A pointer value. The default is NULL (no user-supplied memory pool).	Context.	Useful in asynchronous applications. Cannot be set or cleared when context has connections.

Property name	Meaning	Possible values	Applicable level	Notes
CS_NETIO	Whether network I/O is synchronous, fully asynchronous, or deferred-asynchronous. See “Network I/O” on page 219.	CS_SYNC_IO, CS_ASYNC_IO, or CS_DEFER_IO. The default is CS_SYNC_IO.	Context, connection.	Cannot be set for a context with open connections. CS_DEFER_IO is legal only at the context level. CS_ASYNC_IO cannot be used in an Open Server gateway.
CS_NO_TRUNCATE	Whether Client-Library should truncate or sequence messages that are longer than CS_MAX_MSG. See “No truncate” on page 221.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means that Client-Library truncates long messages.	Context.	
CS_NOAPI_CHK	Whether Client-Library performs argument and state checking when the application calls a Client-Library routine. See “No API checking” on page 221.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means that Client-Library performs API checking.	Context.	
CS_NOCHARSETCN_REQD	Whether the server performs character set conversion if the server’s character set is different from the client’s. See “No character conversion required” on page 222.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means conversion occurs when necessary.	Connection.	Cannot be set after connection is established.

Property name	Meaning	Possible values	Applicable level	Notes
CS_NOINTERRUPT	Whether the application can be interrupted by certain callback events. See “No interrupt” on page 222.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means the application can be interrupted.	Context.	
CS_NOTIF_CMD	A pointer to a command structure containing registered procedure notification parameters.	A pointer value. A default is not applicable.	Connection.	Retrieve only.
CS_PACKETSIZE	The TDS packet size in bytes. See “Packet size” on page 223.	The default is 512 bytes. A server that supports Server Specified Packetsize, for example, ASE 15.0, may choose any packet size between 512 and 65535 bytes. Unless CS_NO_SRVPKTSIZE is set, the packetsize cannot be larger than the value provided here.	Connection.	Negotiated login property. Cannot be set after connection is established.
CS_PARENT_HANDLE	The address of a command or connection structure’s parent structure. See “Parent structure” on page 223.	A pointer value.	Connection, command.	Retrieve only.

Property name	Meaning	Possible values	Applicable level	Notes
CS_PARTIAL_TEXT	Indicates whether or not the client application should perform a partial update. See “Partial updates to text and image data” on page 224.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	This property must be set before a connection to the server is established. If the server does not support partial updates, this property will be reset to CS_FALSE.
CS_PASSWORD	The password used to log in to the server. See “Password” on page 224.	A character string. The default is NULL.	Connection.	Login property.
CS_PROP_ APPLICATION_SPID	ASE SPID is saved during login and is available as the property. See “Extended failover” on page 211.	A CS_INT value corresponding to the server process ID (spid) on the server.	Connection.	Login property.
CS_PROP_EXTENDED FAILOVER	Enables or disables server-provided failover targets. See “Extended failover” on page 211.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context, connection.	Login property.
CS_PROP_ MIGRATABLE	Enables or disables connection migration. See “Connection migration” on page 207.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context, connection.	Login property.
CS_PROP_REDIRECT	Enables or disables login redirection support. See “Login redirection” on page 218.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context, connection.	Login property.
CS_PROP_SSL_ PROTOVERSION	The version of supported SSL/TLS protocols.	CS_INT	Context, connection	Must be one of the following values. CS_SSLVER_20 CS_SSLVER_30 CS_SSLVER_T LS1

Property name	Meaning	Possible values	Applicable level	Notes
CS_PROP_SSL_ CIPHER	Comma-separated list of CipherSuite names.	CS_CHAR	Context, connection	
CS_PROP_SSL_ LOCALID	Property used to specify the path to the Local ID (certificates) file.	Character string	Context connection	A structure containing a file name and a password used to decrypt the information in the file.
CS_PROP_SSL_CA	Specify the path to the file containing trusted CA certificates.	CS_CHAR	Context, connection	
CS_RETRY_COUNT	The number of times to retry a connection to a server's address. See "Retry count" on page 227.	A CS_INT >= 0. The default is 0.	Connection.	Affects only the establishment of a login dialog. Failed logins are not retried.
CS_SEC_ APPDEFINED	Whether the connection will use application-defined challenge/response security handshaking. See "Security handshaking: Challenge/Response" on page 273.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_CHALLENGE	Whether the connection will use Sybase-defined challenge/response security handshaking. See "Security handshaking: Challenge/Response" on page 273.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_CHANBIND	Whether the connection's security mechanism will perform channel binding. See "Requesting login authentication services" on page 256.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.

Property name	Meaning	Possible values	Applicable level	Notes
CS_SEC_ CONFIDENTIALITY	Whether data encryption service will be performed on the connection. See “Requesting per-packet security services” on page 260.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ CREDENTIALS	Used by gateway applications to forward a delegated user credential. See “Requesting login authentication services” on page 256.	A CS_VOID * pointer.	Context, connection.	Cannot be read. Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ CREDTIMEOUT	Whether the user’s credentials have expired. See “Requesting login authentication services” on page 256.	A CS_INT. See Table 2-32 on page 257 for possible values and their meanings.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ DATAORIGIN	Whether the connection’s security mechanism will perform data origin verification. See “Requesting per-packet security services” on page 260.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ DELEGATION	Whether to allow the server to connect to a second server with the user’s delegated credentials. See “Requesting login authentication services” on page 256.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.

Property name	Meaning	Possible values	Applicable level	Notes
CS_SEC_ DETECTREPLAY	Whether the connection's security mechanism will detect replayed transmissions. See "Requesting per-packet security services" on page 260.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ DETECTSEQ	Whether the connection's security mechanism will detect transmissions that arrive out of sequence. See "Requesting per-packet security services" on page 260.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ ENCRYPTION	Whether the connection will use encrypted password security handshaking. See "Security handshaking: encrypted password" on page 274.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_INTEGRITY	Whether the connection's security mechanism will perform data integrity checking. See "Requesting per-packet security services" on page 260.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_KEYTAB	The name and path to the file from which a connection's security mechanism reads the security key to go with the CS_USERNAME property. See "Requesting login authentication services" on page 256.	A character string. The default is NULL, which means the user must have established credentials before the application calls ct_connect.	Connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.

Property name	Meaning	Possible values	Applicable level	Notes
CS_SEC_ MECHANISM	The name of the network security mechanism that performs security services for the connection. See “Choosing a network security mechanism” on page 253.	A string value. The default depends on security driver configuration.	Context, connection.	Cannot be set after connection is established.
CS_SEC_ MUTUALAUTH	Whether the server is required to authenticate itself to the client. See “Requesting login authentication services” on page 256.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_NEGOTIATE	Whether or not the connection will use trusted-user security handshaking to pass security labels to the server.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_ NETWORKAUTH	Whether the connection’s security mechanism will perform network-based user authentication. See “Requesting login authentication services” on page 256.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism and a preestablished credential that matches CS_USERNAM E.
CS_SEC_ SERVERPRINCIPAL	The network security principal name for the server to which a connection will be opened. See “Requesting login authentication services” on page 256.	A string value. The default is NULL, which means that ct_connect assumes the server principal name is the same as its *server_name* parameter.	Connection.	Cannot be set after connection is established. Meaningful only for connections that use network-based user authentication.

Property name	Meaning	Possible values	Applicable level	Notes
CS_SEC_ SESSTIMEOUT	Whether the connection's security session has expired. See "Requesting login authentication services" on page 256.	A CS_INT. See Table 2-32 on page 257 for possible values and their meanings.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SERVERADDR	The address of the server to which you are connected to.	The format "hostname portnumber [filter], where filter is optional.	Connection	Using this property causes ctlib to bypass the host name of the server and the port number of the interfaces.
CS_SERVERNAME	The name of the server to which you are connected. See "Server name" on page 228.	A string value. A default is not applicable.	Connection.	Retrieve only, after connection is established.
CS_STICKY_BINDS	Whether or not bindings between result items and program variables persist when a server command is executed repeatedly. See "Persistent result bindings" on page 224.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Command.	
CS_TDS_VERSION	The version of the TDS protocol that the connection is using. See "TDS version" on page 228.	A symbolic version level. Defaults to a value based on CS_VERSION.	Connection.	Negotiated login property. Cannot be set after connection is established.
CS_TEXTLIMIT	The largest text or image value to be returned on this connection. See "Text and image limit" on page 229.	An integer value. The default is CS_NO_LIMIT.	Context, connection.	
CS_TIMEOUT	The timeout value for reading results from the server. See "Timeout" on page 230.	An integer value. The default is CS_NO_LIMIT.	Context, connection.	

Property name	Meaning	Possible values	Applicable level	Notes
CS_TRANSACTION_NAME	A transaction name to be used over a connection to Open Server for CICS. See “Transaction name” on page 233.	A string value. The default is NULL.	Connection.	
CS_USER_ALLOC	A user-defined memory allocation routine. See “User allocation function” on page 233.	A pointer to a user-defined function. A default is not applicable.	Context.	Useful in asynchronous application.
CS_USER_FREE	A user-defined memory free routine. See “User free function” on page 235.	A pointer to a user-defined function. A default is not applicable.	Context.	Useful in asynchronous application.
CS_USERDATA	User-allocated data. See “User data” on page 235.	User-allocated data.	Connection, command. To set CS_USER DATA at the context level, call cs_config.	
CS_USERNAME	The name used to log in to the server. See “User name” on page 237.	A character string. The default is NULL.	Connection.	Login property. Cannot be set after connection is established.
CS_VER_STRING	Client-Library’s true version string. See “Version string for Client-Library” on page 237.	A character string. A default is not applicable.	Context.	Retrieve only.
CS_VERSION	The version of Client-Library in use by this context. See “Version of Client-Library” on page 237.	A symbolic version level. CS_VERSION gets its value from a context’s ct_init call. See the detailed description for possible values.	Context.	Retrieve only.

About the properties

This section provides a detailed description of each Client-Library property.

ANSI-style binds

CS_ANSI_BINDS determines whether or not Client-Library will use ANSI-style binds and ANSI-style cursor end-data processing.

When ANSI-style binds are in effect, ct_fetch raises an error in the following situations:

- It is considered an error to bind only some of the items in a result set. An application must bind either none of the items or all of the items.
- ct_fetch raises an error when copying a NULL or truncated character string value into a variable which does not have an associated indicator.

In both of these cases, ct_fetch returns CS_ROW_FAIL.

When ANSI-style cursor end-data processing is in effect, ct_fetch does not raise an error when cursor results are being processed, ct_fetch has returned CS_END_DATA, and the following calls are made:

- ct_bind
- ct_fetch

If the CS_ANSI_BINDS property is not CS_TRUE, ct_fetch raises an error and fails when these calls are made in this situation.

Application name

CS_APPNAME specifies an application name, which is used as follows:

- At the context level, CS_APPNAME specifies a configuration file section from which ct_init reads default Client-Library context properties. See “Using the runtime configuration file” on page 305 for a description of this feature. CS_APPNAME is set at the context level by calling the CS-Library routine cs_config.

- At the connection level, CS_APPNAME defines the application name that a connection will use when connecting to a server. If external configuration is enabled for the connection, CS_APPNAME may also identify a configuration file section from which ct_connect reads default properties, server options, and capabilities for the connection. See "Using the runtime configuration file" on page 305 for a description of this feature.

Adaptive Server uses application names to identify connection processes in the *sysprocesses* table of the *master* database.

When a connection structure is allocated, it inherits the CS_APPNAME setting from the parent context structure. If the inherited value is not changed, it becomes the application name when the connection is opened. Applications change the application name for a connection by calling ct_con_props before the connection is opened.

Asynchronous notifications

The CS_ASYNC_NOTIFS connection property controls how a Client-Library application receives registered procedure notifications from an Open Server application. CS_ASYNC_NOTIFS determines whether a connection will receive registered procedure notifications asynchronously.

The Open Server application sends a notification to the client as one or more TDS packets. The client application does not learn of the notification until Client-Library reads the notification packets from the connection and invokes the application's notification callback.

Registered procedure notifications allow clients to watch for execution of one or more registered procedures on an Open Server. When a watched procedure is executed by any client, Open Server sends a notification to each client that is watching that particular registered procedure.

The server sends the notification as one or more Tabular Data Stream packets. For the application to learn about notifications, Client-Library must read these packets and trigger the application's notification callback. The CS_ASYNC_NOTIFS property determines how the application learns about notifications:

- An otherwise synchronous connection receives asynchronous notifications by setting CS_ASYNC_NOTIFS to CS_TRUE.
- An asynchronous connection does not receive notifications asynchronously unless it sets CS_ASYNC_NOTIFS to CS_TRUE.

- On a connection that is used only to receive notifications, ct_poll does not look for notifications unless CS_ASYNC_NOTIFS is CS_TRUE.

When CS_ASYNC_NOTIFS is CS_TRUE

When CS_ASYNC_NOTIFS is set to CS_TRUE, Client-Library interrupts the application to report an arriving registered procedure notification.

On platforms that support interrupt- or thread-driven I/O, Client-Library automatically reads the notification information and invokes the connection's notification callback when a notification arrives on the connection.

On other platforms, if the connection is not otherwise active, it must be polled with ct_poll to trigger the notification callback. CS_ASYNC_NOTIFS must be CS_TRUE for ct_poll to trigger the notification callback on an otherwise idle connection.

When CS_ASYNC_NOTIFS is CS_FALSE

When CS_ASYNC_NOTIFS is CS_FALSE (the default), the application must be reading from the network for Client-Library to report a registered procedure notification. When the server sends a notification, Client-Library reads the notification data and triggers the application's notification callback the next time it interacts with the server.

Likewise, if CS_ASYNC_NOTIFS is CS_FALSE, ct_poll does not read notification data from the network and trigger the application's notification callback. This means that an application must be reading results for ct_poll to report a registered procedure notification. When ct_poll reports the notification, the application's notification callback is automatically called.

Note If a connection is used only for receiving registered procedure notifications, CS_ASYNC_NOTIFS must be set to CS_TRUE to receive the notification. Asynchronous notifications must be enabled even if the connection is polled with ct_poll.

Setting CS_ASYNC_NOTIFS

The following fragment enables asynchronous notifications.

```
/* Turn on read-ahead notifications. */
 boolval = CS_TRUE;
 if (ct_con_props(conn, CS_SET, CS_ASYNC_NOTIFS, &boolval,
                  CS_UNUSED, (CS_INT *)NULL)! CS_SUCCEED)
```

```
{
  fprintf(stderr,
     "Error: ct_con_props(SET, CS_ASYNC_NOTIFS) failed\n");
  (CS_VOID)ct_close(conn, CS_UNUSED);
  (CS_VOID)ct_con_drop(conn);
}
```

Setting CS_ASYNC_NOTIFS to CS_FALSE does not immediately turn off asynchronous notifications. To turn off asynchronous notifications, an application must send a command to the server after setting CS_ASYNC_NOTIFS to CS_FALSE.

CS_ASYNC_NOTIFS is the only property that determines whether notifications are received asynchronously:

- An otherwise synchronous connection receives asynchronous notifications.
- An asynchronous connection does not receive notifications asynchronously unless it sets CS_ASYNC_NOTIFS to CS_TRUE.

For information about registered procedure notifications, see “Registered procedures” on page 238.

Bulk copy operations

CS_BULK_LOGIN describes whether or not a connection can perform bulk copy operations into a database. The default of CS_FALSE prohibits bulk copy operations.

Applications that perform bulk copy operations on a connection must set the CS_BULK_LOGIN connection property to CS_TRUE before calling ct_connect to open the connection.

Applications that allow users to make ad hoc queries may want to avoid setting this property to CS_TRUE, to keep users from initiating a bulk copy sequence through SQL commands. Once a bulk copy sequence has been started, it cannot be stopped with an ordinary SQL command. Applications perform bulk copy operations using Bulk-Library calls. Bulk-Library is described in the Open Client and Open Server *Common Libraries Reference Manual*.

Character set conversion

CS_CHARSETCNV describes whether or not the server is converting between the client and server character sets. This property is retrieve-only, after a connection is established.

A value of CS_TRUE indicates that the server is converting between the client and server character sets; CS_FALSE indicates that no conversion is taking place.

Communications session block

The CS_COMMBLOCK property defines a pointer to a communications block. This property is specific to IBM370 systems and is ignored by all other platforms.

Connection status

CS_CON_STATUS is a CS_INT-sized bitmask that reflects a connection's current status.

The following table lists the symbolic values that make up CS_CON_STATUS:

Symbolic value	Meaning
CS_CONSTAT_CONNECTED	The connection is open.
CS_CONSTAT_DEAD	The connection has been marked as "dead."

Client-Library marks a connection as dead if errors have made it unusable or if an application's client message callback routine returns CS_FAIL. An application must call ct_close and ct_con_drop to close and drop connections that have been marked as "dead," or unusable. An exception to this rule occurs for certain types of results-processing errors. If a connection is marked dead while processing results, the application can try reviving the connection by calling ct_cancel with *type* as CS_CANCEL_ALL or CS_CANCEL_ATTN. If this fails, the application must close and drop the connection.

Configure by server name

CS_CONFIG_BY_SERVERNAME determines whether ct_connect uses its *server_name* parameter or the value of the CS_APPNAME property as the section name to read external configuration data from. For a description this feature, see "Using the runtime configuration file" on page 305.

Configuration file name

CS_CONFIG_FILE specifies the name and location of the Open Client and Open Server runtime configuration file that Client-Library reads to set default values for properties, server options, and capabilities. For a description this feature, see "Using the runtime configuration file" on page 305.

Connection migration

When CS_PROP_MIGRATABLE is CS_TRUE (the default), a connection can be migrated by a server that understands the connection migration protocol and can move a client connection to another server after login has completed.

The CS_PROP_MIGRATABLE property can be set using ct_config and ct_con_props.

Cursor ID

CS_CUR_ID is the server identification number assigned to a cursor.

An application retrieves a cursor's identification number after calling ct_cmd_props(CS_CUR_STATUS) to confirm that a cursor exists in the command space of interest.

CS_CUR_ID is a command structure property and cannot be retrieved at the connection or context levels.

Cursor properties are useful to gateway applications that send cursor information to clients.

For an example fragment that retrieves the cursor ID, see "Example for Cursor Status" on page 320.

Cursor name

CS_CUR_NAME is the name with which a cursor was declared. An application declares a cursor by calling ct_cursor(CS_CURSOR_DECLARE).

An application retrieves a cursor's name any time after its ct_cursor(CS_CURSOR_DECLARE) call returns CS_SUCCEED.

CS_CUR_NAME is a command structure property and cannot be retrieved at the connection or context levels.

Cursor properties are useful to gateway applications that send cursor information to clients.

For an example fragment that retrieves the cursor name, see “Example for Cursor Status” on page 320.

Cursor rowcount

CS_CUR_ROWCOUNT is the current value of cursor rows for a cursor.

Cursor rows is the number of rows returned to Client-Library for each internal fetch request. This is not the number of rows returned to an application for each ct_fetch call. (The latter number is specified by the bindings in place on the command structure. For details, see “Array binding” on page 334.

Cursor rows defaults to one. This implies that for every ct_fetch call made by the application, Client-Library issues one internal cursor fetch command for every row required by the ct_fetch call.

Each internal cursor fetch command requires interaction between the client and the server. Therefore, a larger cursor rows value reduces the number of network round-trips required to fetch from the cursor. However, if the application sends nested cursor commands or sends commands on a different command structure while fetching from the cursor, Client-Library must buffer rows that have not been fetched with ct_fetch to send the new command. Therefore, larger cursor rows values may require increased memory usage by Client-Library.

The application calls ct_cursor to increase the value of cursor rows before a cursor is opened. For details, see “Cursor-Rows commands” on page 432.

An application retrieves CS_CUR_ROWCOUNT after calling ct_cmd_props(CS_CUR_STATUS) to confirm that a cursor exists in the command space of interest.

CS_CUR_ROWCOUNT is a command structure property and cannot be retrieved at the connection or context levels.

Cursor properties are useful to gateway applications that send cursor information to clients.

Cursor status

CS_CUR_STATUS is a CS_INT-sized quantity that reflects a cursor’s current status.

The status may be either CS_CURSTAT_NONE to indicate no cursor exists for the command structure or a status value with bits set to indicate the status.

If CS_CURSTATUS is not CS_CURSTAT_NONE, the application determines the cursor status by applying the bitmask values listed in the table below. For example, to see if a cursor is updatable, apply the following test:

```
if ((cur_status & CS_CURSTAT_UPDATABLE)
     == CS_CURSTAT_UPDATABLE)
```

Table 2-30 lists the symbolic bitmask values for testing a CS_CUR_STATUS value:

Table 2-30: Cursor status bitmask values

Bitmask value	Tests for
CS_CURSTAT_CLOSED	A closed cursor exists in the command space. An application can open or deallocate a closed cursor.
CS_CURSTAT_DECLARED	A cursor is currently declared in this command space. An application can open or deallocate a declared cursor.
CS_CURSTAT_ROWCOUNT	The application has sent a cursor-rows command to the server, but the cursor has not been opened yet.
CS_CURSTAT_OPEN	An open cursor is open in the command space. An application can close an open cursor.
CS_CURSTAT_RDONLY	The cursor is read-only and cannot be used to perform updates.
CS_CURSTAT_UPDATABLE	The cursor can be used to perform updates.
CS_SCROLL_INSENSITIVE	Declares a scrollable, insensitive cursor.
CS_SCROLL_SEMISENSITIVE	Declares a scrollable, semi sensitive cursor.
CS_SCROLL_CURSOR	Declares a scrollable cursor that is insensitive (default).
CS_NOSCROLL_INSENSITIVE	Declares a cursor insensitive and non-scrollable.

The cursor status is reported by the server. An application must send a ct_cursor command and begin processing the results before it will see a change to the CS_CUR_STATUS property value. Cursor status is guaranteed to be accurate:

- After ct_results returns CS_SUCCEED with a **result_type* parameter of CS_CMD_SUCCEED, CS_CMD_FAIL, or CS_CURSOR_RESULT
- After ct_cancel(CS_CANCEL_ALL) returns CS_SUCCEED
- After any Client-Library or CS-Library routine returns CS_CANCELED

Calling ct_cancel may cause a connection's cursors to enter an undefined state. An application uses the cursor status property to determine how a cancel operation has affected a cursor.

CS_CUR_STATUS is a command structure property and cannot be retrieved at the connection or context levels.

Cursor properties are useful to gateway applications that send cursor information to clients.

For an example fragment that retrieves this property and checks the cursor status, see Example for Cursor Status on page 320.

Diagnostic timeout fail

When inline error handling is in effect, the CS_DIAG_TIMEOUT property determines whether Client-Library fails or retries on Client-Library timeout errors.

If CS_DIAG_TIMEOUT is CS_TRUE, Client-Library marks a connection as dead when a Client-Library routine generates a timeout error.

If CS_DIAG_TIMEOUT is CS_FALSE, Client-Library retries indefinitely when a Client-Library routine generates a timeout error.

Disable poll

The CS_DISABLE_POLL property determines whether or not ct_poll reports asynchronous operation completions.

Layered asynchronous applications use CS_DISABLE_POLL to prevent ct_poll from reporting low-level asynchronous completions.

An application cannot call ct_wakeup if the CS_DISABLE_POLL property is set to CS_TRUE.

For more information about CS_DISABLE_POLL, see "Layered applications" on page 19.

Directory service properties

See "Properties for directory services" on page 116 for detailed descriptions of properties related to directory services.

Extended error data command structure

The CS_EED_CMD property defines a pointer to a CS_COMMAND structure containing extended error data.

Within a server message callback, Client-Library indicates that extended error data is available by setting the CS_HASEED bit of the *status* field of the CS_SERVERMSG structure describing the message.

It is an error to retrieve CS_EED_CMD if no extended error data is available.

See “Extended error data” on page 129.

Extended failover

CS_PROP_EXTENDEDFAILOVER is set to TRUE by default, and is only used if CS_HAFAILOVER is also TRUE. If so, HA Aware sends a list of network addresses to be used in place of the information initially retrieved from the directory service. If CS_PROP_EXTENDED FAILOVER is set to FALSE, the failover information is obtained from the directory service layer.

Endpoint polling

CS_ENDPOINT allows an application to get a file descriptor, the number associated with a connection to a remote server. This may be useful to a gateway application that contains both Client-Library and Server-Library calls: After establishing a connection to a remote server with Client-Library, the file descriptor associated with that connection is used by the srv_poll Server-Library routine. A call to srv_poll causes the current thread to be rescheduled until there are results available on the connection.

Use of the CS_ENDPOINT property is discouraged, since it is currently specific only to UNIX platforms.

Expose formats

CS_EXPOSE_FMTS determines whether or not Client-Library exposes format result sets.

A format result set contains format information for the result set with which it is associated. Format information includes the number of items in the result set and a description of each item. There are two types of format result sets:

- CS_ROWFMT_RESULT – contains format information for a regular row result set.

- CS_COMPUTEFMT_RESULT – contains format information for a compute row result set.

All format result sets generated by a command precede the regular row and compute row result sets generated by the command.

If format result sets are not exposed, an application only retrieves format information while it is processing a result set. For example, after ct_results returns with a *result_type* of CS_ROW_RESULT, the application calls ct_res_info to determine the number of columns in the result set, ct_describe to get a description of each column, and so on.

Exposing format result sets allows an application to retrieve format information before processing a result set.

Exposing format result sets is useful in gateway applications that need to repackage Adaptive Server results before sending them on to a foreign client.

An application exposes format result sets by setting the CS_EXPOSE_FMTS property to CS_TRUE.

For more information about format results, see “Format results” on page 244.

External configuration

CS_EXTERNAL_CONFIG specifies whether Client-Library configures default property, server option, and capability values by reading a configuration file. For a description of this feature, see “Using the runtime configuration file” on page 305.

Extra information

CS_EXTRA_INF determines whether or not Client-Library returns the extra information that ct_diag requires to fill in a SQLCA, SQLCODE, or SQLSTATE structure.

This extra information includes the number of rows affected by the most recent command. Applications also retrieve this information by calling ct_res_info(CS_ROW_COUNT).

If an application is not retrieving messages into a SQLCA, SQLCODE, or SQLSTATE, the extra information is returned as ordinary Client-Library messages.

Have bindings

CS_HAVE_BINDS tells whether any saved result bindings are present for the current result set. This property is retrieved with ct_cmd_props.

CS_HAVE_BINDS is always used with the CS_STICKY_BINDS property. Some batch-processing applications that repeatedly execute the same command on a CS_COMMAND structure may set the CS_STICKY_BINDS command property so that Client-Library saves result bindings in between executions of the same command. These applications check the CS_HAVE_BINDS property to see whether saved bindings are in place for the current result set. A value of CS_ TRUE indicates that one or more program variables are bound to one or more items in the current result set.

See “Persistent result bindings” on page 224 for a description of the CS_STICKY_BINDS property.

CS_HAVE_BINDS is guaranteed to be accurate after ct_results indicates the presence of fetchable data on a command structure.

Have resendable command

CS_HAVE_CMD determines whether an application can resend a previously executed server command. The property is read-only, and CS_TRUE indicates the presence of a resendable command.

Client-Library allows applications to resend some types of commands immediately after the results of the previous execution have been processed.

To resend a command, the application:

1. Updates values in the command’s parameter source variables (if any). The address of the parameter source variables must have been specified with ct_setparam after the command was initiated with ct_command, ct_cursors, or ct_dynamic.
2. Calls ct_send to resend the command. ct_send reads the updated parameter values.

Not all command types can be resent. See “Resending commands” on page 584 for more information.

Applications that resend commands may benefit from setting the CS_STICKY_BINDS property to reuse the bindings that were established while processing the results from the original execution of the command. See “Persistent result bindings” on page 224 for a description of this property.

Have restorable cursor-open command

CS_HAVE_CUROPEN determines whether an application may restore a previously executed ct_cursor cursor-open command batch. The property is read-only, and CS_TRUE indicates the presence of a restorable cursor-open command batch.

An application restores a cursor-open command by calling ct_cursor. See “Restoring a cursor-open command” on page 435 for an explanation of this feature.

An open cursor must be closed before the cursor-open command can be restored. CS_HAVE_CUROPEN indicates that Client-Library saved the command information for the original cursor-open command. It does not indicate that the application can legally reopen the cursor while the cursor is in its current state.

The CS_CUR_STATUS property tells an application the current state (if any) of the cursor declared on a command structure. See “Cursor status” on page 208 for a description of this property.

Applications that restore cursor-open commands may benefit from setting the CS_STICKY_BINDS property to reuse the bindings that were established while processing the results from the original execution of the command. See “Persistent result bindings” on page 224 for a description of this property.

Hidden keys

CS_HIDDEN_KEYS determines whether or not Client-Library exposes any “hidden keys” that are part of a result set. Hidden keys are columns that are not explicitly selected in a query, but which are returned to a client because they make up part or all of a table’s key.

Ordinarily, the presence of these columns is suppressed. The client is not aware that they are a part of the result set.

A client exposes hidden keys by setting the CS_HIDDEN_KEYS property to CS_TRUE.

Once hidden keys are exposed, they are returned as ordinary columns. If an application calls ct_res_info to retrieve the number of columns in a result set, for example, the number will include exposed columns. An application binds and fetches the row values of exposed columns.

If a column is an exposed hidden key, ct_describe includes CS_HIDDEN in the *status* field bitmask describing the column.

An application uses ct_keydata with a table's keys to change a cursor's position. For information about how to do this, see ct_keydata on page 521.

An application cannot set the CS_HIDDEN_KEYS property at the command level if results are pending or a cursor is open.

Host name

CS_HOSTNAME is the name of the host machine, used when logging into a server.

Adaptive Server lists a process's host name in the *sysprocesses* table of the *master* database.

Location of the interfaces file

CS_IFILE defines the name and location of the *interfaces* file.

The *interfaces* file contains the name and network address of every server available on the network. It establishes communication between clients and servers. For every server to which a client might connect, the *interfaces* file contains an entry which includes the server name, the machine name, and the address of that server. For Client-Library applications, the *interfaces* file is searched during every call to ct_connect.

On most platforms, if a particular *interfaces* file has not been specified through ct_config, ct_connect attempts to use a file named *interfaces* in the directory named by the SYBASE environment variable or logical name (Windows platforms use the *sql.ini* file). If SYBASE has not been set, ct_connect attempts to use a file named *interfaces* in the home directory of the user named "sybase."

See "Interfaces file" on page 142.

Note Not all platforms use an *interfaces* file. If you do not know whether your platform uses an *interfaces* file, consult your System Administrator or see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

An alternate default file name and path for the *interfaces* file can be specified by the CS_DEFAULT_IFILE property. For detailed information about the CS_DEFAULT_IFILE property, see the Open Client and Open Server *Common Libraries Reference Manual*.

Locale information

CS_LOC_PROP defines a CS_LOCALE structure that contains localization values. Localization values include a language, a character set, datetime formats, and a collating sequence.

An application calls ct_con_props to set or retrieve CS_LOC_PROP at the connection level.

- When setting CS_LOC_PROP, an application passes ct_con_props a CS_LOCALE structure. ct_con_props copies information from the CS_LOCALE and stores it internally. After calling ct_con_props, the application deallocates the CS_LOCALE.
- When retrieving CS_LOC_PROP, an application passes ct_con_props a CS_LOCALE structure. ct_con_props copies current localization information into this CS_LOCALE.

An application calls cs_loc_alloc to allocate a CS_LOCALE structure.

An application calls cs_config to set or retrieve CS_LOC_PROP at the context level.

If an application does not call cs_config to define localization information for a context, the context uses default localization values that are assigned at allocation time. On most platforms, environment variables determine the default values. For specific information about how default localization values are assigned on your platform, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

Login status

CS_LOGIN_STATUS is CS_TRUE if a connection is open, CS_FALSE if it is not. This property can only be retrieved.

ct_connect is used to open a connection.

ct_close is used to close a connection.

Login timeout

CS_LOGIN_TIMEOUT defines the length of time, in seconds, that Client-Library waits for a login response when making a connection attempt. A Client-Library application makes a connection attempt by calling ct_connect.

This timeout specifies the allowable round-trip delay between a client request and the receipt of the server response. Multiple round trips between the client and the server may occur before ct_connect returns. CS_LOGIN_TIMEOUT applies to each round trip.

The default timeout value is 60 seconds. A timeout value of CS_NO_LIMIT represents an infinite timeout period.

Note CS_LOGIN_TIMEOUT applies only to synchronous connections.

You can specify CS_LOGIN_TIMEOUT values on an individual connection basis with ct_con_props.

See “Handling timeout errors” on page 230 for more information.

Loop delay

CS_LOOP_DELAY specifies the delay, in seconds, that ct_connect waits before retrying the sequence of network addresses associated with a server name. The default is 0.

The CS_RETRY_COUNT property specifies how many times Client-Library retries each address in the sequence. See “Retry count” on page 227 for more information.

CS_LOOP_DELAY and CS_RETRY_COUNT affect only the establishment of a login dialog. Once Client-Library has found an address where a server responds, the login dialog between Client-Library and the server begins. Client-Library does not retry any other addresses if the login attempt fails.

Addresses are associated with server names either in a network-based directory or the Sybase *interfaces* file. For more information, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

On UNIX platforms, a server’s *interfaces* file entry can be configured to override application-specified settings for CS_RETRY_COUNT and CS_LOOP_DELAY. For more information, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

Login redirection

CS_PROP_REDIRECT is set to TRUE by default, regardless of the library version in operation. When CS_PROP_REDIRECT is set to TRUE, ct_connect initiates a server login attempt with login redirection enabled. When CS_PROP_REDIRECT is set to FALSE, ct_connect initiates a server login attempt with login redirection disabled.

Should a login redirection occur, the amount of time required to log in may increase as additional data is sent to clients; also, redirected clients may need to restart the login process.

Maximum number of connections

CS_MAX_CONNECT defines the maximum number of simultaneously open connections that a context may have. CS_MAX_CONNECT has a default value of 25. Negative and zero values are not allowed for CS_MAX_CONNECT.

If ct_config is called to set a value for CS_MAX_CONNECT that is less than the number of currently open connections, ct_config raises a Client-Library error and returns CS_FAIL without altering the value of CS_MAX_CONNECT.

Memory pool

CS_MEM_POOL identifies a pool of memory that Client-Library uses to satisfy its memory requirements.

Ordinarily, Client-Library routines satisfy their memory requirements by calling malloc. However, not all implementations of malloc are reentrant, so it is not safe to use malloc in Client-Library routines that are called at the system interrupt level. For this reason, on systems where Client-Library uses signal-driven network I/O, such as UNIX systems, fully asynchronous applications are required to provide an alternate way for Client-Library to satisfy its memory needs. This is not a requirement on platforms that use thread-driven network I/O or for applications that do not use fully asynchronous connections. For a description of the network I/O method used on your platform, see the Open Client and Open Server *Programmers Supplement* for Microsoft Windows and the Open Client and Open Server *Programmers Supplement* for UNIX.

Client-Library provides two mechanisms by which an asynchronous application satisfy Client-Library's memory needs:

- The application uses the CS_MEM_POOL property to provide Client-Library with a memory pool.
- The application uses the CS_USER_ALLOC and CS_USER_FREE properties to install memory allocation routines that Client-Library safely calls at the operating system interrupt level.

If a fully asynchronous application fails to provide Client-Library with a safe way to satisfy memory needs, Client-Library's behavior is undefined.

ct_config returns CS_FAIL if an application attempts to set a memory pool that does not meet Client-Library's minimum pool size requirements.

On UNIX systems, a memory pool should include approximately 6K for each connection.

Client-Library attempts to satisfy memory requirements from the following sources, in the following order:

1 Memory pool

2 User-supplied allocation and free routines

3 System routines

If a connection cannot get the memory it needs, Client-Library marks the connection dead.

An application is responsible for allocating and freeing the memory identified by CS_MEM_POOL.

An application can replace a memory pool by calling ct_config with action as CS_SET and *buffer* as the address of the new pool.

An application clears a memory pool in two ways:

- By calling ct_config with action as CS_SET and *buffer* as NULL
- By calling ct_config with action as CS_CLEAR

An application cannot set or clear a memory pool for a context in which CS_CONNECTION structures currently exist. A context must drop all CS_CONNECTION structures before clearing a memory pool.

Network I/O

CS_NETIO determines whether a connection is synchronous, fully asynchronous, or deferred-asynchronous:

- On a synchronous connection, a routine that requires a server response blocks until the response is received.
- On a fully asynchronous connection, a routine that requires a server response returns CS_PENDING immediately. When the response arrives and the routine completes its work, Client-Library automatically calls the connection's completion callback.

  Depending on the host platform, the completion callback is invoked either at the system interrupt level (on platforms that use signal-driven network I/O) or from a Client-Library runtime thread (on platforms that use thread-driven network I/O). For a description of the network I/O method used for your platform, see the Open Client and Open Server *Programmers Supplement* for Microsoft Windows and the Open Client and Open Server *Programmers Supplement* for UNIX.
- On a deferred-asynchronous connection, a routine that requires a server response returns CS_PENDING immediately. The connection must call ct_poll to find out if the routine has completed. If the application has installed a completion callback and a routine has completed, ct_poll invokes the completion callback before returning.

On platforms that do not support multithreading or signal-driven network I/O, such as Microsoft Windows 98, connections can only be synchronous or deferred-asynchronous. Even if the CS_NETIO property is set to CS_ASYNC_IO, the connection is deferred-asynchronous, and the application must poll for completions with ct_poll.

Warning! In an Open Sever gateway application, the CS_NETIO property cannot be set to CS_ASYNC_IO. The Open Server thread scheduler provides multitasking in an Open Server application.

An application can set up deferred asynchronous connections only at the context level, by calling ct_config with **buffer* as CS_DEFER_IO. CS_DEFER_IO is not a legal value at the connection level.

Asynchronous connections use the type of asynchronous I/O that matches their parent context. For example, suppose an application sets up deferred-asynchronous connections at the context level and then creates a synchronous connection within the context. If the application later calls ct_con_props with **buffer* as CS_ASYNC_IO to make this connection asynchronous, the connection will be deferred-asynchronous, not fully asynchronous.

A context can include both synchronous and asynchronous connections, but the asynchronous connections within a context must all be fully asynchronous or must all be deferred-asynchronous.

The following restrictions apply to an application's use of CS_NETIO:

- An application cannot set CS_NETIO for a context if the context has open connections.
- An application cannot set CS_NETIO for a connection if the connection has any active commands or pending results.

For more information about asynchronous Client-Library programming, see "Asynchronous programming" on page 12.

No truncate

CS_NO_TRUNCATE determines whether Client-Library truncates or sequences Client-Library and server messages that are longer than CS_MAX_MSG - 1 bytes.

Client-Library's default behavior is to truncate messages that are longer than CS_MAX_MSG - 1 bytes. When Client-Library is sequencing messages, however, it uses as many CS_CLIENTMSG or CS_SERVERMSG structures as necessary to return the full text of a message. The message's first CS_MAX_MSG bytes are returned in one structure, its second CS_MAX_MSG bytes in a second structure, and so forth.

Client-Library null terminates only the last chunk of a message. If a message is exactly CS_MAX_MSG bytes long, the message is returned in two chunks: the first containing CS_MAX_MSG bytes of the message and the second containing a null terminator.

For more information about sequenced messages, see "Sequencing long messages" on page 127.

No API checking

CS_NOAPI_CHK determines whether Client-Library performs argument and state checking when the application calls a Client-Library routine.

When CS_NOAPI_CHK is CS_FALSE (the default value), Client-Library performs checking. With this setting, Client-Library performs the following error checking each time you call a Client-Library routine:

- Validates parameter values

- Checks field values in visible structures for illegal combinations
- Verifies that the application is in a correct state for execution of that function

If a problem is discovered, the routine fails and an error message is generated.

When CS_NOAPI_CHK is CS_TRUE, Client-Library's usual checking is disabled. The effect of this is as follows:

- If the application passes an invalid argument or calls a routine at the wrong time, the application experiences memory corruption, memory access violations, or incorrect results.
- With API checking disabled, Client-Library does not check for usage errors. Some usage errors are not trapped with API checking disabled. With API checking enabled, these errors generate error messages; with API checking disabled, they cause incorrect application behavior.

Warning! Do not disable API checking until after you have completely debugged the application.

No character conversion required

CS_NOCHARSETCNV_REQD determines whether the server converts character data to and from its own character set.

When CS_NOCHARSETCNV_REQD is CS_FALSE (the default), and the connection's character set does not match the server's, the server will convert characters to and from its character set when communicating with the client.

When CS_NOCHARSETCNV_REQD is set to CS_TRUE, the server does not perform character set conversion, regardless of the connection's character set.This is useful when the server will be passing data to another server without interpreting it, for example, when the server is a Open Server gateway.

CS_NOCHARSETCNV_REQD cannot be set after a connection is open.

The connection's character set is defined within the connection's CS_LOCALE structure. See "Locale information" on page 216.

No interrupt

CS_NOINTERRUPT determines whether an application can be interrupted by Client-Library completion event.

When CS_NOINTERRUPT is CS_TRUE, completion events are deferred until CS_NOINTERRUPT is reset to CS_FALSE.

An application uses the CS_NOINTERRUPT property to protect critical sections of code.

Note Client-Library's CS_NOINTERRUPT property has no effect on operating system interrupt handling. CS_NOINTERRUPT affects completion events only, not notification events.

Packet size

CS_PACKETSIZE determines the packet size that Client-Library uses when sending Tabular Data Stream (TDS) packets.

If an application needs to send or receive large amounts of data, a larger packet size may improve efficiency.

On Open Client Version 15.0, two capabilities are included: CS_REQ_SRVPKTSIZE and CS_NO_SRVPKTSIZE.

- CS_REQ_SRVPKTSIZE is always set by this version of CT-Library and can be retrieved with ct_capability.
- CS_NO_SRVPKTSIZE is used when clients cannot work with a packet size larger than that requested, and is set and retrieved with ct_capability.

Parent structure

CS_PARENT_HANDLE defines a pointer to a command or connection structure's parent structure.

- If retrieved at the command structure level, CS_PARENT_HANDLE is a pointer to the command structure's parent connection structure.
- If retrieved at the connection structure level, CS_PARENT_HANDLE is a pointer to the connection structure's parent context structure.

Partial updates to *text* and *image* data

Open Client supports the partial update of text and image columns. CS_PARTIAL_TEXT indicates whether or not the client needs to perform a partial update. You can set this property in the connection or context level using ct_con_props() or ct_config(), respectively. The possible values of CS_PARTIAL_TEXT are CS_TRUE and CS_FALSE.

The CS_PARTIAL_TEXT property must be set before a connection to the server is established. If the server does not support partial updates, CS_PARTIAL_TEXT will be reset to CS_FALSE, which is the default value.

Password

CS_PASSWORD defines the password that a connection uses when logging in to a server.

The password is ignored if network-based authentication is requested for the connection. Applications request network-based authentication by setting the CS_SEC_NETWORKAUTH property. See “Requesting login authentication services” on page 256.

Applications that do not use network authentication can set the CS_SEC_ENCRYPT property so that Client-Library sends the password to the server in encrypted form. See “Using password encryption in Client-Library applications” on page 274.

Persistent result bindings

Typically, Client-Library removes the binding between the application’s destination variables and a command after the application has processed the results of the command.

CS_STICKY_BINDS, however, determines whether bindings established by ct_bind persist across repeated executions of a command. If CS_STICKY_BINDS is enabled (CS-TRUE), Client-Library does not remove binds until the application initiates a new command with ct_command, ct_cursor, ct_dynamic, or ct_sendpassthru.

CS_STICKY_BINDS must be set to CS_TRUE before ct_send is called to execute the command whose result bindings will be saved. Once set, the property affects all future command processing on the command structure.

CS_STICKY_BINDS should be set only by applications that repeatedly execute the same command, and only if the result formats returned by the command cannot vary. A command's result format information consists of a sequence of the following result-set characteristics:

- The result type (indicated to the application by the ct_results *result_type* parameter)
- The number of columns available to the application using ct_res_info; applies to fetchable results only.
- The format of each column available to the application using ct_describe for each column; applies to fetchable results only.

If a server command contains conditional logic, it is possible that the format of the results returned by the second and later command executions will not match that of the first execution. In this case, the bindings established in the first execution are cleared automatically by Client-Library. ct_results raises an informational error (and returns CS_SUCCEED) when Client-Library detects a mismatch in the results format.

Program structure for persistent binds

Applications can reuse binds by setting the CS_STICKY_BINDS command property to CS_TRUE before the command is sent to the server. Applications check the CS_HAVE_BINDS command property to see whether binds have been established for a result set.

For example, suppose an application repeatedly executes the same RPC command to run a stored procedure containing a single select statement. Such an application could use the program logic shown below to re-execute the command and reuse the result bindings:

```
/*
 ** Enable persistent result bindings.
 */
 ct_cmd_props to set CS_STICKY_BINDS to CS_TRUE

 /*
 ** Initiate the RPC command.
 */
 ct_command(CS_RPC_COMMAND, proc_name)
 ct_setparam for each parameter
 set values in parameter source variables
 ct_send
 loop while ct_results returns CS_SUCCEED
 switch(result_type)
```

```
 case CS_ROW_RESULT:
 ct_bind for each column
 loop on ct_fetch
           ... process row data ...
        end loop
     case CS_STATUS_RESULT:
        ct_bind for the procedure’s return status
        loop on ct_fetch
           ... process the return status value ...
        end loop
     ... other cases...
   end switch
 end loop
/*
 ** Change the input parameter values and resend the command.
 */
 set values in parameter source variables
 ct_send
 loop while ct_results returns CS_SUCCEED
   switch(result_type)
     case CS_ROW_RESULT:
        (optional) ct_cmd_props to check CS_HAVE_BINDS
        loop on ct_fetch
           ... process row data ...
        end loop
     case CS_STATUS_RESULT:
        (optional) ct_cmd_props to check CS_HAVE_BINDS
        loop on ct_fetch
           ... process the return status value ...
        end loop
     ... other cases...
   end switch
 end loop

 /*
 ** Execute a new command. A call to ct_command, ct_cursor, or
 ** ct_dynamic clears the previous initiated command from the
 ** command structure.
 */
 ct_command
```

```
... and so forth ...
```

Note If a command returns multiple result sets (for example, if the stored procedure in the example above contained multiple select statements), then the results loop logic above would use calls to ct_res_info(CS_CMD_NUMBER) to distinguish between the different result sets.

When CS_STICKY_BINDS is set to CS_TRUE, there is some internal overhead caused by Client-Library's need to save and compare result-set formats. Applications that do not repeatedly execute the same command and reuse the result bindings should leave the property at its default setting, FALSE.

CS_STICKY_BINDS does not affect binds established on command structures that control extended error data or notification parameter values. Applications access these command structure as the CS_EED_CMD and CS_NOTIF_CMD connection properties, respectively. Applications must always rebind when fetching from these command structures.

For detailed usage information on the routines mentioned above, see the reference page for each routine in Chapter 3, "Routines"

Applications check the CS_HAVE_BINDS command property to see if any saved binds are established for the current result set. See "Have bindings" on page 213, "Resending commands" on page 584, and "Restoring a cursor-open command" on page 435.

Retry count

CS_RETRY_COUNT specifies the number of times that ct_connect retries the sequence of network addresses associated with a server name. The default is 0.

The CS_LOOP_DELAY specifies the delay, in seconds, that ct_connect waits before retrying the entire sequence of addresses. See "Loop delay" on page 217.

CS_LOOP_DELAY and CS_RETRY_COUNT affect only the establishment of a login dialog. Once Client-Library has found an address where a server responds, the login dialog between Client-Library and the server begins. Client-Library does not retry any other addresses if the login attempt fails.

Addresses are associated with server names either in a network-based directory or the Sybase *interfaces* file. For more information, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

On UNIX platforms, a server's *interfaces* file entry can be configured to override application-specified settings for CS_RETRY_COUNT and CS_LOOP_DELAY. For more information, see the Open Client and Open Server *Configuration Guide* for UNIX.

Security properties

See "Security features" on page 251 for a description of all the CS_SEC properties.

Server name

CS_SERVERNAME gives the name of the server to which a connection is made.

CS_SERVERNAME is a read-only property, and an application can only retrieve its value after a connection is opened with ct_connect.

Note If external configuration is enabled for the connection, you can change the server name by modifying the CS_SERVERNAME definition in the configuration file. See "Enabling external configuration" on page 307.

To specify the name of a server to connect to, pass the server name to ct_connect.

TDS version

CS_TDS_VERSION defines the version of the Tabular Data Stream (TDS) protocol that the connection is using.

Because CS_TDS_VERSION is a negotiated login property, its value may change during the login process. An application sets CS_TDS_VERSION to request a TDS level before calling ct_connect. When ct_connect creates the connection, if the server cannot provide the requested TDS version, a new (lower) TDS version is negotiated. An application retrieves the value of CS_TDS_VERSION after a connection is established to determine the actual version of TDS in use.

Table 2-31 lists the symbolic values of CS_TDS_VERSION. The supported features for the earlier versions have been carried forward for the later versions:

Table 2-31: Values for CS_TDS_VERSION

Symbolic value	Meaning	Added supported features
CS_TDS_40	4.0 TDS	Browse mode, text and image handling, remote procedure calls, bulk copy
CS_TDS_42	4.2 TDS	Internationalization.
CS_TDS_46	4.6 TDS	Registered procedures, TDS passthrough, negotiable TDS packet size, multibyte character sets
CS_TDS_50	5.0 TDS	Cursors

If not otherwise set, CS_TDS_VERSION defaults to a value based on the CS_VERSION level that an application requested through ct_init.

A connection's CS_TDS_VERSION level will never be higher than the default TDS level associated with its parent context's CS_VERSION level.

For example, 5.0 is the default TDS level associated with a version level of CS_VERSION_110 and later. If an application calls ct_init with *version* as CS_VERSION_110 for a context, all connections created within that context are restricted to CS_TDS_VERSION levels of 5.0 or lower.

If an application sets the CS_TDS_VERSION property, Client-Library overwrites existing capability values with default capability values corresponding to the new TDS version. For this reason, an application should set CS_TDS_VERSION before setting any capabilities for a connection.

Text and image limit

CS_TEXTLIMIT indicates the length, in bytes, of the longest text or image value that an application wants to receive. Client-Library will read but ignore any part of a text or image value that goes over this limit.

The default value of CS_TEXTLIMIT is CS_NO_LIMIT. This means that Client-Library reads and returns all data sent by the server.

In case of huge text values, it takes some time for an entire text value to be returned over the network. To keep an Adaptive Server from sending this extra text in the first place, use the ct_options CS_TEXTSIZE_OPT option to set the server global variable @@*textsize*.

Timeout

CS_TIMEOUT specifies the length of time, in seconds, that Client-Library waits for a server response to a command.

The default timeout value is CS_NO_LIMIT, which represents an infinite timeout period. Negative and zero values are not allowed for CS_TIMEOUT.

Setting timeout values

ct_config is called to set the timeout value before or after a call to ct_connect creates an open connection. It takes effect for all open connections immediately upon being called.

The following code fragment sets a 60-second timeout limit:

```
CS_INT timeval;
 timeval = 60;
 if (ct_config(ctx, CS_SET, CS_TIMEOUT,
             (CS_VOID *)&timeval,
             CS_UNUSED, NULL)
     != CS_SUCCEED)
 {
   fprintf(stdout,“Can't config timeout. Exiting.“);
   (void)ct_exit(ctx, CS_FORCE_EXIT);
   (void)cs_ctx_drop(ctx);
   exit(1);
 }
```

Handling timeout errors

Timeout errors occur in synchronous applications that have set either or both of the CS_TIMEOUT or CS_LOGIN_TIMEOUT properties to values other than CS_NO_LIMIT. CS_LOGIN_TIMEOUT sets the timeout period for reading the server’s response to a login attempt, while CS_TIMEOUT sets the timeout period for reading the results of a server command. The application receives the same Client-Library message for timeouts in both cases. (See “Login timeout” on page 216 for a description of the CS_LOGIN_TIMEOUT property).

Note You can specify CS_TIMEOUT or CS_LOGIN_TIMEOUT values on on a per-connection basis with ct_con_props.

Applications that use inline error handling must set the CS_DIAG_TIMEOUT property to specify whether Client-Library should abort or retry when a timeout occurs. See “Diagnostic timeout fail” on page 210 for more information.

Applications that handle Client-Library messages with a callback can identify the timeout error and either cancel the operation or retry for another timeout period. A client message callback has the following options for handling a timeout message:

- Return CS_FAIL to cancel the operation and mark the connection as dead. This is the only way to abort a login attempt that has timed out.
- (Non-login timeouts only.) Call ct_cancel(CS_CANCEL_ATTN) to cancel the command that is being processed, then return CS_SUCCEED.
- Return CS_SUCCEED to retry for another timeout period.

A timeout error is identified by breaking the error number (identified by the *number* field of the CS_CLIENTMSG structure) into its four components and checking whether the error number matches the following characteristics:

- Severity – CS_SV_RETRY_FAIL
- Number – 63
- Origin – 2
- Layer – 1

An application breaks an error number into components with the CS_SEVERITY, CS_NUMBER, CS_ORIGIN, and CS_LAYER macros. See “Client-Library message numbers” on page 81 for a description of these macros. An example of testing for timeout errors is provided below.

The callback checks the value of the CS_LOGIN_STATUS connection property to see whether the timeout is happening during connection establishment or during command processing. If the property is CS_TRUE, the connection is already established and the server has timed out during command processing.

The following code fragment defines a client message callback that handles timeout errors:

```
/*
 ** ERROR_SNOL(error_numb, severity, number, origin, layer)
 **
 **   Error comparison for Client-Library or CS-Library errors.
 **   Breaks down a message number and compares it to the given
```

```
 **   constants for severity, number, origin, and layer.
 **   Returns non-zero if the error number matches the 4
 **   constants.
 */
 #define ERROR_SNOL(e, s, n, o, l) \
   ( (CS_SEVERITY(e) == s) && (CS_NUMBER(e) == n) \
    && (CS_ORIGIN(e) == o) && (CS_LAYER(e) == l ) )

CS_RETCODE  client_msg_handler(cp, conn, emsgp)
 CS_CONTEXT     *cp;
 CS_CONNECTION  *conn;
 CS_CLIENTMSG   *emsgp;
 {
   CS_RETCODE ret;
   CS_INT     status;

   ... code to print message details and handle any other
       errors besides timeout ...

 /*
   ** Is this a timeout error?
   */
   if (ERROR_SNOL(emsgp->msgnumber, CS_SV_RETRY_FAIL, 63, 2, 1))
   {
     /*
     ** Read from server timed out. Timeouts happen on synchronous
     ** connections only, and you must have set one or both of the
     ** following context properties to see them:
     **  CS_TIMEOUT for results timeouts
     **  CS_LOGIN_TIMEOUT for login-attempt timeouts
     **
     ** If we return CS_FAIL, the connection is marked as dead and
     ** unrecoverable.  If we return CS_SUCCEED, the timeout
     ** continues for another quantum.
     **
     ** We kill the connection for login timeouts, and send a
     ** cancel for results timeouts. We determine which case we
     ** have through the CS_LOGIN_STATUS property.
     */
     status = 0;
     if (ct_con_props(conn, CS_GET, CS_LOGIN_STATUS,
                     (CS_VOID *)&status,
                      CS_UNUSED, NULL) != CS_SUCCEED)
     {
       fprintf(stdout, "ct_con_props() failed in error handler.");
       return CS_FAIL;
     }
```

```
   if (status)
     {
       /* Results timeout */
       fprintf(stdout, "Issuing a cancel on the query...\n");
       (CS_VOID)ct_cancel(conn, (CS_COMMAND *)NULL,
                          CS_CANCEL_ATTN);
     }
   else
     {
       /* Login timeout */
       fprintf(stdout, "Aborting connection attempt...\n");
       return CS_FAIL;
     }
   }
   return (CS_SUCCEED);
}
```

Transaction name

CS_TRANSACTION_NAME defines a transaction name to be used over a connection to Open Server for CICS.

Open Server for CICS uses transaction names to identify executables running under CICS. For more information about Open Server for CICS, see the Open Server for CICS documentation.

Transaction names for Sybase Server applications are determined by the Transact-SQL begin tran statement that marks the transaction's beginning, not by CS_TRANSACTION_NAME. See the Adaptive Server Enterprise *Reference Manual* for more information.

All Client-Library applications can set CS_TRANSACTION_NAME. If a transaction name is not required, CS_TRANSACTION_NAME is ignored.

User allocation function

CS_USER_ALLOC identifies a user-supplied memory allocation routine that Client-Library uses for memory management while operating at the system interrupt level.

Together, CS_USER_ALLOC and CS_USER_FREE allow an asynchronous application to perform its own memory management.

A user-supplied memory allocation routine must be defined as:

```
void     *user_alloc(size)
```

```
size_t    size;
```

Ordinarily, Client-Library routines satisfy their memory requirements by calling malloc. However, not all implementations of malloc are reentrant, so it is not safe to use malloc in Client-Library routines that are called at the system interrupt level. For this reason, on systems where Client-Library uses signal-driven network I/O, such as UNIX systems, fully asynchronous applications are required to provide an alternate way for Client-Library to satisfy its memory needs.

This is not a requirement on platforms that use thread-driven network I/O or for applications that do not use fully asynchronous connections. For a description of the network I/O method used on your platform, see the Open Client and Open Server *Programmers Supplement* for Microsoft Windows and the Open Client and Open Server *Programmers Supplement* for UNIX.

Client-Library provides two mechanisms by which an asynchronous application can satisfy Client-Library's memory requirements:

- The application uses the CS_MEM_POOL property to provide Client-Library with a memory pool.
- The application uses the CS_USER_ALLOC and CS_USER_FREE properties to install memory allocation and free routines that Client-Library safely calls at the interrupt level.

If a fully asynchronous application fails to provide Client-Library with a safe way to satisfy memory requirements, Client-Library's behavior is undefined.

Client-Library attempts to satisfy memory requirements from the following sources, in the following order:

1. Memory pool
2. User-supplied allocation and free routines
3. System routines

If a connection cannot get the memory it needs, Client-Library marks the connection dead.

An application may replace a user-defined memory routine by calling ct_config with action as CS_SET and *buffer* as the address of the new routine.

An application clears a memory routine in two ways:

- By calling ct_config with action as CS_SET and *buffer* as NULL, or
- By calling ct_config with action as CS_CLEAR.

User free function

CS_USER_FREE identifies a user-supplied memory deallocation routine that Client-Library will use for system interrupt-level memory management.

Together, CS_USER_ALLOC and CS_USER_FREE allow an asynchronous application to perform its own interrupt-level memory management.

A user-supplied memory deallocation routine must be defined as:

```
void     user_free(ptr)
void     *ptr;
```

See “User allocation function” on page 233.

User data

The CS_USERDATA property defines user-allocated data. This property allows an application to associate user data with a particular connection or command structure.

There is no default value for CS_USERDATA. If an application retrieves the property when no value is set, then ct_con_props or ct_cmd_props returns with *outlen* set to 0.

CS_USERDATA is useful when a callback routine and the main-line application need to share information without using global variables.

When an application stores data with CS_USERDATA, Client-Library copies the actual data pointed to by the *buffer* parameter of ct_con_props or ct_cmd_props; not a pointer to the data, into internal data space.

CS_USERDATA takes as its value any piece of application-defined data. When setting the property, the application passes a pointer to the data (cast to CS_VOID *) and specifies the exact length of the data in bytes. Most applications actually install the address of an application-allocated data structure as CS_USERDATA. This allows the application to retrieve, as CS_USERDATA, a pointer to the data. The application changes the data through the pointer, and does not need to reinstall the data in the context, connection, or command structure after changing it.

To associate user data with a context structure, an application calls cs_config. CS_USERDATA property values are not inherited at the connection or command levels.

The following code fragment demonstrates the CS_USERDATA property:

```
CS_CHAR          set_charbuf[32];
 CS_CHAR          get_charbuf[32];
```

```
  CS_CONNECTION   *con;
  CS_RETCODE      ret;
  CS_INT          outlen;
  CS_COMMAND      *set_cmd;
  CS_COMMAND      *get_cmd;

/*
  ** Store a character string in the userdata field.
  ** Set the length field to one greater than the length
  ** of the string so that the null terminator will be
  ** stored as part of the user data. If the null
  ** terminator is not explicitly stored as part of the
  ** userdata, then the string will not be null-
  ** terminated when it is retrieved.
  */
  strcpy(set_charbuf, "some userdata");
  ret = ct_con_props(con, CS_SET, CS_USERDATA,
       set_charbuf, strlen(set_charbuf) + 1, NULL);
  if (ret != CS_SUCCEED)
  {
       error("ct_con_props() failed");
  }

ret = ct_con_props(con, CS_GET, CS_USERDATA,
       get_charbuf, sizeof(get_charbuf), &outlen);
  if (ret != CS_SUCCEED)
  {
       error("ct_con_props() failed");
  }

/*
  ** The next example stores a pointer to a CS_COMMAND
  ** structure in the connection’s user data field.
  */
  ret = ct_con_props(con, CS_SET, CS_USERDATA,
       &set_cmd, sizeof(set_cmd), NULL);
  if (ret != CS_SUCCEED)
  {
       error("ct_con_props() failed");
  }

ret = ct_con_props(con, CS_GET, CS_USERDATA,
       &get_cmd, sizeof(get_cmd), &outlen);
  if (ret != CS_SUCCEED)
  {
       error("ct_con_props() failed");
  }
```

User name

CS_USERNAME defines the user login name that the connection will use to log in to a server.

If the application has not requested network-based user authentication, then the application must set the value of CS_PASSWORD connection property to match the user's password. See "Password" on page 224.

If the application has requested network-based authentication with the CS_SEC_NETWORKAUTH property, then the user must already be logged into the connection's network security mechanism under the same name as CS_USERNAME. In this case, the CS_PASSWORD property is ignored.

Applications request network-based authentication by setting the CS_SEC_NETWORKAUTH property. See "Requesting login authentication services" on page 256.

Version string for Client-Library

CS_VER_STRING defines a character string that represents the actual version of Client-Library that an application is using. This property may only be retrieved.

CS_VER_STRING and CS_VERSION indicate different version levels because later versions of Client-Library emulate the behavior of earlier versions.

CS_VER_STRING represents the actual version of Client-Library that is in use. CS_VERSION represents the version of Client-Library behavior that an application has requested with ct_init.

Version of Client-Library

The CS_VERSION property represents the version of Client-Library behavior than an application has requested through ct_init. The value of this property may only be retrieved.

Possible values for CS_VERSION include the following:

- CS_VERSION_100 indicates version 10.0
- CS_VERSION_110 indicates version 11.0
- CS_VERSION_120 indicates version 12.0.
- CS_VERSION_125 indicates version 12.5.

- CS_VERSION_150 indicates version 15.0.

Connections allocated within a context use default CS_TDS_VERSION values that are based on their parent context's CS_VERSION level. See "TDS version" on page 228 for more information.

Both Client-Library and CS-Library have CS_VERSION properties. ct_config returns the value of the Client-Library CS_VERSION. cs_config returns the value of the CS-Library CS_VERSION.

Registered procedures

A registered procedure is a procedure that is defined and installed in a running Open Server application, and extends the functionality of Adaptive Server.

For Client-Library applications, registered procedures provide a means for inter-application communication and synchronization. This is because Client-Library applications connected to an Open Server watches for a registered procedure to execute. When the registered procedure executes, applications watching for it receive a notification that includes the procedure's name and the arguments it was called with.

For example, suppose that:

- stockprice is a real-time Client-Library application monitoring stock prices.
- price_change is a registered procedure created in Open Server by stockprice, and that price_change takes as parameters a stock name and a price differential.
- sellstock, an application that puts stock up for sale, has requested that it be notified when price_change executes.

When stockprice, the monitoring application, becomes aware that the price of Extravagant Auto Parts stock has risen $1.10, it executes price_change with the parameters "Extravagant Auto Parts" and "+1.10".

When price_change executes, Open Server sends sellstock a notification containing the name of the procedure (price_change) and the arguments passed to it ("Extravagant Auto Parts" and "+1.10"). sellstock uses the information contained in the notification to decide whether or not to sell Extravagant Auto Parts stock.

price_change is the means through which the stockprice and sellstock applications communicate.

Registered procedures as a means of communication have the following advantages:

- A single call to execute a registered procedure results in many client applications being notified that the procedure has executed. The application executing the procedure does not need to know how many, or which, clients have requested information.
- The registered procedure communication mechanism is server-based. Open Server acts as a central repository for connection addresses. Because of this, client applications communicate without having to connect directly to each other. Instead, each client simply connects to the Open Server.

A Client-Library application makes remote procedure calls to Open Server system registered procedures to:

- Create a registered procedure in Open Server.

  Note A Client-Library application creates only registered procedures that contain no executable statements. These bodiless procedures are primarily useful for communication and synchronization purposes.

- Drop a registered procedure.
- List all registered procedures defined in Open Server.
- Request to be notified when a particular registered procedure is executed.
- List all registered procedure notifications that the client connection is waiting for.
- Execute a registered procedure.

For more information about Open Server system registered procedures, see the Open Server *Server-Library/C Reference Manual*.

An application calls Client-Library routines to:

- Install a user-supplied notification callback routine to be called when the application receives notification that a registered procedure has executed
- Poll the network (if necessary) to see if any registered procedure notifications are waiting

When Client-Library receives a notification

When Client-Library receives a registered procedure notification, it calls an application's notification callback routine. Depending on the host client platform, the application may have to poll the network (with ct_poll) for Client-Library to invoke the notification callback. See "Receiving notifications asynchronously" on page 240.

The registered procedure's name is available as the second parameter to the notification callback routine.

The arguments with which the registered procedure was called are available inside the notification callback as a parameter result set. To retrieve these arguments, an application:

- Calls ct_con_props(CS_NOTIF_CMD) to retrieve a pointer to the command structure containing the parameter result set
- Calls ct_res_info(CS_NUMDATA), ct_describe, ct_bind, ct_fetch, and ct_get_data to describe, bind, and fetch the parameters

See "Notification callbacks" on page 48.

Receiving notifications asynchronously

The application's receipt of notification events depends on the CS_ASYNC_NOTIFS property and the network I/O methods supported by the client platform.

The CS_ASYNC_NOTIFS property determines whether a connection receives notifications asynchronously. See "Asynchronous notifications" on page 203.

When the connection to the Open Server has little or no activity other than notifications, asynchronous notifications should be enabled by setting the CS_ASYNC_NOTIFS property to CS_TRUE. This property defaults to CS_FALSE, which means that the application must be interacting with the server over the connection (to cause Client-Library to read from the network) to receive a registered procedure notification.

Note If a connection is used only to receive registered procedure notifications, asynchronous notifications must be enabled for a connection even if the connection is polled. On an otherwise idle connection, ct_poll does not trigger the notification callback unless the CS_ASYNC_NOTIFS property is CS_TRUE. The default setting is CS_FALSE.

Finding out about notifications

If asynchronous notifications are enabled on platforms that support signal- or thread-driven I/O, then Client-Library automatically invokes a connection's notification callback when a notification arrives on the connection.

On other platforms, the application must poll the connection with ct_poll if the connection is not otherwise active. CS_ASYNC_NOTIFS must be set to CS_TRUE for ct_poll to report notifications.

Results

When a Client-Library command executes on a server, it generates various types of results, which are returned to the application that sent the command. The result types are as follows:

- Regular row results
- Cursor row results
- Parameter results
- Stored procedure return status results
- Compute row results
- Message results
- Describe results
- Format results

Results are returned to an application in the form of **result sets**. A result set contains only a single type of result data. Regular row and cursor row result sets contain multiple rows of data, but other types of result sets contain at most a single row of data.

An application processes results by calling ct_results, which indicates the type of result available by setting **result_type*.

ct_results sets *result_type to CS_CMD_DONE to indicate that the results of a "logical command" have been completely processed. A logical command is generally considered to be any Client-Library command defined through ct_command, ct_dynamic, or ct_cursor. Exceptions to this rule are documented in "ct_results and logical commands" on page 567.

Some commands, for example a language command containing a Transact-SQL update statement, do not generate results. ct_results sets **result_type* to CS_CMD_SUCCEED or CS_CMD_FAIL to indicate the status of a command that does not return results.

Regular row results

A regular row result set is generated by the execution of a Transact-SQL select statement on a server.

A regular row result set contains zero or more rows of tabular data.

Cursor row results

A cursor row result set is generated when an application executes a Client-Library cursor open command.

Note A cursor row result set is not generated when an application executes language command containing a Transact-SQL fetch statement. Cursor rows from a fetch language statement are returned as CS_ROW_RESULT result set.

A cursor row result set contains zero or more rows of tabular data.

A cursor row result set differs from a regular row result set in that an application uses ct_cursor to update underlying tables while fetching cursor rows. This is not possible with regular rows.

Parameter results

A parameter result set contains a single “row” of parameters. Several types of data are returned as a parameter result set, including:

- Message parameters – a message result set (CS_MSG_RESULT) has parameters associated with it. Message parameters arrive as a CS_PARAM_RESULT result set immediately following the CS_MSG_RESULT result type.

- RPC return parameters – an Adaptive Server stored procedure or an Open Server registered procedure returns output parameter data. This is a CS_PARAM_RESULT result set that contains new values for the procedure's parameters, as set by the procedure code.

Extended error data and registered procedure notification parameters are also returned as parameter result sets, but since an application does not call ct_results to process these types of data, the application never sees a result type of CS_PARAM_RESULT. Instead, the row of parameters is simply available to be fetched after the application retrieves the CS_COMMAND structure containing the data.

For information about extended error data, see "Extended error data" on page 129. For information about registered procedure notification parameters, see "Registered procedures" on page 238.

Stored procedure return status results

A status result set consists of a single row which contains a single value—a return status.

All stored procedures that run on a Adaptive Server return a status number. Stored procedures usually return 0 to indicate normal completion. For a list of Adaptive Server default return status numbers, see the return reference page in the Adaptive Server Enterprise *Reference Manual*.

Because return status numbers are a feature of stored procedures, only an RPC command or a language command containing an execute statement generates a return status.

Compute row results

A compute row result set contains a single row of tabular data with a number of columns equal to the number of columns listed in the compute clause that generated the compute row.

For more information about compute rows, see compute clause in the Adaptive Server Enterprise *Reference Manual*.

Message results

A message result set does not actually contain any data. Instead, a message has an ID. To get a message's ID, an application calls ct_res_info after ct_results returns with a *result_type* of CS_MSG_RESULT.

If parameters are associated with a message, they are returned as a separate parameter result set, immediately following the message result set.

Describe results

A describe result set does not contain fetchable data; instead, it indicates the existence of descriptive information returned as the result of a dynamic SQL describe input or describe output command.

An application retrieves this descriptive information with any of the methods below:

- Call ct_res_info to get the number of items and ct_describe to get a description of each item.
- Call ct_dyndesc several times to get the number of items and a description of each.
- Call ct_res_info to get the number of items, and call ct_dynsqlda once to get item descriptions.

For more information about dynamic SQL, see Chapter 8, "Using Dynamic SQL Commands," in the Open Client *Client-Library/C Programmers Guide*.

Format results

There are two types of format results: regular row format results and compute row format results.

Format result sets do not contain fetchable data, but rather indicate the availability of format information for the regular row and compute row result sets with which they are associated.

All format information for a command is returned before any data. That is, the row format and compute format result sets for a command precede the regular row and compute row result sets that the command generates.

Format information is primarily of use in gateway applications, which need to repackage Adaptive Server results before sending them on to a foreign client.

A gateway application typically processes a format result set one column at a time, retrieving format information for the column by calling ct_describe and ct_compute_info and sending the format information through Server-Library routines.

A connection receives format results only if its CS_EXPOSE_FMTS property is set to CS_TRUE.

Program structure for processing results

The following pseudocode demonstrates how a typical application might process the various types of result data:

```
while ct_results returns CS_SUCCEED
     case CS_ROW_RESULT
          ct_res_info to get the number of columns
          for each column:
               ct_describe to get a description of the
                    column
               ct_bind to bind the column to a program
                    variable
          end for
          while ct_fetch returns CS_SUCCEED or
               CS_ROW_FAIL
               if CS_SUCCEED
                    process the row
               else if CS_ROW_FAIL
                    handle the row failure;
               end if
          end while
          switch on ct_fetch's final return code
               case CS_END_DATA...
               case CS_CANCELED...
               case CS_FAIL...
          end switch
     end case
     case CS_CURSOR_RESULT
          ct_res_info to get the number of columns
          for each column:
               ct_describe to get a description of the
                    column
               ct_bind to bind the column to a program
                    variable
          end for
          while ct_fetch returns CS_SUCCEED or
```

```
                CS_ROW_FAIL
          (while ct_scroll_fetch returns CS_SUCCEED or
              CS_CURSOR_BEFORE_FIRST or CS_CURSOR_AFTER_LAST
                for scrollable cursors)

                     process the row
                     /*
                     ** Nested cursor commands are legal
                     ** here.
                     */
                else if CS_ROW_FAIL
                     handle the row failure
                end if

        end while

    switch on ct_fetch's final return code
          case CS_END_DATA...
          case CS_CANCELED...
          case CS_FAIL...
      end switch
end case
case CS_PARAM_RESULT
    ct_res_info to get the number of parameters
    for each parameter:
        ct_describe to get a description of the
            parameter
        ct_bind to bind the parameter to a
            variable
    end for
    while ct_fetch returns CS_SUCCEED or
        CS_ROW_FAIL
        if CS_SUCCEED
            process the row of parameters
        else if CS_ROW_FAIL
            handle the failure
        end if
    end while
    switch on ct_fetch's final return code
        case CS_END_DATA...
        case CS_CANCELED...
        case CS_FAIL...
    end switch
end case
case CS_STATUS_RESULT
    ct_bind to bind the status to a program
        variable
    while ct_fetch returns CS_SUCCEED or
```

```
                CS_ROW_FAIL
                if CS_SUCCEED
                    process the return status
                else if CS_ROW_FAIL
                    handle the failure
                end if
            end while
            switch on ct_fetch's final return code
                case CS_END_DATA...
                case CS_CANCELED...
                case CS_FAIL...
            end switch
        end case
        case CS_COMPUTE_RESULT
            (optional:  ct_compute_info to get bylist
                length, bylist, or compute row id)
            ct_res_info to get the number of columns
            for each column:
                ct_describe to get a description of the
                    column
                ct_bind to bind the column to a program
                    variable
                (optional: ct_compute_info to get the
                    compute column id or the aggregate
                    operator for the compute column)
            end for
            while ct_fetch returns CS_SUCCEED or
                CS_ROW_FAIL
                if CS_SUCCEED
                    process the compute row
                else if CS_ROW_FAIL
                    handle the failure
                end if
            end while
          switch on ct_fetch's (or ct_scroll_fetch for scrollable cursors)
                final return code
                case CS_END_DATA (or CS_SCROLL_CURSOR_ENDS for scrollable
                cursors)...
                case CS_CANCELED...
                case CS_FAIL...
            end switch
        end case
        case CS_MSG_RESULT
            ct_res_info to get the message id
            code to handle the message
        end case
```

```
        case CS_DESCRIBE_RESULT
            ct_res_info to get the number of columns
            for each column:
                ct_describe to get a
                    description
            end for
        end case
        case CS_ROWFMT_RESULT
            ct_res_info to get the number of columns
            for each column:
                ct_describe to get a column description
                send the information on to the gateway
                    client
            end for
        end case
        case CS_COMPUTEFMT_RESULT
            ct_res_info to get the number of columns
            for each column:
                ct_describe to get a column description
                (if required:
                    ct_compute_info for compute
                        information
                end if required)
                send the information on to the gateway
                    client
            end for
        end case
        case CS_CMD_DONE
            indicates a command’s results are completely
                processed
        end case
        case CS_CMD_SUCCEED
            indicates the success of a command that
                returns no results
        end case
        case CS_CMD_FAIL
            indicates a command failed
        end case
    end while
    switch on ct_results’ final return code
        case CS_END_RESULTS
            indicates no more results
        end case
        case CS_CANCELED
            indicates results were canceled
        end case
```

```
        case CS_FAIL
            indicates ct_results failed
        end case
    end switch
```

Retrieving an item's value

When processing a result set, there are four ways for an application to retrieve a result item's value:

- It calls ct_bind to associate a result item with a program variable. When the program calls ct_fetch to fetch a result row, the item's value is automatically converted to the destination variable's format and the result is placed into the bound destination variable. Most applications use this method for all result items except large text or image values. See "text and image data handling" on page 284 for more information.
- It calls ct_get_data to retrieve a result item's value in chunks. After calling ct_fetch to fetch the row, the application calls ct_get_data in a loop. Each ct_get_data call retrieves a chunk of the result item's value. Most applications use ct_get_data only to retrieve large text or image values.
- It calls ct_dyndesc(CS_USE_DESC) to associate a dynamic descriptor with the result set. After a dynamic descriptor is associated with a result set, an application repeatedly calls ct_fetch to fetch each row, and for each row, calls ct_dyndesc once for each result item. Typical applications do not use ct_dyndesc, which is intended for precompiler support.
- It calls ct_dynsqlda(CS_USE_DESC) to associate an application-managed SQLDA structure with the result columns. An application calls ct_dynsqlda once to bind all result columns to the value buffers pointed at by the SQLDA structure. Subsequent calls to ct_fetch place column values in the value buffers. Typical applications do not use ct_dynsqlda, which is intended for precompiler support.

Keeping result bindings for batch processing

Batch processing applications resends the same server command over and over again. Applications resend a command by calling ct_send immediately after the results of the previous execution have been processed. See "Resending commands" on page 584.

Batch processing applications that resend commands may benefit from setting the CS_STICKY_BINDS command property. When this property is set to CS_TRUE (the default is CS_FALSE), Client-Library reuses result bindings when a command is resent. This eliminates redundant ct_bind calls in the application.

For more information, see:

- "Persistent result bindings" on page 224 for a description of the CS_STICKY_BINDS property, and
- The reference page for ct_bind on page 323.

Selecting multiple rows of variable length data into an array

When multiple rows of a variable length data (VARCHAR or VARBINARY) are selected into a buffer, each new item begins at an index that is a multiple of *datafmt->maxlength*, even if the preceding item is less than *datafmt->maxlength* bytes long. This is illustrated in the code fragment below.

```
/* This example demonstrates selecting multiple rows of
variable-length data into a buffer. In this case, the
first row to be returned will have one column with the
value "first string" and a second row with a column with
the value "second string". */
datafmt.count = 2;
datafmt.maxlength = 25;
retcode = ct_results(cmd, &restype);
if (retcode != CS_SUCCEED)
{
       /* error handling code deleted */
       . . .
}
if (restype == CS_ROW_RESULT)
{
       retcode = ct_bind(cmd, 1, datafmt, buffer,
CS_NULL, CS_NULL);
       if (retcode != CS_SUCCEED)
       {
           /* error handling code deleted */
           . . .
       }
       retcode = ct_fetch(cmd, CS_UNUSED, CS_UNUSED,
           &nrows);
       if (retcode != CS_SUCCEED)
```

```
        {
            /* error handling code deleted */
            . . .
        }

/* At this point, the string "first string" begins at
buffer[0] and the string "second string" begins at
buffer[25], even though the first data item was less
than 25 characters long.*/
}
```

Security features

Client-Library provides three categories of security features:

- Network-based security – Client-Library and Server-Library applications may be integrated with the security services provided by network system software such as DCE, or Microsoft LAN Manager. Among other services, this feature provides unified login (users connect to a Sybase server using their network user name and password), and per-packet security services (such as encrypting all communications between the client and the server).

  This feature requires separate Sybase-supported network security software and a Sybase-supplied security driver for that software. Network-based security was introduced to Client-Library at version 11.1 and requires a server based on Open Server 11.1 or later.

  Note Adaptive Server 11.0 and above do not support network-based security services.

- Secure Sockets Layer (SSL) network-based security – From version 12.5, Client-Library and Server-Library applications include a network-library driver to enable SSL, session-based security.

  SSL is an industry standard for sending wire- or socket-level encrypted data over client-to-server and server-to server connections. A client sends a connection request to the server along with its supported SSL options. The server responds with a server certificate that proves that the server is what it claims to be, along with a list of its supported CipherSuites. An SSL-enabled session begins when the client and the server agree upon a CipherSuite, and all transmitted data is protected by session-based encryption.

- Sybase security features – these features include password encryption and challenge/response security handshakes.

  Client-Library encrypts user passwords if an application requests it. Passwords are encrypted with a handshaking protocol where the server sends an encryption key and the client uses the key to encrypt the user's passwords.

  Challenge/response handshaking allows applications to implement a security strategy where the server challenges clients at connect time. In this strategy, the server refuses connections from clients who cannot provide the expected response to the challenge.

  These features are part of the TDS protocol and require no external software. Adaptive Server and Open Server version 10.0 or later support these features.

Network-based security

A distributed client/server computing environment introduces security concerns that go beyond those of a local system. Because users are out of sight and data is moving from system to system, even across public data networks, intruders may view or tamper with confidential data. Security services allow client/server applications to create secure connections.

Network-based security takes advantage of third-party distributed security software to authenticate network users and to protect data transmitted over the network.

Security mechanisms and security drivers

Sybase defines a **security mechanism** as external software that provides security services for a connection. For example, these are some security mechanisms that can be used on a Client-Library connection:

- DCE security servers and security clients provide security services for clients and servers within a DCE cell.
- CyberSafe Kerberos provides security services for clients on Windows and UNIX and servers on UNIX.

- Windows NT LAN Manager Security Services Provider Interface (SSPI) provides security services for servers on Windows NT (3.5 and later) and clients on NT (3.5 and later), and Windows 95.

Note Certain security systems, such as DCE, use the Data Encryption Standard (DES), which is banned for export by the United States State Department. If you are in the United States and writing software for export, verify that the security mechanism is available.

Sybase provides security drivers that allow Client-Library and Server-Library applications to take advantage of an installed network security system. By using security drivers, Client-Library and Server-Library provide a portable interface for implementing secure applications that work with several different network security systems.

To use a security mechanism on a connection, each item below must be true:

- The client and server must be configured to use compatible security drivers. For example, if the server runs on a Windows NT machine and uses the Microsoft SSPI driver for NT, then a Windows 95 client application must use the Microsoft SSPI driver for Windows 95.
- The client must request services by setting connection properties before connecting to the server.
- The underlying security mechanism must support the requested services.

Choosing a network security mechanism

The value of the CS_SEC_MECHANISM connection property determines the name of the security mechanism to be used to establish a connection. The default depends on the Sybase security driver configuration for your system.

Client-Library uses a driver configuration file to map security mechanism names to security driver file names. On most platforms, this file is named *libtcl.cfg*. For a full description of the driver configuration file, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

Determining the default security mechanism

The default security mechanism name corresponds to the first entry in the [SECURITY] section of the *libtcl.cfg* driver configuration file. This section has entries of the form:

```
[SECURITY]
 mechanism_name = driver_file_name init_string
 mechanism_name = driver_file_name init_string
```

where *mechanism_name* specifies a possible value for the CS_SEC_MECHANISM property, *driver_file_name* is a file name for the driver, and *init_string* specifies start-up settings for the driver.

If no driver configuration file is present on the system, or the file lacks a [SECURITY] section, the CS_SEC_MECH property defaults to NULL.

For a full description of the driver configuration on your system, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

Loading the default security driver

If an application does not request a driver by name, Client-Library loads the default security driver (if any) when needed. If a security driver is not loaded, ct_con_props or ct_config load the default driver when called with *action* as CS_SET or CS_SUPPORTED and any of the following values for *property*:

- CS_SEC_CHANBIND (only when setting to CS_TRUE)
- CS_SEC_CONFIDENTIALITY (only when setting to CS_TRUE)
- CS_SEC_CREDTIMEOUT
- CS_SEC_DATAORIGIN (only when setting to CS_TRUE)
- CS_SEC_DELEGATION (only when setting to CS_TRUE)
- CS_SEC_DETECTREPLAY (only when setting to CS_TRUE)
- CS_SEC_DETECTSEQ (only when setting to CS_TRUE)
- CS_SEC_INTEGRITY (only when setting to CS_TRUE)
- CS_SEC_KEYTAB
- CS_SEC_MECHANISM (CS_CLEAR always loads the default driver. CS_GET loads the default driver if no driver is loaded yet. CS_SET loads the requested driver.
- CS_SEC_MUTUALAUTH (only when setting to CS_TRUE)
- CS_SEC_NETWORKAUTH (only when setting to CS_TRUE)
- CS_SEC_SESSTIMEOUT

Global mechanism names

The security mechanism names in the driver configuration file are local names that may vary from system to system. For the client and the server to both determine the identity of the connection's security mechanism, they require invariant global names for security mechanisms.

When setting the CS_SEC_MECHANISM property or when loading the default security driver, Client-Library reads a configuration file, the global object identifiers file, to map local security mechanism names to object identifier (OID) strings. On most platforms, this file is called *objectid.dat*. Client-Library looks for security mechanism OIDs in the section [SECMECH]. The entries in this section have the form:

```
[SECMECH]
 mechanism_oid = local_name1, local_name2, ...
```

where *mechanism_oid* is the OID string that globally identifies the security mechanism and *local_name1*, *local_name2*, and so forth are local security provider names from the *libtcl.cfg* file. For more information on the global object identifiers file, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX.

Requesting network security services

Each security mechanism provides a set of security services. Each security service addresses some security concern. In a Client-Library application, the requested services correspond to context or connection properties.

Not all of the security services are supported by all security mechanisms. To find out whether a given service is supported by the current security mechanism, the application calls ct_config or ct_con_props with *action* as CS_SUPPORTED, *buffer* as the address of a CS_BOOL variable, and *property* as the symbolic property constant that represents the security service. **buffer* is set to CS_TRUE if the service is supported. ct_config and ct_con_props both fail when the application requests a service that is not supported by the current security mechanism.

Network security services are split into two categories:

- Login authentication services allow an application to establish a secure connection.
- Per-packet security services protect data transmitted over an established connection.

Requesting login authentication services

The fundamental security service is **login authentication**, or confirming that users are who they say they are. Login authentication involves user names and passwords. Users identify themselves by their user name, then supply their password as proof of their identity.

In Sybase applications, each connection between a client and a server has one user name associated with it. If the application uses a security mechanism, then Sybase uses the mechanism to authenticate this user name when the connection is established. The advantage of this service is that the user name/password pairs are managed in a central repository, and not in the system catalogs of individual servers.

When an application requests to connect to a server using network-based authentication, Client-Library queries the connection's security mechanism to confirm that the given user name represents the authenticated user that is running the application. This means that users do not have to supply a password to connect to the server. Instead, users prove their identity to the network security system before the connection attempt is made. When connecting, Client-Library obtains a **credential token** from the security mechanism and sends it to the server in lieu of a password. The server then passes the token to the security mechanism again to confirm that the user name has been authenticated.

The following connection properties are related to login authentication. To take effect, these properties must be set before a connection is established. At the connection level, all the following properties are retrieve-only when the connection is open:

- CS_USERNAME specifies the name of the user to connect with. If the application requests network-based authentication, then the user must be logged in to the network security system. Otherwise, the CS_PASSWORD property must be set to the user's server password.
- CS_SEC_NETWORKAUTH enables network-based authentication. The default is CS_FALSE, which means network-based authentication is disabled.
- CS_SEC_CREDTIMEOUT and CS_SEC_SESSTIMEOUT allow applications to specify or check whether the user's network credentials or security session have expired. Both apply only when network-based authentication is enabled on the connection.

The credential timeout period begins when the user obtains the credentials (that is, when the user logs in to the network). Some network security systems allow an administrator to specify a timeout value for user credentials. If the credentials expire, they are no longer valid. In addition, some systems allow applications to set credential timeout values.

The session timeout period begins when the connection is opened. Some network security systems allow an administrator to specify a timeout value for all security sessions. In addition, some systems allow applications to set session timeout values.

Table 2-32 lists the possible values for the credential and session timeout properties:

Table 2-32: Values for CS_SEC_SESSTIMEOUT and CS_SEC_CREDTIMEOUT

Value	Meaning
A positive integer	The number of seconds remaining before the credential expires.
0	The credential has expired.
CS_UNEXPIRED	The credential is valid. Remaining time is unknown.
CS_NO_LIMIT	The credential will not expire.

Some security mechanisms do not support credential or session timeouts. If either type is not supported, the retrieved timeout value is always CS_NO_LIMIT. Some security mechanisms support timeouts, but do not report timeout values to applications. With these mechanisms, the retrieved timeout value is always either CS_UNEXPIRED or 0.

Applications can request a different credential or session timeout value by setting the corresponding property to a positive integer or CS_NO_LIMIT. However, the security system's administrative settings restrict application-requested values. For example, if the system is configured so that all sessions timeout after 10 minutes, then an application's request for a 20-minute (1200-second) session timeout has no effect.

No error is raised if an application's request for a specific credential or session timeout value cannot be granted. If a connection's security mechanism does not support credential or session timeouts, then calls to set the CS_SEC_CREDTIMEOUT or CS_SEC_SESSTIMEOUT properties have no effect.

When the user's credential or session expires, the connection is closed either by Client-Library or the server, as follows:

- Client-Library checks for credential or session expiration prior to writing to the network, and closes the connection if the session has expired.
- The server checks for credential or session expiration before sending data to the client, and closes the connection if the session has expired. When the server closes the connection because of an expired session, the server does not send a warning message to the client.

- CS_SEC_MUTUALAUTH requests that the connection's security mechanism perform mutual authentication. For mutual authentication, the server is required to provide proof of its identity to the client before a connection is opened. The default is CS_FALSE, which means mutual authentication is not performed.

  When mutual authentication is requested, the server provides proof of its identity to the client when a connection is established. This proof consists of a credential token sent by the server to Client-Library. The token is an opaque chunk of data that encodes the server principal name and proof that the name is authentic. Client-Library queries the security mechanism to verify that the received token is genuine. If it is not, Client-Library aborts the connection attempt.

- CS_SEC_SERVERPRINCIPAL specifies the network security principal name for the server to which a connection will be opened. The default is NULL, which means ct_connect assumes that the server principal name matches the server's directory entry name. CS_SEC_SERVERPRINCIPAL is meaningful only when network-based authentication is requested.

- CS_SEC_DELEGATION determines whether the server is allowed to connect to a remote server using delegated credentials. The default is CS_FALSE, which means the credential delegation is not allowed.

  Delegation applies only to applications that use network-based user authentication to connect to an Open Server gateway.

  When a client connects to a gateway server, the gateway may establish a connection to a second, remote server that supports network-based authentication with an identical security mechanism. Credential delegation allows the gateway to connect to the remote server using the client's delegated credential.

- CS_SEC_CREDENTIALS allows a gateway application to forward user credentials to a remote server. The client application must have permitted credential delegation by setting the CS_SEC_DELEGATION connection property to CS_TRUE.

  Gateways support delegation by retrieving the value of the SRV_T_SEC_DELEGCRED Open Server thread property and setting the CS_SEC_CREDENTIALS Client-Library connection property to the retrieved value. The gateway's client, the gateway, and the gateway's remote server must use an identical security mechanism for delegation to work.

  The CS_SEC_CREDENTIALS property can only be set or cleared.

- CS_SEC_CHANBIND determines whether the connection's security mechanism performs channel binding. The default is CS_FALSE, which means channel binding is not performed.

  When channel binding is enabled, Client-Library and the server both provide a network channel identifier (consisting of the network addresses of the client and the server) to the connection's security mechanism.

- CS_SEC_KEYTAB specifies the name and path to an operating system file (called a **keytab file**) from which a connection's security mechanism reads the security key to go with the user name that is specified by the CS_USERNAME property.

  Note Only the DCE security driver supports keytab files.

  CS_SEC_KEYTAB is meaningful only for connections that use DCE as their security mechanism and that have requested network-based authentication. An application specifies a keytab file to connect to a server under a different user name than the DCE user that is running the application. The application sets the CS_USERNAME property to the new user name and sets CS_SEC_KEYTAB to indicate the keytab file that specifies the security key for the user. The default for CS_SEC_KEYTAB is NULL, which means that no keytab file is read, that CS_USERNAME must represent the DCE name of the application user, and that the user must already be logged into DCE.

  A keytab file is created with the DCE `dcecp` utility (see your DCE documentation). The keytab file must be readable by the user who is running the Client-Library application.

Requesting per-packet security services

In some environments, distributed application designers have to deal with the fact that the network is not physically secure. For example, unauthorized parties may listen to a dialog by attaching analyzers to a physical line or capturing wireless transmissions.

In these environments, applications require protection of transmitted data to assure a secure dialog. Per-packet security services protect transmitted data.

All per-packet services require that one or both of the following operations be performed for each TDS packet to be sent over a connection:

- Encryption of the packet's contents
- Computation of a digital signature that encodes the packet contents as well as other needed information

Note Applications that use the services described in this section incur a per-packet overhead on all communication between the client and the server. Do not use per-packet security services unless application security is more important than application performance.

If an application selects multiple per-packet services, each operation is performed only once per packet. For example, if the application selects the data confidentiality, sequence verification, data integrity, and channel binding services, then each packet is encrypted and accompanied by a digital signature that encodes the packet contents, packet sequence information, and a network channel identifier.

All per-packet services, except data confidentiality, require the connection's security mechanism to compute a digital signature for each packet that is sent over the connection. The signature encodes information about the packet's contents, and may encode other information as well. The client and the server both compute packet signatures and send them with each TDS packet. When the packet and signature are received, the security mechanism verifies the received signature. If packet signature is rejected, the connection is closed as follows:

- If the error occurs when Client-Library is reading results from the network, Client-Library raises an error and closes the connection.
- If the error occurs when the server is reading packets sent by the client, the server closes the connection. In this case, the client application will not discover the error until it tries to read from the network.

The following connection properties control the use of the per-packet services. To take effect, these properties must be set before a connection is established. At the connection level, all of the following properties are retrieve-only when the connection is open. All of the following properties take CS_BOOL **buffer* values, and all are CS_FALSE by default:

- CS_SEC_CONFIDENTIALITY requests encryption of all transmitted data. All commands sent to the server and all results returned by the server are encrypted.

  Data confidentiality protects data that is sent over public networks where the transmission medium is not physically secure. For example, strangers may attach analyzers to a physical line or capture wireless transmissions.

- CS_SEC_INTEGRITY requests that integrity checking be performed on all data transmitted over the connection. This service checks all TDS packets sent to the server and all sent from the server to assure that the contents were not modified.

  Data integrity checking is used only when the connection is also using network-based user authentication.

- CS_SEC_DATAORIGIN determines whether the connection's security mechanism performs data origin stamping. This service stamps each TDS packet transmitted over the connection with a digital signature that encodes information about the packet's sender and contents.

- CS_SEC_DETECTREPLAY determines whether the connection's security mechanism detects invalid repetition of transmitted TDS packets.

  Replay detection assures that attempts to capture packets and replay them are detected. For example, a stranger could capture the packets that represent a command sent to the server and replay them in an attempt to cause an unauthorized repeat of the command.

- CS_SEC_DETECTSEQ determines whether the connection's security mechanism detects transmitted TDS packets that arrive in a different order than the order in which they were sent.

  The replay detection and the sequence verification services are similar. However, they are distinct services. For example, consider the case where packets sent by the client are numbered in the sending order as P1, P2, P3, and so forth. If the server receives the packets in the order P1, P2, P2, then this is a replay error but not an out-of-sequence error. If the server receives the packets in the order P1, P3, P2, this is an out-of-sequence error but not a replay error.

Secure Sockets Layer in Open Client and Open Server

From Open Client and Open Server version 12.5, a network-library driver is added to enable Secure Sockets Layer (SSL), session-based security.

SSL is the standard for securing the transmission of sensitive information, such as credit card numbers, stock trades, and banking transactions, over the Internet.

While a comprehensive discussion on public-key cryptography is beyond the scope of this document, the fundamentals are worth describing so that you have an understanding of how SSL secures Internet communication channels. This document should not be considered comprehensive or complete.

The implementation of Open Client and Open Server SSL functionality assumes that there is a knowledgeable System Security Officer who is familiar with the security policies and needs of your site, and who has a general understanding of SSL and public-key cryptography.

Internet communications overview

TCP/IP is the primary transport protocol used in client/server computing and governs the transmission of data over the Internet. TCP/IP uses intermediate computers to transport communications from sender to recipient. The intermediate computers introduce weak links to the communication system where data may be subjected to tampering, theft, eavesdropping, and impersonation.

An SSL-enabled client application uses standard techniques of public-key cryptography to authenticate a server's certificate, and verify that the server certificate was issued by a trusted CA before sending private information, such as a credit card number, over the connection.

Public-key cryptography

To secure Internet communications, several mechanisms, known collectively as public-key cryptography, have been developed and implemented to protect sensitive data during transmission over the Internet. Public-key cryptography consists of data encryption, key exchange, digital signatures, and digital certificates.

Encryption

Encryption is a process wherein a cryptographic algorithm is used to encode information to safeguard it from anyone except the intended recipient. There are two types of keys used for encryption:

- Symmetric-key encryption is where the same algorithm (key) is used to encrypt and decrypt the message. This form of encryption provides minimal security because the key is simple, and therefore easy to decipher. However, transfer of data that is encrypted with a symmetric key is fast because the computation required to encrypt and decrypt the message are minimal.
- Public/private keys, also known as asymmetric keys, are a pair of keys that are made up of public and private components to encrypt and decrypt messages. Typically, the message is encrypted by the sender with a private key, and decrypted by the recipient with the sender's public key, although this may vary. It is quite possible to use a recipient's public key to encrypt a message, who then uses his private key to decrypt the message.

  The algorithms used to create public and private keys are more complex, and therefore harder to decipher. However, public/private key encryption requires more computation, sends more data over the connection, and noticeably slows the transfer of data.

Key exchange

The solution for reducing computation overhead and speeding transactions without sacrificing security is to use a combination of both symmetric key and public/private key encryption in what is known as a key exchange.

For large amounts of data, a symmetric key is used to encrypt the original message. The sender then uses either his private key or the recipient's public key to encrypt the symmetric key. Both the encrypted message and the encrypted symmetric key are sent to the recipient. Depending on what key was used to encrypt the message (public or private) the recipient uses the opposite to decrypt the symmetric key. Once the key has been exchanged, the recipient uses the symmetric key to decrypt the message.

Digital signatures

Digital signatures are used for tamper detection and non-repudiation. Digital signatures are created with a mathematical algorithm that generates a unique, fixed-length string of numbers from a text message; the result is called a hash or message digest.

To ensure message integrity, the message digest is encrypted by the signer's private key, then sent to the recipient along with information about the hashing algorithm. The recipient decrypts the message with the signer's public key. This process also regenerates the original message digest. If the digests match, the message proves to be intact and tamper free. If they do not match, the data has either been modified in transit or the data was signed by an imposter.

Further, the digital signature provides non-repudiation—senders are prevented from denying, or repudiating, that they sent the message, because the sender's private key encrypted the message. Obviously, if the private key has been compromised (stolen or deciphered), the digital signature is worthless for non-repudiation.

Certificates

Certificates are like passports: once you have been assigned one, the authorities have all your identification information in the system. Immigration control can access your information as you travel from country to country. Like a passport, the certificate is used to verify the identity of one entity (server, router, Web site, and so on) to another.

There are two types of certificates:

- Server certificates – A server certificate authenticates the server that holds it. Certificates are issued by a trusted third-party Certificate Authority (CA), much like the U.S. Department of State issues passports. The CA validates the holder's identity, and embeds the holder's public key and other identification information into the digital certificate. Certificates also contain the digital signature of the issuing CA, verifying the integrity of the data contained therein and validating its use.
- CA certificates – Also known as trusted root certificates, CA certificates are used by servers when they function as a client, such as during remote procedure calls (RPCs). When connecting to a remote server for RPCs, Adaptive Server verifies that the CA that signed the remote server's certificate is a "trusted" CA listed in its own CA trusted roots file. If it is not, the connection fails.

The combination of these mechanisms protect data transmitted over the Internet from eavesdropping and tampering. These mechanisms also protect users from impersonation, where one entity pretends to be another (spoofing), or where a person or an organization says it is set up for a specific purpose when the real intent is to capture private information (misrepresentation).

SSL overview

SSL is an industry standard for sending wire- or socket-level encrypted data over client-to-server and server-to-server connections. Before the SSL connection is established, the server and the client exchange a series of I/O round trips to negotiate and agree upon a secure encrypted session. This is called the SSL handshake.

SSL handshake

When a client application requests a connection, the SSL-enabled server presents its certificate to prove its identity before data is transmitted. Essentially, the SSL handshake consists of the following steps:

- The client sends a connection request to the server. The request includes the SSL (or Transport Layer Security, TLS) options that the client supports.
- The server returns its certificate and a list of supported CipherSuites, which includes SSL/TLS support options, the algorithms used for key exchange, and digital signatures.
- A secure, encrypted session is established when both client and server have agreed upon a CipherSuite.

For more specific information about the SSL handshake and the SSL/TLS protocol, see the Internet Engineering Task Force Web site at http://www.ietf.org.

Performance

There is additional overhead required to establish a secure session, because data increases in size when it is encrypted, and it requires additional computation to encrypt or decrypt information. Typically, the additional I/O accrued during the SSL handshake may make user login 10-20-times slower.

CipherSuites

During the SSL handshake, the client and server negotiate a common security protocol through a CipherSuite. CipherSuites are preferential lists of key-exchange algorithms, hashing methods, and encryption methods used by the SSL protocol. For a complete description of CipherSuites, go to the IETF organization Web site at http://www.ietf.org/rfc/rfc2246.txt.

By default, the strongest CipherSuite supported by both the client and the server is the CipherSuite that is used for the SSL-based session.

Server connection attributes are specified with directory services, such as LDAP or DCE, or with the traditional Sybase *interfaces* file.

Note The CipherSuites listed below conform to the TLS specification. TLS, or Transport Layer Security, is an enhanced version of SSL 5.0, and is an alias for the SSL version 5.0 CipherSuites.

Open Client and Open Server and Adaptive Server support the CipherSuites that are available with the SSL Plus library API and the cryptographic engine, Security Builder, both from Certicom Corp.

```
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_NULL_MD5
SSL_RSA_EXPORT_WITH_RC4_40_MD5
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA RSA
TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
TLS_DHE_DSS_EXPORT1024_WITH_RC4_56_SHA
TLS_DHE_DSS_WITH_RC4_128_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
```

SSL in Open Client and Open Server

SSL provides several levels of security:

- When establishing a connection to an SSL-enabled server, the server authenticates itself—proves that it is the server you intended to contact—and an encrypted SSL session begins before any data is transmitted.
- Once the SSL session is established, user name and password are transmitted over a secure, encrypted connection.
- A comparison of the server certificate's digital signature can determine if any information received from the server was modified in transit.

SSL filter

When establishing a connection to an SSL-enabled Adaptive Server, the SSL security mechanism is specified as a filter on the master and query lines in the *interfaces* file (*sql.ini* on Windows). SSL is used as an Open Client and Open Server protocol layer that sits on top of the TCP/IP connection.

The SSL filter is different from other security mechanisms, such as DCE and Kerberos, which are defined with SECMECH (security mechanism) lines in the *interfaces* file (*sql.ini* on Windows). The master and query lines determine the security protocols that are enforced for the connection.

For example, a typical *interfaces* file on a UNIX machine using SSL looks like the following:

```
[SERVER]
query tcp /dev/tcp add1 ssl
master tcp /dev/tcp add1 ssl
```

A typical *sql.ini* file on Windows using SSL looks like the following:

```
[SERVER]

query=TCP,hostname,address1, ssl
master=TCP,hostname,address1, ssl
```

where *hostname* is the name of the server to which the client is connecting and *address1* is the port number of the host machine. All connection attempts to a master or query entry in the *interfaces* file with an SSL filter must support the SSL protocol. A server can be configured to accept SSL connections and have other connections that accept plain text (unencrypted data), or use other security mechanisms.

For example, an Adaptive Server *interfaces* file on UNIX that supports both SSL-based connections and plain-text connections looks like:

```
SYBSRV1
    master tcp /dev/tcp \x00020abc1234567800000000000000000 ssl
    query tcp /dev/tcp \x00020abc1234567800000000000000000 ssl
    master tcp /dev/tcp \x00020abd1234567800000000000000000
```

Or, the same entry with the new style of Sybase *interfaces* file on UNIX looks like the following:

```
SYBSRV1
    master tcp hostname 2748 ssl
    query tcp hostname 2748 ssl
    master tcp hostname 2749
```

An example of a socket-style *interfaces* file looks like the following:

```
SYBSRV1
    master tcp ether hostname 2748 ssl
    query tcp ether hostname 2748 ssl
    master tcp ether hostname 2749
```

In these examples, the SSL security service is specified on port number 2748(0x0abc). On SYBSRV1, Adaptive Server listens for clear text on port number 2749(0x0abd), which has no security mechanism or security filter.

Validating the server by its certificate

Any Open Client/ Open Server connection to an SSL-enabled server requires that the server have a certificate file, which consists of the server's certificate and an encrypted private key. The certificate must also be digitally signed by a CA.

Open Client applications establish a socket connection to Adaptive Server similarly to the way that existing client connections are established. Before any user data is transmitted, an SSL handshake occurs on the socket when the network transport-level connect call completes on the client side and the accept call completes on the server side.

To make a successful connection to an SSL-enabled server:

- The SSL-enabled server must present its certificate when the client application makes a connection request.
- The client application must recognize the CA that signed the certificate. A list of all "trusted" CAs is in the trusted roots file. See "The trusted roots file" on page 270.
- For connections to SSL-enabled servers, the common name in the server's certificate must match the server name in the *interfaces* file as well.

Note You may choose to install SSL validation callback, which intercepts SSL handshakes and overrides SSL validation checks. SSL validation callback is installed with ct_callback using CS_SSLVALIDATE_CB.

When establishing a connection to an SSL-enabled Adaptive Server, Adaptive Server loads its own encoded certificates file at start-up from:

UNIX – *$SYBASE/$SYBASE_ASE/certificates/servername.crt*

Windows – *%SYBASE%\%SYBASE_ASE%\certificates\servername.crt*

where *servername* is the name of the Adaptive Server as specified on the command line when starting the server with the -S flag or from the server's environment variable $DSLISTEN.

Other types of servers may store their certificate in a different location. See the vendor-supplied documentation for the location of your server's certificate.

Validation in an SDC environment

The default behavior for SSL validation in Open Client and Open Server is to compare the common name in the server certificate with the server name specified by ct_connect. However, in a Shared Disk Cluster (SDC) environment, a client may specify the SSL certificate common name independent of the server name or the SDC instance name. A client may connect to an SDC by its cluster name, which represents multiple server instances, or to a specific server instance.

Open Client and Open Server support common name validation in an SDC environment by allowing the client to use a transport address to specify the common name used in certificate validation. The ASE SSL certificate common name can therefore be different from the server or cluster name. The transport address can be specified in one of the directory services like the *interfaces* file, an LDAP or NT registry, or through the connection property CS_SERVERADDR.

The following is an example of an *interfaces* file for an SSL-enabled ASE and cluster for UNIX:

```
CLUSTERSSL
query tcp ether hostname1 5000 ssl="CN=name1"
query tcp ether hostname2 5000 ssl="CN=name2"
query tcp ether hostname3 5000 ssl="CN=name3"

ASESSL1
master tcp ether hostname1 5000 ssl="CN=name1"
query tcp ether hostname1 5000 ssl="CN=name1"

ASESSL2
master tcp ether hostname2 5000 ssl="CN=name2"
query tcp ether hostname2 5000 ssl="CN=name2"

ASESSL3
master tcp ether hostname3 5000 ssl="CN=name3"
query tcp ether hostname3 5000 ssl="CN=name3"
```

The following is an example of an *interfaces* file for an SSL-enabled ASE and cluster for Windows:

```
[CLUSTERSSL]
query=tcp,hostname1,5000, ssl="CN=name1"
query=tcp,hostname2,5000, ssl="CN=name2"
query=tcp,hostname3,5000, ssl="CN=name3"

[ASESSL1]
master=tcp,hostname1,5000, ssl="CN=name1"
query=tcp,hostname1,5000, ssl="CN=name1"

[ASESSL2]
master=tcp,hostname2,5000, ssl="CN=name2"
query=tcp,hostname2,5000, ssl="CN=name2"

[ASESSL3]
master=tcp,hostname3,5000, ssl="CN=name3"
query=tcp,hostname3,5000, ssl"CN=name3"
```

The trusted roots file

The list of known and trusted CAs is maintained in the trusted roots file. The trusted roots file is similar in format to a certificate file, except that it contains certificates for CAs known to the entity (client applications, servers, network resources, and so on). The System Security Officer adds and deletes CAs using a standard ASCII-text editor.

The trusted roots file for Open Client and Open Server is as follows:

- For UNIX – *$SYBASE/config/trusted.txt*
- For Windows – *%SYBASE%\ini\trusted.txt*

Currently, the recognized CAs are Thawte, Entrust, Baltimore, VeriSign and RSA.

By default, Adaptive Server stores its own trusted roots file in:

- UNIX – *$SYBASE/$SYBASE_ASE/certificates/servername.txt*
- Windows – *%SYBASE%\%SYBASE_ASE%\certificates\servername.txt*

Both Open Client and Open Server allow you to specify an alternate location for the trusted roots file:

- Open Client:

```
ct_con_props (connection, CS_SET, CS_PROP_SSL_CA,
"$SYBASE/config/trusted.txt", CS_NULLTERM, NULL);
```

where *$SYBASE* is the installation directory. CS_PROP_SSL_CA can be set at the context level using ct_config, or at the connection level using ct_con_props.

- Open Server:

```
srv_props (context, CS_SET, SRV_S_CERT_AUTH,
"$SYBASE/config/trusted.txt", CS_NULLTERM, NULL);
```

where *$SYBASE* is the installation directory.

bcp and isql utilities also allow you to specify an alternative location for the trusted roots file.The parameter -x is included in the syntax, allowing you to specify an alternative location for the *trusted.txt* file.

Obtaining a certificate

The System Security Officer installs signed server certificates and private keys in the server. You can get a server certificate by:

- Using third-party tools provided with existing public-key infrastructure already deployed in the customer environment.
- Using the Sybase certificate request tool in conjunction with a trusted third-party CA.

To obtain a certificate, you must request a certificate from a CA. If you request a certificate from a third-party and that certificate is in PKCS #12 format, use the certpk12 utility to convert the certificate into a format that is understood by Open Client and Open Server.

To test the certificate request tool and to verify that the authentication methods are working on your server, Open Client and Open Server provides a certreq and certauth tool, for testing purposes, that allows you to function as a CA and issue a CA-signed certificate to yourself.

Following are the main steps to creating a certificate for use with a server:

1. Generate the certificate request.
2. Generate the public and private key pair.
3. Securely store the private key.
4. Send the certificate request to the CA.

5 After the CA signs and returns the certificate, append the private key to the certificate.

6 Store the certificate in the server's installation directory.

Third-party tools to request certificates

Most third-party PKI vendors and some browsers have utilities to generate certificates and private keys. These utilities are typically graphical wizards that prompt you through a series of questions to define a distinguished name and a common name for the certificate.

Follow the instructions provided by the wizard to create certificate requests. Once you receive the signed PKCS #12-format certificate, use certpk12 to generate a certificate file and a private key file. Concatenate the two files into a *servername.crt* file, where *servername* is the name of the server, and place it in the server's installation directory. By default, the certificates for Adaptive Server are stored in *$SYBASE/$SYBASE_ASE/certificates*.

Using Sybase tools to request and authorize certificates

Sybase provides tools for requesting and authorizing certificates. certreq generates public and private key pairs and certificate requests. certauth converts a server certificate request to a CA-signed certificate.

- UNIX – *$SYBASE/$SYBASE_OCS/bin*
- Windows – *%SYBASE%\%SYBASE_OCS%\bin*

Warning! Use certauth only for testing purposes. Sybase recommends that you use the services of a commercial CA because it provides protection for the integrity of the root certificate, and because a certificate that is signed by a widely accepted CA facilitates the migration to the use of client certificates for authentication.

Preparing a server's trusted root certificate is a five-step process. Perform all five steps to create a test trusted root certificate so you can verify that you are able to create server certificates. Once you have a test CA certificate (trusted roots certificate) repeat steps 3 through 5 to sign server certificates.

1 Use certreq to request a certificate.

2 Use certauth to convert the certificate request to a CA self-signed certificate (trusted root certificate).

3 Use certreq to request a server certificate and private key.

4 Use certauth to convert the certificate request to a CA-signed server certificate.

5 Append the private key text to the server certificate and store the certificate in the server's installation directory.

See "Using Sybase tools to request and authorize certificates" on page 272 for more information.

Note certauth and certreq are dependent on RSA and DSA algorithms. These tools only work with vendor-supplied crypto modules that use RSA and DSA algorithms to construct the certificate request.

For information on adding, deleting, or viewing server certificates on Adaptive Server, see the *System Administration Guide*.

Adaptive Server security features

Client applications that connect to Adaptive Server or Open Server version 10.0 or later can take advantage of password encryption and challenge/response security handshakes.

Security handshaking: Challenge/Response

Servers use challenge/response security handshaking to provide an additional level of login security checking.

To provide the response that this handshake method requires, an application must be coded as follows:

- Before calling ct_connect, the application must call ct_con_props to set one of the following properties:
  - CS_SEC_CHALLENGE to request Sybase-defined challenge/response security handshaking.
  - CS_SEC_APPDEFINED to request Open Server application-defined challenge/response security handshaking.

  If either or both of these properties is CS_TRUE, ct_connect invokes the application's negotiation callback in response to server challenges.
- The application must contain a negotiation callback that is coded to return the required response.

- The application calls ct_callback to install the callback either at the context level or for a specific connection.

See “Defining a negotiation callback” on page 46.

Security handshaking: encrypted password

Sybase Servers uses encrypted password handshakes if the client requests password encryption. Encrypted password security handshaking occurs while the connection to the server is being established.

Note Applications must request password encryption by setting the CS_SEC_EXTENDED_ENCRYPTION or CS_SEC_ENCRYPTION connection property to CS_TRUE (the default is CS_FALSE). Otherwise, the password is sent to the server as plain text.

The password encryption process

When password encryption is enabled, the server receives the user passwords and remote-server passwords as follows:

1. Client-Library initially sends a dummy password to the server consisting of a zero-length string.
2. The server responds by asking the client for the encrypted passwords and sending an encryption key to the client.
   - If the client program has installed an encryption callback, Client-Library invokes the callback once for the local password and once for each remote-server password. Each time Client-Library invokes the encryption callback, it supplies the password to be encrypted and the encryption key as arguments.
   - If the client program has not installed an encryption callback, Client-Library performs the default encryption for all passwords.

Using password encryption in Client-Library applications

Password encryption is disabled by default, so applications that need password encryption must set the CS_SEC_EXTENDED_ENCRYPTION or CS_SEC_ENCRYPTION property to CS_TRUE before calling ct_connect. Below are sample codes you can use to enable password encryption.

Enabling normal password encryption

```
CS_BOOL boolval;
/* Enable password encryption for the connection attempt. */
boolval = CS_TRUE;

if (ct_con_props(conn, CS_SET, CS_SEC_ENCRYPTION, (CS_VOID *)&boolval,
    CS_UNUSED,(CS_INT *)NULL) != CS_SUCCEED)
{
    fprintf(stdout,"ct_con_props(SEC_ENCRYPTION) failed. Exiting\n");
    (CS_VOID)ct_con_drop(conn);
    (CS_VOID)ct_exit(ctx, CS_FORCE_EXIT);
    (CS_VOID)cs_ctx_drop(ctx);
    exit(1);
}
```

Enabling extended password encryption

```
...
CS_INT Ex_encryption = CS_TRUE;
CS_INT Ex_nonencryptionretry = CS_FALSE;
...
main()
{
    ...
    /*
    ** This needs to be called before calling ct_connect()
    */
    ret = ct_con_props(connection, CS_SET, CS_SEC_EXTENDED_ENCRYPTION,
                       &Ex_encryption, CS_UNUSED, NULL);
    EXIT_ON_FAIL(context, ret, "Could not set extended encryption");

    ret = ct_con_props(connection, CS_SET, CS_SEC_NON_ENCRYPTION_RETRY,
                       &Ex_nonencryptionretry, CS_UNUSED, NULL);
    EXIT_ON_FAIL(context, ret, "Could not set non encryption retry");

    ....
}
```

Password encryption is performed either by Client-Library's default encryption handler or by an application handler installed with ct_callback.

The default encryption handler performs the encryption expected by Adaptive Server. Applications that connect to Adaptive Server or an Open Server gateway to Adaptive Server should rely on the default encryption. Most applications fall into this category.

Applications that require an encryption handler include the following:

- Open Server gateways that connect to an Adaptive Server must support password encryption with an encryption callback that obtains encrypted passwords from the gateway's client (through srv_negotiate) and forwards each password to the remote server (through the callback's output parameters).
- Client applications that require a custom password encryption technique (for example, applications that connect to a custom Open Server) must install a custom encryption callback that performs the encryption expected by the server.

For information about defining a password encryption callback, see "Defining an encryption callback" on page 42.

Server directory object

The **server directory object** is a generalized description of the logical content of directory entries that describe Sybase servers.

For more information on directories, see the "Directory services" on page 104.

Use of the server directory object

Server directory objects are implicitly accessed when connecting to a server with ct_connect. An application can also search for server entries in a directory using ct_ds_lookup and a directory callback.

Client-Library applications inspect the contents of a directory object using ct_ds_objinfo.

Contents of the server directory object

Client-Library maps server entries in the directory onto the server directory object described here. The server directory object provides a view of directory entries that is independent of their actual storage format. The object is defined as a set of attributes for which a server entry can contain values.

The actual storage format of directory entries varies depending on the directory service being used. Each **directory driver** converts entries from their native storage format into the Server Directory Object format. The object format provides a generic view of directory entries to Client-Library applications.

Format of object attributes

Each directory object specifies the set of attributes that are stored in a directory entry of that type. Attributes have metadata and one or more values. An attribute's metadata is represented by a CS_ATTRIBUTE structure, and consists of:

- A name that identifies the attribute

  Because attribute-naming schemes can vary among directory providers, Client-Library uses an object identifier (or OID) to identify each attribute. Client-Library provides a predefined OID-string macro for each attribute.

- A value syntax specifier

  This is an integer code that identifies which C datatype holds the attribute's values.

- The number of values in this instance of the attribute

  Values are retrieved with a CS_ATTRVALUE union. Applications use the syntax specifier to know which member of the union holds the value.

See "Retrieving object attributes and attribute values" on page 472 for a description of the CS_ATTRIBUTE and CS_ATTRVALUE structures.

List of attributes

Table 2-33 summarizes the attributes of the server directory object and gives the syntax and OID string for each. Detailed descriptions follow the table.

Note Applications that inspect server directory objects with ct_ds_objinfo should be coded to accept unexpected attributes. Sybase may add attributes to the server directory object that are not listed here.

Table 2-33: Attributes of the server directory object

Attribute and corresponding OID string	Value syntax	Description
Server entry version CS_OID_ATTRVERSION	Integer	The server's version level.
Server name attribute CS_OID_ATTRSERVNAME	String	The server's name. The value of the name attribute can differ from the fully qualified name for the server's directory entry.
Service description CS_OID_ATTRSERVICE	String	A description of the service provided by the server. The value may be any meaningful description.
Server status CS_OID_ATTRSTATUS	Integer	The operating status of the server. See "Server status" on page 279 for possible values and their meanings. **Note** Adaptive Server version 11.0 or earlier always has an unknown status.
Transport address CS_OID_ATTRADDRESS	Transport Address	One or more transport addresses for the server. The transport address attribute has three elements: • Transport type • Access type • Transport address
Security mechanisms CS_OID_ATTRSECHMECH	OID	The security mechanisms supported by the server or servers. This attribute is optional.

Server entry version

The server entry version holds a symbolic integer code for the server's software version. The version attribute is provided for the convenience of directory users

The version attribute is for administrative use only; the value of the attribute does not affect any capabilities of a connection to the server.

Server name attribute

The server name attribute provides a server name that will be visible to applications that search the directory with ct_ds_lookup.

The name can be any string that is CS_MAX_DS_STRING or fewer bytes long. By convention, the name attribute should match the name the server uses for itself (for Adaptive Servers, the local server name is given by sp_addname).

Do not confuse a server's name attribute with the name used to locate the directory entry. The latter is the fully qualified name for the directory entry, expressed in the name syntax of the directory provider. ct_connect uses the fully qualified name to find the directory entry. The name attribute is an arbitrary string value provided for the convenience of directory users. To avoid confusion, the directory administrator should ensure that the name attribute at least partially matches the server's fully qualified name (for example, the attribute value could be the entry's common name).

Note When the directory provider is the *interfaces* file, the value of the name attribute is the same as the entry's name.

Service description

The service description attribute describes the service that the server provides. The service type value can be any string that is CS_MAX_DS_STRING or fewer bytes long.

When the Sybase *interfaces* file is the directory source, this value is always "Adaptive Server".

Server status

The server status is a symbolic integer code which describes the operating status of the server. Possible values are listed in Table 2-34.

Table 2-34: Status attribute values (server directory object)

Status value	Meaning
CS_STATUS_ACTIVE	Server is up and running.
CS_STATUS_STOPPED	Server has been taken offline and has not been restarted.
CS_STATUS_FAILED	Server is offline because of an error.
CS_STATUS_UNKNOWN	Status of the server is unknown. See "Unknown status values" on page 280 for an explanation.

Unknown status values

The value of the status attribute may be unknown for the following reasons:

- The server is an Adaptive Server – Adaptive Server version 11.0 or earlier does not register its status with the directory service. The status attribute for Adaptive Server directory entries is always CS_STATUS_UNKNOWN.
- Use of the *interfaces* file in directory lookups – if the directory object being inspected came from the *interfaces* file, the status attribute is always unknown. The *interfaces* file does not support status attributes, so the status attribute defaults to CS_STATUS_UNKNOWN when ct_ds_lookup retrieves file entries and converts them to directory objects.
- Unregistered Open Server Applications – an Open Server application registers itself to use the directory service as part of its initialization.

  If an Open Server registers itself, then Server-Library automatically sets the status attribute value to reflect the current operating condition of the server. If the application does not register itself, its status attribute value will always be CS_STATUS_UNKNOWN.

  For more information on how an Open Server registers with the directory service, see the Open Server *Server Library/C Reference Manual*.

Transport address

The transport address attribute is used by ct_connect to establish a connection to the server. The transport address attribute may have multiple transport address values.

Note In an SDC environment in which the client specifies the SSL certificate common name independent of the server name or the SDC instance name, the client uses the transport address to specify the common name used in the certificate validation.

Client-Library applications view the transport address value as a CS_TRANADDR structure. For details on the format of the structure, see "Transport address values" on page 474.

Multiple transport address types

The server may allow connecting over multiple network transport types. Your network installation and the Sybase network driver configuration determines which transport types are used by Client-Library on your system. For more information, see the Open Client and Open Server *Configuration Guide* for Microsoft Windows or Open Client and Open Server *Configuration Guide* for UNIX more information.

Standby server addressing

The server entry may contain multiple address values for use with your network configuration. In this case, ct_connect tries to connect to each eligible address in turn, repeating if necessary, until one of the following conditions are satisfied:

- A connection dialog is accepted at a given address.
- Each address has been tried *retry_count* times, where *retry_count* is the value of the CS_RETRY_COUNT connection property.

The CS_LOOP_DELAY connection property sets a time in seconds for Client-Library to wait before beginning the sequence again. Client-Library does not wait between trying individual addresses in the sequence.

See "Retry count" on page 227 and "Loop delay" on page 217 for a description of the CS_RETRY_COUNT and CS_LOOP_DELAY properties.

Security mechanisms

The security mechanism attribute is an optional, multivalued attribute that contains one or more OID strings for Sybase security mechanisms supported by the server.

Client applications specify a connection's security mechanism by setting the CS_SEC_MECHANISM connection property (or accepting the default). See "Choosing a network security mechanism" on page 253.

Security mechanism OIDs are mapped to local security mechanism names by the Sybase global objects file. See "Global mechanism names" on page 255.

If the security mechanism attribute is present in a server's directory entry, then clients connect to the indicated server using only the listed services. If no security mechanism attribute is present, then clients connect using any mechanism that the server is configured to support.

Server objects from the interfaces file

An application that is not configured to read from a network-based directory will read server directory objects from the Sybase *interfaces* file.

See "Server objects from the Interfaces file" on page 144 for a description of how Client-Library maps *interfaces* file entries to the server directory object.

See the "Interfaces file" on page 142 for general information about the Sybase *interfaces* file.

Server restrictions

Client-Library is a generic programming interface. This means that it is functionally independent of the servers to which it connects. Such independence allows Open Client applications to communicate with not only Adaptive Server, and Open Server applications, but also with non-Sybase servers if the Open Server application is a gateway.

Being functionally independent means that Open Client has no knowledge of the way in which a server may choose to implement certain functionality. It is possible that the same feature, implemented by multiple servers, will exhibit various different behaviors. The behavior of a server feature is specific to the server currently being accessed.

As an Open Client application developer, you should have a thorough understanding of the behavior of the server(s) for which you are writing an application. This includes knowing what functionality is supported and what restrictions are enforced.

Open Server restrictions

Open Client and Open Server do not inherit Adaptive Server restrictions. This means that communication between Open Client applications and Open Server applications is not constrained by rules that govern Sybase server behavior.

Communication *is* constrained, however, by the implementation of the Open Server application. For example, an Open Server application developer may decide not to support remote procedure calls (RPCs) by not installing a SRV_RPC event handler. This is a constraint of which an Open Client application developer must be aware.

Open Client and Open Server are mirror images of each other. Open Server can receive anything that Open Client sends, and vice versa. Restrictions arise not only when implementation-specific limitations are imposed on an Open Server application, but when functionality available in Open Server is not enabled.

Adaptive Server restrictions

It is only when an Open Client application accesses Adaptive Server that the application must be aware of Adaptive Server restrictions. For example, Adaptive Server has login name requirements: the login name must follow the rules for Adaptive Server identifiers and it must be unique. When an Open Client application accesses an Adaptive Server, it must adhere to such requirements.

Note Sybase SQL Server changed names to Adaptive Server Enterprise at version level 11.5. The following restriction also apply to SQL Server.

Following are some important Adaptive Server restrictions:

- Dynamic SQL is implemented using temporary stored procedures, and therefore inherits the restrictions of stored procedures.
- Long variable-length binary datatypes and long variable-length character datatypes are not supported.

- By definition, a cursor is associated with only one select statement. This means that a stored procedure on which a Client-Library cursor is declared contains only a single statement: a select statement.
- Stored procedures do not support parameters of datatype text or image.
- Event notifications are not supported.
- Message commands are not supported.
- The POSIX locale method of localization is not supported.

Supported client/server features

To ascertain some of the client and server features supported by a particular connection, an application calls ct_capability. The ct_capability *value* parameter returns information about whether the capability is enabled.

This retrieves, among other things:

- What datatypes are supported
- What types of requests are valid

For more information about getting (and setting) client and server features, see the ct_capability reference page.

text and *image* data handling

text and image are Adaptive Server datatypes that hold large text or image values. The text datatype holds up to 2,147,483,647 bytes of printable characters. The image datatype holds up to 2,147,483,647 bytes of binary data.

Because they can be so large, text and image values are not actually stored in database tables. Instead, a **text pointer** to the text or image value is stored in the table.

To ensure that competing applications do not wipe out one another's modifications to the database, a timestamp is associated with each text or image column. This timestamp is called a **text timestamp**.

Client-Library stores the text pointer and text timestamp for a text or image column in an I/O descriptor structure, the CS_IODESC. The I/O descriptor for a column also contains other information about the column, including its name and datatype.

For detailed information about the CS_IODESC structure, see "CS_IODESC structure" on page 91.

Retrieving a *text* or *image* column

An application retrieves text and image columns in two ways:

- It selects the columns, binds the columns, and fetches rows. In other words, an application retrieves and process text and image columns in the same way it retrieves and processes any other type of column.
- It selects the columns, uses ct_fetch to loop through result rows, and uses ct_get_data to retrieve data in the text and image columns. An application uses this method when processing text or image values that are too large for convenient binding.

Using *ct_get_data* to fetch *text* and *image* values

Only columns that follow the last column bound with ct_bind are available for use with ct_get_data.

For example, if an application selects four columns, all of which are text, and binds the first and third columns to program variables, then the application cannot use ct_get_data to retrieve the text contained in the second column. However, it can use ct_get_data to retrieve the text in the fourth column. Applications that control the select statement can reorder the select list so that the text and image columns come at the end.

To retrieve a text or image value using ct_get_data, an application follows these steps:

1. Executes a command that generates a result set that contains text or image columns.

   An application uses a language command, RPC command, or dynamic SQL command to generate a result set containing text or image columns.

   For example, the *pic* column in the *au_pix* table of the *pubs2* database contains authors' pictures. To retrieve them, an application might execute the following language command:

```
ct_command(cmd, CS_LANG_CMD,
         "select pic from au_pix”,
         CS_NULLTERM, CS_UNUSED);
 ct_send(cmd);
```

2 Processes the result set containing the text or image column.

An application uses ct_fetch to loop through the rows contained in the result set. Inside the loop, for each unbound text or image column:

- The application calls ct_get_data in a loop to retrieve the text or image data for the column.
- The application calls ct_get_info to get an I/O descriptor that updates the column at a later time.

Most applications use a program structure similar to the following:

```
while ct_fetch is returning rows
     process any bound columns
     for each unbound text or image column
         while ct_get_data is returning data
             process the data
         end while
         ct_data_info to get the column’s CS_IODESC
     end for
 end while
```

Alternatively, for each unbound text or image column, an application:

- Calls ct_get_data with the parameter *buflen* as 0, so that it returns no data but does refresh the I/O descriptor for the column.
- Calls ct_get_data to get the I/O descriptor for the column. The *total_txtlen* field in this structure represents the total length of the text or image value.
- Calls ct_get_data as many times as necessary to retrieve the value.

This method has the advantage of allowing an application to determine the total length of a text or image value before retrieving it.

Updating a *text* or *image* column

Text or image columns are updated two ways:

- Embed the new value in the text of an update language command. The advantage of this method is simplicity. The disadvantage is that the application must send the entire value at once. This method may not be appropriate for very large columns (that is, larger than the program can allocate space for). Adaptive Server requires the value to be embedded in the command text, and not passed as a command parameter. Adaptive Server does not allow parameters of type text or image.
- Initiate a send-data command (with ct_command) and send the value in chunks with ct_send_data). This method handles values that are larger than the program's buffer space, but it is more complicated. This method may be more natural than the embedded method for applications that read the value in chunks from an external source such as an operating system file.

An application only updates a text or image column using ct_send_data if it has defined (using ct_data_info) current I/O descriptor settings for the column that it intends to update. The I/O descriptor settings are contained in a CS_IODESC structure. (See "CS_IODESC structure" on page 91.) Adaptive Server requires a correctly initialized I/O descriptor to perform the update, and the client application must retrieve the required I/O descriptor settings from the server.

Retrieving the I/O descriptor settings

An application retrieves the I/O descriptor settings by calling ct_data_info. If the Adaptive Server is version 11.0 or later, select the I/O descriptor directly using the server *ss*.

To retrieve the current I/O descriptor with ct_data_info, an application must first select the column of interest in the row of interest. While processing the row results returned from the select, the application gets the I/O descriptor as follows:

1. Calls ct_fetch to fetch the row of interest.
2. Calls ct_get_data to retrieve the column's value and refresh the I/O descriptor for the column. To refresh the I/O descriptor without retrieving any data for the column, call ct_get_data with *buflen* as 0.
3. Calls ct_data_info to retrieve the I/O descriptor.

Beginning with version 11.0, Adaptive Server, provides an alternative method for setting the I/O descriptor fields. This method is used by applications that only update one text or image column at a time. See "Using global variables to update a text or image column" on page 292.

Sending the new column value

Once it has the current I/O descriptor for a column value, the application performs the update:

1. Calls ct_command to initiate a send-data command.
2. Modifies the I/O descriptor, if necessary. Most applications change only the values of the *locale*, *total_txtlen*, or *log_on_update* fields.
3. Calls ct_get_data to set the I/O descriptor for the column value. The *textptr* field of the I/O descriptor structure identifies the target column of the send-data operation.
4. Calls ct_send_data in a loop to write the entire text or image value. Each call to ct_send_data writes a portion of the text or image value.
5. Calls ct_send to send the command.
6. Calls ct_results to process the results of the command. An update of a text or image column generates a a parameter result set containing a single parameter, the new text timestamp for the value. If the application plans to update this column value again, it must save the new timestamp and copy it into the CS_IODESC for the column value before calling ct_data_info (step 3, above) to set the I/O descriptor for the new update.

Most applications use a program structure similar to the following to update text or image columns:

```
ct_con_alloc to allocate connection1 and connection2
ct_cmd_alloc to allocate cmd1 and cmd2

ct_command(cmd1) to select columns
     (including text) from table
ct_send to send the command
while ct_results returns CS_SUCCEED
     (optional) ct_res_info to get description of result set
     (optional) ct_describe to get descriptons of columns
     (optional) ct_bind if binding any columns

     while ct_fetch(cmd1) returns rows
         for each text column
             /* Retrieve the current CS_IODESC for the column */
             if you want the column’s data, loop on ct_get_data
                     while there’s data to retrieve
             if you don’t want the column’s data, call
                     ct_get_data once with buflen of 0 to
                     refresh the CS_IODESC
             ct_data_info(cmd1, CS_GET) to get the CS_IODESC
```

```
            /* Update the column */
            ct_command(cmd2) to initiate a send-data command
            if necessary, modify fields in the CS_IODESC
            ct_data_info(cmd2, CS_SET) to set the CS_IODESC for
                    the column
            while there is data to send
                ct_send_data(cmd2) to send a chunk of data
            end while
            ct_send(cmd2) to send the send-data command
            ct_results(cmd2) to process the send-data results
        end for
    end while
end while
```

Partial updates to *text* and *image* data

Open Client supports the partial update of text and image columns. A partial update allows you to specify the part of the text or image field that you want to replace, delete, or insert, and to update that part only instead of modifying the entire field.

Note Currently, ASE does not support partial updates of text or image data. However, Open Server does support partial updates of text or image data. For more information, see Chapter 2, “Topics“ in the Open Server *Server-Library/C Reference Manual*.

To perform a partial update, use ct_data_info to set *iotype*, *delete_length*, and *offset*. The values of *delete_length* and of the data passed to the server through ct_send_data determine the behavior of the partial update:

delete_length	**Text data**	**Server action**
0	Provided	Insert the text data at *offset*.
!= 0	Provided	Starting from *offset*, overwrite *delete_length* bytes of data with text data.
!= 0	Not provided	Starting at *offset*, delete *delete_length* bytes of data.
NULL	Provided / Not provided	Delete data starting from *offset* to the end of the text or image column.

Sending partial updates with *ct_send_data*

The ct_send_data routine can be used to send partially updated data. For partially updated data, the ct_send_data constructs a Transact-SQL updatetext statement, and the data is sent in chunks using multiple ct_send_data calls. The updatetext syntax is:

```
updatetext table_name.column_name text_pointer
   {NULL | offset} {NULL | delete_length} [with_log]
```

Note The updatetext statement is created only if the *iotype* value of the CS_IODESC structure is set to is set to CS_IOPARTIAL.

Handling of unitext data

If your client application performs partial updates on 2-byte Unicode datatypes, the application must make sure that it sends an even number of bytes to avoid a character split. You can use the *buflen* parameter of ct_send_data and the *total_txtlen* field of CS_IODESC to specify the length, in bytes, of the Unicode data. For Unitext, the *offset* and *delete_length* values must be specified as a character count while *total_txtlen* must be specified in bytes. For other datatypes, the *offset*, *delete_length*, and *total_txtlen* must be in bytes.

Populating a table containing *text* or *image* columns

An application's method of populating a table containing text or image columns depends on the size of the data values to be inserted.

Smaller *text* and *image* values

Most applications embed text or image values of less than 100K in an insert statement:

```
insert blurbs values (“486-29-1786”, “If Chastity
     Locksley didn’t exist, this troubled...")
insert au_pix values ("486-29-1786", 0x67f44c...,
     "ICT", "30220", "626", "635")
```

Larger *text* and *image* values

The following method is recommended for populating an Adaptive Server table with text or image values larger than 100K:

1. insert all data into the row except the text or image values.
2. update the row, setting the value of the text or image columns to NULL. This step is necessary because a text or image column row that contains a null value will have a valid text pointer only if the null value was explicitly entered with the update statement.
3. Retrieve I/O descriptor settings for the column. This is done two ways:
   - Select the text or image column of interest, then call ct_data_info after the column's value has been retrieved. For a description of this method, see "Retrieving the I/O descriptor settings" on page 287. This method works with all Sybase Servers that support the text and image datatypes.
   - Use the text and image global variables provided by Adaptive Server. For a description of this method, see "Using global variables to update a text or image column" on page 292. This method requires Adaptive Server version 11.0 or later.
4. Update the columns as described in "Sending the new column value" on page 288.

Server global variables for *text* and *image* updates

Adaptive Server version 11.0 and later have global variables specifically for text and image support. These variables are:

Variable	Explanation	Datatype
@@textptr	The text pointer of the last text or image column inserted or updated by a process.	binary(16)
@@textts	Text timestamp of the column referenced by *@@textptr*.	varbinary(8)
@@textcolid	ID of the column referenced by *@@textptr*.	tinyint
@@textdbid	ID of the database containing the object with the column referenced by *@@textptr*.	smallint
@@textobjid	ID of the object containing the column referenced by *@@textptr*.	int

Each connection to Adaptive Server has its own instance of these variables. The variables are set at 0 at the beginning of a session. Adaptive Server updates the variables:

- When the application updates a text or image data column during the session. If multiple text or image columns in the same row are updated at the same time, the variables describe the last text/image column in the row that was updated.
- When the application inserts a row containing a non-NULL text or image value. If multiple non-NULL text or image columns are inserted in the same row, the variables describe the last non-NULL text or /. column in the row.

Using global variables to update a *text* or *image* column

In applications that only insert or update one text or image column at a time, the text/image global variables provide an alternative way to fill in the I/O descriptor fields required for updating a text or image column with ct_send_data.

As mentioned in "Updating a text or image column" on page 286, ct_data_info cannot be called to set an I/O descriptor's fields until after the application has selected and retrieved the text or image column of interest. Instead of calling ct_data_info, the application retrieves the text and image global variables and uses their values to fill in the I/O descriptor. To do this, the application must:

- Issue a language command to update the column or to insert a new row.
- In the same language batch, select the current values of the text and image global variables.
- Process the results and retrieve the values into the I/O descriptor fields.

Most applications follow the steps below to perform a text/image update using the Server text and image global variables:

1. Call ct_command to initiate a language command containing an update or insert statement that causes Adaptive Server to place the desired I/O descriptor values in the text and image global variables. This is done by sending a language command that updates the column to a dummy value. The command must also select Transact-SQL expressions that are appropriate for the *textptr*, *timestamp*, and the *name* fields of the CS_IODESC structure. For example, if the key of the my_table table is the int_col column, an appropriate language command batch is:

```
update my_table set text_col = NULL
         where int_col = 23
 if @@rowcount != 0
    select @@textptr,
           @@textts,
```

```
colname = object_name(@@textobjid) +
'.' + col_name(@@textobjid,
               @@textcolid,
                @@textdbid)
```

For inserts of a new row, the update is preceded or replaced by an insert command in the same batch. If the insert command specifies NULL for the text or image column, it must be followed by an update that updates the column to NULL. Otherwise, the server does not update the *@@text* variables to describe the column. An insert that specifies a non-NULL value for the text or image column need not be followed by an update.

If the update in the example command above succeeds, the required information for the I/O descriptor is selected and returned as three columns. The first column is the text pointer value, the second is the new timestamp, and the third is a string of the form *table_name.column*.

2 Process the results in a ct_results loop.

The selected expressions are returned as regular result rows (result type CS_ROW_RESULT). The application calls ct_bind to bind the values to the fields of a CS_IODESC structure and retrieve the values with ct_fetch. The application binds the structure fields according to the following table:

CS_IODESC field	Column value
timestamp, timestamplen	Call ct_bind to bind *timestamp* to *@@textts* and pass the address of *timestamplen* as ct_bind's *copied* parameter.
textptr, textptrlen	Call ct_bind to bind to*@@textptr* and pass the address of *textptrlen* as ct_bind's *copied* parameter.

name, namelen	Call ct_bind to bind *name* to the value returned for:
	`object_name(@@textobjid) +`
	`"." + col_name(@@textobjid,`
	`@@textcolid,`
	`@@textdbid)`
	In the ct_bind call, pass the address of *namelen* as the ct_bind's *copied* parameter when binding to the *name* field.

3 Set all remaining I/O descriptor fields to appropriate values:

```
iodesc->iotype = CS_IODATA;
 iodesc->usertype = 0;
 iodesc->offset = 0;
 iodesc->locale = (CS_LOCALE *) NULL;
 iodesc->total_txtlen = length_of_new_value;
 iodesc->log_on_update = CS_TRUE; /* or CS_FALSE */
```

After following these steps, the application is ready to send the new text or image value as described under “Sending the new column value” on page 288.

Datatypes support

Client-Library supports a wide range of datatypes. These datatypes are shared with Open Client CS-Library and Server-Library. In most cases, they correspond directly to Adaptive Server datatypes.

Table 2-35 lists Open Client and Open Server type constants, their corresponding C datatypes, and their corresponding Adaptive Server, if any.

Following Table 2-35 is a list of Open Client routines that are useful in manipulating datatypes and more detailed information about each datatype.

For additional information about datatypes, see Chapter 3, “Using Open Client and Open Server Datatypes,” in the Open Client *Client-Library/C Programmers Guide*.

Datatype summary

Table 2-35 lists Open Client and Open Server type constants, their corresponding C datatypes, and their corresponding Adaptive Server datatypes, if any:

Table 2-35: Datatype summary

Type category	Open Client and Open Server type constant	Description	Corresponding C datatype	Corresponding server datatype
Binary types	CS_BINARY_TYPE	Binary type	CS_BINARY	binary, varbinary
	CS_LONGBINARY_TYPE	Long binary type	CS_LONGBINARY	None
	CS_VARBINARY_TYPE	Variable-length binary type	CS_VARBINARY	None
Bit types	CS_BIT_TYPE	Bit type	CS_BIT	bit

Type category	Open Client and Open Server type constant	Description	Corresponding C datatype	Corresponding server datatype
Character types	CS_CHAR_TYPE	Character type	CS_CHAR	char, varchar
	CS_LONGCHAR_TYPE	Long character type	CS_LONGCHAR	None
	CS_VARCHAR_TYPE	Variable-length character type	CS_VARCHAR	None
	CS_UNICHAR_TYPE	Fixed-length or variable-length character type	CS_UNICHAR	unichar univarchar
XML type	CS_XML_TYPE	Variable-length character type	CS_XML	xml
Datetime types	CS_DATE_TYPE	4-byte date type	CS_DATE	date
	CS_TIME_TYPE	4-byte time type	CS_TIME	time
	CS_DATETIME_TYPE	8-byte datetime type	CS_DATETIME	datetime
	CS_DATETIME4_TYPE	4-byte datetime type	CS_DATETIME4	smalldatetime
Numeric types	CS_TINYINT_TYPE	1-byte unsigned integer type	CS_TINYINT	tinyint
	CS_SMALLINT_TYPE	2-byte integer type	CS_SMALLINT	smallint
	CS_INT_TYPE	4-byte integer type	CS_INT	int
	CS_BIGINT_TYPE	8-byte integer type	CS_BIGINT	bigint
	CS_USMALLINT_TYPE	2-byte unsigned integer type	CS_USMALLINT	usmallint
	CS_UINT_TYPE	4-byte unsigned integer type	CS_UINT	uint
	CS_UBIGINT_TYPE	8-byte unsigned integer type	CS_UBIGINT	ubigint
	CS_DECIMAL_TYPE	Decimal type	CS_DECIMAL	decimal
	CS_NUMERIC_TYPE	Numeric type	CS_NUMERIC	numeric
	CS_FLOAT_TYPE	8-byte float type	CS_FLOAT	float
	CS_REAL_TYPE	4-byte float type	CS_REAL	real
Money types	CS_MONEY_TYPE	8-byte money type	CS_MONEY	money
	CS_MONEY4_TYPE	4-byte money type	CS_MONEY4	smallmoney

Type category	Open Client and Open Server type constant	Description	Corresponding C datatype	Corresponding server datatype
Text and image types	CS_TEXT_TYPE	Text type	CS_TEXT	text
	CS_UNITEXT_TYPE	Variable-length character type	CS_UNITEXT	unitext
	CS_IMAGE_TYPE	Image type	CS_IMAGE	image

Routines that manipulate datatypes

Open Client CS-Library provides several routines that are useful for manipulating datatypes. They include:

- cs_calc, which performs arithmetic operations on decimal, money, and numeric datatypes
- cs_cmp, which compares datetime, decimal, money, and numeric datatypes
- cs_convert, which converts a data value from one datatype to another
- cs_dt_crack, which converts a machine readable datetime value into a user-accessible format
- cs_dt_info, which sets or retrieves language-specific datetime information
- cs_strcmp, which compares two strings

These routines are documented in the Open Client and Open Server *Common Libraries Reference Manual*.

Open Client datatypes

This section describes the datatypes in Open Client, and provides definitions for the datatypes.

Binary types

Open Client has three binary types, CS_BINARY, CS_LONGBINARY, and CS_VARBINARY.

- CS_BINARY corresponds to the Adaptive Server types *binary* and *varbinary*. That is, Client-Library interprets both the server *binary* and *varbinary* types as CS_BINARY. For example, ct_describe returns CS_BINARY_TYPE when describing a result column that has the server datatype *varbinary*.

  CS_BINARY is defined as:

  ```
  typedef unsigned char    CS_BINARY;
  ```

  Warning! CS_LONGBINARY and CS_VARBINARY do not correspond to any Adaptive Server datatypes.

- Some Open Server applications may support CS_LONGBINARY. An application uses the CS_DATA_LBIN capability to determine whether an Open Server connection supports CS_LONGBINARY. If it does, then ct_describe returns CS_LONGBINARY when describing a result data item.

  A CS_LONGBINARY value has a maximum length of 2,147,483,647 bytes. CS_LONGBINARY is defined as:

  ```
  typedef unsigned char    CS_LONGBINARY;
  ```

- CS_VARBINARY does not correspond to any Adaptive Server type. For this reason, Open Client routines do not return CS_VARBINARY_TYPE. CS_VARBINARY is provided to enable non-C programming language veneers to be written for Open Client. Typical client applications will not use CS_VARBINARY.

  CS_VARBINARY is defined as:

  ```
  typedef struct _cs_varybin
   {
       CS_SMALLINT      len;
       CS_BYTE          array[CS_MAX_CHAR];
   } CS_VARBINARY;
  ```

  where:

  - *len* is the length of the binary array.
  - *array* is the array itself.

Although CS_VARBINARY variables are used to store variable-length values, CS_VARBINARY is considered to be a fixed-length type. This means that an application does not typically need to provide Client-Library with the length of a CS_VARBINARY variable. For example, ct_bind ignores the value of *datafmt->maxlength* when binding to a CS_VARBINARY variable.

Bit types

Open Client supports a single bit type, CS_BIT. This type is intended to hold server bit (or boolean) values of 0 or 1. When converting other types to bit, all non-zero values are converted to 1:

```
typedef unsigned char    CS_BIT;
```

Character types

Open Client has four character types, CS_CHAR, CS_LONGCHAR, CS_VARCHAR, and CS_UNICHAR:

- CS_CHAR corresponds to the Adaptive Server types *char* and *varchar*. That is, Client-Library interprets both the server *char* and *varchar* types as CS_CHAR. For example, ct_describe returns CS_CHAR_TYPE when describing a result column that has the server datatype *varchar*.

  CS_CHAR is defined as:

  ```
  typedef char    CS_CHAR;
  ```

  Warning! CS_LONGCHAR and CS_VARCHAR do not correspond to any Adaptive Server datatypes. Specifically, CS_VARCHAR does not correspond to the Adaptive Server datatype varchar.

- CS_LONGCHAR does not correspond to any Adaptive Server type, but some Open Server applications support CS_LONGCHAR. An application uses the CS_DATA_LCHAR capability to determine whether an Open Server connection supports CS_LONGCHAR. If it does, then ct_describe returns CS_LONGCHAR when describing a result data item.

  A CS_LONGCHAR value has a maximum length of 2,147,483,647 bytes. CS_LONGCHAR is defined as:

  ```
  typedef unsigned char    CS_LONGCHAR;
  ```

- CS_VARCHAR does not correspond to any Adaptive Server type. For this reason, Open Client routines do not return CS_VARCHAR_TYPE. CS_VARCHAR is provided to enable non-C programming language veneers to be written for Open Client. Typical client applications will not use CS_VARCHAR.

  CS_VARCHAR is defined as:

  ```
  typedef struct _cs_varchar
   {
       CS_SMALLINT      len;
       CS_CHAR          str[CS_MAX_CHAR];
   } CS_VARCHAR;
  ```

  where:

  - *len* is the length of the string.
  - *str* is the string itself. Note that *str* is not null-terminated.

    Although CS_VARCHAR variables are used to store variable-length values, CS_VARCHAR is considered to be a fixed-length type. This means that an application does not typically need to provide Client-Library with the length of a CS_VARCHAR variable. For example, ct_bind ignores the value of *datafmt–>maxlength* when binding to a CS_VARCHAR variable.

- CS_UNICHAR corresponds to the Adaptive Server unichar fixed-width and univarchar variable-width datatypes. CS_UNICHAR is a shared, C-programming datatype that can be used anywhere the CS_CHAR datatype is used. The CS_UNICHAR datatype stores character data in the two-byte Unicode UTF-16 format.

  CS_UNICHAR is defined as follows:

  ```
  typedef unsigned char       CS_UNICHAR;
  ```

XML type

CS_XML corresponds directly to Adaptive Server xml variable-length datatype. CS_XML can be used anywhere CS_TEXT and CS_IMAGE are used to represent XML documents and contents.

CS_XML is defined as follows:

```
typedef unsigned char       CS_XML
```

Datetime types

Open Client supports four datetime types, CS_DATE, CS_TIME, CS_DATETIME, and CS_DATETIME4. These datatypes are intended to hold 4-byte and 8-byte datetime values.

An Open Client application uses the CS-Library routine cs_dt_crack to extract date parts (year, month, day, etc.) from a datetime structure.

- CS_DATE corresponds to the Adaptive Server date datatype. The range of legal CS_DATE values is from January 1, 0001 to December 31, 9999. The definition of CS_DATE is:

  ```
  typedef CS_INT CS_DATE; /* 4-byte date type*/
  ```

- CS_TIME corresponds to the Adaptive Server time datatype. The range of legal CS_TIME values is from 12:00:00.000 to 11:59:59:999 with a precision of 1/300th of a second (3.33 ms.). The definition of CS_TIME is:

  ```
  typedef CS_INT CS_TIME; /* 4-byte time type*/
  ```

- CS_DATETIME corresponds to the Adaptive Server datetime datatype. The range of legal CS_DATETIME values is from January 1, 1753 to December 31, 9999, with a precision of 1/300th of a second (3.33 ms.). The definition of CS_DATETIME is:

  ```
  typedef struct _cs_datetime
   {
       CS_INT    dtdays;
       CS_INT    dttime;
   } CS_DATETIME;
  ```

  where:

  - *dtdays* is the number of days since 1/1/1900.
  - *dttime* is the number of 300ths of a second since midnight.

- CS_DATETIME4 corresponds to the Adaptive Server *smalldatetime* datatype. The range of legal CS_DATETIME4 values is from January 1, 1900 to June 6, 2079, with a precision of 1 minute. The definition of CS_DATETIME is:

  ```
  typedef struct _cs_datetime4
   {
       CS_USHORT    days;
       CS_USHORT    minutes;
   } CS_DATETIME4;
  ```

  where:

- *days* is the number of days since 1/1/1900.
- *minutes* is the number of minutes since midnight.

Integer types

Open Client supports seven integer types: CS_TINYINT, CS_SMALLINT, CS_INT, CS_BIGINT, CS_USMALLINT, CS_UINT, and CS_UBIGINT.

Integer types include CS_TINYINT, a 1-byte integer; CS_SMALLINT, a 2-byte integer, CS_INT, a 4-byte integer, CS_BIGINT, an 8-byte integer, CS_USMALLINT, an unsigned 2-byte integer, CS_UINT, an unsigned 4-byte integer and CS_UBIGINT, an unsigned 8-byte integer:

```
typedef unsigned char      CS_TINYINT;
typedef short              CS_SMALLINT;
typedef int                CS_INT;
typedef long long          CS_BIGINT;
typedef unsigned char      CS_USMALLINT;
typedef unsigned int       CS_UINT;
typedef unsigned long long CS_UBIGINT;
```

Real, float, numeric, and decimal types

- CS_REAL corresponds to the Adaptive Server datatype *real*. It is implemented as a C-language *float* type:

  ```
  typedef float    CS_REAL;
  ```

  Note When converting 6-digit precision bigint or ubigint datatypes to *real* datatypes, note the following maximum and minimum values:

  - -9223370000000000000.0 < bigint < 9223370000000000000.0
  - 0 < ubigint < 18446700000000000000.0

  Values outside of these ranges cause overflow errors.

- CS_FLOAT corresponds to the Adaptive Server datatype *float*. It is implemented as a C-language *double* type:

```
typedef double   CS_FLOAT;
```

Note When converting 15-digit precision bigint or ubigint datatypes to *float* datatypes, note the following maximum and minimum values:

- -9223372036854770000.0 < bigint < 9223372036854770000.0
- 0 < ubigint < 18446744073709500000.0

Values outside of these ranges cause overflow errors.

- CS_NUMERIC and CS_DECIMAL correspond to the Adaptive Server datatypes *numeric* and *decimal*. These types provide platform-independent support for numbers with precision and scale.

Warning! For output parameters CS_DECIMAL and CS_NUMERIC in Client-Library and ESQL/C programs, the precision and scale must be defined before making a call to ct_param. This is required because the output parameters have no values associated with them at definition time and have an invalid precision and scale associated with them. You must do one of the following before calling ct_param:

```
CS_NUMERIC numeric_vaa
..numeric_var.precision =18;
numeric..
ct_param (..)
```

Failure to initialize the values will result in an invalid precision or scale message.

The Adaptive Server datatypes numeric and decimal are equivalent; and CS_DECIMAL is defined as CS_NUMERIC:

```
typedef struct_cs_numeric
 {
     CS_BYTE         precision;
     CS_BYTE         scale;
     CS_BYTE         array[CS_MAX_NUMLEN];
 } CS_NUMERIC;

 typedef CS_NUMERIC   CS_DECIMAL;
```

where:

- *precision* is the maximum number of decimal digits that can be represented by the corresponding number of digits in base-256 numbering. For example, four digits of decimal precision (0-9999) can be represented by two base-256 digits. At the current time, legal values for *precision* are from 1 to 77. The default precision is 18. CS_MIN_PREC, CS_MAX_PREC, and CS_DEF_PREC define the minimum, maximum, and default precision values, respectively.
- *array* is a base-256 representation of the numeric value. The byte at index 0 denotes the sign, where 0 (or a byte value of 00000000) represents a positive number, and 1 (or a byte value of 0000001) represents a negative number. The remaining bytes, 1-n, represent the base-256 number in little-endian order, with the byte at index 1 being the most significant byte.

  The number of bytes used in *array* is based on the selected precision of the numeric. Mapping is performed based on the precision of the numeric to the length of array that is used.
- *scale* is the maximum number of digits to the right of the decimal point. At the current time, legal values for *scale* are from 0 to 77. The default scale is 0. CS_MIN_SCALE, CS_MAX_SCALE, and CS_DEF_SCALE define the minimum, maximum, and default scale values, respectively.
- *scale* must be less than or equal to *precision*.

  CS_DECIMAL types use the same default values for *precision* and *scale* as CS_NUMERIC types.

Money types

Open Client supports two money types, CS_MONEY and CS_MONEY4. These datatypes are intended to hold 8-byte and 4-byte money values, respectively.

- CS_MONEY corresponds to the Adaptive Server *money* datatype. The range of legal CS_MONEY values is between +$922,337,203,685,477.5807 and -$922,337,203,685,477.5807:

```
typedef struct _cs_money
 {
     CS_INT      mnyhigh;
     CS_UINT     mnylow;
 } CS_MONEY;
```

- CS_MONEY4 corresponds to the Adaptive Server *smallmoney* datatype. The range of legal CS_MONEY4 values is between -$214,748.3648 and +$214,748.3647:

```
typedef struct _cs_money4
 {
     CS_INT    mny4;
 } CS_MONEY4;
```

Text and image types

Open Client supports text datatypes, CS_TEXT and CS_UNITEXT, as well as an image datatype, CS_IMAGE.

- CS_TEXT corresponds to the Adaptive Server datatype text, which describes a variable-length column containing up to 2,147,483,647 bytes of printable character data. CS_TEXT is defined as unsigned character:

```
typedef unsigned char    CS_TEXT;
```

- CS_UNITEXT corresponds to the Adaptive Server unitext variable-length datatype. CS_UNITEXT exhibits identical syntax and semantics to CS_TEXT. The difference is that CS_UNITEXT encodes character data in the 2-byte Unicode UTF-16 format. CS_UNITEXT can be used anywhere CS_TEXT is used. The maximum length of the CS_UNITEXT string parameter is half of the maximum length of CS_TEXT.

  CS_UNITEXT is defined as follows:

```
typedef unsigned short       CS_UNITEXT;
```

- CS_IMAGE corresponds to the Adaptive Server datatype image, which describes a variable-length column containing up to 2,147,483,647 bytes of binary data. CS_IMAGE is defined as unsigned character:

```
typedef unsigned char    CS_IMAGE;
```

Open Client user-defined datatypes

An application that needs to use a datatype that is not included in the standard Open Client type set may create a user-defined datatype.

A Client-Library application creates a user-defined type by declaring it:

```
typedef char CODE_NAME;
```

The Open Client routines ct_bind and cs_set_convert use integer symbolic constants to identify datatypes, so 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 CODE_NAME_TYPE (CS_USERTYPE + 2)
```

Once a user-defined type has been created, an application:

- Calls cs_set_convert to install custom conversion routines to convert between standard Open Client types and the user-defined type
- Calls cs_setnull to define a null substitution value for the user-defined type.

After conversion routines are installed, an application binds server results to a user-defined type:

```
mydatafmt.datatype = CODE_NAME_TYPE;
 ct_bind(cmd, 1, &mydatafmt, mycodename, NULL, NULL);
```

Custom conversion routines are called transparently, whenever required, by ct_fetch (following a call to ct_bind specifying the conversion) and cs_convert.

Note Do not confuse Open Client user-defined types with Adaptive Server user-defined types. Open Client user-defined types are C-language types, declared within an application. Adaptive Server user-defined types are database column datatypes, created using the system stored procedure sp_addtype.

Using the runtime configuration file

By default, Client-Library reads the Open Client and Open Server runtime configuration file to set runtime values for the following:

- Property values (normally set with ct_config or ct_con_props calls)
- Server option values (normally set with ct_options calls after a connection is opened)
- Server capabilities (normally set with ct_capability calls before a connection is opened)
- Debugging options (normally set with ct_debug calls)

Applications that read the configuration file to apply these settings eliminate several calls to ct_con_props or the other routines mentioned above. Another benefit is that the application's runtime settings are changed without recompiling code.

The environment variable, SYBOCS_DBVERSION, allows you to externally configure the DB-Library version level at runtime. It does this by calling dbsetversion, thereby changing the application code.

Note If there is an external Sybase configuration file, add these sections to enable bcp and isql:

[BCP]

[isql]

Enabling debugging

Table 2-36 lists the keywords for configuring the debugging options for a connection.

Table 2-36: Configuration file keywords for debugging options

Keyword	Value
CS_DBG_FILE	A character string specifying the destination file name for text-format debugging information.
CS_DBG_PROTOCOL_FILE	This ct_debug parameter may be set without *devlib* libraries. If the parameter is not set on connection, mktemp is called, generating a unique file name to dump the protocol packets into. The prefix string passed to mkstemp is *capture*. Ribo can decode the resulting protocol file.
CS_PROTOCOL_FILE	A character string specifying the destination file name for binary format debugging information.
CS_DEBUG	A character string giving a comma-separated list of debug flags.

CS_DEBUG specifies the data to be written to the file CS_DBG_FILE. Its value can be a list of flags that correspond to the bitmasks for ct_debug's *flag* parameter. For meanings of these debug flags, see "ct_debug" on page 442.

The possible flags are:

- CS_DBG_ALL
- CS_DBG_API_LOGCALL

- CS_DBG_API_STATES
- CS_DBG_ASYNC
- CS_DBG_DIAG
- CS_DBG_ERROR
- CS_DBG_MEM
- CS_DBG_NETWORK
- CS_DBG_PROTOCOL
- CS_DBG_PROTOCOL_FILE
- CS_DBG_PROTOCOL_STATES

Enabling external configuration

The following properties control the use of the Open Client and Open Server runtime configuration file:

- *CS_EXTERNAL_CONFIG* – when this property is CS_TRUE at the context level, ct_init reads default Client-Library context property values from the Open Client and Open Server runtime configuration file.

  At the context level, CS_EXTERNAL_CONFIG defaults to CS_TRUE if the default Open Client and Open Server runtime configuration file exists, and to CS_FALSE otherwise. The name of the external configuration file is determined by the CS_CONFIG_FILE property. Applications can override the context-level default by calling cs_config.

  At the connection level, allocated connection structures inherit CS_EXTERNAL_CONFIG from the parent context. If CS_EXTERNAL_CONFIG is CS_TRUE at the connection level, ct_connect reads default connection properties, capabilities, server options, and debugging options from the Open Client and Open Server runtime configuration file.

- *CS_CONFIG_FILE* – specifies the name and location of the Open Client and Open Server runtime configuration file. CS_CONFIG_FILE is set at the context level with cs_config or at the connection level with ct_con_props. The default value is NULL, which means that a platform-specific default file will be used:

  - On UNIX platforms, the default configuration file is *$SYBASE/$SYBASE_OCS/config/ocs.cfg*.

$SYBASE is the path to the Sybase installation directory, specified in the SYBASE environment variable. *$SYBASE_OCS* is the Open Client and Open Server subdirectory, specified in the SYBASE_OCS environment variable.

- On Windows platforms, the default configuration file is *%SYBASE%\%SYBASE_OCS%\ini\ocs.cfg*.

  %SYBASE% is the path to the Sybase installation directory, specified in the SYBASE environment variable. *%SYBASE_OCS%* is the Open Client and Open Server subdirectory, specified in the SYBASE_OCS environment variable.

For other platforms, see the Open Client and Open Server *Configuration Guide* for the name of the default Open Client and Open Server runtime configuration file.

- *CS_CONFIG_BY_SERVERNAME* – controls whether ct_connect uses the value of the connection's CS_APPNAME property or the server name as the file section name. By default, the value of CS_APPNAME is used. CS_CONFIG_BY_SERVERNAME is set at the connection level with ct_con_props.

  For example, if external configuration is enabled for the connection, the application name is "Monthly Report," and the value of *server_name* is "FinancialDB," then:

  - If CS_CONFIG_BY_SERVERNAME is CS_FALSE, ct_connect looks for a section labeled [Monthly Report].
  - If CS_CONFIG_BY_SERVERNAME is CS_TRUE, ct_connect looks for a section name labeled [FinancialDB].

  Note CS_SERVERNAME cannot be changed in the external configuration file unless CS_CONFIG_BY_SERVERNAME is set to CS_TRUE.

  The server and application names are changed by the configuration section. This allows an administrator to override a server or application name that is hard-coded in the application. For example, if the application is set up to read the section name FinancialDB, the section could contain the following:

```
[FinancialDB]
 CS_APPNAME = "Monthly Financial Report"
 CS_SERVERNAME = "Dev_FinancialDB" ; redirect to
                                   ; development
```

```
                                                    ; server
```

- *SYBOCS_CFG* – specifies the configuration file to be used, overriding the default *ocs.cfg* file located as follows:
  - In UNIX: *$SYBASE/$SYBASE_OCS/config/ocs.cfg*
  - In Windows: *%SYBASE%\%SYBASE_OCS%\ini\ocs.cfg*
- *CS_APPNAME* – at the context level, specifies from which section of the file to read values. Applications call cs_config to set CS_APPNAME at the context level. If the application does not set CS_APPNAME for the context structure, ct_init looks for a section labeled [DEFAULT]. At the connection level, ct_connect reads the file section indicated by CS_APPNAME when external configuration is enabled and the CS_CONFIG_BY_SERVERNAME property has its default value of CS_FALSE.

Open Client and Open Server runtime configuration file syntax

The Open Client and Open Server runtime configuration file is a text file. The file is separated into sections, each of which begins with a section name enclosed in square brackets ([]) and ends with the next section name or the end of the file, whichever appears first.

Each section contains one or more settings, as illustrated below:

```
[section name]
 keyword = value ; comment
 keyword = value
; more comments
[next section name]
... and so forth ...
```

In general, all supported keywords in the file match the names of the symbolic constants that would identify the property, option, or capability in a Client-Library/C program. However, not all properties can be set in the configuration file. If a keyword is not supported, the setting is ignored.

The syntax is as follows:

- ; – Signifies a comment line.

- [section_name] – Section names are wrapped in square brackets. The Open Client/Server configuration file comes with a section named DEFAULT. The application name will be used as the section name for an application that has been compiled with the -x option. For an application that has been compiled with the -e option, the server name will be used for the section name. Any name can be used as a section name for the sections that contain settings that will be used in multiple sections. The following example shows a section arbitrarily named “GENERIC,” and how that section is included in other sections:

```
[GENERIC]
    CS_OPT_ANSINULL=CS_TRUE

[APP_PAYROLL]
    include=GENERIC
    CS_CAP_RESPONSE=CS_RES_NOSTRIPBLANKS

[APP_HR]
    include=GENERIC
    CS_OPT_QUOTED_IDENT=CS_TRUE
```

- entry_name=entry_value
  - Entry values can be anything: integers, strings and so on. If an entry value line ends with '\'<newline>, the entry value continues to the next line.
  - White spaces are trimmed from the beginning and end of entry values.
  - If white spaces are required at the beginning or end of an entry value, wrap them in double quotes.
  - An entry that begins with a double quote must end with a double quote. Two double quote characters in a row within a quoted string represent a single double quote in the value string. If a newline is encountered within double quotes, it is considered to be literally part of the value.
  - Entry names and section names can consist of alphabetic characters (both upper and lower case), the digits 0-9, and any of the following punctuation characters:

    ! " # $ % & ' ( ) * + , - . / : ; < > ? @ \ ^ _ ` { | } ~

    Square brackets, spaces, and the equal symbol (=) are not supported. The first letter MUST be alphabetic.
  - Entry and section names are case sensitive.
  - Include=*earlier_section*

If a section contains the entry include, then the entire contents of that previously defined section are considered to be replicated within this section. In other words, the properties defined in the previous section are inherited by this section.

Note The included section must have been defined prior to it being included in another section. This allows the configuration file parsing to happen in a single pass and eliminates the need to detect recursive included directives.

If an included section in turn includes another section, the order of entry values is defined by a "depthfirst" search of the included sections.

Sections cannot include a reference to themselves. In other words, recursion is not possible because you must include a previously defined section—you cannot include the section being defined.

All direct entry values defined in a given section supersede any values which may have been included from another section. In the following example, CS_OPT_ANSINULL will be set to false in the APP.PAYROLL application.

Note The position of the include statement does not affect this rule.

```
[GENERIC]
 CS_OPT_ANSINULL=CS_TRUE
[APP_PAYROLL]
 CS_OPT_ANSINULL=CS_FALSE
 include=GENERIC
```

- If an entry's value in a C program takes symbolic constants, then the legal values are the names of these constants. For example:

  ```
  CS_NETIO = CS_SYNC_IO
  ```

- If an entry's value in a C program takes integer values, then legal values match the legal range of integer values. For example:

  ```
  CS_TIMEOUT = 60
  ```

- If an entry's value in a C program takes boolean values, then legal values are CS_TRUE and CS_FALSE. For example:

  ```
  CS_DIAG_TIMEOUT = CS_TRUE
  ```

- If an entry's value in a C program takes character strings, then the string is typed directly into the file. For example:

  ```
  CS_USERNAME = winnie
  ```

  Some string values must be quoted. If a string contains leading white space, trailing white space, or the semicolon (;) comment character, then the value must be quoted. Also, null string values must be indicated by consecutive quotes. For example:

  ```
  CS_APPNAME = "   Monthly report; Financials   "
   CS_PASSWORD = ""
  ```

  Long string values are continued on a subsequent line by escaping the end-of-line with a backslash (\). If an unescaped end-of-line occurs in a quoted string, it is read as part of the value. Finally, literal backslashes in a string value must be doubled.

- If a property's value in a C program takes pointers to a datatype other than CS_CHAR, the property cannot be set through external configuration. The sole exception is the CS_LOCALE keyword, which has the same effect as configuring a CS_LOCALE structure and installing it as a context or connection's CS_LOC_PROP property. For example, this line would assign the French locale to the context or connection:

  ```
  CS_LOCALE = french
  ```

- If a keyword occurs twice in a section, only the first definition is used.

- A section can include the keywords in another section using this syntax:

  ```
  [section name]
   include = previous section name
  ... more settings ...
  ```

  All settings defined under an included section name are defined in a section that includes that section. An included setting is always replaced by an explicit setting in the including section. For example, the Finance section, below, defines CS_TIMEOUT as 30. The included setting from the DEFAULT section is replaced by an explicit setting:

  ```
  [DEFAULT]
   CS_TIMEOUT = 45

   [Finance]
   include = DEFAULT
  ```

```
CS_TIMEOUT = 30
```

Runtime configuration file keywords

The tables below contain the legal keywords for configuring Client-Library's runtime behavior and the recognized values for each.

Keywords for localization

The following table describes the keywords for configuring a context or a connection's locale. These settings replace calls necessary to set the CS_LOC_PROP property for a context or connection.

Keyword	Legal value
CS_LOCALE	Any locale name defined in the locales file for the host platform.

Keywords for context or connection properties

When the application calls them, ct_init and ct_connect each read a section of the configuration file if the application has requested external configuration.

If a context property is set when ct_init reads a section, then any calls to ct_con_props to set the same property override the configured setting.

If a property is set when ct_connect reads a section, then calls to ct_con_props to set the same property either:

- Get replaced by the file's value if the ct_con_props call occurs before ct_connect, or
- Replace the file's value if the ct_con_props call occurs after ct_connect.

For example, values for CS_USERNAME and CS_PASSWORD that are set in a configuration section always override hard-coded values in the application code. This is true because the application must set these properties before ct_connect is called.

Table 2-37 lists the keywords that set context or connection properties. For descriptions of what each property controls, see "Properties" on page 180.

Table 2-37: Configuration file keywords to set properties

Keyword	Read by	Legal value
CS_ANSI_BINDS	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_APPNAME	ct_connect	A character string
CS_ASYNC_ NOTIFS	ct_connect	CS_TRUE or CS_FALSE
CS_BULK_LOGIN	ct_connect	CS_TRUE or CS_FALSE
CS_CON_KEEPALIVE	ct_connect	CS_TRUE or CS_FALSE
CS_CON_TCP_NODELAY	ct_connect	CS_TRUE or CS_FALSE
CS_DIAG_TIMEOUT	ct_connect	CS_TRUE or CS_FALSE
CS_DISABLE_POLL	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_DS_COPY	ct_connect	CS_TRUE or CS_FALSE
CS_DS_DITBASE	ct_connect	A character string
CS_DS_FAILOVER	ct_connect	CS_TRUE or CS_FALSE
CS_DS_PASSWORD	ct_connect	A character string
CS_DS_PRINCIPAL	ct_connect	A character string
CS_DS_PROVIDER	ct_connect	A character string
CS_DS_RAND_OFFSET	ct_config, ct_con_props	CS_TRUE or CS_FALSE
CS_EXPOSE_FMTS	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_EXTENDED_ENCRYPT_CB	ct_connect	CS_TRUE or CS_FALSE
CS_EXTRA_INF	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_HAFAILOVER	ct_config, ct_con_props	CS_TRUE or CS_FALSE
CS_HIDDEN_KEYS	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_HOSTNAME	ct_connect	A character string
CS_IFILE	ct_init	A character string
CS_LOGIN_TIMEOUT	ct_init	An integer value
CS_LOOP_DELAY	ct_connect	An integer value
CS_MAX_CONNECT	ct_init	An integer value
CS_NETIO	ct_init, ct_connect	CS_SYNC_IO, CS_ASYNC_IO, or CS_DEFER_IO
CS_NOAPI_CHK	ct_init	CS_TRUE or CS_FALSE
CS_NO_TRUNCATE	ct_init	CS_TRUE or CS_FALSE

Keyword	Read by	Legal value
CS_NOINTERRUPT	ct_init	CS_TRUE or CS_FALSE
CS_PACKETSIZE	ct_connect	An integer value
CS_PASSWORD	ct_connect	A character string
CS_PROP_EXTENDEDFAILOVER	ct_config, ct_con_props	CS_TRUE or CS_FALSE
CS_PROP_REDIRECT	ct_config, ct_con_props	CS_TRUE or CS_FALSE
CS_RETRY_COUNT	ct_connect	An integer value
CS_SEC_APPDEFINED	ct_connect	CS_TRUE or CS_FALSE
CS_SEC_CHALLENGE	ct_connect	CS_TRUE or CS_FALSE
CS_SEC_CHANBIND	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_SEC_CONFIDENTIALITY	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_SEC_CREDTIMEOUT	ct_init, ct_connect	A positive integer or CS_NO_LIMIT
CS_SEC_DATAORIGIN	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_SEC_DELEGATION	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_SEC_DETECTREPLAY	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_SEC_DETECTSEQ	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_SEC_ENCRYPTION	ct_connect	CS_TRUE or CS_FALSE
CS_SEC_EXTENDED_ENCRYPTION	ct_connect	CS_TRUE or CS_FALSE
CS_SEC_INTEGRITY	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_SEC_KEYTAB	ct_connect	A character string
CS_SEC_MECHANISM	ct_init, ct_connect	A character string
CS_SEC_MUTUALAUTH	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_SEC_NETWORKAUTH	ct_init, ct_connect	CS_TRUE or CS_FALSE
CS_SEC_NON_ENCRYPTION_RETRY	ct_connect	CS_TRUE or CS_FALSE
CS_SEC_SERVERPRINCIPAL	ct_connect	A character string

Keyword	Read by	Legal value
CS_SEC_SESSTIMEOUT	ct_init, ct_connect	A positive integer or CS_NO_LIMIT.
CS_TDS_VERSION	ct_connect	CS_TDS_40, CS_TDS_42, CS_TDS_46, or CS_TDS_50
CS_TEXTLIMIT	ct_init, ct_connect	An integer value or CS_NO_LIMIT
CS_TIMEOUT	ct_init	An integer value
CS_USERNAME	ct_connect	A string value

Keywords for server options

Table 2-38 lists the keywords for configuring the server options for a connection.

Application calls to ct_options always override equivalent settings in the configuration file.

The keywords for setting server options are listed in Table 2-38. For descriptions of what each option controls, see “Options” on page 174.

Table 2-38: Configuration file keywords for server options

Keyword	Legal value
CS_OPT_ANSINULL	CS_TRUE or CS_FALSE
CS_OPT_ANSIPERM	CS_TRUE or CS_FALSE
CS_OPT_ARITHABORT	CS_TRUE or CS_FALSE
CS_OPT_ARITHIGNORE	CS_TRUE or CS_FALSE
CS_OPT_AUTHOFF	A string value
CS_OPT_AUTHON	A string value
CS_OPT_CHAINXACTS	CS_TRUE or CS_FALSE
CS_OPT_CURCLOSEONXACT	CS_TRUE or CS_FALSE
CS_OPT_CURREAD	A string value.
CS_OPT_CURWRITE	A string value
CS_OPT_DATEFIRST	CS_OPT_SUNDAY, CS_OPT_MONDAY, CS_OPT_TUESDAY, CS_OPT_WEDNESDAY, CS_OPT_THURSDAY, CS_OPT_FRIDAY, or CS_OPT_SATURDAY
CS_OPT_DATEFORMAT	CS_OPT_FMTMDY, CS_OPT_FMTDMY, CS_OPT_FMTYMD, CS_OPT_FMTYDM, CS_OPT_FMTMYD, or CS_OPT_FMTDYM
CS_OPT_FIPSFLAG	CS_TRUE or CS_FALSE
CS_OPT_FORCEPLAN	CS_TRUE or CS_FALSE
CS_OPT_FORMATONLY	CS_TRUE or CS_FALSE
CS_OPT_GETDATA	CS_TRUE or CS_FALSE
CS_OPT_IDENTITYOFF	A string value
CS_OPT_IDENTITYON	A string value
CS_OPT_ISOLATION	CS_OPT_LEVEL0, CS_OPT_LEVEL1, or CS_OPT_LEVEL3
CS_OPT_NOCOUNT	CS_TRUE or CS_FALSE
CS_OPT_NOEXEC	CS_TRUE or CS_FALSE
CS_OPT_PARSEONLY	CS_TRUE or CS_FALSE
CS_OPT_QUOTED_IDENT	CS_TRUE or CS_FALSE
CS_OPT_RESTREES	CS_TRUE or CS_FALSE
CS_OPT_ROWCOUNT	An integer value

Keyword	Legal value
CS_OPT_SHOWPLAN	CS_TRUE or CS_FALSE
CS_OPT_SORTMERGE	CS_TRUE or CS_FALSE
CS_OPT_STATS_IO	CS_TRUE or CS_FALSE
CS_OPT_STATS_TIME	CS_TRUE or CS_FALSE
CS_OPT_TEXTSIZE	An integer value
CS_OPT_TRUNCIGNORE	CS_TRUE or CS_FALSE

Keywords for server capabilities

Only response capabilities are configured externally. If response capabilities are read from the file, they replace any response capabilities set by application calls to ct_capability for the connection.

The following table lists the keywords for configuring the server capabilities for a connection. For descriptions of what each capability controls, see the reference page for ct_capability.

Keyword	Legal value
CS_CAP_RESPONSE	A comma-separated list of capabilities the client does not want to receive. List values include any symbolic constant listed in Table 3-6 on page 353.

Keywords for *ct_debug* options

The following table lists the keywords for configuring the debugging options for a connection.

The CS_DBG_FILE keyword specifies the name of the file to which Client-Library writes text-format debug information. Client-Library only records the debug information that is requested.

Debug information is requested with the other keywords. These correspond to the bitmasks for the ct_debug *flag* parameter. For meanings of these debug flags, see the reference page for ct_debug.

Table 2-39: Configuration file keywords for debugging options

Keyword	Legal value
CS_DBG_FILE	A character string specifying the file name for text-format debugging information
CS_DEBUG	A character string giving a comma-delimited list of debug flags
CS_PROTOCOL_FILE	A character string specifying the destination file name for binary-format debugging information

CS_DEBUG specifies the data to be written to the file CS_DBG_FILE. Its value can be a list of flags that correspond to the bitmasks for ct_debug's *flag* parameter. For meanings of these debug flags, see the reference page for ct_debug in the Open Client *Client-Library/C Reference Manual*.

The possible flags are:

- CS_DBG_ALL
- CS_DBG_API_LOGCALL
- CS_DBG_API_STATES
- CS_DBG_ASYNC
- CS_DBG_DIAG
- CS_DBG_ERROR
- CS_DBG_MEM
- CS_DBG_NETWORK
- CS_DBG_PROTOCOL
- CS_DBG_PROTOCOL_FILE
- CS_DBG_PROTOCOL_STATES

CHAPTER 3

Routines

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

ct_bind

Description

Bind server results to program variables.

Syntax

CS_RETCODE ct_bind(cmd, item, datafmt, buffer, copied, indicator)

```
CS_COMMAND   cmd;
CS_INT       item;
CS_DATAFMT   *datafmt;
CS_VOID      *buffer;
CS_INT       *copied;
CS_SMALLINT  *indicator;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

item

An integer representing the number of the column, parameter, or status to bind.

When binding a column, *item* is the column's column number. The first column in a select statement's select list is column number 1, the second number 2, and so forth.

When binding a compute column, *item* is the column number of the compute column. Compute columns are returned in the order in which they are listed in the compute clause. The first column returned is number 1.

When binding a return parameter, *item* is the parameter number. The first parameter returned by a stored procedure is number 1. Stored procedure return parameters are returned in the same order as the parameters were originally specified in the stored procedure's create procedure statement. This is not necessarily the same order as specified in the RPC command that invoked the stored procedure. In determining what number to pass as *item*, do not count non-return parameters. For example, if the second parameter in a stored procedure is the only return parameter, pass *item* as 1.

When binding a stored procedure return status, *item* must be 1, as there can be only a single status in a return status result set.

To clear all bindings, pass *item* as CS_UNUSED, with *datafmt*, *buffer*, *copied*, and *indicator* as NULL.

datafmt

The address of a CS_DATAFMT structure that describes the destination variable or array. ct_bind copies the contents of **datafmt* before returning. Client-Library does not reference the address in *datafmt* after ct_bind returns.

The chart below lists the fields in **datafmt* that are used by ct_bind and contains general information about the fields. ct_bind ignores fields that it does not use.

Table 3-1: CS_DATAFMT field settings for ct_bind

Field name	When used	Set to
name	Not used.	Not applicable.
namelen	Not used.	Not applicable.
datatype	When binding all types of results.	A type constant (CS_xxx_TYPE) representing the datatype of the destination variable. For valid type constants, see “Datatypes support” on page 294. Open Client user-defined types are valid, provided that user-supplied conversion routines have been installed using cs_set_convert. If *datatype* is an Open Client user-defined type, ct_bind does not validate any CS_DATAFMT fields except *count*. ct_bind supports a wide range of type conversions, so *datatype* can be different from the type returned by the server. For instance, by specifying a destination type of CS_FLOAT_TYPE, a CS_MONEY result can be bound to a CS_FLOAT program variable. The appropriate data conversion happens automatically. ct_bind can perform any conversion supported by cs_convert. For a list of the supported conversions, see the cs_convert reference page in the Open Client and Open Server. If *datatype* is CS_BOUNDARY_TYPE or CS_SENSITIVITY_TYPE, the **buffer* program variable must be of type CS_CHAR.
format	When binding result items to character, binary, text, or image destination variables; otherwise CS_FMT_UNUSED.	A bitmask of the following symbols: For character and text destinations only: 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: CS_FMT_PADNULL to pad to the full length of the variable with nulls. For any type of destination: CS_FMT_UNUSED if no format information is being provided.

Field name	When used	Set to
maxlength	When binding all types of results to non-fixed-length types. When binding to fixed-length types, *maxlength* is ignored.	The length of the **buffer* destination variable. If *buffer* points to an array, set *maxlength* to the length of a single element of the array. When binding to character or binary destinations, *maxlength* must describe the total length of the destination variable, including any space required for special terminating bytes, such as a null terminator. If *maxlength* indicates that **buffer* is not large enough to hold a result data item, then at fetch time ct_fetch discards the result item that is too large, fetches any remaining items in the row, and returns CS_ROW_FAIL. If this occurs, the contents of **buffer* are undefined.
scale	When binding to numeric or decimal destinations.	The maximum number of digits to the right of the decimal point in the destination 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	When binding to numeric or decimal destinations.	The maximum number of decimal digits that can be represented in the destination 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	Set to
count	When binding all types of results.	The number of result rows to be copied to program variables per ct_fetch or ct_scroll_fetch call. If *count* is larger than the number of available rows, only the available rows are copied. (Note that only regular row and cursor row result sets contain multiple rows. *count* must have the same value for all columns in a result set, with one exception: an application can intermix *count*s of 0 and 1. If *count* is 0, 1 row is fetched. For ct_scroll_fetch calls, the *count* value must be equal or greater than CS_CURSOR_ROWS. The *count* value cannot be less than CS_CURSOR_ROWS, as unpredictable results may occur.
usertype	Not used.	Not applicable.
locale	When binding all types of results.	A pointer to a CS_LOCALE structure containing locale information for the *buffer destination variable. If custom locale information is not required for the variable, pass *locale* as NULL.

buffer

The address of an array of *datafmt–>count* variables, each of which is of size *datafmt–>maxlength*.

**buffer* is the program variable or variables to which ct_bind binds the server results. When the application calls ct_fetch to fetch the result data, it is copied into this space.

If *buffer* is NULL, ct_bind clears the binding for this result item. Note that if *buffer* is NULL, *datafmt*, *copied*, and *indicator* must also be NULL.

Note The *buffer* address must remain valid as long as binds are active on the command structure.

---

copied

The address of an array of *datafmt–>count* integer variables. At fetch time, ct_fetch fills this array with the lengths of the copied data. *copied* is an optional parameter and can be passed as NULL.

indicator

The address of an array of *datafmt–>count* CS_SMALLINT *variables*. At fetch time, each variable is used to indicate certain conditions about the fetched data. *indicator* is an optional parameter and can be passed as NULL.

The following table lists the values that an indicator variable can have:

Indicator value	Meaning
-1	The fetched data was NULL. In this case, no data is copied to **buffer*.
0	The fetch was successful.
integer value > 0	The actual length of the server data, if the fetch resulted in truncation.

Return value

ct_bind returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see “Asynchronous programming” on page 12.

Common reasons for a ct_bind failure include:

- An illegal datatype specified using *datafmt–>datatype*.
- A bad *datafmt–>locale* pointer. Initialize *datafmt–>locale* to NULL if it is not used.
- Requested conversion is not available.

Examples

```
CS_RETCODE      retcode;
 CS_INT          num_cols;
 CS_INT          i;
 CS_INT          j;
 CS_INT          row_count = 0;
 CS_INT          rows_read;
 CS_INT          disp_len;
 CS_DATAFMT      *datafmt;
 EX_COLUMN_DATA  *coldata;
/* Determine the number of columns in this result set */
 .... ct_res_info code deleted ....
/*
 ** Our program variable, called 'coldata', is an array of
```

```
 ** EX_COLUMN_DATA structures. Each array element represents
 ** one column.  Each array element will be re-used for each
 ** row.
 **
 ** First, allocate memory for the data element to process.
 */
 coldata = (EX_COLUMN_DATA *)malloc(num_cols *
            sizeof (EX_COLUMN_DATA));

if (coldata == NULL)
 {
    ex_error("ex_fetch_data: malloc() failed");
    return CS_MEM_ERROR;
 }
 datafmt = (CS_DATAFMT *)malloc(num_cols *
           sizeof (CS_DATAFMT));
 if (datafmt == NULL)
 {
    ex_error("ex_fetch_data: malloc() failed");
    free(coldata);
    return CS_MEM_ERROR;
 }

/*
 ** Loop through the columns, getting a description of each
 ** one and binding each one to a program variable.
 **
 ** We're going to bind each column to a character string;
 ** this will show how conversions from server native
 ** datatypes to strings can occur using bind.
 **
 ** We're going to use the same datafmt structure for both
 ** the describe and the subsequent bind.
 **
 ** If an error occurs within the for loop, a break is used
 ** to get out of the loop and the data that was allocated
 ** is freed before returning.
 */

for (i = 0; i < num_cols; i++)
 {
    /*
    ** Get the column description.  ct_describe() fills
    ** the datafmt parameter with a description of the
    ** column.
    */
    retcode = ct_describe(cmd, (i + 1), &datafmt[i]);
    if (retcode != CS_SUCCEED)
```

```
 {
     ex_error("ex_fetch_data: ct_describe() failed");
     break;
 }

/*
 ** Update the datafmt structure to indicate that we
 ** want the results in a null terminated character
 ** string.
 **

** First, update datafmt.maxlength to contain the
 ** maximum possible length of the column. To do this,
 ** call ex_display_len() to determine the number of
 ** bytes needed for the character string
 ** representation, given the datatype described
 ** above. Add one for the null termination character.
 */

datafmt[i].maxlength
       = ex_display_dlen(&datafmt[i]) + 1;

/*
 ** Set datatype and format to tell bind we want things
 ** converted to null terminated strings.
 */
 datafmt[i].datatype = CS_CHAR_TYPE;
 datafmt[i].format = CS_FMT_NULLTERM;


/*
 ** Allocate memory for the column string
 */
 coldata[i].value = (CS_CHAR *)malloc
     (datafmt[i].maxlength);
 if (coldata[i].value == NULL)
 {
     ex_error("ex_fetch_data: malloc() failed");
     retcode = CS_MEM_ERROR;
     break;
 }

/* Now bind. */
 retcode = ct_bind(cmd, (i + 1), &datafmt[i],
     coldata[i].value, &coldata[i].valuelen,
     &coldata[i].indicator);
 if (retcode != CS_SUCCEED)
 {
     ex_error("ex_fetch_data: ct_bind() failed");
```

```
            break;
        }
    }
```

This code excerpt is from the function *ex_fetch_data()* routine in the *exutils.c* sample program. For further examples of using ct_bind, see the *compute.c*, *ex_alib.c*, *getsend.c*, and *i18n.c* sample programs.

Usage

- ct_bind can be used to bind a regular or cursor result column, a compute column, a return parameter, or a stored procedure status number. When binding a regular or cursor column, multiple rows of the column can be bound with a single call to ct_bind.

  Note Message, describe, row format, and compute format results are not bound. This is because result sets of type CS_MSG_RESULT, CS_DESCRIBE_RESULT, CS_ROWFMT_RESULT, and CS_COMPUTEFMT_RESULT contain no fetchable data. Instead, these result sets indicate that certain types of information are available. An application can retrieve the information by calling other Client-Library routines, such as ct_res_info. For more information about processing these types of results, see “Results” on page 241.

- Binding associates a result data item with a program variable. At fetch time, each ct_fetch call copies a row instance of the data item into the variable with which the item is associated.

  If a result data item is very large (for example, a large text or image column), it is often more convenient for an application to use ct_get_data to retrieve the data item’s value in chunks, rather than copying the entire value to a bound variable. For more information about ct_get_data, see the ct_get_data reference page, and “text and image data handling” on page 284.

- ct_bind binds only the current result type. ct_results indicates the current result type through its *result_type* parameter. For example, if ct_results sets **result_type* to CS_STATUS_RESULT, a return status is available for binding.

- An application can call ct_res_info to determine the number of items in the current result set and ct_describe to get a description of each item.

- An application can only bind a result item to a single program variable. If an application binds a result item to multiple variables, only the last binding has any effect.

- An application can use ct_bind to bind to Open Client user-defined datatypes for which conversion routines have been installed. To install a conversion routine for a user-defined datatype, an application calls cs_set_convert. For more information about Open Client user-defined types, see “Open Client user-defined datatypes” on page 304.

Replacing existing binds

- An application can rebind while actively fetching rows. That is, an application can call ct_bind inside a ct_fetch loop if it needs to change a result item’s binding.
- Applications do not have to rebind interspersed regular row results and compute row results that are generated by the same command. If not changed, binding for a particular type of result remains in effect until ct_results returns CS_CMD_DONE to indicate that the results of a logical command are completely processed.

  For example, a language command containing a select statement with compute and order by clauses can generate multiple regular row result sets intermixed with compute row result sets. Because they are generated by the same command, each regular row result set and each compute row result set will contain identical columns. An application need only bind each one time (before fetching the first result set of each type). These bindings will remain in effect until both result sets are completely processed (that is, until ct_results returns a result_type of CS_CMD_DONE).

  This behavior is independent of the CS_STICKY_BINDS property value.

Clearing bindings

- To clear the binding for a result item, call ct_bind with *buffer*, *datafmt*, *copied*, and *indicator* as NULL. If the CS_STICKY_BINDS property is enabled (CS_TRUE) for the command structure, then the result-item binding is cleared for all subsequent executions of the command.
- To clear all bindings, call ct_bind with item as CS_UNUSED and *buffer*, *datafmt*, *copied*, and *indicator* as NULL. If the CS_STICKY_BINDS property is enabled (CS_TRUE) for the command structure, then the result-item bindings are cleared only until ct_results returns CS_CMD_DONE (in other words, only for the current execution of the command). If the same command is executed again, the command structure reverts to the previous bindings.
- It is not an error to clear a non-existent binding.

Duration of bindings

- By default, the binding between a result item and a program variable remains active until:
  - ct_results returns CS_CMD_DONE,
  - The application rebinds the result item, or
  - The application clears the binding.
- The application can change the default duration of bindings by setting the CS_STICKY_BINDS command property. When this property is set to CS_TRUE, then result item bindings remain active across executions of the same server command. Specifically, the binding between a result item and a program variable remains active until:
  - The application initiates a new server on the same command structure with ct_command, ct_cursor, ct_dynamic, or ct_sendpassthru (but nested cursor-close, cursor-update, or cursor-delete commands do not clear bindings),
  - The application rebinds the result item,
  - The application clears the binding, or
  - The application calls ct_results and it finds a format mismatch between the result set format when the binds were established and the current result set.

  The CS_STICKY_BINDS property is useful in batch-processing applications that repeatedly execute the same command.
- Commands can return multiple result sets. When the CS_STICKY_BINDS property is CS_TRUE, then Client-Library preserves all bindings for all result sets returned by the first execution of a command for use with later executions of the same command. During first-time command execution, Client-Library also saves information about the formats and sequencing of the returned result sets. After subsequent executions of the same command, each call to ct_results compares the current result formats to the saved result formats. If ct_results finds a mismatch, then it clears all bindings, raises an informational error, and returns CS_SUCCEED.

  The result formats from repeated execution of the same command can only vary if the command contains conditional server-side logic (for example, an Adaptive Server stored procedure that contains an if or a while statement).

- Applications can check the value of the CS_HAVE_BINDS command property to see if binds were saved from a previous execution of the current command. A value of CS_TRUE indicates one or more binds is active for the current result set. For example, a batch processing application might use the following logic to retrieve result rows:

  ```
  retrieve CS_HAVE_BINDS property with ct_cmd_props
   if CS_HAVE_BINDS is CS_FALSE
       bind variables with ct_bind
   end if
   while ct_fetch returns CS_SUCCEED or CS_ROW_FAIL
     process row data
   end while
  ```

  Calling ct_bind does not change the value of CS_HAVE_BINDS. The property reflects whether binds established during a previous execution of the command are still in effect.

- As long as a result item binding remains active, the memory addresses given as ct_bind's *buffer* parameter must remain valid. Each call to ct_fetch writes data to the *buffer* address. If the address is invalid, the application will experience memory corruption or a memory access violation. For example, if an application's C routine binds the address of an automatic variable, and the routine returns before the application calls ct_fetch, then the bound address will be invalid.

Array binding

- Array binding is the process of binding a result column to an array of program variables. At fetch time, multiple rows are copied to an array of variables with a single ct_fetch or ct_scroll_fetch call. An application indicates array binding by setting *datafmt–>count* to a value greater than 1.

- Array binding is only practical for regular row and cursor results. This is because other types of results are considered to be the equivalent of a single row.

- When binding columns to arrays, all ct_bind calls in the sequence of calls binding the columns must use the same value for *datafmt–>count*. For example, when binding three columns to arrays, it is an error to use a *count* of five in the first two ct_bind calls and a *count* of three in the last.

  However, an application can intermix *count*s of 0 and 1. *count*s of 0 and 1 are considered to be equivalent because they both cause ct_fetch to fetch a single row.

- If the CS_CURSOR_ROWS value is greater than 1, and you are using a scrollable client cursor, array binding must be used. Failure to use array binding may result in lost efficiency and undefined behavior.

  For non-scrollable cursors, either arrays or regular program variables can be used.

See also

ct_describe, ct_fetch, ct_res_info, ct_results, Datatypes support

ct_br_column

Description

Retrieve information about a column generated by a browse-mode select.

Syntax

CS_RETCODE ct_br_column(cmd, colnum, browsedesc)

```
CS_COMMAND        *cmd;
CS_INT            colnum;
CS_BROWSEDESC     *browsedesc;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

colnum

The number of the column to describe. The first column in a select statement's select-list is column number 1, the second is number 2, and so forth.

browsedesc

A pointer to a CS_BROWSEDESC structure. ct_br_column fills this structure with information about the column specified by *colnum*.

For information about the CS_BROWSEDESC structure, see "CS_BROWSEDESC structure" on page 76.

Return value

ct_br_column returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see "Asynchronous programming" on page 12.

ct_br_column fails if the current result set was not generated by a select...for browse language command.

Usage

- ct_br_column fills **browsedesc* with information about the column specified by *colnum*.
- A column can be updated through browse mode only if it:
  - Belongs to a browsable table,
  - Is the result of a select...for browse, and
  - Is not the result of a SQL expression, such as max(colname).
- Is an error to call ct_br_column if browse-mode information is not available. Generally, browse mode information is available if the current result set is a CS_ROW_RESULT result set that was generated by a select...for browse.

  Before calling ct_br_column, an application can call ct_res_info with *type* as CS_BROWSE_INFO to check whether browse mode information is available.

See also

"Browse mode" on page 21, ct_br_table

ct_br_table

Description

Return information about browse mode tables.

Syntax

CS_RETCODE ct_br_table(cmd, tabnum, type, buffer, buflen, outlen)

```
CS_COMMAND      *cmd;
CS_INT          tabnum;
CS_INT          type;
CS_VOID         *buffer;
CS_INT          buflen;
CS_INT          *outlen;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

tabnum

The number of the table of interest. The first table in a select statement's from list is table number 1, the second number is 2, and so forth.

type

The type of information to return. The following table lists the symbolic values for *type*:

Value of type	ct_br_table return value	*buffer set to
CS_ISBROWSE	Whether or not the table is browsable. A table is browsable if it has a unique index and a timestamp column.	CS_TRUE or CS_FALSE
CS_TABNAME	The name of the table whose number is *tabnum*.	A string value
CS_TABNUM	The number of tables named in the browse-mode select. If *type* is CS_TABNUM, pass *tabnum* as CS_UNUSED.	An integer value.

buffer

A pointer to the space in which ct_br_table will place the requested information.

buflen

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

If *type* is CS_ISBROWSE or CS_TABNUM, pass *buflen* as CS_UNUSED.

outlen

A pointer to an integer variable.

If supplied, ct_br_table sets **outlen* to the length, in bytes, of the requested information.

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

Return value

ct_br_table returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see “Asynchronous programming” on page 12.

ct_br_table fails if the current result set was not generated by a select...for browse language command.

Usage

- ct_br_table returns either the number of tables named in the select statement or information about a particular table.
- A table is browsable if it has a unique index and a timestamp column.
- It is an error to call ct_br_table if browse-mode information is not available. Generally, browse mode information is available if the current result set is a CS_ROW_RESULT result set that was generated by a select...for browse.
- Before calling ct_br_table, an application can call ct_res_info with *type* as CS_BROWSE_INFO to check whether browse mode information is available.

See also

"Browse mode" on page 21, ct_br_column

ct_callback

Description

Install or retrieve a Client-Library callback routine.

Syntax

```
CS_RETCODE ct_callback(context, connection, action, type, func)

CS_CONTEXT      *context;
CS_CONNECTION   *connection;
CS_INT          action;
CS_INT          type;
CS_VOID         *func;
```

Parameters

context

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

Either *context* or *connection* must be NULL:

- If *context* is supplied, the callback is installed as a "default" callback for the specified context. Once installed, a default callback is inherited by all connections subsequently allocated within the context.
- If *context* is NULL, the callback is installed for the individual connection specified by *connection*.

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

Either *context* or *connection* must be NULL:

- If *connection* is supplied, the callback is installed for the specified connection.
- If *connection* is NULL, the callback is installed for the application context specified by *context*.

action

One of the following symbolic values:

Value of action	Meaning
CS_SET	Installs a callback
CS_GET	Retrieves the currently installed callback of this type

type

The type of callback routine of interest. The following table lists the symbolic values for *type*:

Table 3-2: Values for ct_callback type parameter

Value of type	Meaning
CS_CLIENTMSG_CB	A client message callback, as described in “Client message callbacks” on page 31.
CS_COMPLETION_CB	A completion callback, as described in “Completion callbacks” on page 34.
CS_DS_LOOKUP_CB	A directory callback, as described in “Directory callbacks” on page 39.
CS_ENCRYPT_CB	An encryption callback, as described in “Encryption callbacks” on page 41.
CS_EXTENDED_ENCRYPT_CB	An encryption callback, as described in “Encryption callbacks” on page 41.
CS_CHALLENGE_CB	A negotiation callback, as described in “Negotiation callbacks” on page 45.
CS_NOTIF_CB	A registered procedure notification callback, as described in “Notification callbacks” on page 48.
CS_SECSESSION_CB	A security session callback, as described in “Security session callbacks” on page 50.
CS_SERVERMSG_CB	A server message callback, as described in “Server message callbacks” on page 53.
CS_SIGNAL_CB + *signal_number*	A signal callback, as described in “Signal callbacks” on page 57. Signal callbacks are identified by adding the signal number of interest to the manifest constant CS_SIGNAL_CB. For example, to install a signal callback for a SIGALRM signal, pass *type* as CS_SIGNAL_CB + SIGALRM.
CS_SSLVALIDATE_CB	An SSL validation callback, as described in “SSL validation callbacks” on page 59.

func

A pointer variable.

If a callback routine is being installed, *func* is the address of the callback routine to install.

If a callback routine is being retrieved, ct_callback sets **func* to the address of the currently installed callback routine.

Return value

ct_callback returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Return value	Meaning
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see “Asynchronous programming” on page 12.

Examples

```
/*
 ** Install message and completion handlers.
 */
 retstat = ct_callback(Ex_context, NULL, CS_SET,
      CS_CLIENTMSG_CB,(CS_VOID *)ex_clientmsg_cb);
 if (retstat != CS_SUCCEED)
 {
      ex_panic("ct_callback failed");
 }
 retstat = ct_callback(Ex_context, NULL, CS_SET,
      CS_SERVERMSG_CB,(CS_VOID *)ex_servermsg_cb);
 if (retstat != CS_SUCCEED)
 {
      ex_panic("ct_callback failed");
 }

retstat = ct_callback(Ex_context, NULL, CS_SET,
      CS_COMPLETION_CB,(CS_VOID *)CompletionCB);
 if (retstat != CS_SUCCEED)
 {
      ex_panic("ct_callback failed");
 }
```

This code excerpt is from the *ex_amain.c* sample program. For additional examples of using ct_callback, see the *ex_alib.c* and *exutils.c* sample programs.

Usage

- A typical application will use ct_callback only to install callback routines. However, some applications may need to retrieve previously installed callbacks.
- To install a callback routine, an application calls ct_callback with *action* as CS_SET and *func* as the address of the callback to install.
- To retrieve the address of a previously installed callback, an application calls ct_callback with *action* as CS_GET and *func* as a pointer to a pointer. In this case, ct_callback sets **func* to the address of the current callback of the specified type. An application can save this address for use again at a later time. Note that retrieving the address of a callback does not de-install it.

- ct_callback can be used to install a callback routine either for a context or for a particular connection. To install a callback for a context, pass *connection* as NULL. To install a callback for a connection, pass *context* as NULL.
- When a context is allocated, it has no callback routines installed. An application must specifically install any callbacks that are required.
- When a connection is allocated, it picks up default callback routines from its parent context. An application can override these default callbacks by calling ct_callback to install new callbacks at the connection level.
- To deinstall an existing callback routine, an application can call ct_callback with *func* as NULL. An application can also install a new callback routine at any time. The new callback automatically replaces any existing callback.
- For most types of callbacks, if no callback of a particular type is installed for a connection, Client-Library discards callback information of that type.

  The client message callback is an exception to this rule. When an error or informational message is generated for a connection that has no client message callback installed, Client-Library calls the connection's parent context's client message callback (if any) rather than discarding the message. If the context has no client message callback installed, then the message is discarded.
- A connection picks up its parent context's callback routines only once, when it is allocated. This has two important implications:
  - Existing connections are not affected by changes to their parent context's callback routines.
  - If a callback routine of a particular type is de-installed for a connection, the connection does not pick up its parent context's callback routine. Instead, the connection is considered to have no callback routine of this type installed.

- An application can use the CS_USERDATA property to transfer information between a callback routine and the program code that triggered it. The CS_USERDATA property allows an application to save user data in internal Client-Library space and retrieve it later.

Note On Digital UNIX, Client-Library uses interrupt-driven I/O for all network I/O modes, including synchronous mode. This affects the coding of some applications.

On Digital UNIX, Client-Library applications that require their own signal handler must install any needed signal handlers with ct_callback. Programs that make UNIX system calls should check for system-call failure caused by system interrupts, and reissue any interrupted system calls.

See also

"Callbacks" on page 24, ct_capability, ct_config, ct_con_props, ct_connect

ct_cancel

Description

Cancel a command or the results of a command.

Syntax

```
CS_RETCODE ct_cancel(connection, cmd, type)

CS_CONNECTION      *connection;
CS_COMMAND         *cmd;
CS_INT             type;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

For CS_CANCEL_CURRENT cancels, *connection* must be NULL.

For CS_CANCEL_ATTN and CS_CANCEL_ALL cancels, one of *connection* or *cmd* must be NULL. If *connection* is supplied and *cmd* is NULL, the cancel operation applies to all commands pending for this connection.

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

For CS_CANCEL_CURRENT cancels, *cmd* must be supplied. The cancel operation applies only to the results pending for this command structure.

For CS_CANCEL_ATTN and CS_CANCEL_ALL cancels, if *cmd* is supplied and *connection* is NULL, the cancel operation applies only to the command pending for this command structure. If *cmd* is NULL and *connection* is supplied, the cancel operation applies to all commands pending for this connection.

type

The type of cancel. The following table lists the symbolic values that are legal for *type*

Table 3-3: Values for ct_cancel type parameter

Value of type	Result	Notes
CS_CANCEL_ALL	ct_cancel sends an attention to the server, instructing it to cancel the current command. Client-Library immediately discards all results generated by the command.	Causes this connection's cursors to enter an undefined state. To determine the state of a cursor, an application can call ct_cmd_props with *property* as CS_CUR_STATUS.
CS_CANCEL_ATTN	ct_cancel sends an attention to the server, instructing it to cancel the current command. The next time the application reads from the server, Client-Library discards all results generated by the canceled command.	Causes this connection's cursors to enter an undefined state. To determine the state of a cursor, an application can call ct_cmd_props with *property* as CS_CUR_STATUS.
CS_CANCEL_CURRENT	ct_cancel discards the current result set.	Safe to use on connections with open cursors.

Return value

ct_cancel returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. For more information, see "Asynchronous programming" on page 12.
CS_CANCELED	The cancel operation was canceled. Only a CS_CANCEL_CURRENT type of cancel can be canceled.

Return value	Meaning
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see “Asynchronous programming” on page 12.
CS_TRYING	A cancel operation is already pending for this connection.

Examples

```
if (query_code == CS_FAIL)
 {
      /*
      ** Terminate results processing and break out of
      ** the results loop.
      */
      retcode = ct_cancel(NULL, cmd, CS_CANCEL_ALL);
      if (retcode != CS_SUCCEED)
      {
         ex_error("ex_execute_cmd: ct_cancel() failed");
      }
      break;
 }
```

This code excerpt is from the *exutils.c* sample program.

Usage

- Canceling a command is equivalent to sending an attention to the server, instructing it to halt execution of the current command. When a command is canceled, any results generated by it are no longer available to an application.
- Canceling results is equivalent to fetching and then discarding a result set.Once results are canceled, they are no longer available to an application. If the result set has not been completely processed, subsequent results remain available.

Canceling a command

- To cancel the current command and all results generated by it, an application calls ct_cancel with *type* as CS_CANCEL_ATTN or CS_CANCEL_ALL. Both of these calls tell Client-Library to:
  - Send an attention to the server, instructing it to halt execution of the current command.
  - Discard any results already generated by the command.
- Both types of cancels return CS_SUCCEED immediately, without sending an attention to the server, if no command is in progress.

- If an application has not yet called ct_send to send an initiated command or command batch:
  - A CS_CANCEL_ALL cancel discards the initiated command or command batch without sending an attention to the server. A CS_CANCEL_ATTN cancel has no effect.
- A connection can become unusable due to error. If this occurs, Client-Library marks the connection as “dead.” An application can use the CS_CON_STATUS property to determine if a connection has been marked “dead.”

  If a connection has been marked “dead” because of a results-processing error, an application can try calling ct_cancel(CS_CANCEL_ALL or CS_CANCEL_ATTN) to revive the connection. If this fails, the application must close the connection and drop its CS_CONNECTION structure.
- The difference between CS_CANCEL_ALL and CS_CANCEL_ATTN is:
  - CS_CANCEL_ALL causes Client-Library to discard the canceled command’s results immediately.
  - CS_CANCEL_ATTN causes Client-Library to wait until the application attempts to read from the server before discarding the results.
- This difference is important because Client-Library must read from the result stream to discard results, and it is not always safe to read from the result stream.

  It is not safe to read from the result stream from within callbacks or interrupt handlers, or when an asynchronous routine is pending. It is safe to read from the result stream anytime an application is running in its mainline code, except when an asynchronous operation is pending.

  Use CS_CANCEL_ATTN from within callbacks or interrupt handlers or when an asynchronous operation is pending.

  Use CS_CANCEL_ALL in mainline code, except when an asynchronous operation is pending.
- CS_CANCEL_ALL leaves the command structure in a “clean” state, available for use in another operation. When a command is canceled with CS_CANCEL_ATTN, however, the command structure cannot be reused until a Client-Library routine returns CS_CANCELED.

The Client-Library routines that can return CS_CANCELED are:

- ct_cancel(CS_CANCEL_CURRENT)
- ct_fetch
- ct_get_data
- ct_options
- ct_recvpassthru
- ct_results
- ct_send
- ct_sendpassthru

- CS_CANCEL_ATTN has two primary uses:
  - To cancel commands from within an application's interrupt handlers or callback routines.
  - In asynchronous applications, to cancel pending calls to the result-processing routines ct_results and ct_fetch.
- If a command has been sent and ct_results has not been called, a ct_cancel(CS_CANCEL_ATTN) call has no effect.
- Canceling commands on a connection that has an open cursor may affect the state of the cursor in unexpected ways. For this reason, it is recommended that the CS_CANCEL_ALL and CS_CANCEL_ATTN types of cancels not be used on connections with open cursors. Instead of canceling a cursor command, an application can simply close the cursor.

Canceling current results

- To cancel current results, an application calls ct_cancel with *type* as CS_CANCEL_CURRENT. This tells Client-Library to discard the current results; it is equivalent to calling ct_fetch until it returns CS_END_DATA.
- The next buffer's worth of results, if any, remains available to the application, and the current command is not affected.
- Canceling results clears the bindings between the result items and program variables.
- A CS_CANCEL_CURRENT type of cancel is legal for all types of result sets, even those that contain no fetchable results. If a result set contains no fetchable results, a cancel has no effect.

See also ct_fetch, ct_results

ct_capability

Description Set or retrieve a client/server capability.

Syntax CS_RETCODE ct_capability(connection, action, type, capability, value)

```
CS_CONNECTION    *connection;
CS_INT           action;
CS_INT           type;
CS_INT           capability;
CS_VOID          *value;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

action

One of the following symbolic values:

Value of action	Meaning
CS_SET	Sets a capability
CS_GET	Retrieves a capability

type

The type category of the capability. The following table lists the symbolic values for *type*:

Table 3-4: Values for ct_capability type parameter

Value of type	Meaning
CS_CAP_REQUEST	Request capabilities. These capabilities describe the types of requests that a connection can support. Request capabilities are retrieve-only.
CS_CAP_RESPONSE	Response capabilities. These capabilities describe the types of responses that a server can send to a connection. An application can set response capabilities before a connection is open and can retrieve response capabilities at any time.

capability

The capability of interest. The following two tables list the symbolic values that are legal for *capability*:

Note In addition to the values listed in the tables, *capability* can have the special value CS_ALL_CAPS, to indicate that an application is setting or retrieving all response or request capabilities simultaneously. CS_ALL_CAPS is primarily of use in gateway applications. A typical Client-Library application needs to set or retrieve only a small number of capabilities.

Table 3-5 summarizes the CS_CAP_REQUEST capabilities.

Table 3-5: Request capabilities

CS_CAP_REQUEST capability	Meaning	Capability relates to
CS_CON_INBAND	In-band (non-expedited) attentions.	Connections
CS_CON_LOGICAL	Logical mapping.	Connections
CS_CON_OOB	Out-of-band (expedited) attentions.	Connections
CS_CSR_ABS	Fetch of specified absolute cursor row.	Cursors
CS_CSR_FIRST	Fetch of first cursor row.	Cursors
CS_CSR_LAST	Fetch of last cursor row.	Cursors
CS_CSR_MULTI	Multi-row cursor fetch.	Cursors
CS_CSR_PREV	Fetch previous cursor row.	Cursors
CS_CSR_REL	Fetch specified relative cursor row.	Cursors
CS_CUR_IMPLICIT	TDS optimized read-only cursor.	Cursors
CS_DATA_BIN	Binary datatype.	Datatypes
CS_DATA_VBIN	Variable-length binary type.	Datatypes
CS_DATA_LBIN	Long binary datatype.	Datatypes
CS_DATA_BIT	Bit datatype.	Datatypes
CS_DATA_BITN	Nullable bit values.	Datatypes
CS_DATA_BOUNDARY	Boundary datatype.	Datatype
CS_DATA_CHAR	Character datatype.	Datatypes
CS_DATA_VCHAR	Variable-length character datatype.	Datatypes
CS_DATA_LCHAR	Long character datatype.	Datatypes
CS_DATA_DATE	Date datatype.	Datatypes
CS_DATA_DATE4	Short datetime datatype.	Datatypes
CS_DATA_DATE8	Datetime datatype.	Datatypes
CS_DATA_DATETIMEN	NULL datetime values.	Datatypes
CS_DATA_DEC	Decimal datatype.	Datatypes
CS_DATA_FLT4	4-byte float datatype.	Datatypes
CS_DATA_FLT8	8-byte float datatype.	Datatypes
CS_DATA_FLTN	Nullable float values.	Datatypes
CS_DATA_IMAGE	Image datatype.	Datatypes
CS_DATA_INT1	Tiny integer datatype.	Datatypes
CS_DATA_INT2	Small integer datatype.	Datatypes
CS_DATA_INT4	Integer datatype.	Datatypes

CS_CAP_REQUEST capability	Meaning	Capability relates to
CS_DATA_INTN	NULL integers.	Datatypes
CS_DATA_INT8	8-integer datatype	Datatypes
CS_DATA_LBIN	Long binary datatype	Datatypes
CS_DATA_LCHAR	Long character datatype	Datatypes
CS_DATA_UINT2	Unsigned 2-byte integer datatype	Datatypes
CS_DATA_UINT4	Unsigned 4-byte integer datatype	Datatypes
CS_DATA_UINT8	Unsigned 8-byte integer datatype	Datatypes
CS_DATA_UINTN	Unsigned datatype	Datatypes
CS_DATA_UCHAR	unsigned character.	Datatypes
CS_DATA_UNITEXT	unsigned character.	Datatypes
CS_DATA_MNY4	Short money datatype.	Datatypes
CS_DATA_MNY8	Money datatype.	Datatypes
CS_DATA_MONEYN	NULL money values.	Datatypes
CS_DATA_NUM	Numeric datatype.	Datatypes
CS_DATA_SENSITIVITY	Secure Server sensitivity datatypes.	Datatypes
CS_DATA_TEXT	Text datatype.	Datatypes
CS_DATA_TIME	Time datatype.	Datatypes
CS_DATA_XML	Variable-width character datatype.	Datatypes
CS_DOL_BULK	Token for bulk copy on DOL table.	Bulk copy
CS_OBJECT_CHAR	Specifies whether the server can send/recieve streaming character data.	Java objects.
CS_OBJECT_BINARY	Specifies whether the server can send/receive streaming binary data.	Streaming data.
CS_OBJECT_JAVA1	Specifies whether Java object serializations can be sent/received by the server.	Streaming data.
CS_OPTION_GET	Whether the client can get current option values from the server.	Options
CS_PROTO_BULK	Tokenized bulk copy.	Bulk copy
CS_PROTO_DYNAMIC	Descriptions for prepared statements come back at prepare time.	Dynamic SQL

CS_CAP_REQUEST capability	Meaning	Capability relates to
CS_PROTO_DYNPROC	Client-Library prepends “create proc” to a Dynamic SQL prepare statement.	Dynamic SQL
CS_REQ_BCP	Bulk copy requests.	Commands
CS_REQ_CURSOR	Cursor requests.	Commands
CS_REQ_DBRPC2	Large RPC name requests.	Commands
CS_REQ_DYN	Dynamic SQL requests.	Commands
CS_REQ_LANG	Language requests.	Commands
CS_REQ_MSG	Message commands.	Commands
CS_REQ_MSTMT	Multiple server commands per Client-Library language command.	Commands
CS_REQ_NOTIF	Registered procedure notifications.	Commands
CS_REQ_PARAM	Use PARAM/PARAMFMT TDS streams for requests.	Commands
CS_REQ_RESERVED1	Reserved for future use.	Commands
CS_REQ_RESERVED2	Reserved for future use.	Commands
CS_REQ_URGNOTIF	Send notifications with the “urgent” bit set in the TDS packet header.	Registered procedures
CS_REQ_RPC	Remote procedure requests.	Commands
CS_WIDETABLES	Wide table support.	Connection

Table 3-6 summarizes the CS_CAP_RESPONSE capabilities.

Table 3-6: Response capabilities

CS_CAP_RESPONSE capability	Meaning	Capability relates to
CS_CON_NOINBAND	No in-band (non-expedited) attentions.	Connections
CS_CON_NOOOB	No out-of-band (expedited) attentions.	Connections
CS_DATA_NOBIN	No binary datatype.	Datatypes
CS_DATA_NOBOUNDARY	No security boundary datatype.	Datatypes
CS_DATA_NOVBIN	No variable-length binary type.	Datatypes
CS_DATA_NOLBIN	No long binary datatype.	Datatypes
CS_DATA_NOBIT	No bit datatype.	Datatypes
CS_DATA_NOCHAR	No character datatype.	Datatypes
CS_DATA_NOVCHAR	No variable-length character datatype.	Datatypes
CS_DATA_NOLCHAR	No long character datatype.	Datatypes
CS_DATA_NODATE	No date datatype.	Datatypes
CS_DATA_NODATE4	No short datetime datatype.	Datatypes
CS_DATA_NODATE8	No datetime datatype.	Datatypes
CS_DATA_NODATETIMEN	No NULL datetime values.	Datatypes
CS_DATA_NODEC	No decimal datatype.	Datatypes
CS_DATA_NOFLT4	No 4-byte float datatype.	Datatypes
CS_DATA_NOFLT8	No 8-byte float datatype.	Datatypes
CS_DATA_NOIMAGE	No image datatype.	Datatypes
CS_DATA_NOINT1	No tiny integer datatype.	Datatypes
CS_DATA_NOINT2	No small integer datatype.	Datatypes
CS_DATA_NOINT4	No integer datatype.	Datatypes
CS_DATA_NOINT8	No 8-byte integer datatype.	Datatypes
CS_DATA_NOINTN	No NULL integers.	Datatypes
CS_DATA_NOLBIN	No long binary	Datatypes
CS_DATA_NOLCHAR	No long character	Datatypes
CS_DATA_NOMNY4	No short money datatype.	Datatypes
CS_DATA_NOMNY8	No money datatype.	Datatypes
CS_DATA_NOMONEYN	No NULL money values.	Datatypes
CS_DATA_NONUM	No numeric datatype.	Datatypes
CS_DATA_NOSENSITIVITY	No Secure Server sensitivity datatypes.	Datatypes
CS_DATA_NOTEXT	No text datatype.	Datatypes
CS_DATA_NOTIME	No time datatype.	Datatypes

CS_CAP_RESPONSE capability	Meaning	Capability relates to
CS_DATA_NOUCHAR	No unsigned character.	Datatypes
CS_DATA_NOUINT2	No unsigned 2-byte integer datatype.	Datatypes
CS_DATA_NOUINT4	No unsigned 4-byte integer datatype.	Datatypes
CS_DATA_NOUINT8	No unsigned 8-byte integer datatype.	Datatypes
CS_DATA_NOUINTN	No unsigned integer datatype.	Datatypes
CS_DATA_NOUNITEXT	No unsigned short datatype.	Datatypes
CS_DATA_NOXML	No variable-width character datatype.	Datatypes
CS_DATA_NOZEROLEN	No zero length datatype.	Datatypes
CS_NOWIDETABLES	No wide tables.	Connection
CS_RES_NOEED	No extended error results.	Results
CS_RES_NOMSG	No message results.	Results
CS_RES_NOPARAM	Do not use PARAM/PARAMFMT TDS streams for RPC results.	Results
CS_RES_NOSTRIPBLANKS	The server should not strip blanks when returning data from nullable fixed-length character columns.	Results
CS_RES_NOTDSDEBUG	No TDS debug token in response to certain dbcc commands.	Results
CS_RES_RESERVED	Reserved for future use	Future

value

If a capability is being set, *value* points to a CS_BOOL variable that has the value CS_TRUE or CS_FALSE.

If a capability is being retrieved, value points to a CS_BOOL-sized variable, which ct_capability sets to CS_TRUE or CS_FALSE.

CS_TRUE indicates that a capability is enabled. For example, if the CS_RES_NOEED capability is set to CS_TRUE, no extended error data will be returned on the connection.

Note If capability is CS_ALL_CAPS, the value must point to a CS_CAP_TYPE structure.

Return value

ct_capability returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see “Asynchronous programming” on page 12.

Usage

- Capabilities describe client/server features that a connection supports.
- There are two types of capabilities: CS_CAP_RESPONSE capabilities, also called **response capabilities**, and CS_CAP_REQUEST capabilities, also called **request capabilities**.
  - An application uses request capabilities to determine what kinds of requests a server connection supports. For example, an application can retrieve the CS_REQ_CURSOR capability to find out whether a connection supports cursor requests.
  - An application uses response capabilities to prevent the server from sending a type of response that the application cannot process. For example, an application can prevent a server from sending NULL money values by setting the CS_DATA_NOMONEYN response capability to CS_TRUE.
- Before a connection is open, an application can:
  - Retrieve request or response capabilities, to determine what request and response features are normally supported at the application’s current TDS (Tabular Data Stream) version level. An application’s TDS level defaults to a value based on the CS_VERSION level that the application requested in its call to ct_init.
  - Set response capabilities, to indicate that a connection does not wish to receive particular types of server responses. An application *cannot* set request capabilities, which are retrieve-only.
- After a connection is open, an application can:
  - Retrieve request capabilities to find out what types of requests the connection supports.
  - Retrieve response capabilities to find out whether the server has agreed to withhold the previously indicated response types from the connection.

- Capabilities are determined by a connection's TDS version level. Not all TDS versions support the same capabilities. For example, 4.0 TDS does not support registered procedure notifications or cursor requests. However, 4.0 TDS does support bulk copy requests, remote procedure call requests, row results, and compute row results. A connection's TDS version level is negotiated during the connection process.
- If an application sets the CS_TDS_VERSION property, Client-Library overwrites existing capability values with default capability values corresponding to the new TDS version. For this reason, an application should set CS_TDS_VERSION before setting any capabilities for a connection.

  Because CS_TDS_VERSION is a negotiated login property, the server can change its value at connection time. If this occurs, Client-Library overwrites existing capability values with default capability values corresponding to the new TDS version.
- Because capability values can change at connection time, an application must call ct_capability after a connection is open to determine what capability values are in effect for the connection.
- When a connection is closed, Client-Library resets its capability values to values corresponding to the application's default TDS version.

Setting and retrieving multiple capabilities

- Gateway applications often need to set or retrieve all capabilities of a type category with a single call to ct_capability. To do this, an application calls ct_capability with:
  - *type* as the type category of interest
  - *capability* as CS_ALL_CAPS
  - *value* as a pointer to a CS_CAP_TYPE structure
- Client-Library provides the following macros to enable an application to set, clear, and test bits in a CS_CAP_TYPE structure:
  - CS_SET_CAPMASK(*mask*, *capability*)
  - CS_CLR_CAPMASK(*mask*, *capability*)
  - CS_TST_CAPMASK(*mask*, *capability*)

  where *mask* is a pointer to a CS_CAP_TYPE structure and *capability* is the capability of interest.

Configuring capabilities externally

- ct_connect optionally reads a section from the Open Client and Open Server runtime configuration file to set server capabilities for a connection.

  For a description of this feature, see “Using the runtime configuration file” on page 305.

See also

“Capabilities” on page 60, ct_con_props, ct_connect, ct_options, “Properties” on page 180

ct_close

Description

Close a server connection.

Syntax

CS_RETCODE ct_close(connection, option)

```
CS_CONNECTION    *connection;
CS_INT           option;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

option

The option to use for the close. The following table lists the symbolic values for *option*:

Value of option	Meaning
CS_UNUSED (10.0+ servers only)	Default behavior. ct_close sends a logout message to the server and reads the response to this message before closing the connection. If the connection has results pending, ct_close returns CS_FAIL.
CS_FORCE_CLOSE	The connection is closed whether or not results are pending, and without notifying the server. This option is primarily for use when an application is hung waiting for a server response. It is also useful if ct_results, ct_fetch, or ct_cancel returns CS_FAIL.

Return value

ct_close returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. See “Asynchronous programming” on page 12. If asynchronous network I/O is in effect and ct_close is called with option as CS_FORCE_CLOSE, it returns CS_SUCCEED or CS_FAIL immediately to indicate the network response. In this case, no completion callback event occurs.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12. Note that ct_close does not return CS_BUSY when called with *option* as CS_FORCE_CLOSE.

The most common reason for a ct_close(CS_UNUSED) failure is pending results on the connection.

Examples

```
CS_RETCODE retcode;
CS_INT close_option;
close_option = (status != CS_SUCCEED)? CS_FORCE_CLOSE :
               CS_UNUSED;

retcode = ct_close(connection, close_option);
if (retcode != CS_SUCCEED)
{
    ex_error("ex_con_cleanup: ct_close() failed");
    return retcode;
}
```

This code excerpt is from the *exutils.c* sample program.

Usage

- To deallocate a CS_CONNECTION, an application can call ct_con_drop after the connection has been successfully closed.
- A connection can become unusable due to error. If this occurs, Client-Library marks the connection as “dead.” An application can use the CS_CON_STATUS property to determine if a connection has been marked “dead.”

  If a connection has been marked dead, an application must call ct_close(CS_FORCE_CLOSE) to close the connection and ct_con_drop to drop its CS_CONNECTION structure.

An exception to this rule occurs for certain types of results-processing errors. If a connection is marked "dead" while processing results, the application can try calling ct_cancel(CS_CANCEL_ALL or CS_CANCEL_ATTN) to revive the connection. If this fails, the application must close the connection and drop its CS_CONNECTION structure.

- When a connection is closed, all open cursors on that connection are automatically closed.
- If the connection is using asynchronous network I/O, ct_close returns CS_PENDING. When the server response arrives, Client-Library closes the connection and then calls the completion callback installed for the connection.
- The behavior of ct_close depends on the value of *option*, which determines the type of close. Each section below contains information about a specific type of close.

Default close behavior

- If the connection has any pending results, ct_close returns CS_FAIL. If the connection has any open cursor(s), the server closes the cursor(s) when Client-Library closes the connection.
- When connected to a 10.0+ server, ct_close sends a logout message to the server and reads the response to this message before terminating the connection. The contents of this message do not affect ct_close's behavior.
- An application cannot call ct_close(CS_UNUSED) when an asynchronous operation is pending.

CS_FORCE_CLOSE behavior

- The connection is closed whether or not it has an open cursor or pending results.
- ct_close does not behave asynchronously when called with the CS_FORCE_CLOSE option. When ct_close(CS_FORCE_CLOSE) is called to close an asynchronous connection, it returns CS_SUCCEED or CS_FAIL immediately, to indicate the network response. In this case, no completion callback event occurs.
- CS_FORCE_CLOSE is useful when:
  - A connection has been marked as dead.
  - An application is hung, waiting for a server response.

- An application cannot call ct_close(CS_UNUSED) because results are pending.

- Because no logout message is sent to the server, the server cannot tell whether the close is intentional or whether it is the result of a lost connection or crashed client.
- An application can call ct_close(CS_FORCE_CLOSE) when an asynchronous operation is pending.

See also

ct_callback, ct_con_drop, ct_connect, ct_con_props

ct_cmd_alloc

Description

Allocate a CS_COMMAND structure.

Syntax

CS_RETCODE ct_cmd_alloc(connection, cmd_pointer)

```
CS_CONNECTION     *connection;
CS_COMMAND        **cmd_pointer;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

cmd_pointer

The address of a pointer variable. ct_cmd_alloc sets **cmd_pointer* to the address of a newly allocated CS_COMMAND structure.

Return value

ct_cmd_alloc returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

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

Examples

```
/* Allocate a command handle to send the text with */
 if ((retcode = ct_cmd_alloc(connection, &cmd)) !=
      CS_SUCCEED)
 {
      ex_error("UpdateTextData: ct_cmd_alloc() failed");
```

```
        return retcode;
}
```

This code excerpt is from the *getsend.c* sample program.

Usage

- A CS_COMMAND structure, also called a command structure, is a control structure that a Client-Library application uses to send commands to a server and process the results of those commands.
- An application must call ct_con_alloc to allocate a connection structure before calling ct_cmd_alloc to allocate command structures for the connection.

  However, it is not necessary that the connection structure represent an open connection. (An application opens a connection by calling ct_connect to connect to a server.)

See also

ct_command, ct_cmd_drop, ct_cmd_props, ct_con_alloc, ct_cursor, ct_dynamic

ct_cmd_drop

Description

Deallocate a CS_COMMAND structure.

Syntax

CS_RETCODE ct_cmd_drop(cmd)

CS_COMMAND *cmd;

Parameters

cmd

A pointer to a CS_COMMAND structure.

Return value

ct_cmd_drop returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

ct_cmd_drop returns CS_FAIL if:

- **cmd* has an active command. A command that has been initialized but not yet sent is considered to be active.
- **cmd* has an open cursor.

- **cmd* has pending results.

Examples

```
if ((retcode = ct_cmd_drop(cmd)) != CS_SUCCEED)
 {
      ex_error("DoCompute: ct_cmd_drop() failed");
      return retcode;
 }
```

This code excerpt is from the *compute.c* sample program.

Usage

- A CS_COMMAND structure is a control structure that a Client-Library application uses to send commands to a server and process the results of those commands.
- Once a command structure has been deallocated, it cannot be reused. To allocate a new CS_COMMAND structure, an application can call ct_cmd_alloc.
- Before deallocating a command structure, an application should cancel any active commands, process or cancel any pending results, and close and deallocate any open cursors on the command structure.

See also

ct_command, ct_cmd_alloc

ct_cmd_props

Description

Set or retrieve command structure properties. For use by applications that resend commands.

Syntax

CS_RETCODE ct_cmd_props(cmd, action, property, buffer, buflen, outlen)

```
CS_COMMAND    *cmd;
CS_INT        action;
CS_INT        property;
CS_VOID       *buffer;
CS_INT        buflen;
CS_INT        *outlen;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

action

One of the following symbolic values:

Value of action	Meaning
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 Client-Library default value.

property

The symbolic name of the property whose value is being set or retrieved. The Properties lists all Client-Library properties. For a summary of the properties that are legal with ct_cmd_props, see Table 3-7 on page 367.

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 ct_cmd_props will place the requested information.

buflen

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

If a property value is being set and the value in **buffer* is a null-terminated string, 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, ct_cmd_props sets **outlen* to the length, in bytes, of the requested information.

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

Return value

ct_cmd_props returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Return value	Meaning
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

Example for Command-Level User Data

- The following code fragment retrieves the CS_USERDATA property value. This code excerpt is from the *ex_alib.c* sample program. For another example using ct_cmd_props, see the *rpc.c* sample program.

```
 /*
 ** Extract the user area out of the command handle.
 */
 retstat = ct_cmd_props(cmd, CS_GET, CS_USERDATA,
     &ex_async, CS_SIZEOF(ex_async), NULL);
 if (retstat != CS_SUCCEED)
 {
     return retstat;

}
```

Example for Cursor Status

- This code fragment shows a function *cursor_status* that calls ct_cmd_props to retrieve status information about a Client-Library cursor.

```
#define RETURN_IF(a,b) if (a != CS_SUCCEED)\
   { fprintf(stderr, "Error in: %s line %d\n", \
             b, __LINE__); return a ;}

 /*
 ** cursor_status() -- Print status information about the
 **   Client-Library cursor (if any) declared on a CS_COMMAND
 **   structure.
 **
 ** PARAMETERS:
 **   cmd -- an allocated CS_COMMAND structure.
 **
 **
 ** RETURNS
 **   CS_FAIL if an error occurred.
 **   CS_SUCCEED if everything went ok.
 */

 CS_RETCODE
 cursor_status(cmd)
 CS_COMMAND *cmd;
 {
```

```
CS_RETCODE ret;
CS_INT     cur_status;
CS_INT     cur_id;
CS_CHAR    cur_name[CS_MAX_NAME];
CS_CHAR    updatability[CS_MAX_NAME];
CS_CHAR    status_str[CS_MAX_NAME];
CS_INT     outlen;

/*
** Get the cursor status property.
*/
ret = ct_cmd_props(cmd, CS_GET, CS_CUR_STATUS, &cur_status,
                   CS_UNUSED, (CS_INT *) NULL);
RETURN_IF(ret, "cursor_status: ct_cmd_props(CUR_STATUS)");

/*
** Is there a cursor?
** Note that CS_CURSTAT_NONE is not a bitmask, but the
** other values are.
*/
if (cur_status == CS_CURSTAT_NONE)
  fprintf(stdout,
     "cursor_status: no cursor on this command structure\n");
else
{
  /*
 ** A cursor exists, so check its state. Is it
 ** declared, opened, or closed?
 */
 if ((cur_status & CS_CURSTAT_DECLARED)
      == CS_CURSTAT_DECLARED)
    strcpy(status_str, "declared");
  if ((cur_status & CS_CURSTAT_OPEN) == CS_CURSTAT_OPEN)
    strcpy(status_str, "open");
  if ((cur_status & CS_CURSTAT_CLOSED) == CS_CURSTAT_CLOSED)
    strcpy(status_str, "closed");

  /*
  ** Is the cursor updatable or read only?
  */
  if ((cur_status & CS_CURSTAT_RDONLY) == CS_CURSTAT_RDONLY)
    strcpy(updatability, "read only");
  else if ((cur_status & CS_CURSTAT_UPDATABLE)
           == CS_CURSTAT_UPDATABLE)
    strcpy(updatability, "updatable");
  else
```

```
        updatability[0] = '\0';

      /*
      ** Get the cursor id.
      */
      ret = ct_cmd_props(cmd, CS_GET, CS_CUR_ID, &cur_id,
                         CS_UNUSED, (CS_INT *) NULL);
      RETURN_IF(ret, "cursor_status: ct_cmd_props(CUR_ID)");

      /*
      ** Get the cursor name.
      */
      ret = ct_cmd_props(cmd, CS_GET, CS_CUR_NAME, cur_name,
                         CS_MAX_NAME, &outlen);
      RETURN_IF(ret, "cursor_status: ct_cmd_props(CUR_NAME)");

      /*
      ** Null terminate the name.
      */
      if (outlen < CS_MAX_NAME)
        cur_name[outlen] = '\0';
      else
        RETURN_IF(CS_FAIL, "cursor_status: name too long");

      /* Print it all out                                        */
      fprintf(stdout, "Cursor '%s' (id %d) is %s and %s.\n",
              cur_name, cur_id, updatability, status_str);
    }
    return CS_SUCCEED;
  } /* cursor_status */
```

Example for Scrollable Cursor Status

- The code fragment for scrollable cursors is identical to the previous example, with the exception of the following extract:

```
      /*
      ** Is the cursor scrollable or read-only?
      */
      if (( cur_status & CS_CURSTAT_SCROLLABLE == CS_CURSTAT_SCROLLABLE )
          strcpy(updatability, "scrollable")
      else if ((cur_status & CS_CURSTAT_RDONLY) == CS_CURSTAT_RDONLY)
          strcpy(updatability, "read-only")
      else
          updatability[0] = '\0';
```

Usage

For information about *action*, *buffer*, *buflen*, and *outlen*, see Chapter 2, "Understanding Structures, Constants, and Conventions," in the Open Client *Client-Library/C Programmer's Guide*.

- Command structure properties affect the behavior of an application at the command structure level.
- All command structures allocated for a connection pick up default property values from the parent connection. An application can override these default values by calling ct_cmd_props.

  If an application changes connection property values after allocating command structures for the connection, the existing command structures will not pick up the new property values. New command structures allocated for the connection will use the new property values as defaults.
- See "Properties" on page 180 for more information about properties.
- An application can use ct_cmd_props to set or retrieve the properties listed in Table 3-7:

Table 3-7: Command structure properties

Property	Meaning	*buffer value	Level	Notes
CS_CUR_ID	The cursor's identification number.	Set to an integer value.	Command.	Retrieve only, after CS_CUR_STATUS indicates an existing cursor.
CS_CUR_NAME	The cursor's name, as defined in an application's ct_cursor(CS_CUR SOR_DECLARE) call.	Set to a null-terminated character string.	Command.	Retrieve only, after ct_cursor(CS_CURSO R_DECLARE) returns CS_SUCCEED.
CS_CUR_ ROWCOUNT	The current value of cursor rows. Cursor rows is the number of rows returned to Client-Library per internal fetch request.	Set to an integer value.	Command.	Retrieve only, after CS_CUR_STATUS indicates an existing cursor.
CS_CUR_STATUS	The cursor's status.	A CS_INT value. See "Cursor status" on page 208 for possible values.	Command.	Retrieve only.

Property	Meaning	*buffer value	Level	Notes
CS_HAVE_BINDS	Whether any saved result bindings are present for the current result set.	CS_TRUE or CS_FALSE. A default is not applicable.	Command.	Retrieve only.
CS_HAVE_CMD	Whether a resendable command exists for the command structure.	CS_TRUE or CS_FALSE	Command.	Retrieve only.
CS_HAVE_ CUROPEN	Whether or not a restorable cursor-open command exists for the command structure.	CS_TRUE or CS_FALSE	Command.	Retrieve only.
CS_HIDDEN_KEYS	Whether or not to expose hidden keys.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection, or command.	Cannot be set at the command level if results are pending or a cursor is open.
CS_PARENT_ HANDLE	The address of the command structure's parent connection.	Set to an address.	Connection or command.	Retrieve only.
CS_STICKY_BINDS	Whether or not bindings between result items and program variables persist when a command is executed repeatedly.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Command.	
CS_USERDATA	User-allocated data.	User-allocated data.	Connection or command. To set CS_ USERDAT A at the context level, call cs_config.	None.

See also ct_config, ct_cmd_alloc, ct_con_props, ct_res_info

ct_command

Description Initiate a language, package, RPC, message, or send-data command.

Syntax

```
CS_RETCODE ct_command(cmd, type, buffer, buflen,
                      option)
```

```
CS_COMMAND    *cmd;
CS_INT        type;
CS_VOID       *buffer;
CS_INT        buflen;
CS_INT        option;
```

Parameters

cmd
A pointer to the CS_COMMAND structure managing a client/server operation.

type
The type of command to initiate. Table 3-9 lists the symbolic values for *type*.

buffer
A pointer to data space. Table 3-9 lists the datatypes and meanings for **buffer* values.

buflen
The length, in bytes, of the **buffer* data, or CS_UNUSED if **buffer* represents a fixed-length or symbolic value.

option
The option associated with this command.

Language, RPC (remote procedure call), send-data, and send-bulk-data commands take options. For all other types of commands, pass *option* as CS_UNUSED.

The following table lists the symbolic values for *option*:

Table 3-8: Values for ct_command option parameter

type is	Value of option	Meaning
CS_LANG_CMD	CS_MORE	The text in buffer is only part of the language command to be executed.
	CS_END	The text in buffer is the last part of the language command to be executed.
	CS_UNUSED	Equivalent to CS_END.
CS_RPC_CMD	CS_RECOMPILE	Recompile the stored procedure before executing it.
	CS_NO_RECOMPILE	Do not recompile the stored procedure before executing it.
	CS_UNUSED	Equivalent to CS_NO_RECOMPILE.
CS_SEND_DATA_CMD	CS_COLUMN_DATA	The data will be used for a text or image column update.
	CS_BULK_DATA	For internal Sybase use only. The data will be used for a bulk copy operation.
CS_SEND_BULK_CMD	CS_BULK_INIT	For internal Sybase use only. Initialize a bulk copy operation.
	CS_BULK_CONT	For internal Sybase use only. Continue a bulk copy operation.

Return value

ct_command returns the following values:

Returns	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

```
/*
 ** ex_execute_cmd()
 **
 ** Type of function:
```

```
**      example program utility api
**
** Purpose:
** Sends a language command to the server.
*/
CS_RETCODE CS_PUBLIC
ex_execute_cmd(connection, cmdbuf)
CS_CONNECTION   *connection;
CS_CHAR         *cmdbuf;
{
CS_RETCODE      retcode;
CS_INT          restype;
CS_COMMAND      *cmd;
CS_RETCODE      query_code;

  /*
   ** Get a command structure,store the command string in it,
   ** and send it to the server.
   */
   if ((retcode = ct_cmd_alloc(connection, &cmd)) !=
        CS_SUCCEED)
   {
       ex_error("ex_execute_cmd: ct_cmd_alloc() failed");
       return retcode;
   }

  if ((retcode = ct_command(cmd, CS_LANG_CMD, cmdbuf,
        CS_NULLTERM, CS_UNUSED)) != CS_SUCCEED)
   {
       ex_error("ex_execute_cmd: ct_command() failed");
       (void)ct_cmd_drop(cmd);
       return retcode;
   }
  /* Now send the command and process the results */
   ... ct_send, ct_results, and so forth deleted ...
  return CS_SUCCEED;
}
```

This code excerpt is from the *exutils.c* sample program.

Usage

Table 3-9 summarizes ct_command usage.

Table 3-9: Summary of ct_command parameters

Value of type	Command initiated	buffer is	buflen is
CS_LANG_CMD	A language command	A pointer to a CS_CHAR array that contains all or part of the language command text. Use the CS_MORE and CS_END options to build the command text in pieces. See Table 3-8 for details.	The length of the **buffer* data or CS_NULLTERM.
CS_MSG_CMD	A message command	A pointer to a CS_INT variable that contains the message ID.	CS_UNUSED
CS_PACKAGE_CMD	A package command	A pointer to a CS_CHAR array that contains the name of the package.	The length of the **buffer* data or CS_NULLTERM.
CS_RPC_CMD	A remote procedure call command	A pointer to a CS_CHAR array that contains the name of the remote procedure.	The length of the **buffer* data or CS_NULLTERM.
CS_SEND_DATA_CMD	A send-data command	NULL.	CS_UNUSED.
CS_SEND_BULK_CMD	A Sybase internal send-bulk-data command	A pointer to a CS_CHAR array that contains the database table name.	The length of the **buffer* data or CS_NULLTERM.

- ct_command initiates several types of server commands.

  For an overview of Client-Library command types, see Chapter 5, “Choosing Command Types,” in the Open Client *Client-Library/C Programmer's Guide*.

- Initiating a command is the first step in sending it to a server. For a client application to execute a server command, Client-Library must convert the command to a symbolic command stream that can be sent to the server. The command stream contains information about the type of the command and the data needed for execution. For example, an RPC command requires a procedure name, the number of parameters, and (if needed) parameter values.

  The steps for executing a server command with ct_command are as follows:

  a Initiate the command by calling ct_command. This step sets up internal structures that are used in building a command stream to send to the server.

b Pass parameters for the command (if required) by calling ct_param or ct_setparam once for each parameter that the command requires.

Not all commands require parameters. For example, a remote procedure call command may or may not require parameters, depending on the stored procedure being called.

c Send the command to the server by calling ct_send. ct_send writes the symbolic command stream onto the command structure's parent connection.

d Handle the results of the command by calling ct_results repeatedly until it no longer returns CS_SUCCEED. See Chapter 6, "Writing Results-Handling Code," in the Open Client *Client-Library/C Programmer's Guide* for a discussion of processing results.

- ct_command command types other than send-data commands or send-bulk commands can be resent by calling ct_send immediately after the application has processed the results of the previous execution. Client-Library saves the initiated command information in the command structure until the application initiates a new command with ct_command, ct_cursor, ct_dynamic, or ct_sendpassthru. If parameter source variables for the command were specified with ct_setparam, then the application can change the parameter values without calling ct_setparam again. See "Resending commands" on page 580 for more information on this feature.
- An application can call ct_cancel with *type* as CS_CANCEL_ALL to clear a command that has been initiated but not yet sent.
- The following rules apply to the use of ct_command:
  - When a command structure is initiated, an application must either send the initiated command or clear it before a new command can be initiated with ct_command, ct_cursor, ct_dynamic, or ct_sendpassthru.
  - After sending a command, an application must completely process or cancel all results returned by the command's execution before initiating a new command on the same command structure.
  - An application cannot call ct_command to initiate a command on a command structure that is managing a cursor. The application must deallocate the cursor first (or use a different command structure).
- Each section below contains information about one of the types of commands that ct_command can initiate.

Language commands

- A language command contains a character string that represents one or more command in a server's own language. For example, the following language command contains a Transact-SQL command:

```
ct_command(cmd, CS_LANG_CMD,
           "select * from pubs2..authors",
          CS_NULLTERM,
          CS_UNUSED);
```

- ct_command's CS_MORE and CS_END *option* values allow the application to append text to the language buffer. Language command text can be assembled in pieces with consecutive calls.

- The language buffer can represent more than one server commands. For example, the following sequence builds a language command containing three Transact-SQL statements:

```
ct_command(cmd, CS_LANG_CMD,
           “select * from pubs2..titles ”,
           CS_NULLTERM, CS_MORE);
 ct_command(cmd, CS_LANG_CMD,
           “select * from pubs2..authors ”,
           CS_NULLTERM, CS_MORE);
 ct_command(cmd, CS_LANG_CMD,
           “select * from pubs2..publishers ”,
           CS_NULLTERM, CS_END);
```

  ct_command does not add white space when appending to the command buffer and the space must therefore be specified by the user.

- When the CS_UNUSED option is specified, Client-Library requires the application to pass the entire language text in one call to ct_command.

- A language command can be in any language, as long as the server to which it is directed can understand it. Adaptive Server understands Transact-SQL, but an Open Server application constructed with Server-Library can be written to understand any language.

- If the language command string contains host variables, an application can pass values for these variable by calling ct_param or ct_setparam once for each variable that the language string contains. Use ct_setparam if the command will be sent to the server more than once. See “Resending commands” on page 580 for more information.

- Transact-SQL variables must begin with an @ sign.

- An Adaptive Server language cursor generates a regular row result set when an application calls ct_command to execute a fetch language command against the cursor. The Transact-SQL fetch command generates regular row results containing a number of rows equal to the current "cursor rows" setting for the language cursor.

Message commands

- Message commands and results provide a way for clients and servers to communicate specialized information to one another. A message command is similar to an RPC command, but the command is identified by a number (called the **message ID**) rather than by an RPC name.
- A message command has an ID, which an application provides in a CS_INT variable. The application passes the address of the CS_INT as ct_command's *buffer* parameter.
- A custom Open Server application can be programmed with an event handler that responds to user-defined messages. IDs for user-defined messages must be greater than or equal to CS_USER_MSGID and less than or equal to CS_USER_MAX_MSGID.
- If a message requires parameters, the application calls ct_param or ct_setparam once for each parameter that the message requires. Use ct_setparam if the RPC command will be sent to the server more than once. See "Resending commands" on page 580 for more information.

Package commands

- A package command instructs an IBM DB/2 database server to execute a package. A package is similar to a remote procedure. It contains precompiled SQL statements that are executed as a unit when the package is invoked.
- If the package requires parameters, the application calls ct_param or ct_setparam once for each parameter that the package requires. Use ct_setparam if the package command will be sent to the server more than once. See "Resending commands" on page 580 for more information.

RPC commands

- An RPC (remote procedure call) command instructs a server to execute a stored procedure either on this server or a remote server.
- An application initiates an RPC command by calling ct_command with **buffer* as the name of the stored procedure to execute.

- If an application is using an RPC command to execute a stored procedure that requires parameters, the application must call ct_param or ct_setparam once for each parameter required by the stored procedure. Use ct_setparam if the RPC command will be sent to the server more than once. See "Resending commands" on page 580 for more information.
- After sending an RPC command with ct_send, an application processes the stored procedure's results with ct_results and ct_fetch. ct_results and ct_fetch are used to process both the result rows generated by the stored procedure and the return parameters and status from the procedure, if any.
- An alternative way to call a stored procedure is by executing a language command containing a Transact-SQL execute statement. When a stored procedure is executed through a language command, parameter values may be converted to character format (if necessary) and passed as part of the language command text, or they may be included in the command as host variables. With an RPC command, parameters can be passed in their declared datatypes with ct_param or ct_setparam.

Send-data commands

- An application uses a send-data command to write large amounts of text or image data to a server.
- An application typically calls:
  - ct_command to initiate the send-data command.
  - ct_data_info to set the I/O descriptor for the operation.
  - ct_send_data to write the value, in chunks, to the data stream.
  - ct_send to send the command to the server.
- Send-data commands cannot be re-sent.
- For more information about writing text or image values, see "text and image data handling" on page 284.

Send-bulk-data commands

- Internally, Sybase uses send-bulk-data commands as part of its implementation of the Bulk-Library routines.
- Send-bulk-data commands cannot be re-sent.

See also

ct_cmd_alloc, ct_cursor, ct_dynamic, ct_param, ct_send, ct_setparam

ct_compute_info

Description

Retrieve compute result information.

Syntax

```
CS_RETCODE ct_compute_info(cmd, type, colnum, buffer,
            buflen, outlen)
 CS_COMMAND     *cmd;
 CS_INT          type;
 CS_INT         colnum;
 CS_VOID        *buffer;
 CS_INT          buflen;
 CS_INT         *outlen;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server command.

type

The type of information to return. For a list of the symbolic values for *type*, see Table 3-10 on page 379.

colnum

The number of the compute column of interest, as it appears in the compute row result set. Compute columns appear in the order in which they are listed in the compute clause of a select statement. The first column is number 1, the second is number 2, and so forth.

buffer

A pointer to the space in which ct_compute_info will place the requested information.

If *buflen* indicates that **buffer* is not large enough to hold the requested information, ct_compute_info returns CS_FAIL.

buflen

The length, in bytes, of the **buffer* data space or CS_UNUSED if **buffer* represents a fixed-length or symbolic value.

outlen

A pointer to an integer variable.

Return value

ct_compute_info returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

Assume that the following command has been executed:

```
select dept, name, year, sales from employee
          order by dept, name, year
          compute count(name) by dept, name
```

1 The call:

```
CS_INT      mybuffer;
ct_compute_info(cmd, CS_BYLIST_LEN, CS_UNUSED,
        &mybuffer, CS_UNUSED, CS_UNUSED);
```

sets *mybuffer* to 2, because there are two items in the bylist.

2 The call:

```
CS_SMALLINT    mybuffer[2];
 CS_INT         outlength;
ct_compute_info(cmd, CS_COMP_BYLIST, CS_UNUSED,
         mybuffer, sizeof(mybuffer), &outlength)
```

copies the CS_SMALLINT values 1 and 2 into *mybuffer[0]* and *mybuffer[1]* to indicate that the bylist is composed of columns 1 and 2 from the select list.

3 The call:

```
CS_INT      mybuffer;
ct_compute_info(cmd, CS_COMP_COLID, 1, &mybuffer,
        CS_UNUSED,NULL);
```

sets *mybuffer* to 2, since *name* is the second column in the select list.

4 The call:

```
CS_INT     mybuffer;
ct_compute_info(cmd, CS_COMP_ID, CS_UNUSED,
         &mybuffer, CS_UNUSED, NULL);
```

sets *mybuffer* to 1 because there is only a single compute clause in the select statement.

5 The call:

```
CS_INT    mybuffer;
ct_compute_info(cmd, CS_COMP_OP, 1, &mybuffer,
        CS_UNUSED, NULL);
```

sets *mybuffer* to the symbolic value CS_OP_COUNT, since the aggregate operator for the first compute column is a *count*.

Usage

Table 3-10 summarizes ct_compute_info usage.

Table 3-10: Summary of ct_compute_info parameters

Value of type	Value of colnum	Information retrieved	*buffer is set to	*outlen is set to
CS_BYLIST_LEN	CS_UNUSED	The number of elements in the bylist array	An integer valu.	sizeof(CS_INT)
CS_COMP_BYLIST	CS_UNUSED	An array containing the bylist that produced this compute row	An array of CS_SMALLINT values	The length of the array, in bytes
CS_COMP_COLID	The column number of the compute column	The select-list column ID of the column from which the compute column derives	An integer value	sizeof(CS_INT)
CS_COMP_ID	CS_UNUSED	The compute ID for the current compute row	An integer value	sizeof(CS_INT)
CS_COMP_OP	The column number of the compute column	The aggregate operator type for the compute column	One of the following symbolic values: CS_OP_SUM CS_OP_AVG CS_OP_COUNT CS_OP_MIN CS_OP_MAX	sizeof(CS_INT)

- Compute rows result from the compute clause of a select statement. A compute clause generates a compute row every time the value of its by column-list changes. A compute row contains one column for each aggregate operator in the compute clause. If a select statement contains multiple compute clauses, separate compute rows are generated by each clause.

  Each compute row returned by the server is considered to be a distinct result set. That is, each result set of type CS_COMPUTE_RESULT will contain exactly one row.

- It is only legal to call ct_compute_info when compute information is available; that is, after ct_results returns CS_COMPUTE_RESULT or CS_COMPUTEFMT.

- Each section below contains information about a particular type of compute result information.

The bylist for a compute row

- A select statement's compute clause may contain the keyword by, followed by a list of columns. This list, known as the **bylist**, divides the results into subgroups, based on changing values in the specified columns. The compute clause's aggregate operators are applied to each subgroup, generating a compute row for each subgroup.

The select-list column ID for a compute column

- The select-list column ID for a compute column is the position within the select-list of the column from which the compute column derives.

The compute ID for a compute row

- A SQL select statement can have multiple compute clauses, each of which returns a separate compute row. The compute ID corresponding to the first compute clause in a select statement is 1.

The aggregate operator for a particular compute row column

- When called with *type* as CS_COMP_OP,ct_compute_info sets **buffer* to one of the following aggregate operator types:

Table 3-11: Aggregate operator types

*buffer setting	Meaning
CS_OP_AVG	Average aggregate operator
CS_OP_COUNT	Count aggregate operator
CS_OP_MAX	Maximum aggregate operator
CS_OP_MIN	Minimum aggregate operator
CS_OP_SUM	Sum aggregate operator.

See also

ct_bind, ct_describe, ct_res_info, ct_results

ct_con_alloc

Description

Allocate a CS_CONNECTION structure.

Syntax

CS_RETCODE ct_con_alloc(context, con_pointer)

```
CS_CONTEXT        *context;
CS_CONNECTION     **con_pointer;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

con_pointer
The address of a pointer variable. ct_con_alloc sets **con_pointer* to the address of a newly allocated CS_CONNECTION structure.

Return value

ct_con_alloc returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully
CS_FAIL	The routine failed

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

Examples

```
/*
 ** DoConnect()
 **
 ** Type of function:
 ** async example program api
 */
 CS_STATIC CS_CONNECTION CS_INTERNAL *
 DoConnect(argc, argv)
 int      argc;
 char     **argv;
 {
     CS_CONNECTION   *connection;
     CS_INT          netio_type = CS_ASYNC_IO;
     CS_RETCODE      retcode;
    /* Open a connection to the server */
     retcode = ct_con_alloc(Ex_context, &connection);
     if (retcode != CS_SUCCEED)
     {
         ex_panic("ct_con_alloc failed");
     }
    /* Set properties for the connection */
     ...ct_con_props calls deleted ...
    /* Open the connection */
     ...ct_connect call deleted.....
 }
```

Usage

- A CS_CONNECTION structure, also called a connection structure, contains information about a particular client/server connection.
- Before calling ct_con_alloc, an application must allocate a context structure by calling the CS-Library routine cs_ctx_alloc, and must initialize Client-Library by calling ct_init.

- Connecting to a server is a three-step process. To connect to a server, an application:
  - a Calls ct_con_alloc to allocate a CS_CONNECTION structure.
  - b Calls ct_con_props to set the values of connection-specific properties, if desired.
  - c Calls ct_connect to create the connection and log in to the server.
- An application can have multiple open connections to one or more servers at the same time.

  For example, an application can simultaneously have two connections to the server MARS, one connection to VENUS, and one connection to PLUTO. The context property CS_MAX_CONNECT, set by ct_config, determines the maximum number of open connections allowed per context.

  Each server connection requires a separate CS_CONNECTION structure.
- To send commands to a server, one or more command structures must be allocated for a connection. ct_cmd_alloc allocates a command structure.

See also

cs_ctx_alloc, ct_cmd_alloc, ct_close, ct_connect, ct_con_props

ct_con_drop

Description

Deallocate a CS_CONNECTION structure.

Syntax

```
CS_RETCODE ct_con_drop(connection)
CS_CONNECTION    *connection;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

Return value

ct_con_drop returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

The most common reason for a ct_con_drop failure is that the connection is still open.

Examples

```
/* ex_con_cleanup() */
CS_RETCODE CS_PUBLIC
 ex_con_cleanup(connection, status)
 CS_CONNECTION             *connection;
 CS_RETCODE                status;
 {
     CS_RETCODE    retcode;
     CS_INT        close_option;
    /* Close connection */
     ...CODE DELETED.....
    retcode = ct_con_drop(connection);
     if (retcode != CS_SUCCEED)
     {
         ex_error("ex_con_cleanup: ct_con_drop()
             failed");
         return retcode;
     }
    return retcode;
 }
```

This code excerpt is from the *exutils.c* sample program.

Usage

- When a CS_CONNECTION structure is deallocated, all CS_COMMAND structures associated with it are deallocated.
- A CS_CONNECTION structure contains information about a particular client/server connection.
- Once a CS_CONNECTION has been deallocated, it cannot be reused. To allocate a new CS_CONNECTION, an application can call ct_con_alloc.
- An application cannot deallocate a CS_CONNECTION structure until the connection it represents is closed. To close a connection, an application can call ct_close.
- A connection can become unusable due to error. If this occurs, Client-Library marks the connection as “dead.” An application can use the CS_CON_STATUS property to determine if a connection has been marked dead.

  If a connection has been marked dead, an application must call ct_close(CS_FORCE_CLOSE) to close the connection and ct_con_drop to drop its CS_CONNECTION structure.

An exception to this rule occurs for certain types of results-processing errors. If a connection is marked dead while processing results, the application can try calling ct_cancel(CS_CANCEL_ALL or CS_CANCEL_ATTN) to revive the connection. If this fails, the application must close the connection and drop its CS_CONNECTION structure.

See also

ct_con_alloc, ct_close, ct_connect, ct_con_props

ct_con_props

Description

Set or retrieve connection structure properties.

Syntax

```
CS_RETCODE ct_con_props(connection, action, property,
            buffer, buflen, outlen)

CS_CONNECTION    *connection;
CS_INT           action;
CS_INT           property;
CS_VOID          *buffer;
CS_INT           buflen;
CS_INT           *outlen;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

action

One of the following symbolic values:

Value of action	Result
CS_SET	Sets the value of the property.
CS_GET	Retrieves the value of the property.
CS_CLEAR	Resets the property to its default value.
CS_SUPPORTED	Checks whether a distributed-service driver supports the property. Use only with properties that affect the behavior of a directory or security driver. See “Checking whether a property is supported” on page 182.

property

The symbolic name of the property whose value is being set or retrieved. Table 3-12 on page 388 lists the properties that can be set with ct_con_props. “Properties” on page 180 lists all Client-Library properties.

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

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

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 and should be passed as NULL.

If a property value is being retrieved and outlen is supplied, ct_con_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

ct_con_props returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

Example 1 This code excerpt is from the *blktxt.c* sample program.

```
/*

 ** EstablishConnection()
 **
 ** Purpose:
 ** This routine establishes a connection to the server
 ** identified in example.h and sets the CS_USER,
 ** CS_PASSWORD, and CS_APPNAME properties for the
 ** connection.
 **
 ** NOTE: The user name, password, and server are defined
 ** in the example header file.
 */
CS_STATIC CS_RETCODE
 EstablishConnection(context, connection)
```

```
CS_CONTEXT      *context;
CS_CONNECTION   *connection;
{
    CS_INT      len;
    CS_RETCODE  retcode;
    CS_BOOL     bool;

   /* Allocate a connection structure */
    ...CODE DELETED.....

   /*
    ** If a user name is defined in example.h, set the
    ** CS_USERNAME property.
    */
    if (retcode == CS_SUCCEED && Ex_username != NULL)
    {
        if ((retcode = ct_con_props(*connection, CS_SET,
            CS_USERNAME, Ex_username, CS_NULLTERM, NULL))
            != CS_SUCCEED)
        {
            ex_error("ct_con_props(username) failed");
        }
    }

   /*
    ** If a password is defined in example.h, set the
    ** CS_PASSWORD property.
    */
    if (retcode == CS_SUCCEED && Ex_password != NULL)
    {
        if ((retcode = ct_con_props(*connection, CS_SET,
            CS_PASSWORD, Ex_password, CS_NULLTERM, NULL))
            != CS_SUCCEED)
        {
            ex_error("ct_con_props(passwd) failed");
        }
    }

   /* Set the CS_APPNAME property */
    ...CODE DELETED.....

   /* Enable the bulk login property */
    if (retcode == CS_SUCCEED)
    {
        bool = CS_TRUE;
        retcode = ct_con_props(*connection, CS_SET,
            CS_BULK_LOGIN, &bool, CS_UNUSED, NULL);
        if (retcode != CS_SUCCEED)
        {
```

```
                        ex_error("ct_con_props(bulk_login) failed");
                    }
                }

               /* Open a server connection */
                ...CODE DELETED.....

          }
```

Example 2 In the following example, CS_SEC_EXTENDED_ENCRYPTION is disabled:

```
...
CS_INT Ex_encryption = CS_FALSE;
CS_INT Ex_nonencryptionretry = CS_FALSE;
...
main()
{
    ...
    /*

    ** This needs to be called before calling ct_connect()
    */
     ret = ct_con_props(connection, CS_SET, CS_SEC_EXTENDED_ENCRYPTION,
           &Ex_encryption, CS_UNUSED, NULL);
    EXIT_ON_FAIL(context, ret, "Could not set extended encryption");
    ret = ct_con_props(connection, CS_SET, CS_SEC_NON_ENCRYPTION_RETRY,
          &Ex_nonencryptionretry, CS_UNUSED, NULL);
    EXIT_ON_FAIL(context, ret, "Could not set non encryption retry");
    ...
}
```

Usage

For information about *action*, *buffer*, *buflen*, and *outlen*, see Chapter 2, “Understanding Structures, Constants, and Conventions,” in the Open Client *Client-Library/C Programmer's Guide*.

- Connection properties define aspects of Client-Library behavior at the connection level. To determine whether a property is supported, an application can call ct_con_props on an established connection. The call must use the CS_SUPPORTED action parameter and must use the buffer parameter as the address of a CS_BOOL variable.
- All connections created within a context pick up default property values from the parent context. An application can override these default values by calling ct_con_props.

If an application changes context property values after allocating connections for the context, existing connections will not pick up the new property values. New connections allocated within the context will use the new property values as defaults.

- All command structures allocated for a connection pick up default property values from the parent connection. An application can override these default values by calling ct_cmd_props to set property values at the command structure level.

  If an application changes connection property values after allocating command structures for the connection, the existing command structures will not pick up the new property values. New command structures allocated for the connection will use the new property values as defaults.

- Some connection properties only take effect if they are set before an application calls ct_connect to establish the connection. These are indicated the "Notes" column in Table 3-12.

- See "Properties" on page 180 for more information.

- An application can use ct_con_props to set or retrieve the following properties:

Table 3-12: Connection structure properties

Property	Meaning	*buffer value	Level	Notes
CS_ANSI_BINDS	Whether to use ANSI-style binds.	CS_TRUE or CS_FALSE	Context, connection.	
CS_APPNAME	The application name used when logging into the server.	A character string	Context, connection. To set at the context level, call cs_config.	Login property. Cannot be set after connection is established.
CS_ASYNC_ NOTIFS	Whether a connection will receive registered procedure notifications asynchronously.	CS_TRUE or CS_FALSE.	Connection.	
CS_BULK_LOGIN	Whether a connection is enabled to perform bulk copy "in" operation.	CS_TRUE or CS_FALSE.	Connection.	Login property. Cannot be set after connection is established.

Property	Meaning	*buffer value	Level	Notes
CS_CHARSETCNV	Whether character set conversion is taking place.	CS_TRUE or CS_FALSE.	Connection.	Retrieve only, after connection is established.
CS_COMMBLOCK	A pointer to a communication sessions block. This property is specific to IBM370 systems and is ignored on all other platforms.	A pointer value.	Connection.	Cannot be set after connection is established.
CS_CONNECTED_ ADDR	The transport address of the server to which the current connection is established.	A valid transport address.	Connection.	This property cannot be set. It requires a pointer to a CS_TRANADDR structure that will be filled in with the server's address.
CS_CON_ KEEPALIVE	Whether to use the KEEPALIVE option.	CS_TRUE (default) or CS_FALSE	Context or connection	Some Net-Library protocol drivers do not support this property. After completing a connection on such a protocol driver, calling ct_con_props with CS_GET or CS_SET returns CS_FAIL.
CS_CON_STATUS	The connection’s status.	A CS_INT-sized bit-mask.	Connection.	Retrieve only.
CS_CON_TCP_ NODELAY	Whether to use the TCP_NODELAY option.	CS_TRUE (default) or CS_FALSE	Context or connection	Some Net-Library protocol drivers do not support this property. After completing a connection on such a protocol driver, calling ct_con_props with CS_GET or CS_SET returns CS_FAIL.

Property	Meaning	*buffer value	Level	Notes
CS_CONFIG_ BY_SERVERNAME	Whether ct_connect uses its *server_name* parameter or the value of the CS_APPNAME property as the section name to read external configuration data from.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means that CS_APPNAME is used.	Connection.	Requires initialization with CS_VERSION_110 or later.
CS_CONFIG_FILE	The name and path of the Open Client and Open Server runtime configuration file. See “Using the runtime configuration file” on page 305 for more information.	A character string. The default is NULL, which means a platform-specific default is used.	Connection.	Requires initialization with CS_VERSION_110 or later.
CS_DIAG_TIMEOUT	When inline error handling is in effect, whether Client-Library should fail or retry on timeout errors.	CS_TRUE or CS_FALSE.	Connection.	
CS_DISABLE_POLL	Whether to disable polling. If polling is disabled, ct_poll does not report asynchronous operation completions. Registered procedure notification will still be reported.	CS_TRUE or CS_FALSE.	Context, connection.	Useful in layered asynchronous applications.
CS_DS_COPY	Whether the directory service is allowed to satisfy an applications request with cached copies of directory entries.	CS_TRUE or CS_FALSE. The default is CS_TRUE, which allows cache use.	Connection.	Not supported by all directory providers.
CS_DS_DITBASE	Fully qualified name of directory node where directory searches begin.	A character string. The default is directory-provider specific.	Connection.	Not supported by all directory providers.

Property	Meaning	*buffer value	Level	Notes
CS_DS_EXPAND ALIAS	Whether the directory service expands directory alias entries.	CS_TRUE or CS_FALSE. The default is CS_TRUE, which allows alias expansion.	Connection.	Not supported by all directory providers.
CS_DS_FAILOVER	Whether to allow failover to the next *libtcl.cfg* entry or the *interfaces* file when a directory service driver can not be initialized.	CS_TRUE or CS_FALSE The default is CS_TRUE.	Connection.	
CS_DS_PASSWORD	Password to go with the directory user ID specified as CS_DS_PRINCIPAL.	A character string. The default is NULL.	Connection.	Not supported by all directory providers.
CS_DS_PRINCIPAL	A directory user ID for use of the directory service to go with the password specified as CS_DS_PASSWORD .	A character string. The default is NULL.	Connection.	Not supported by all directory providers.
CS_DS_PROVIDER	The name of the directory provider for the connection.	A character string. The default depends on directory driver configuration.	Connection.	
CS_DS_RAND_ OFFSET	Enables or disables random offset in connection lists.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Determined when the network address list is retrieved from the directory service.
CS_DS_SEARCH	Restricts the depth of a directory search.	A CS_INT sized symbolic value. For a list of possible values, see “Directory service search depth” on page 122.	Connection.	Not supported by all directory providers.

Property	Meaning	*buffer value	Level	Notes
CS_DS_SIZELIMIT	Restricts the number of directory entries that can be returned by a search started with ct_ds_lookup.	A CS_INT value greater than or equal to 0. A value of 0 indicates there is no size limit.	Connection.	
CS_DS_TIMELIMIT	Sets an absolute time limit, in seconds, for completion of directory searches.	A CS_INT value greater than or equal to 0. A value of 0 indicates there is no time limit.	Connection.	Not supported by all directory providers.
CS_EED_CMD	A pointer to a command structure containing extended error data.	A pointer value.	Connection.	Retrieve only.
CS_ENDPOINT	The file descriptor for a connection.	An integer value, or -1 if the platform does not support CS_END POINT	Connection.	Retrieve only, after connection is established.
CS_EXPOSE_FMTS	Whether to expose results of type CS_ROWFMT_RES ULT and CS_COMPUTEFMT_ RESULT.	CS_TRUE or CS_FALSE.	Context, connection.	Cannot be set after connection is established.
CS_EXTENDED_ ENCRYPT_CB	Whether the connection will set the asymmetrical password encryption using the non-default public key encryption handler.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_EXTERNAL_ CONFIG	Whether ct_connect reads an external configuration file to set properties and options for the connection to be opened.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Requires initialization with CS_VERSION_110 or later.

Property	Meaning	*buffer value	Level	Notes
CS_EXTRA_INF	Whether to return the extra information that's required when processing Client-Library messages inline using SQLCA, SQLCODE, and SQLSTATE structures.	CS_TRUE or CS_FALSE.	Context, connection.	
CS_HIDDEN_KEYS	Whether to expose hidden keys.	CS_TRUE or CS_FALSE.	Context, connection, command.	
CS_HOSTNAME	The host name of the client machine.	A character string.	Connection.	Login property. Cannot be set after connection is established.
CS_LOC_PROP	A CS_LOCALE structure that defines localization information.	A CS_LOCALE structure previously allocated by the application.	Connection.	Login property. Cannot be set after connection is established.
CS_LOGIN_STATUS	Whether the connection is open.	CS_TRUE or CS_FALSE.	Connection.	Retrieve only.
CS_LOOP_DELAY	The delay, in seconds, that ct_connect waits before retrying the sequence of addresses associated with a server name.	A CS_INT >= 0. The default is 0.	Connection.	CS_RETRY_ COUNT specifies the number of times to retry.
CS_NETIO	Whether network I/O is synchronous, fully asynchronous, or deferred-asynchronous	CS_SYNC_IO, CS_ASYNC_IO, CS_DEFER_IO.	Context, connection.	Asynchronous connections are either fully or deferred asynchronous, to match their parent context.
CS_ NOCHARSETCNV_ REQD	Whether the server performs character set conversion if the server's character set is different from the client's.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.

Property	Meaning	*buffer value	Level	Notes
CS_NOTIF_CMD	A pointer to a command structure containing registered procedure notification parameters.	A pointer value.	Connection.	Retrieve only.
CS_PACKETSIZE	The TDS packet size.	An integer value.	Connection.	Negotiated login property. Cannot be set after connection is established.
CS_PARENT_ HANDLE	The address of the connection structure's parent context.	Set to an address.	Connection, command.	Retrieve only.
CS_PARTIAL_TEXT	Indicates whether or not the client application should perform a partial update.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	This property must be set before a connection to the server is established. If the server does not support partial updates, this property will be reset to CS_FALSE.
CS_PASSWORD	The password used to log in to the server.	A character string.	Connection.	Login property.
CS_PROP_ APPLICATION_SPID	ASE SPID is saved during login and is available as the property. See "Extended failover" on page 211.	A CS_INT value corresponding to the server process ID (spid) on the server.	Connection.	Login property.
CS_PROP_ EXTENDED FAILOVER	Enables or disables server-provided failover targets.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context, connection.	Login property.
CS_PROP_ MIGRATABLE	Enables or disables connection migration.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context, connection.	Login property.
CS_PROP_ REDIRECT	Enables or disables login redirection support.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context, connection.	Login property.

Property	Meaning	*buffer value	Level	Notes
CS_PROP_SSL_ PROTOVERSION	The version of supported SSL/TLS protocols.	CS_INT	Context, connection	Must be one of the following values. • CS_SSLVER_20 • CS_SSLVER_30 • CS_SSLVER_TLS1
CS_PROP_SSL_ CIPHER	Comma-separated list of CipherSuite names.	CS_CHAR	Context, connection	
CS_PROP_SSL_ LOCALID	Property used to specify the path to the Local ID (certificates) file.	Character string	Context connection	A structure containing a file name and a password used to decrypt the information in the file.
CS_PROP_SSL_CA	Specify the path to the file containing trusted CA certificates.	CS_CHAR	Context, connection	
CS_RETRY_COUNT	The number of times to retry a connection to a server's address.	A CS_INT >= 0. The default is 0.	Connection.	Affects only the establishment of a login dialog. Failed logins are not retried.
CS_SEC_ APPDEFINED	Whether the connection will use application-defined challenge/response security handshaking.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_ CHALLENGE	Whether the connection will use Sybase-defined challenge/response security handshaking.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_ CHANBIND	Whether the connection's security mechanism will perform channel binding.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ CONFIDENTIALITY	Whether data encryption service will be performed on the connection.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.

Property	Meaning	*buffer value	Level	Notes
CS_SEC_ CREDENTIALS	Used by gateway applications to forward a delegated user credential.	A CS_VOID * pointer.	Context, connection.	Cannot be read. Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ CREDTIMEOUT	Whether the user's credentials have expired.	A CS_INT. See Table 2-32 on page 257 for possible values and their meanings.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ DATAORIGIN	Whether the connection's security mechanism will perform data origin verification.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ DELEGATION	Whether to allow the server to connect to a second server with the user's delegated credentials.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ DETECTREPLAY	Whether the connection's security mechanism will detect replayed transmissions.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ DETECTSEQ	Whether the connection's security mechanism will detect transmissions that arrive out of sequence.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ ENCRYPTION	Whether the connection will use symmetrical password encryption.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.

Property	Meaning	*buffer value	Level	Notes
CS_SEC_ EXTENDED_ ENCRYPTION	Whether the connection will use asymmetrical password encryption.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_NON_ ENCRYPTION_ RETRY	Whether the connection will use plain text password retries when the server cannot use symmetrical or asymmetrical password encryption.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_ INTEGRITY	Whether the connection's security mechanism will perform data integrity checking.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ KEYTAB	The name and path to the file from which a connection's security mechanism reads the security key to go with the CS_USERNAME property.	A character string. The default is NULL, which means the user must have established credentials before the application calls ct_connect.	Connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ MECHANISM	The name of the network security mechanism that performs security services for the connection.	A string value. The default depends on security driver configuration.	Context, connection.	Cannot be set after connection is established.
CS_SEC_ MUTUALAUTH	Whether the server is required to authenticate itself to the client.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SEC_ NEGOTIATE	Whether the connection will use trusted-user security handshaking.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.

Property	Meaning	*buffer value	Level	Notes
CS_SEC_ NETWORKAUTH	Whether the connection's security mechanism will perform network-based user authentication.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism and a preexisting credential that matches CS_USERNAME.
CS_SEC_ SERVERPRINCIPAL	The network security principal name for the server to which a connection will be opened.	A string value. The default is NULL, which means that ct_connect assumes the server principal name is the same as its *server_name* parameter.	Connection.	Cannot be set after connection is established. Meaningful only for connections that use network-based user authentication.
CS_SEC_ SESSTIMEOUT	Whether the connection's security session has expired.	A CS_INT. See Table 2-32 on page 257 for possible values and their meanings.	Context, connection.	Cannot be set after connection is established. Requires a supporting network security mechanism.
CS_SERVERADDR	The address of the server to which you are connected to.	The format "hostname portnumber [filter], where filter is optional.	Connection	Using this property causes ctlib to bypass the host name of the server and the port number of the interfaces.
CS_SERVERNAME	The name of the server to which you are connected.	A string value.	Connection.	Retrieve only, after connection is established.
CS_TDS_VERSION	The version of the TDS protocol that the connection is using.	A symbolic version level.	Connection.	Negotiated login property. Cannot be set after connection is established.
CS_TEXTLIMIT	The largest text or image value to be returned on this connection.	An integer value.	Context, connection.	

Property	Meaning	*buffer value	Level	Notes
CS_TRANSACTION_NAME	A transaction name to be used over a connection to Open Server for CICS.	A string value.	Connection.	
CS_USERDATA	User-allocated data.	User-allocated data.	Connection, command.	
CS_USERNAME	The name used to log in to the server.	A character string.	Connection.	Login property. Cannot be set after connection is established.
CS_VALIDATE_CB	A Client-Library routine, registered through ct_callback	An integer value	Connection, command	

See also

ct_capability, ct_cmd_props, ct_connect, ct_config, ct_init, “Properties” on page 180

ct_config

Description

Set or retrieve context properties.

Syntax

```
CS_RETCODE ct_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:

Value of action	Result
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 Client-Library default value.

Value of action	Result
CS_SUPPORTED	Checks whether a distributed-service driver supports the property. Use only with properties that affect the behavior of a security or directory driver. See “Checking whether a property is supported” on page 182.

property

The symbolic name of the property whose value is being set or retrieved. Table 3-13 on page 402 lists the Client-Library context properties. “Properties” on page 180 lists all Client-Library properties.

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 ct_config will place the requested information.

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 value, symbolic value, or function, 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, ct_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.

Return value

ct_config returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully
CS_FAIL	The routine failed

Examples

```
/* Set the input/output type to asynchronous */
 CS_INT        propvalue;
 if (retcode == CS_SUCCEED)
 {
     propvalue = CS_ASYNC_IO;
     retcode = ct_config(*context, CS_SET, CS_NETIO,
         (CS_VOID *)&propvalue, CS_UNUSED, NULL);
     if (retcode != CS_SUCCEED)
     {
         ex_error("ex_init: ct_config(netio) failed");
     }
 }
```

This code excerpt is based on code in the *exutils.c* sample program.

Usage

For information about *action*, *buffer*, *buflen*, and *outlen*, see Chapter 2, “Understanding Structures, Constants, and Conventions,” in the Open Client *Client-Library/C Programmer's Guide*.

- Context properties define aspects of Client-Library behavior at the context level.
- ct_config takes precedence over the *libtcl*.cfg* file for all connections established within the CS_CONTEXT
- ct_config controls connection properties and the use of external files that configure context. See “Using the runtime configuration file” on page 305.
- All connections created within a context pick up default property values from the parent context. An application can override these default values by calling ct_con_props to set property values at the connection level.

  If an application changes context property values after allocating connections for the context, existing connections will not pick up the new property values. New connections allocated within the context will use the new property values as defaults.
- 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-specific context properties. Properties set through cs_config affect only CS-Library.

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

srv_props sets and retrieves the values of Server-Library-specific context properties. Properties set through srv_props affect only Server-Library.

- See “Properties” on page 180 for more information.
- An application can use ct_config to set or retrieve the following properties:

Table 3-13: Client-Library context structure properties

Property	Meaning	*buffer value	Level	Notes
CS_ANSI_BINDS	Whether to use ANSI-style binds.	CS_TRUE or CS_FALSE.	Context, connection	
CS_DISABLE_POLL	Whether to disable polling. If polling is disabled, ct_poll does not report asynchronous operation completions.	CS_TRUE or CS_FALSE.	Context, connection	Useful in layered asynchronous applications.
CS_DS_RAND_ OFFSET	Enables or disables random offset in connection lists.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Determined when the network address list is retrieved from the directory service.
CS_EXPOSE_FMTS	Whether to expose results of type CS_ROWFMT_RESULT and CS_COMPUTEFMT_ RESULT.	CS_TRUE or CS_FALSE.	Context, connection	Takes effect only if set before connection is established.
CS_EXTERNAL_ CONFIG	Whether ct_connect reads an external configuration file to set properties and options for the connection to be opened.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	Requires initialization with CS_VERSION_11 0 or later.
CS_EXTRA_INF	Whether to return the extra information that’s required when processing Client-Library messages inline using SQLCA, SQLCODE, and SQLSTATE structures.	CS_TRUE or CS_FALSE.	Context, connection	
CS_HIDDEN_KEYS	Whether to expose hidden keys.	CS_TRUE or CS_FALSE.	Context, connection, command	
CS_IFILE	The path and name of the *interfaces* file.	A character string.	Context	

Property	Meaning	*buffer value	Level	Notes
CS_LOGIN_ TIMEOUT	The login timeout value.	An integer value.	Context	
CS_MAX_CONNECT	The maximum number of connections for this context.	An integer value.	Context	
CS_MEM_POOL	A memory pool that Client-Library will use to satisfy interrupt-level memory requirements.	If *action* is CS_SET, **buffer* is a pool of bytes. If *action* is CS_GET, **buffer* is set to the address of a pool of bytes.	Context	Useful in asynchronous applications. Cannot be set or cleared when context has connections.
CS_NETIO	Whether network I/O is synchronous, fully asynchronous, or deferred asynchronous.	CS_SYNC_IO, CS_ASYNC_IO, or CS_DEFER_IO.	Context, connection	Cannot be set for a context with open connections.
CS_NO_TRUNCATE	Whether Client-Library should truncate or sequence messages that are longer than CS_MAX_MSG.	CS_TRUE, which means sequence or CS_FALSE, which means truncate.	Context	
CS_NOAPI_CHK	Whether Client-Library performs argument and state checking when the application calls a Client-Library routine.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means that Client-Library performs API checking.	Context	
CS_NOINTERRUPT	Whether the application can be interrupted by certain callback events.	CS_TRUE or CS_FALSE.	Context	Affects completion events only, not notification events.
CS_PARTIAL_TEXT	Indicates whether or not the client application should perform a partial update.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	This property must be set before a connection to the server is established. If the server does not support partial updates, this property will be reset to CS_FALSE.

Property	Meaning	*buffer value	Level	Notes
CS_PROP_EXTENDEDFAILOVER	Enables or disables server-provided failover targets.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context, connection	Login property.
CS_PROP_MIGRATABLE	Enables or disables connection migration.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context, connection	Login property.
CS_PROP_REDIRECT	Enables or disables login redirection support.	CS_TRUE or CS_FALSE. The default is CS_TRUE.	Context, connection	Login property.
CS_PROP_SSL_PROTOVERSION	The version of supported SSL/TLS protocols.	CS_INT	Context, connection	Must be one of the following values: • CS_SSLVER_20 • CS_SSLVER_30 • CS_SSLVER_TLS1
CS_PROP_SSL_CIPHER	Comma-separated list of CipherSuite names.	CS_CHAR	Context, connection	
CS_PROP_SSL_LOCALID	Property used to specify the path to the Local ID (certificates) file.	Character string	Context, connection	A structure containing a file name and a password used to decrypt the information in the file.
CS_PROP_SSL_CA	Specify the path to the file containing trusted CA certificates.	CS_CHAR	Context, connection	
CS_SEC_CHANBIND	Whether the connection's security mechanism will perform channel binding.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Requires a supporting network security mechanism.
CS_SEC_CONFIDENTIALITY	Whether data encryption service will be performed on the connection.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Requires a supporting network security mechanism.

Property	Meaning	*buffer value	Level	Notes
CS_SEC_ CREDENTIALS	Used by gateway applications to forward a delegated user credential.	A CS_VOID * pointer.	Context, connection	Cannot be read. Requires a supporting network security mechanism.
CS_SEC_ CREDTIMEOUT	Whether the user's credentials have expired.	A CS_INT. See Table 2-32 on page 257 for possible values and their meanings.	Context, connection	Requires a supporting network security mechanism.
CS_SEC_ DATAORIGIN	Whether the connection's security mechanism will perform data origin verification.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Requires a supporting network security mechanism.
CS_SEC_ DELEGATION	Whether to allow the server to connect to a second server with the user's delegated credentials.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Requires a supporting network security mechanism.
CS_SEC_ DETECTREPLAY	Whether the connection's security mechanism will detect replayed transmissions.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Requires a supporting network security mechanism.
CS_SEC_ DETECTSEQ	Whether the connection's security mechanism will detect transmissions that arrive out of sequence.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Requires a supporting network security mechanism.
CS_SEC_ INTEGRITY	Whether the connection's security mechanism will perform data integrity checking.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Requires a supporting network security mechanism.
CS_SEC_ MECHANISM	The name of the network security mechanism that performs security services for the connection.	A string value. The default depends on security driver configuration.	Context, connection	
CS_SEC_ MUTUALAUTH	Whether the server is required to authenticate itself to the client.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Requires a supporting network security mechanism.

Property	Meaning	*buffer value	Level	Notes
CS_SEC_ NETWORKAUTH	Whether the connection's security mechanism will perform network-based user authentication.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Requires a supporting network security mechanism.
CS_SEC_ SESSTIMEOUT	Whether the connection's security session has expired.	A CS_INT. See Table 2-32 on page 257 for possible values and their meanings.	Context, connection	Requires a supporting network security mechanism.
CS_TEXTLIMIT	The largest text or image value to be returned on this connection.	An integer value.	Context, connection	
CS_TIMEOUT	The timeout value.	An integer value.	Context	
CS_USER_ALLOC	A user-defined memory allocation routine.	If *action* is CS_SET, **buffer* is the user-defined function to install. If *action* is CS_GET, **buffer* is set to the address of the user-defined function that is currently installed.	Context	Useful in asynchronous application.
CS_USER_FREE	A user-defined memory free routine.	If *action* is CS_SET, **buffer* is the user-defined function to install. If *action* is CS_GET, **buffer* is set to the address of the user-defined function that is currently installed.	Context	Useful in asynchronous applications.
CS_VER_STRING	Client-Library's true version string.	A character string.	Context	Retrieve only.
CS_VERSION	The version of Client-Library in use by this context.	A symbolic version level.	Context	Retrieve only.

See also

cs_config, ct_cmd_props, ct_capability, ct_con_props, ct_connect, ct_init, "Properties" on page 180

ct_connect

Description

Connect to a server.

Syntax

```
CS_RETCODE ct_connect(connection, server_name,
            snamelen)

 CS_CONNECTION    *connection;
 CS_CHAR          *server_name;
 CS_INT            snamelen;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

Use ct_con_alloc to allocate a CS_CONNECTION structure, and ct_con_props to initialize that structure with login parameters.

server_name

A pointer to the name of the server to connect to. **server_name* is the name of the server's entry in the connection's directory source. ct_connect looks up **server_name* in the connection's directory source to determine how to connect to that server. A connection's directory source is specified with the CS_DS_PROVIDER property. See "Directory service provider" on page 120. This can be either the Sybase *interfaces* file or a network-based directory service.

server_name must use the naming syntax recognized by the connection's directory provider. Most network-based directory providers allow a base directory path (DIT base) to be specified with the CS_DS_DITBASE connection property. If *server_name* is a partially qualified name, the directory provider combines it with the DIT base to form a fully qualified name.

If *server_name* is NULL, ct_connect uses a platform-specific default for the server name. On platforms that support environment variables or logical names, this is the value of the DSQUERY environment variable or logical name. On these platforms, if DSQUERY has not been set, ct_connect looks for a server with the name SYBASE.

snamelen

The length, in bytes, of **server_name*. If **server_name* is null-terminated, pass *snamelen* as CS_NULLTERM. If *server_name* is NULL, pass *snamelen* as 0 or CS_UNUSED.

Return value

ct_connect returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. See “Asynchronous programming” on page 12.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Common reason for a ct_connect failure include:

- Unable to allocate sufficient memory.
- The maximum number of connections is already established. Use ct_config to increase the maximum number of connections allowed per context.
- Unable to open socket.
- Server name not found in *interfaces* file.
- Unknown host machine name.
- Adaptive Server is unavailable or does not exist.
- Login incorrect.
- Cannot open *interfaces* file or a directory service session.
- Cannot load requested directory driver.

When ct_connect returns CS_FAIL, it generates a Client-Library error number that indicates the error.

Note If ct_connect is called in the entry functions of a DLL, a deadlock may arise. This system creates OS threads and tries to synchronize them using system utilities. This synchronization conflicts with the operating system’s serialization process.

Examples

```
/* ex_connect() */
 CS_RETCODE CS_PUBLIC
 ex_connect(context, connection, appname,username, password,
       server)
   CS_CONTEXT       context;
   CS_CONNECTION  *connection;
   CS_CHAR      *appname;
   CS_CHAR      *username;
   CS_CHAR      *password;
```

```
CS_CHAR      *server;
{
    CS_INT        len;
    CS_RETCODE    retcode;

   /* Allocate a connection structure */
    ...CODE DELETED.....

   /* Set properties for new connection */
    ...CODE DELETED.....

   /* Open the connection */
    if (retcode == CS_SUCCEED)
    {
        len = (server == NULL) ? 0 : CS_NULLTERM;
        retcode = ct_connect(*connection, server, len);
        if (retcode != CS_SUCCEED)
        {
            ex_error("ct_connect failed");
        }
    }

   if (retcode != CS_SUCCEED)
    {
        ct_con_drop(*connection);
        *connection = NULL;
    }

   return retcode;
}
```

This code excerpt is from the *exutils.c* sample program.

Usage

- Information about the connection is stored in a CS_CONNECTION structure, which uniquely identifies the connection. In the process of establishing a connection, ct_connect sets up communication with the network, logs into the server, and communicates any connection-specific property information to the server.
- Because creating a connection involves logging into a server, an application must define login parameters (such as a server user name and password) before calling ct_connect. An application can call ct_con_props to define login parameters.
- A connection can be either synchronous or asynchronous. The Client-Library property CS_NETIO determines whether a connection is synchronous or asynchronous.

  For more information about asynchronous connections, see “Asynchronous programming” on page 12.

- The maximum number of open connections per context is determined by the CS_MAX_CONNECT property (set by ct_config). If not explicitly set, the maximum number of connections defaults to a platform-specific value. For information about platform-specific property values, see the Open Client and Open Server *Programmer's Supplement* for Microsoft Windows or Open Client and Open Server *Programmer's Supplement* for UNIX.
- When a connection attempt is made between a client and a server, there are two ways in which the process can fail (assuming that the system is correctly configured):
  - The machine that the server is supposed to be on is running correctly and the network is running correctly.

    In this case, if no server is listening on the specified port, the machine that the server is supposed to be on will inform the client, through a network error, that the connection cannot be formed. Regardless of the login timeout value, the connection will fail.
  - The machine that the server is on is down.

    In this case, the machine that the server is supposed to be on will not respond. Because “no response” is not considered to be an error, the network will not inform the client that an error has occurred. However, if a login timeout period has been set, a timeout error will occur when the client fails to receive a response within the set period.

    The CS_LOGIN_TIMEOUT property specifies a login timeout period. See “Login timeout” on page 216.
- To close a connection, an application calls ct_close.

Server address information

- Client-Library requires a directory source that contains the network addresses associated with a given server name. The directory source can be either the Sybase *interfaces* file or a network-based directory service.
- The directory source used by ct_connect depends on the setting of the CS_DS_PROVIDER connection property. See “Directory service provider” on page 120 for a description of the CS_DS_PROVIDER property.
- For information on network-based directory services, see “Directory services” on page 104 and “Server directory object” on page 276.
- More than one address can be associated with a server name. ct_connect begins a login dialog at the first address where a server responds.

- The CS_RETRY_COUNT property controls how many times ct_connect retries each server address.
- The CS_LOOP_DELAY property controls how long ct_connect waits before retrying the sequence again.

See “Retry count” on page 227 and “Loop delay” on page 217 for descriptions of these properties.

Configuring connection defaults externally

- ct_connect optionally reads the Open Client and Open Server runtime configuration file to set connection properties, server options, and debugging options for the connection. This feature allows a programmer to externalize settings rather than hard-coding calls to ct_con_props, ct_options, and ct_debug.
- By default, ct_connect does not read the configuration file. The application must set the CS_EXTERNAL_CONFIG property to enable external configuration. For more information on this feature, see “Using the runtime configuration file” on page 305.

See also

ct_close, ct_con_alloc, ct_con_drop, ct_con_props, ct_remote_pwd, “Directory services” on page 104, “Interfaces file” on page 142, “Properties” on page 180, “Server directory object” on page 276

ct_cursor

Description

Initiate a Client-Library cursor command.

Syntax

```
CS_RETCODE ct_cursor(cmd, type, name, namelen, text,
            textlen, option)

CS_COMMAND      *cmd;
 CS_INT          type;
 CS_CHAR        *name;
 CS_INT          namelen;
 CS_CHAR        *text;
 CS_INT          textlen;
 CS_INT          option;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

type
The type of cursor command to initiate. Table 3-14 lists the symbolic values for *type*.

name
A pointer to the name associated with the cursor command, if any. Table 3-14 on page 419 indicates which types of commands require names.

namelen
The length, in bytes, of **name*. If **name* is null-terminated, pass *namelen* as CS_NULLTERM. If *name* is NULL pass *namelen* as CS_UNUSED.

text
A pointer to the text associated with the cursor command. Table 3-14 indicates which commands require text and what that text must be.

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

option
The option associated with this command. Table 3-14 indicates which commands take an option and what that option can be.

Return value

ct_cursor returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully
CS_FAIL	The routine failed
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

Example 1 The following code excerpt is from the *csr_disp.c* sample program and describes the functionality of regular cursors:

```
/* DoCursor(connection) */
 CS_STATIC CS_RETCODE
 DoCursor(connection)
 CS_CONNECTION  *connection;

{
     CS_RETCODE       retcode;
     CS_COMMAND       *cmd;
     CS_INT            res_type;

    /* Use the pubs2 database */
     ...CODE DELETED.....

    /*
```

```
 ** Allocate a command handle to declare the
 ** cursor on.
 */
 retcode = ct_cmd_alloc(connection, &cmd)
 if (retcode != CS_SUCCEED)
 {
     ex_error("DoCursor: ct_cmd_alloc() failed");
     return retcode;
 }

/*
 ** Declare the cursor. SELECT is a select
 ** statement defined in the header file.
 */
 retcode = ct_cursor(cmd, CS_CURSOR_DECLARE,
     "cursor_a", CS_NULLTERM, SELECT, CS_NULLTERM,
     CS_READ_ONLY);
 if (retcode != CS_SUCCEED)
 {
     ex_error("DoCursor: ct_cursor(declare)
         failed");
         return retcode;
 }

/* Set cursor rows to 10*/
 retcode = ct_cursor(cmd, CS_CURSOR_ROWS, NULL,
     CS_UNUSED, NULL, CS_UNUSED, (CS_INT)10);
 if (retcode != CS_SUCCEED)
 {
     ex_error("DoCursor: ct_cursor(currows)
         failed");
     return retcode;
 }

/* Open the cursor */
 retcode = ct_cursor(cmd, CS_CURSOR_OPEN, NULL,
     CS_UNUSED, NULL,     CS_UNUSED, CS_UNUSED);
 if (retcode != CS_SUCCEED)
 {
     ex_error("DoCursor: ct_cursor() failed");
     return retcode;
 }

/*
 ** Send (batch) the last 3 cursor commands to
 ** the server
 */
 retcode = ct_send(cmd)
```

```
    if (retcode != CS_SUCCEED)
    {
        ex_error("DoCursor: ct_send() failed");
        return retcode;
    }

   /*
    ** Process the results.  Loop while ct_results()
    ** returns CS_SUCCEED, and then check ct_result's
    ** final return code to see if everything went ok.
    */
    ...CODE DELETED.....

   /*
    ** Close and deallocate the cursor. Note that we
    ** don't have to do this, since it is done
    ** automatically when the connection is closed.
    */
    retcode = ct_cursor(cmd, CS_CURSOR_CLOSE, NULL,
        CS_UNUSED, NULL, CS_UNUSED, CS_DEALLOC);
    if (retcode != CS_SUCCEED)
    {
        ex_error("DoCursor: ct_cursor(dealloc)
            failed");
        return retcode;
    }

   /* Send the cursor command to the server */
    retcode = ct_send(cmd)
    if (retcode != CS_SUCCEED)
    {
        ex_error("DoCursor: ct_send() failed");
        return retcode;
    }

   /*
    ** Check its results.  The command won't generate
    ** fetchable results.
    */
    ...CODE DELETED.....

   /* Drop the cursor's command structure */
    ...CODE DELETED.....

   return retcode;
}
```

Example 2 The following code excerpt is from the *csr_disp_scrollcurs.c* sample program and describes the functionality of scrollable cursors:

```
CS_STATIC CS_RETCODE
DoCursor(connection)
CS_CONNECTION*connection;
{
    CS_RETCODEretcode;
    CS_COMMAND*cmd;
    CS_INT res_type;

        if ((retcode = ex_use_db(connection, Ex_dbname)) != CS_SUCCEED)
        {
                ex_error("DoCursor: ex_use_db(pubs2) failed");
                return retcode;
        }

        if ((retcode = ct_cmd_alloc(connection, &cmd)) != CS_SUCCEED)
        {
                ex_error("DoCursor: ct_cmd_alloc() failed");
                return retcode;
        }

    /*
    ** Declare an insensitive, scrollable cursor. The same result
    ** would be obtained by using CS_SCROLL_INSENSITIVE.
    */
    retcode = ct_cursor(cmd, CS_CURSOR_DECLARE, "cursor_a", CS_NULLTERM,
          SELECT, CS_NULLTERM, CS_SCROLL_CURSOR);

    if (retcode != CS_SUCCEED)
    {
                ex_error("DoCursor: ct_cursor(declare) failed");
       return retcode;
    }

    /*
    ** This example relies on CS_CURSOR_ROWS set to 1, e.g. fetch a single
    ** row at any time for the server. No row buffering here.
    */
    retcode = ct_cursor(cmd, CS_CURSOR_ROWS, NULL, CS_UNUSED, NULL,
          CS_UNUSED, (CS_INT)1);
    if (retcode != CS_SUCCEED)
    {
                ex_error("DoCursor: ct_cursor(currows) failed");
       return retcode;
    }

    retcode = ct_cursor(cmd, CS_CURSOR_OPEN, NULL, CS_UNUSED, NULL,
```

```
            CS_UNUSED, CS_UNUSED);
      if (retcode != CS_SUCCEED)
      {
                  ex_error("DoCursor: ct_cursor() failed");
         return retcode;
      }

      if ((retcode = ct_send(cmd)) != CS_SUCCEED)
      {
                  ex_error("DoCursor: ct_send() failed");
         return retcode;
      }

      while((retcode = ct_results(cmd, &res_type)) == CS_SUCCEED)
      {
         switch ((int)res_type)
         {
         case CS_CMD_SUCCEED:
            break;

         case CS_CMD_DONE:
            break;

         case CS_CMD_FAIL:
            ex_error("DoCursor: ct_results() returned CMD_FAIL");
            break;

         case CS_CURSOR_RESULT:
            retcode = ex_scroll_fetch_1(cmd);
            if (retcode != CS_SUCCEED)
            {
               if (retcode == CS_SCROLL_CURSOR_ENDS ||
                  retcode == CS_CURSOR_BEFORE_FIRST ||
                  retcode == CS_CURSOR_AFTER_LAST)
               {
                  retcode = CS_SUCCEED;
               }
            }
            else
            {
               ex_error("DoCursor: ex_scroll_fetch_1() failed on
CS_CURSOR_RESULT ");
               return retcode;
            }
            break;
```

```
        default:
            ex_error("DoCursor: ct_results() returned unexpected result
type");
            return CS_FAIL;
        }
    }

    switch ((int)retcode)
    {
        case CS_SUCCEED:
        case CS_END_RESULTS:
            break;

        case CS_FAIL:
            ex_error("DoCursor: ct_results() failed");
            return retcode;

        default:
            ex_error("DoCursor: ct_results() returned unexpected result
code");
            return retcode;
    }

    /*
    ** cursor close only
    */
    retcode = ct_cursor(cmd, CS_CURSOR_CLOSE, NULL, CS_UNUSED, NULL,
            CS_UNUSED, CS_UNUSED);

    if (retcode != CS_SUCCEED)
    {
                ex_error("DoCursor: ct_cursor(close) failed");
        return retcode;
    }

    if ((retcode = ct_send(cmd)) != CS_SUCCEED)
    {
                ex_error("DoCursor: ct_send() for close failed");
        return retcode;
    }

    while((retcode = ct_results(cmd, &res_type)) == CS_SUCCEED)
    {
        switch ((int)res_type)
        {
            case CS_CMD_SUCCEED:
```

```
            case CS_CMD_DONE:
           break;

        case CS_CMD_FAIL:
           ex_error("DoCursor: ct_results() close returned CMD_FAIL");
           break;

        default:
           ex_error("DoCursor: ct_results() close returned unexpected result
type");
           return CS_FAIL;
        }
    }

    if (retcode != CS_END_RESULTS)
    {
        ex_error("DoCursor: close ENDRESULTS ct_results() failed");
        return retcode;
    }

    /*
    ** cursor dealloc only, but this could be combined with the close.
    */
    retcode = ct_cursor(cmd, CS_CURSOR_DEALLOC, NULL, CS_UNUSED, NULL,
           CS_UNUSED, CS_UNUSED);

    if (retcode != CS_SUCCEED)
    {
                ex_error("DoCursor: ct_cursor(cursor_dealloc) failed");
        return retcode;
    }

    if ((retcode = ct_send(cmd)) != CS_SUCCEED)
    {
                ex_error("DoCursor: ct_send() for dealloc failed");
        return retcode;
    }

    while((retcode = ct_results(cmd, &res_type)) == CS_SUCCEED)
    {
        switch ((int)res_type)
        {
            case CS_CMD_SUCCEED:
            case CS_CMD_DONE:
           break;
```

```
        case CS_CMD_FAIL:
           ex_error("DoCursor: ct_results() returned CMD_FAIL");
           break;
        default:
           ex_error("DoCursor: ct_results() returned unexpected result
type");
           return CS_FAIL;
        }
     }

     if (retcode != CS_END_RESULTS)
     {
        ex_error("DoCursor: cursor_dealloc ENDRESULTS ct_results() failed");
        return retcode;
     }

     if ((retcode = ct_cmd_drop(cmd)) != CS_SUCCEED)
     {
                 ex_error("DoCompute: ct_cmd_drop() failed");
                 return retcode;
     }

     return retcode;
  }
```

Usage

Table 3-14: Summary of ct_cursor parameters

Value of type	Command initiated	name value	text value	option value
CS_CURSOR_ DECLARE	A cursor declare command.	A pointer to the cursor name.	A pointer to the SQL text that is the body of the cursor.	CS_UNUSED, or a bitwise OR of the values in Table 3-15 on page 426.

Value of type	Command initiated	name value	text value	option value
CS_CURSOR_ OPTION	A cursor set options command.	NULL	NULL	• CS_FOR_UPDATE to indicate that the cursor is “for update.” • CS_READ_ONLY to indicate that the cursor is “read-only.” • CS_UNUSED to indicate that the server should decide whether a cursor is updatable. • CS_SCROLL_INSENSITIVE to declare a scrollable cursor insensitive. • CS_SCROLL_SEMISENSITIVE to declare a scrollable, semi sensitive cursor. • CS_SCROLL_CURSOR to declare a scrollable cursor that is insensitive (default). • CS_NOSCROLL_INSENSITIVE to declare a cursor insensitive and non-scrollable. See Specifying updatability on page 373 for more information.
CS_CURSOR_ROWS	A cursor set rows command.	NULL	NULL	An integer representing the number of rows to be returned with a single fetch request. The default is 1 if not specified in subsequent ct_cursor calls. For multi-row returns from ct_scroll_fetch, the CS_CURSOR_ROWS value must be greater than 1. For best performance, set CS_CURSOR_ROWS to the same value as the count field in the ct_bind call. See “ct_bind” on page 323 for more information.

Value of type	Command initiated	name value	text value	option value
CS_CURSOR_OPEN	A cursor open command.	NULL	NULL	• CS_RESTORE_OPEN restores parameter-binding information for a previously sent cursor-open command. See "Restoring a cursor-open command" on page 432 for an explanation. • CS_UNUSED should be passed the first time a cursor is opened.
CS_CURSOR_ UPDATE	A cursor update command.	A pointer to the name of the table to update.	A pointer to the SQL update statement.	• CS_UNUSED if *text is the entire update statement. • CS_MORE if *text is part of the update statement. • CS_END if *text is the last piece of the update statement.
CS_CURSOR_ DELETE	A cursor delete command.	A pointer to the name of the table to delete from.	NULL	CS_UNUSED
CS_CURSOR_ CLOSE	A cursor close command.	NULL	NULL	• CS_DEALLOC to close and deallocate the cursor. • CS_UNUSED to close the cursor without deallocating it.
CS_CURSOR_ DEALLOC	A deallocate cursor command.	NULL	NULL	CS_UNUSED

- Initiating a command is the first step in sending it to a server. Client-Library cursor commands include commands to declare, open, set cursor rows, close, and deallocate a cursor as well as commands to update and delete rows in an underlying table. Chapter 7, "Using Client-Library Cursors," in the Open Client *Client-Library/C Programmer's Guide* contains additional information on Client-Library cursors.
- To send a cursor command to a server, an application must:

  a Initiate the command by calling ct_cursor. This sets up internal structures that are used in building a command stream to send to the server.

  b Pass parameters for the command (if required) by calling ct_param or ct_setparam once for each parameter that the command requires.

c Cursor-declare, cursor-open, and cursor-update commands may require parameters. Other cursor commands do not.

d Send the command to the server by calling ct_send.

e Handle the results of the command by calling ct_results until it returns CS_END_RESULTS, CS_CANCELED, or CS_FAIL. A cursor-open command returns a CS_CURSOR_RESULT result type (and possibly other result types indicating status information). Other cursor commands do not return fetchable results, but they do return result types that indicate command status. See “Results” on page 241 for a discussion of processing results.

- Client-Library allows an application to resend commands by calling ct_send immediately after the results of the previous execution have been processed. An application can resend any command that was initiated with ct_cursor. However, only cursor-update and cursor-delete commands can be reexecuted successfully on the server. Other cursor commands must be executed in a specific sequence and resending them can cause server processing errors.

Sequencing cursor commands

- Servers require cursor commands to be executed in the sequence described below. Each step is a separate server command that generates distinct results:

  a Declare the cursor. This step identifies the source query for the cursor and optionally identifies which (if any) columns in the cursor’s result set can be updated. Cursors can be declared with ct_cursor or ct_dynamic. ct_cursor details for this step are under “Cursor-declare commands” on page 423. For ct_dynamic cursor declarations, see “Declaring a cursor on a prepared statement” on page 476.

  b Specify cursor options (only for cursors declared with ct_dynamic). For details, see “Dynamic SQL cursor option” on page 428.

  c Specify the cursor rows setting. For details, see “Cursor-Rows commands” on page 429.

  d Open the cursor. The first time a cursor is opened, the commands in steps 1–4 can be batched to reduce the number of network round-trips to the server and back. For details, see “Cursor-open commands” on page 430 and “Batching cursor-open commands” on page 431.

e Process the cursor-open results with ct_results and ct_fetch, or in the case of a scrollable cursor, with ct_results and ct_scroll_fetch. Each time ct_fetch returns CS_SUCCEED or CS_ROW_FAIL, the application can issue nested cursor-update or cursor-delete commands on the same CS_COMMAND structure. The application can also send new commands (unrelated to the cursor), as long as the application uses a different CS_COMMAND structure and processes the results of the command before fetching from the cursor again. Results processing is described in "Results" on page 241. For details on nested cursor commands, see "Cursor-update commands" on page 433 and "Cursor-delete commands" on page 434.

f Close the cursor as described by "Cursor-close commands" on page 434. Closed cursors can be reopened: steps c through f can be repeated indefinitely. A cursor can be reopened by initiating a new cursor-open command or by restoring the previously initiated cursor-open command. For details, see "Restoring a cursor-open command" on page 432.

g Deallocate the cursor. For details, see "Cursor-deallocate commands" on page 435.

Cursor-declare commands

- Declaring a Client-Library cursor is equivalent to associating the cursor name with a select statement. This SQL statement is called the **body** of the cursor.
- The following rules apply to ct_cursor cursor-declare commands:
  - Only one cursor may be declared for each CS_COMMAND structure. However, another cursor can be declared on a separate CS_COMMAND structure that shares the same connection.
  - All operations on a Client-Library cursor, from its declaration to its deallocation, must reference the command structure with which the cursor was created.
  - When a cursor is declared on a CS_COMMAND structure, the structure can not be used to execute ct_command, ct_dynamic, or ct_sendpassthru server commands until the cursor is deallocated.
  - Cursors associated with a dynamic SQL statement are declared with ct_dynamic, not with ct_cursor.

- The cursor body can either be specified directly as the *text parameter, or indirectly as the text of a stored procedure. In the case of the stored procedure, the *text parameter must be a command to execute the stored procedure. A cursor declared with a stored procedure is called an **execute cursor**.
  - The following example declares a cursor named *title_cursor* on rows from the *titles* table.

    ```
    ct_cursor (cmd, CS_CURSOR_DECLARE,
         "title_cursor",CS_NULLTERM,
         "select * from titles", CS_NULLTERM,
         CS_UNUSED);

    ct_send(cmd);
    ```

  - The following example declares an execute cursor on the stored procedure title_cursor_proc:

    ```
    ct_cursor (cmd, CS_CURSOR_DECLARE,
         "mycursor", CS_NULLTERM,
         "exec title_cursor_proc", CS_NULLTERM,
         CS_UNUSED);

    ct_send(cmd);
    ```

    In this case, the body of the cursor is the text that makes up the stored procedure. The stored procedure text must contain a single select statement only. In the example above, title_cursor_proc could be created as:

    ```
    create proc title_cursor_proc as
       select * from titles for read only
    ```

    Note A stored procedure used with an execute cursor must consist of a single select statement. The stored procedure's return status is not available to the client program. Output parameter values are also not available to the client program.

- A select statement associated with a cursor can contain host variables. If it does, you must describe the format for each variable after declaring the cursor. To describe the format of each host variable, first initialize a CS_DATAFMT structure to describe the variable's format; then call ct_param with the CS_DATAFMT as a parameter.

At cursor-declare time, ct_param only provides format information for the host-language variables. At cursor-open time, actual values are provided by calling ct_param with parameter values or ct_setparam with pointers to parameter source variables.

- An execute statement associated with a cursor should not contain host language variables, and you do not need to specify variable formats with ct_param at cursor-declare time. At cursor-open time, supply values for the procedure's parameters using ct_param or ct_setparam. For execute cursors, the declaration of the stored procedure determines the formats of the procedure's parameters.

Option values for ct_cursor(CS_CURSOR_DECLARE)

- The following values can be passed for the ct_cursor *option* parameter when initiating a cursor-declare command:

Table 3-15: Option values for ct_cursor(CS_CURSOR_DECLARE)

Value of option	Meaning
CS_MORE	Indicates that *text is only part of the cursor body, with the rest to be supplied in subsequent calls. If this bit is set, all other options are ignored. If this bit is not set, then *text is taken to be the entire cursor body.
CS_END	Indicates that *text is the last piece of the cursor body.
CS_FOR_UPDATE	Indicates that the cursor is "for update." Can be used with CS_END or by itself. When this option appears by itself, the entire cursor body must be specified with one call. **Note** For Adaptive Server connections, use the for update of or for read only clauses in the cursor body to specify whether the cursor rows can be updated. Adaptive Server does not recognize *option* values.
CS_READ_ONLY	Indicates that the cursor is read-only. Can be used with CS_END or by itself.
CS_UNUSED	Equivalent to setting the CS_END bit (only).
CS_IMPLICIT_ CURSOR	This is a TDS-based client cursor with optimizations to potentially reduce the number of network round-trips. New rows inserted after the last row fetch are not seen by subsequent fetches.
CS_SCROLL_ INSENSITIVE	Declares insensitive scrollable cursors. At cursor open time, the cursor result set is static and the number of rows known. Changes to the base table are not visible.
CS_SCROLL_ SEMISENSITIVE	Declares semi-sensitive scrollable cursors. At cursor open time, the number of rows in the cursor result set is not known. The cursor result set changes when data in the base table changes.
CS_SCROLL_ CURSOR	Maps to insensitive scrollable cursors.
CS_NOSCROLL_ INSENSITIVE	Declares insensitive, read-only cursors with forward direction only. This option is only offered through ct_fetch and cannot be used with ct_scroll_fetch.

- To build the cursor's text value in pieces, use the CS_MORE and CS_END option values. A sequence of one or more ct_cursor calls that use CS_MORE must be ended with a call that specifies CS_END, as demonstrated below:

```
ct_cursor(cmd, CS_CURSOR_DECLARE,
       "select title_id, contract ", ..., CS_MORE);
 ct_cursor(cmd, CS_CURSOR_DECLARE,
```

```
        "from titles ", ... , CS_MORE);
ct_cursor(cmd,  CS_CURSOR_DECLARE,
        "where contract=FALSE ", ...., CS_MORE);
ct_cursor(cmd, CS_CURSOR_DECLARE,
         "for update of contract", ...., CS_END);
```

The last cursor-declare call must specify CS_END. Note that CS_READ_ONLY and CS_FOR_UPDATE are illegal with CS_MORE. If you need to set either of these option bits, set it in the last call (for example, use CS_END | CS_READONLY).

Client-Library does not add white space when appending the *text values.

- The CS_FOR_UPDATE and CS_READ_ONLY options are passed to the server. If neither option is set, then the server decides whether the cursor is updatable based on the content of the cursor body specified by *text.

Specifying updatability

- When declaring a Client-Library cursor, the application must specify whether the cursor is **updatable**; that is, whether the application intends to update the retrieved cursor rows using ct_cursor update commands. Depending on the destination server, this is done either by the content of the cursor's body statement or with the ct_cursor option parameter.

  If the server is a Adaptive Server, the select statement associated with the cursor defines whether the table rows can be updated. Applications use the Transact-SQL clauses for update of or for read only to specify whether the cursor is updatable. For example, the statement in the call below specifies that the price column will be updated, and all other columns will not:

```
#define TITLE_CUR \
 "select title_id, title, price from titles \
 for update of price"

ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          "titles_cursor", CS_NULLTERM,
          TITLE_CUR, CS_NULLTERM, CS_END);
```

  If the server is an Open Server application, it may require that the client use the CS_READ_ONLY and CS_FOR_UPDATE options, or it may parse the select statement. The choice depends on the design of the Open Server application. If the server requires *option* to determine whether the cursor is updatable, then the ct_cursor usage is as follows:

  - To declare a cursor as "read-only," an application specifies *option* as CS_READ_ONLY.
  - To declare a cursor "for update," an application specifies *option* as CS_FOR_UPDATE.

If some of a cursor's columns are "for update," an application indicates which columns are "for update" by calling ct_param once for each update column. If all of a cursor's columns are "for update," an application does not have to call ct_param to identify the update columns.

For example, to indicate that the *au_id* and *au_lname* columns are "for update":

```
    ct_cursor(cmd, CS_CURSOR_DECLARE,
"au_cursor",
         CS_NULLTERM, "select * from authors"
         CS_NULLTERM, CS_FOR_UPDATE);
     format.status = CS_UPDATECOL;
     ct_param(cmd, &format, "au_id",
              CS_NULLTERM, 0);
     format.status = CS_UPDATECOL;
     ct_param(cmd, &format, "au_lname",
              CS_NULLTERM, 0);
     ct_send(cmd);
```

To indicate that all columns returned by a cursor are "for update":

```
    ct_cursor(cmd, CS_CURSOR_DECLARE,
 "au_cursor",
           CS_NULLTERM, "select * from authors"
            CS_NULLTERM, CS_FOR_UPDATE);
     ct_send(cmd);
```

Dynamic SQL cursor option

- A dynamic SQL application can declare a cursor on a prepared statement. To declare a cursor on a prepared statement, call ct_dynamic(CS_CURSOR_DECLARE); from that point on, use ct_cursor calls to manipulate the cursor.
- The dynamic SQL cursor declare command does not provide a way to specify cursor options. To set options, call ct_cursor(CS_CURSOR_OPTION) after calling ct_dynamic and before calling ct_send.
- If the server is an Adaptive Server, the CS_READ_ONLY and CS_FOR_UPDATE options do not affect the underlying server tables. The select statement associated with the cursor defines whether the table rows can be updated.
- If the server is an Open Server application, the CS_READ_ONLY and CS_FOR_UPDATE options may be used by the server.

In this case, if some but not all of a cursor's columns are "for update," an application must indicate which columns are "for update" by calling ct_param once for each update column. If all of a cursor's columns are "for update," an application does not have to call ct_param to identify update columns.

- Cursor options must be specified before the cursor-declare command is sent.

Cursor-Rows commands

- A ct_cursor(CS_CURSOR_ROWS) command specifies the number of rows that the server returns to Client-Library per internal fetch request. Note that this is not the number of rows returned to an application per ct_fetch call. The number of rows returned to an application per ct_fetch call is determined by the value of the *count* field in the CS_DATAFMT structures used in binding the cursor result columns.
- An application can set cursor rows only before opening a cursor.
- The cursor rows setting defaults to one row.

Using implicit cursors

You can use implicit cursors with Client-Library version 12.5 and later. Implicit cursors function in the same way as read-only cursors during row-fetching, but they use system resources more efficiently.

This example uses read-only cursors:

```
ct_cursor(cmd, CS_CURSOR_DECLARE, "cursor_a",
CS_NULLTERM, SELECT, CS_READ_ONLY)

ct_cursor(cmd, CS_CURSOR_ROWS, NULL, CS_UNUSED, NULL,
CS_UNUSED, CS_INT)5)

ct_cursor(cmd, CS_CURSOR_OPEN, NULL, CS_UNUSED, NULL,
CS_UNUSED, CS_UNUSED)
```

This example uses implicit cursors:

```
ct_cursor(cmd, CS_CURSOR_DECLARE, "cursor_a",
CS_NULLTERM, SELECT, CS_IMPLICIT CURSOR)

ct_cursor(cmd, CS_CURSOR_ROWS, NULL, CS_UNUSED, NULL,
CS_UNUSED, CS_INT)5)

ct_cursor(cmd, CS_CURSOR_OPEN, NULL, CS_UNUSED, NULL,
CS_UNUSED, CS_UNUSED)
```

To use implicit cursors, you must set `cs_ctx_alloc(CS_VERSION_125, context)` or `ct_init(*context, CS_VERSION_125)`. You must set CS_CURSOR_ROWS to a minimum value of 2 for single-row fetches, and a higher value if more rows are to be retrieved.

Warning! You can use implicit cursors only with Client-Library version 12.5 and later. If you use them with an earlier version of Client-Library, they are converted to read-only cursors.

Cursor-open commands

- A ct_cursor(CS_CURSOR_OPEN) command executes the body of a Client-Library cursor, generating a CS_CURSOR_RESULT result set.
  - To access the cursor rows, an application processes the cursor result set by calling ct_results, ct_bind, and ct_fetch.
  - While fetching rows in a cursor result set, the application can send nested cursor commands (cursor update, cursor delete, cursor close) using the same CS_COMMAND structure.
  - While fetching rows in a cursor result set, the application can also send non-cursor commands to the server (or declare and open another cursor) by using a separate CS_COMMAND structure.
- The cursor must have been declared with ct_cursor(CS_DECLARE) or ct_dynamic(CS_CURSOR_DECLARE) before it can be opened. A closed cursor can be reopened.

  If the cursor is declared with ct_cursor, the declare and open commands can be batched. For more information, see “Batching cursor-open commands” on page 431.
- Cursors may require parameter values at cursor-open time. An application can pass input parameter values for a cursor-open command by calling ct_param or ct_setparam after calling ct_cursor. A cursor-open command requires parameters if any of the following conditions is true:
  - The body of the cursor is a SQL statement that contains host variables.
  - The body of the cursor is a stored procedure that requires input parameter values.
  - The body of the cursor is a dynamic SQL statement that contains dynamic parameter markers.

- To open a cursor on a dynamic SQL prepared statement, specify the same command structure used to dynamically declare the cursor (ct_dynamic(CS_CURSOR_DECLARE)).
- The first time a cursor is opened, all the server commands to declare the cursor, set cursor rows, and open the cursor can be sent with a single call to ct_send. For subsequent cursor-open commands, the application can use the CS_RESTORE_OPEN option to eliminate redundant ct_cursor(CS_CURSOR_ROWS) and ct_param calls. For a description of these features, see:
  - "Batching cursor-open commands" on page 431, and
  - "Restoring a cursor-open command" on page 432.
  - Text for cursor-open commands can be assembled in pieces with multiple ct_cursor calls. To specify the open statement in pieces, use the CS_MORE and CS_END values for the *option* parameter.

Batching cursor-open commands

- When opening a cursor, an application can batch ct_cursor commands to reduce network traffic and improve application performance. All the commands required to declare and open the cursor can be sent with one call to ct_send.

  To batch commands to declare, set rows for, and open a Client-Library cursor, the application:

  a Calls ct_cursor to declare the cursor

  b If necessary, calls ct_param or ct_setparam to define the format(s) of host variables.

  c If desired, calls ct_cursor to set rows for the cursor.

  d Calls ct_cursor to open the cursor.

  e If necessary, calls ct_param or ct_setparam to supply value(s) for the host variable(s). The application should use ct_setparam if it will reopen the cursor using the CS_RESTORE_OPEN option. ct_setparam binds program variables to input parameters, allowing the application to change parameter values when resending a command. If the application uses ct_param, the parameter values cannot be changed when the cursor-open command is restored.

  f Calls ct_send to send the command batch to the server.

  The sequence of calls is:

```
ct_cursor(CS_CURSOR_DECLARE)
ct_param or ct_setparam for each parameter
ct_cursor(CS_CURSOR_ROWS)
ct_cursor(CS_CURSOR_OPEN)
ct_param or ct_setparam for each parameter
ct_send
ct_results
```

Each of the batched commands generates separate results, and several calls to ct_results are required.

Restoring a cursor-open command

- When reopening a cursor, an application can use the CS_RESTORE_OPEN option to restore the most recently sent cursor-open command.
- If the application used ct_param to supply parameter values for the original cursor-open command, then the restored cursor-open command will use the same parameter values. If the application used ct_setparam, then the application can change the parameter values for the restored cursor-open command.
- If the application batched a cursor-rows command with the previous cursor-open command, then Client-Library resends the cursor-rows command with the cursor-open command. The cursor is reopened with the same cursor-rows setting.
- The sequence of calls for restoring a cursor-open command is:

```
/*
 ** Assign new variables in the program variables
 ** bound with ct_setparam.
 */
  ... assignment statement for each parameter
      source value ...


ct_cursor(CS_CURSOR_OPEN, CS_RESTORE_OPEN)
ct_send
... handle cursor results ...
```

- An application cannot restore a cursor that has been deallocated.
- An application can check the CS_HAVE_CUROPEN property to see whether a restorable cursor-open command exists for a command structure. See “Have restorable cursor-open command” on page 214 for a description.

- Applications that restore cursor-open commands may benefit from setting the CS_STICKY_BINDS command property. When CS_TRUE, this property allows the application to reuse the original cursor result bindings and eliminate redundant ct_bind calls. See "Persistent result bindings" on page 224 for a description of this property.

Cursor-update commands

- A ct_cursor(CS_CURSOR_UPDATE) command defines new column values for the current cursor row. These new values are used to update an underlying table.
- A cursor update command is always "nested"; that is, the command is sent from within the ct_results loop while the cursor's rows are being processed by ct_fetch.

  A nested cursor command can be sent after ct_results returns a *result_type* value of CS_CURSOR_RESULT. At least one row must be fetched before a cursor update command is allowed, and cursor update commands are not allowed after ct_fetch returns CS_END_DATA.
- By default, the last-fetched row is updated. The application can redirect the update to another row in the cursor result set. To redirect the update, specify different key values with ct_keydata before sending the cursor update command.
- When updating an Adaptive Server table, an application must specify the name of the table to update twice: once as the value of ct_cursor's **name* parameter and a second time in the update statement itself (update *tablename*).
- Text for cursor-open and cursor-update commands can be assembled in pieces with multiple ct_cursor calls. To specify the update statement in pieces, use the CS_MORE and CS_END values for the *option* parameter. CS_MORE indicates that the application intends to append more text to the update statement. To be specified in pieces, an update statement must update only a single table.
- The text of the update statement can contain host-language variables. If it does, the application must specify values for the variables with ct_param or ct_setparam before calling ct_send. Use ct_setparam if the cursor-update command requires parameters and will be sent to the server more than once.
- A cursor-update command generates results like any other command. The application must process the results before it can fetch from the cursor again.

- Cursor-update commands can be resent by calling ct_send immediately after the results of the previous execution have been handled. A cursor-update command can be resent as long as
  - The application has not initiated a new nested cursor command,
  - The cursor is still open, and
  - ct_fetch has not returned CS_END_DATA.

Cursor-delete commands

- A ct_cursor(CS_CURSOR_DELETE) command deletes a row from the cursor result set. The delete is propagated back to the underlying server tables.
- A cursor-delete command is always "nested"; that is, the command is sent from within the ct_results loop while the cursor's rows are being processed by ct_fetch.

  A nested cursor command can be sent after ct_results returns a *result_type* value of CS_CURSOR_RESULT. At least one row must be fetched before a cursor delete command is allowed, and cursor delete commands are not allowed after ct_fetch returns CS_END_DATA.
- By default, the last-fetched row is deleted. The application can redirect the deletion to another row in the cursor-result set. To redirect the deletion, specify different key values with ct_keydata before sending the cursor-delete command.
- A cursor-delete command generates results like any other command. The application must process the results before it can fetch from the cursor again.
- Cursor-delete commands can be resent, with the same restrictions as for cursor-update commands.

Cursor-close commands

- A ct_cursor(CS_CURSOR_CLOSE) command abandons the cursor result set that was generated when the cursor was opened. If all the cursor's rows have been fetched, a cursor-close command must be issued before the application can reopen the cursor.
- An application can reopen a closed cursor.
- A cursor-close command can be "nested "; that is, a cursor-close command can be sent from within the ct_results loop while the cursor's rows are being processed by ct_fetch.

- A nested cursor-close command can be sent after ct_results returns a *result_type* value of CS_CURSOR_RESULT and before ct_fetch returns CS_END_DATA.
- After ct_fetch returns CS_END_DATA, the cursor-close command can no longer be nested, and cannot be sent until ct_results has returned CS_END_RESULTS or CS_CANCELED.

A nested cursor-close command is the preferred way to abandon rows returned from a cursor-open command, since ct_cancel can put a connection's cursors into an undefined state.

- A non-nested cursor-close command must be sent when the CS_COMMAND structure is idle, that is, after ct_results has returned CS_END_RESULTS or CS_CANCELED.

Cursor-deallocate commands

- A ct_cursor(CS_CURSOR_DEALLOC) command deallocates a Client-Library cursor. If a cursor has been deallocated, it cannot be reopened.
- An application cannot deallocate an open cursor.
- To initiate a command to both close and deallocate a Client-Library cursor, call ct_cursor with *type* as CS_CURSOR_CLOSE and *option* as CS_DEALLOC.

See also

"Commands" on page 100, ct_cmd_alloc, ct_keydata, ct_param, ct_results, ct_send, ct_setparam, ct_scroll_fetch.

ct_data_info

Description

Define or retrieve a data I/O descriptor structure.

Syntax

CS_RETCODE ct_data_info(cmd, action, colnum, iodesc)

```
CS_COMMAND    *cmd;
CS_INT        action;
CS_INT        colnum;
CS_IODESC     *iodesc;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

action
One of the following symbolic values:

Value	Meaning
CS_SET	Define an I/O descriptor
CS_GET	Retrieve an I/O descriptor

colnum
The number of the text or image column whose I/O descriptor is being retrieved.

If *action* is CS_SET, pass *colnum* as CS_UNUSED.

If *action* is CS_GET, *colnum* refers to the select-list ID of the text or image column. The first column is column number 1, the second is number 2, and so forth. An application must select a text or image column before it can update the column.

colnum must represent a text or image column.

iodesc
A pointer to a CS_IODESC structure. A CS_IODESC structure contains information describing text or image data. For more information about this structure, see “CS_IODESC structure” on page 91.

Return value

ct_data_info returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

```
/*
 ** FetchResults()
 **
 ** The result set contains four columns: integer, text,
 ** float, and integer.
 */

CS_STATIC CS_RETCODE
 FetchResults(cmd, textdata)
 CS_COMMAND            *cmd;
 TEXT_DATA            *textdata;
 {
 CS_RETCOD    retcode;
```

```
 CS_DATAFMT   fmt;
 CS_INT       firstcol;
 CS_TEXT      *txtptr;
 CS_FLOAT     floatitem;
 CS_INT       count;
 CS_INT       len;

/*
 ** Before we call ct_get_data(), we can only bind
 ** columns that come before the column on which we
 ** perform the ct_get_data().
 ** To demonstrate this, bind the first column
 ** returned.
 */
 ...CODE DELETED.....

/* Retrieve and display the result */
 while(((retcode = ct_fetch(cmd, CS_UNUSED, CS_UNUSED,
     CS_UNUSED,&count)) == CS_SUCCEED) ||
     (retcode == CS_ROW_FAIL) )
 {
     /* Check for a recoverable error */
     ...CODE DELETED.....

    /* Get the text data item in the 2nd column */
     ...CODE DELETED.....

 /*
  ** Retrieve the descriptor of the text data. It
  ** is available while retrieving results of a select
  ** query. The information will be needed for later
  ** updates.
  */
     retcode = ct_data_info(cmd, CS_GET,  2,
         &textdata->iodesc);
     if (retcode != CS_SUCCEED)
     {
         ex_error("FetchResults: cs_data_info()
             failed");
         return retcode;
     }

    /* Get the float data item in the 3rd column */
     ...CODE DELETED.....

    /* Last column not retrieved */
 }

/*
 ** We're done processing rows. Check the final return
```

```
        ** value of ct_fetch().
        */
        ...CODE DELETED.....

       return retcode;
}
```

This code excerpt is from the *getsend.c* sample program.

Usage

- ct_data_info defines or retrieves a CS_IODESC, also called an **I/O descriptor structure**, for a text or image column.
- An application calls ct_data_info to retrieve an I/O descriptor after calling ct_get_data to retrieve a text or image column value that it plans to update at a later time. This I/O descriptor contains the text pointer and text timestamp that the server uses to manage updates to text or image columns.

  After retrieving an I/O descriptor, a typical application changes only the values of the *locale*, *total_txtlen*, and *log_on_update* fields before using the I/O descriptor in an update operation:

  - The *total_txtlen* field of the CS_IODESC represents the total length, in bytes, of the new text or image value.
  - The *6log_on_update* field in the CS_IODESC indicates whether or not the server should log the update.
  - The *locale* field of the CS_IODESC points to a CS_LOCALE structure containing localization information for the value.
- An application calls ct_data_info to define an I/O descriptor before calling ct_send_data to send a chunk or image data to the server. Both of these calls occur during a text or image update operation.
- A successful text or image update generates a parameter result set that contains the new text timestamp for the text or image value. If an application plans to update the text or image value a second time, it must save this new text timestamp and copy it into the CS_IODESC for the value before calling ct_data_info to define the CS_IODESC for the update operation.

- In most cases, an application must call ct_get_data for a column before calling ct_data_info. However, when ct_get_data is used with the Open Server srv_send_data routine to transfer text, image, and XML columns in chunks in Gateway Open Server applications, the application must call ct_data_info before calling ct_get_data. This allows Open Server to retrieve fixed I/O fields, such as object names, before a column is read and to send a row's data format before the whole row is read. The changeable fields in I/O descriptors, such as pointers to text data and the length of text data, are still retrievable only after the column is read.

  A call to ct_get_data does not have to retrieve any data: An application can call ct_get_data with a buffer length of 0 and then call ct_data_info to retrieve the descriptor. This technique is useful when an application needs to determine the length of a text or image value before retrieving it.

  For more information about srv_send_data, see the Open Server *Server-Library/C Reference Manual*.

- For more information about the I/O descriptor structure, see the "CS_IODESC structure" on page 91.

See also

ct_get_data, ct_send_data, "text and image data handling" on page 284

ct_debug

Description

Manage debug library operations.

Syntax

```
CS_RETCODE ct_debug(context, connection, operation,
            flag, filename, fnamelen)

CS_CONTEXT      *context;
CS_CONNECTION   *connection;
CS_INT          operation;
CS_INT          flag;
CS_CHAR         *filename;
CS_INT          fnamelen;
```

Parameters

context

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

When *operation* is CS_SET_DBG_FILE, *context* must be supplied and *connection* must be NULL.

When setting or clearing flags, see Table 3-16 to determine whether or not to supply *context*.

connection

A pointer to a CS_CONNECTION structure. *connection* must point to a valid CS_CONNECTION structure, but no actual connection to a server is necessary to enable debug operations.

When *operation* is CS_SET_PROTOCOL_FILE, *connection* must be supplied and *context* must be NULL.

When setting or clearing flags, see Table 3-16 on page 441 to determine whether or not to supply *connection*.

operation

The operation to perform. Table 3-17 on page 442 lists the symbolic values for *operation*.

flag

A bitmask representing debug subsystems. The following table lists the symbolic values that can make up *flag*:

Table 3-16: Values for ct_debug flag parameter

Value	Required	Resulting Client-Library behavior
CS_DBG_ALL	*context* and *connection*	Takes all possible debug actions.
CS_DBG_API_ LOGCALL	*context*	Prints out information each time the application calls a Client-Library routine, including the routine name and the parameter values.
CS_DBG_API_STATES	*context*	Prints information relating to Client-Library function-level state transitions.
CS_DBG_ASYNC	*context*	Prints function trace information each time an asynchronous function starts or completes.
CS_DBG_DIAG	*connection*	Prints message text whenever a Client-Library or server message is generated.
CS_DBG_ERROR	*context*	Prints trace information whenever a Client-Library error occurs. This allows a programmer to determine exactly where an error is occurring.
CS_DBG_MEM	*context*	Prints information relating to memory management.
CS_DBG_NETWORK	*context*	Prints information relating to Client-Library's network interactions.
CS_DBG_PROTOCOL	*connection*	This ct_debug parameter may be set without *devlib* libraries. The parameter captures information exchanged with a server in protocol-specific (for example, TDS) format. This information is not human-readable.
CS_DBG_PROTOCOL_ FILE	*connection*	This ct_debug parameter may be set without *devlib* libraries. If the parameter is not set on connection, mktemp is called, generating a unique file name to dump the protocol packets into. The prefix string passed to mktemp is *capture*. The resulting protocol file is decodable by Ribo.
CS_DBG_PROTOCOL_ STATES	*connection*	Prints information relating to Client-Library protocol-level state transitions.

filename

The full path and name of the file to which ct_debug should write the generated debug information.

fnamelen
The length, in bytes, of *filename*, or CS_NULLTERM if *filename* is a null-terminated string.

Return value

ct_debug returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

```
...CODE DELETED.....
 #ifdef EX_API_DEBUG
     /*
     ** Enable this function right before any call to
     ** Client-Library that is returning failure.
     */
     retcode = ct_debug(*context, NULL, CS_SET_FLAG,
         CS_DBG_API_STATES, NULL, CS_UNUSED);
     if (retcode != CS_SUCCEED)
     {
         ex_error("ex_init: ct_debug() failed");
     }
 #endif
 ...CODE DELETED.....
```

This code excerpt is from the *exutils.c* sample program.

Usage

Table 3-17: Summary of ct_debug parameters

Value of operation	Flag is	File name is	Result
CS_SET_FLAG	Supplied	NULL	Enables the subsystems specified by flag.
CS_CLEAR_FLAG	Supplied	NULL	Disables the subsystems specified by flag.
CS_SET_DBG_FILE	CS_UNUSED	Supplied	Records the name of the file to which it will write character-format debug information.
CS_SET_PROTOCOL_FILE	CS_UNUSED	Supplied	Records the name of the file to which it will write protocol-format debug information.

- ct_debug manages debug library operations, allowing an application to enable and disable specific diagnostic subsystems and send the resultant trace information to files.

- ct_debug functionality is available only from within the debug version of Client-Library. When called from within the standard Client-Library, it returns CS_FAIL.
- Some debug flags can be enabled only at the connection level, while others can be enabled only at the context level. Table 3-16 on page 441 indicates the level at which each flag can be enabled.
- If an application does not call ct_debug to specify debug files, ct_debug writes character-format debug information to *stdout* (where available) and protocol-form debug information to:
  - On Windows:
    capXXXX.tmp
    where *XXXX* is a unique code.
  - On Unix:
    captureXXXXXX
    where *XXXXXX* is a unique code.

  These files are found in the application's working directory.
- When the debug version of Client-Library is linked in with an application, the following behaviors automatically take place:
  - Memory reference checks: Client-Library verifies that all memory references, both internal and application-specific, are valid.
  - Data structure validation: each time a Client-Library function accesses a data structure, Client-Library first validates the structure.
  - Special assertion checking: Client-Library checks that all array references, including strings, are in bounds.
- Because the debug version of Client-Library performs extensive internal checking, application performance will decrease when the debug library is in use. The level of performance decrease depends on the type and number of tracing subsystems that are enabled. To minimize performance decrease, an application programmer can selectively enable tracing subsystems, limiting heavy tracing to problem areas of code.
- Use of the debug library will change the behavior of asynchronous applications that are experiencing timing problems. In this case, the use of external tracing tools (for example, a network protocol analyzer) is recommended.

See also

"Error handling" on page 123, "Enabling debugging" on page 306.

ct_describe

Description

Return a description of result data.

Syntax

```
CS_RETCODE ct_describe(cmd, item, datafmt)

CS_COMMAND   *cmd;
CS_INT        item;
CS_DATAFMT   *datafmt;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

item

An integer representing the result item of interest.

When retrieving a column description, item is the column's column number. The first column in a select-list is column number 1, the second is number 2, and so forth.

When retrieving a compute column description, item is the column number of the compute column. Compute columns are returned in the order in which they are listed in the compute clause. The first column returned is number 1.

When retrieving a return parameter description, item is the parameter number of the parameter. The first parameter returned by a stored procedure is number 1. Stored procedure return parameters are returned in the same order as the parameters were originally specified in the stored procedure's create procedure statement. This is not necessarily the same order as specified in the RPC command that invoked the stored procedure. In determining what number to pass as *item* do not count non-return parameters. For example, if the second parameter in a stored procedure is the only return parameter, pass *item* as 1.

When retrieving a stored procedure return status description, item must be 1, as there can be only a single status in a return status result set.

When retrieving format information, item takes a column or compute column number.

Note An application cannot call ct_describe after ct_results indicates a result set of type CS_MSG_RESULT. This is because a result type of CS_MSG_RESULT has no data items associated with it. Parameters associated with a message are returned as a CS_PARAM_RESULT result set. Likewise, an application cannot call ct_describe after ct_results sets its **result_type* parameter to CS_CMD_DONE, CS_CMD_SUCCEED, or CS_CMD_FAIL to indicate command status information.

---

datafmt

A pointer to a CS_DATAFMT structure. ct_describe fills **datafmt* with a description of the result data item referenced by *item*.

ct_describe fills in the following fields in the CS_DATAFMT:

Table 3-18: Fields in the CS_DATAFMT structure as set by ct_describe

Field name	Types of result items	Value of field after ct_describe call
name	Regular columns, column formats, and return parameters.	The null-terminated name of the data item, if any. A NULL name is indicated by a *namelen* of 0.
namelen	Regular columns, column formats, and return parameters.	The actual length of the name, not including the null terminator. 0 to indicate a NULL *name*.
datatype	Regular columns, column formats, return parameters, return status, compute columns, and compute column formats.	A type constant (CS_xxx_TYPE) representing the datatype of the item. All type constants listed in “Datatypes support” on page 294 are valid, except CS_VARCHAR_TYPE and CS_VARBINARY_TYPE. A return status has a datatype of CS_INT_TYPE. A compute column’s datatype depends on the type of the underlying column and the aggregate operator that created the column.
format	Not used.	
maxlength	Regular columns, column formats, and return parameters.	The maximum possible length (in bytes) of the data for the column or parameter.
scale	Regular columns, column formats, return parameters, compute columns, or compute column formats of type numeric or decimal.	The maximum number of digits to the right of the decimal point in the result data item.
precision	Regular columns, column formats, return parameters, compute columns, or compute column formats of type numeric or decimal.	The maximum number of decimal digits that can be represented in the result data item.

Field name	Types of result items	Value of field after ct_describe call
status	Regular columns and column formats.	A bitmask of the following values: • CS_CANBENULL to indicate that the column can contain NULL values. • CS_HIDDEN to indicate that the column is a “hidden” column that has been exposed. For information about hidden columns, see “Hidden keys” on page 214. • 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. • CS_TIMESTAMP to indicate the column is a timestamp column. • CS_UPDATABLE to indicate that the column is an updatable cursor column. • CS_UPDATECOL to indicate that the column is in the update clause of the cursor declare commandC. • CS_RETURN to indicate that the column is a return parameter of an RPC command.
count	Regular columns, column formats, return parameters, return status, compute columns, and compute column formats.	*count* represents the number of rows copied to program variables per ct_fetch call. ct_describe sets *count* to 1 to provide a default value in case an application uses ct_describe’s return CS_DATAFMT as ct_bind’s input CS_DATAFMT.
usertype	Regular columns, column formats, and return parameters.	The Adaptive Server user-defined datatype of the column or parameter, if any. *usertype* is set in addition to (not instead of) *datatype*.

Field name	Types of result items	Value of field after ct_describe call
locale	Regular columns, column formats, return parameters, return status, compute columns, and compute column formats.	A pointer to a CS_LOCALE structure that contains locale information for the data. This pointer can be NULL.

- When ct_describe is called, it fills **datafmt* with information about the column or parameter being described. The *status* field of **datafmt* is a bitmask of the following values:
  - CS_CANBENULL to indicate that the column can contain NULL values.
  - CS_HIDDEN to indicate that the column is a “hidden” column that has been exposed.
  - CS_IDENTITY to indicate that the column is an identity column.
  - CS_KEY to indicate that the column is part of the key for a table.
  - CS_VERSION_KEY to indicate that the column is part of the version key for the row.
  - CS_TIMESTAMP to indicate that the column is a timestamp column.
  - CS_UPDATABLE to indicate that the column is an updatable cursor column.
  - CS_UPDATECOL to indicate that the column is in the update clause of a cursor declare command.
  - CS_RETURN to indicate that the column is a return parameter of an RPC command.

Return value

ct_describe returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

ct_describe returns CS_FAIL if *item* does not represent a valid result data item.

Examples

```
/* ex_fetch_data()*/

CS_RETCODE CS_PUBLIC
 ex_fetch_data(cmd)
 CS_COMMAND   *cmd;
 {
     CS_RETCODE      retcode;
     CS_INT          num_cols;
     CS_INT          i;
     CS_INT          j;
     CS_INT          row_count = 0;
     CS_DATAFMT      *datafmt;
     EX_COLUMN_DATA  *coldata;

    /*
     ** Determine the number of columns in this result
     ** set
     */
     ...CODE DELETED...

    for (i = 0; i < num_cols; i++)
     {
         /*
         ** Get the column description.  ct_describe()
         ** fills the datafmt parameter with a
         ** description of the column.
         */
         retcode = ct_describe(cmd, (i + 1),
             &datafmt[i]);
         if (retcode != CS_SUCCEED)
         {
             ex_error("ex_fetch_data: ct_describe()
                 failed");
             break;
         }

        /* Now bind columns */
         ...CODE DELETED.....
     }

    /* Now fetch rows */
     ...CODE DELETED.....

    return retcode;
 }
```

This code excerpt is from the *exutils.c* sample program.

Usage

- An application can use ct_describe to retrieve a description of a regular result column, a return parameter, a stored procedure return status number, or a compute column.

  An application can also use ct_describe to retrieve format information. Client-Library indicates that format information is available by setting ct_results' **result_type* to CS_ROWFMT_RESULT or CS_COMPUTEFMT_RESULT.

- An application cannot call ct_describe after ct_results sets its **result_type* parameter to CS_MSG_RESULT, CS_CMD_SUCCEED, CS_CMD_DONE, or CS_CMD_FAIL. This is because, in these cases, there are no result items to describe.
- An application can call ct_res_info to find out how many result items are present in the current result set.
- An application generally needs to call ct_describe to describe a result data item before it binds the result item to a program variable using ct_bind.
- See "CS_DATAFMT structure" on page 85 for a description.
- See "Results" on page 241 for a description of result types.

See also

ct_bind, ct_fetch, ct_res_info, ct_results, "Results" on page 241

ct_diag

Description

Manage inline error handling.

Syntax

```
CS_RETCODE ct_diag(connection, operation, type, index,
        buffer)

CS_CONNECTION   *connection;
CS_INT          operation;
CS_INT          type;
CS_INT          index;
CS_VOID         *buffer;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

operation

The operation to perform. Table 3-20 lists the symbolic values for *operation*.

type

Depending on the value of *operation*, *type* indicates either the type of structure to receive message information, the type of message on which to operate, or both. Table 3-19 lists the symbolic values for *type*:

Table 3-19: Values for ct_diag type parameter

Value of type	Meaning
SQLCA_TYPE	A SQLCA structure.
SQLCODE_TYPE	A SQLCODE structure, which is a long integer.
SQLSTATE_TYPE	A SQLSTATE structure, which is a six-element character array.
CS_CLIENTMSG_TYPE	A CS_CLIENTMSG structure. Also used to indicate Client-Library messages.
CS_SERVERMSG_TYPE	A CS_SERVERMSG structure. Also used to indicate server messages.
CS_ALLMSG_TYPE	Client-Library and server 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.

If type is CS_CLIENTMSG_TYPE, then index refers to Client-Library messages only. If type is CS_SERVERMSG_TYPE, then index refers to server messages only. If type is CS_ALLMSG_TYPE, then index refers to Client-Library and server messages combined.

buffer

A pointer to data space.

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

Return value

ct_diag returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed. ct_diag returns CS_FAIL if the original error has made the connection unusable.
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 are queued.

Return value	Meaning
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see “Asynchronous programming” on page 12.

Common reasons for a ct_diag failure include:

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

Usage

Table 3-20: Summary of ct_diag parameters

Value of operation	Resulting action	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 to limit Client-Library messages only. CS_SERVERMSG_TYPE to limit server messages only. CS_ALLMSG_TYPE to limit the total number of Client-Library and server messages combined.	CS_UNUSED	A pointer to an integer value.
CS_CLEAR	Clears message information for this connection. If *buffer* is not NULL and type is not CS_ALLMSG_TYPE, ct_diag also clears the **buffer* structure by initializing it with blanks and/or NULLs, as appropriate.	Any valid *type* value. If *type* is CS_CLIENTMSG_TYPE, ct_diag clears Client-Library messages only. If *type* is CS_SERVERMSG_TYPE, ct_diag clears server messages only. If *type* has any other valid value, ct_diag clears both Client-Library and server messages.	CS_UNUSED	A pointer to a structure whose type is defined by *type*, or NULL.

Value of operation	Resulting action	Type is	Index is	Buffer is
CS_GET	Retrieves a specific message.	Any valid *type* value except CS_ALLMSG_TYPE. If *type* is CS_CLIENTMSG_TYPE, a Client-Library message is retrieved into a CS_CLIENTMSG structure. If *type* is CS_SERVERMSG_TYPE, a server message is retrieved into a CS_SERVERMSG structure. If *type* has any other valid value, then either a Client-Library or server message is retrieved.	The one-based index of the message to retrieve.	A pointer to a structure whose type is defined by *type*.
CS_STATUS	Returns the current number of stored messages.	CS_CLIENTMSG_TYPE to retrieve the number of Client-Library messages. CS_SERVERMSG_TYPE to retrieve the number of server messages. CS_ALLMSG_TYPE to retrieve the total number of Client-Library and server messages combined.	CS_UNUSED	A pointer to an integer variable.
CS_EED_CMD	Sets **buffer* to the address of the CS_COMMAND structure containing extended error data.	CS_SERVERMSG_TYPE	The one-based index of the message for which extended error data is available.	A pointer to a pointer variable.

- A Client-Library application can handle Client-Library and server messages in two ways:
  - The application can call ct_callback to install client message and server message callbacks to handle Client-Library and server messages.

- The application can handle Client-Library and server messages inline, using ct_diag.

An application can switch back and forth between the two methods. For information about how to do this, see “Error handling” on page 123.

- ct_diag manages inline message handling for a specific connection. If an application has more than one connection, it must make separate ct_diag calls for each connection.
- An application cannot use ct_diag at the context level. That is, an application cannot use ct_diag to retrieve messages generated by routines that take a CS_CONTEXT (and no CS_CONNECTION) as a parameter. These messages are unavailable to an application that is using inline error handling.
- An application can perform operations on either Client-Library messages, server messages, or both.

  For example, an application can clear Client-Library messages without affecting server messages by using:

  ```
  ct_diag(connection, CS_CLEAR, CS_CLIENTMSG,
          CS_UNUSED, NULL);
  ```

- ct_diag allows an application to retrieve message information into standard Client-Library structures (CS_CLIENTMSG and CS_SERVERMSG) or a SQLCA, SQLCODE, or SQLSTATE. When retrieving messages, ct_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 must set the Client-Library property CS_EXTRA_INF to CS_TRUE. This is because the SQL structures require information that is not ordinarily returned by Client-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 Client-Library messages.

- If ct_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, ct_diag starts saving messages again.

Initializing inline error handling

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

Clearing messages

- To clear message information for a connection, an application calls ct_diag with *operation* as CS_CLEAR.
  - To clear Client-Library messages only, an application passes *type* as CS_CLIENTMSG_TYPE.
  - To clear server messages only, an application passes *type* as CS_SERVERMSG.
  - To clear both Client-Library and server messages, pass *type* as SQLCA_TYPE, SQLCODE_TYPE, SQLSTATE_TYPE, or CS_ALLMSG_TYPE.
- If *type* is not CS_ALLMSG_TYPE:
  - ct_diag assumes that *buffer* points to a structure whose type corresponds the value of *type*.
  - ct_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 ct_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 ct_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.
- If *type* is CS_CLIENTMSG_TYPE, then index refers only to Client-Library messages. If *type* is CS_SERVERMSG_TYPE, *index* refers only to server messages. If *type* has any other value, *index* refers to the collective “queue” of both types of messages combined.
- ct_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, ct_diag returns CS_NOMSG to indicate that no message is available.
- For information about these structure, see:
  - “SQLCA structure” on page 97
  - “SQLCODE structure” on page 99
  - “SQLSTATE structure” on page 100
  - “CS_CLIENTMSG structure” on page 78
  - “CS_SERVERMSG structure” on page 95

Limiting messages

- Applications running on platforms with limited memory may want to limit the number of messages that Client-Library saves.
- An application can limit the number of saved Client-Library messages, the number of saved server messages, and the total number of saved messages.
- To limit the number of saved messages, an application calls ct_diag with *operation* as CS_MSGLIMIT and *type* as CS_CLIENTMSG_TYPE, CS_SERVERMSG_TYPE, or CS_ALLMSG_TYPE:
  - If *type* is CS_CLIENTMSG_TYPE, then the number of Client-Library messages is limited.
  - If *type* is CS_SERVERMSG_TYPE, then the number of server messages is limited.
  - If *type* is CS_ALLMSG_TYPE, then the total number of Client-Library and server messages combined is limited.
  - When a specific message limit is reached, Client-Library discards any new messages of that type. When a combined message limit is reached, Client-Library discards any new messages. If Client-Library discards messages, it saves a message to this effect.
  - An application cannot set a message limit that is less than the number of messages currently saved.
  - Client-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 ct_diag with *operation* as CS_STATUS and *type* as the type of message of interest.

Getting the CS_COMMAND for extended error data

- To retrieve a pointer to the CS_COMMAND structure containing extended error data (if any), call ct_diag with *operation* as CS_EED_CMD and *type* as CS_SERVERMSG_TYPE. ct_diag sets **buffer* to the address of the CS_COMMAND structure containing the extended error data.
- When an application retrieves a server message into a CS_SERVERMSG structure, Client-Library indicates that extended error data is available for the message by setting the CS_HASEED bit in the *status* field in the CS_SERVERMSG structure.
- It is an error to call ct_diag with *operation* as CS_EED_CMD when extended error data is not available.
- For more information about extended error data, see “Extended error data” on page 129.

Sequenced messages and *ct_diag*

- If an application is using sequenced error messages, ct_diag acts on message chunks instead of messages. This has the following effects:
  - A ct_diag(CS_GET, *index*) call returns the message chunk with number *index*.
  - A ct_diag(CS_MSGLIMIT) call limits the number of chunks, not the number of messages, that Client-Library stores.
  - A ct_diag(CS_STATUS) call returns the number of currently stored chunks, not the number of currently stored messages.
- For more information about sequenced messages, see “Sequencing long messages” on page 127.

See also

“Error handling” on page 123 “CS_CLIENTMSG structure” on page 78, “CS_SERVERMSG structure” on page 95, “SQLCA structure” on page 97, “SQLCODE structure” on page 99, “SQLSTATE structure” on page 100, ct_callback, ct_options

ct_ds_dropobj

Description

Release the memory associated with a directory object.

Syntax

```
CS_RETCODE ct_ds_dropobj(connection, ds_object)

CS_CONNECTION   *connection;
CS_DS_OBJECT    *ds_object;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. ct_ds_lookup returns search results to the application's directory callback that has been installed in the CS_CONNECTION structure.

ds_object

A pointer to the directory object being discarded.

Return value

ct_ds_dropobj returns the following values:

Return value	Meaning
CS_SUCCEED	The routine succeeded.
CS_FAIL	The routine failed.

Usage

- ct_ds_dropobj discards a CS_DS_OBJECT hidden structure and frees the memory associated with it. The directory entry associated with the object is not affected in any way by ct_ds_dropobj.
- To keep the information associated with a directory object, copy it into application memory before dropping the object.
- If an application does not explicitly drop directory objects with ct_ds_dropobj, Client-Library drops them automatically when the application calls ct_con_drop to drop the parent connection.

See also

ct_ds_lookup, ct_ds_objinfo

ct_ds_lookup

Description

Initiate or cancel a directory lookup operation.

Syntax

```
CS_RETCODE ct_ds_lookup(connection,
        action, reqid, lookup_info,
        userdata)

CS_CONNECTION       connection;
CS_INT              action;
```

```
CS_INT                  *reqid;
CS_DS_LOOKUP_INFO       *lookup_info;
CS_VOID                 *userdata;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular connection.

action

One of the following symbolic values:

Action	Function performed
CS_CLEAR	Cancel the directory lookup operation specified by *reqid*. Supported for asynchronous connections only.
CS_SET	Initiate a directory lookup operation.

reqid

A pointer to a CS_INT variable.

When *action* is CS_SET, Client- Library returns the request identifier in **reqid*.

When *action* is CS_CLEAR, **reqid* specifies the request ID of the operation to cancel.

lookup_info

The address of a CS_DS_LOOKUP_INFO structure.

A CS_DS_LOOKUP_INFO structure is defined as follows:

```
typedef struct _cs_ds_lookup_info
 {
        CS_OID          *objclass;
        CS_CHAR         *path;
        CS_INT          pathlen;
        CS_DS_OBJECT    *attrfilter;
        CS_DS_OBJECT    *attrselect;
 } CS_DS_LOOKUP_INFO;
```

When *action* is CS_SET, set the fields of **lookup_info* as follows:

Table 3-21: Contents of *lookup_info for a ct_ds_lookup(CS_SET) call

Field	Set to
objclass	The address of a CS_OID structure that specifies the class of directory objects to search for. *objclass->oid_buffer* contains the OID string for the object class and *objclass->oid_length* specifies the length of the OID string (not counting any null terminator). ct_ds_lookup finds only those directory entries whose class matches the contents of *lookup_info->objclass*.
path	Reserved. Set to NULL to ensure compatibility with future versions of Client-Library.
pathlen	Reserved. Set to 0 to ensure compatibility with future versions of Client-Library.
attrfilter	Reserved. Set to NULL to ensure compatibility with future versions of Client-Library.
attrselect	Reserved. Set to NULL to ensure compatibility with future versions of Client-Library.

Note In asynchronous mode, the contents of **lookup_info* and the pointers contained in it must remain valid until the connection's completion callback or ct_poll indicates that the request has completed or was canceled.

When *action* is CS_CLEAR, *lookup_info* must be passed as NULL.

userdata

The address of user-allocated data to pass into the directory callback.

When action is CS_SET, *userdata* is optional and can be passed as NULL. If ct_ds_lookup finds matching directory entries, Client-Library invokes the connection's directory callback. The directory callback receives the address specified as *userdata*. *userdata* provides a means for the callback to communicate the search results back to the mainline code where ct_ds_lookup was called.

When action is CS_CLEAR, *userdata* must be passed as (CS_VOID *) NULL.

Return value

ct_ds_lookup returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully
CS_FAIL	The routine failed

Return value	Meaning
CS_PENDING	Asynchronous network I/O is in effect. See “Asynchronous programming” on page 12. **Note** On platforms where Client-Library does not use thread-driven I/O, applications must always poll for ct_ds_lookup completions even when the connection’s CS_NETIO setting is CS_ASYNC_IO.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.
CS_CANCELED	The lookup request was canceled by the application. Lookup requests can be canceled only on asynchronous connections.

Examples

For an explanation of the steps in this example, see Chapter 9, “Using Directory Services,” in the Open Client *Client-Library/C Programmer's Guide*.

Usage

- ct_ds_lookup initiates or cancels a directory lookup request.
- ct_ds_lookup, ct_ds_objinfo, and the connection’s directory callback routine provide a mechanism for Client-Library applications to view directory entries. The typical process is outlined below.

  a The application installs a directory callback to handle the search results.

  b (Network-based directories only.) The application sets the CS_DS_DITBASE connection property to specify the subtree to be searched.

  c (Network-based directories only.) The application sets any other necessary directory service properties to constrain the search.

  d The application calls ct_ds_lookup to initiate the directory search.

  e Client-Library calls the application’s directory callback once for each directory entry found. Each invocation of the callback receives a CS_DS_OBJECT pointer, that provides an abstract view of the directory entry’s contents.

  f The application examines each object by calling ct_ds_objinfo as many times as necessary. This can be done in the callback or in the mainline code.

g When the application is finished with the directory objects returned by the search, it frees the memory associated with them by calling ct_ds_dropobj.

Directory callbacks

- The results of directory searches are returned to the connection's directory callback. Client-Library invokes the directory callback once for each entry found in the search, and each invocation receives a pointer to a CS_DS_OBJECT that describes the entry.
- Before beginning a search, the application must install a directory callback with ct_callback(CS_DS_LOOKUP_CB) to receive the search results. Otherwise, the results are discarded.
- See "Defining a directory callback" on page 39 for a description of directory callbacks.

Initiating directory lookups

- ct_ds_lookup(CS_SET) passes a request to the directory driver specified by the current setting of the CS_DS_PROVIDER connection property. See "Directory service provider" on page 120.
- If the search uses a directory service rather than the *interfaces* file, then the search finds all directory entries which match these criteria:
  - The object class of the entry matches the OID specified by the *objclass* field in ct_ds_lookup's *lookup_info* parameter.
  - The entry is under the directory node that is specified as the CS_DS_DITBASE connection property. See "Base for directory searches" on page 117.
  - The entry is within the depth limit defined by the CS_DS_SEARCH connection property. See "Directory service search depth" on page 122. By default, ct_ds_lookup returns only those entries that are located directly beneath the DIT-base node.
- If the *interfaces* file is searched, the search must be for server (CS_OID_SERVERCLASS) objects. A search returns a description of all servers defined in the *interfaces* file.
- Some directory service providers may have access restrictions for directory entries. In this case, the application must provide a value for the CS_DS_PRINCIPAL connection property. See "Directory service principal name" on page 119.

Synchronous vs. asynchronous directory lookups

- ct_ds_lookup(CS_SET) passes a lookup request to the underlying directory service. The request returns the matching objects to Client-Library. The processing cycle differs for asynchronous and synchronous connections.
- If the connection is synchronous, ct_ds_lookup blocks until the lookup request has completed and the application has finished viewing returned objects in the directory callback. Synchronous processing happens as follows:

  a The application's main-line code calls ct_ds_lookup(CS_SET) to initiate the lookup operation.

  b When the search is complete, Client-Library begins invoking the directory callback and passing the returned objects to the application.

  If the search failed for any reason, then Client-Library passes CS_FAIL as the value of the *status* callback argument.

  The callback is invoked repeatedly, once for each object found or until the directory callback returns CS_SUCCEED.

  c ct_ds_lookup returns control to the mainline code.

  Note To provide fully asynchronous support, ct_ds_lookup requires a version of Client-Library that uses thread-driven I/O. With other versions, ct_ds_lookup gives deferred-asynchronous behavior when CS_NETIO is set to CS_ASYNC_IO or CS_DEFER_IO.

- If the connection is fully asynchronous or deferred asynchronous, then ct_ds_lookup returns immediately. The detailed process is as follows:

  a The application's mainline code calls ct_ds_lookup(CS_SET) to initiate the lookup operation.

  b Client-Library passes the request to the directory service driver.

  If the directory driver accepts the request, ct_ds_lookup returns CS_PENDING. On platforms where Client-Library uses threads, Client-Library spawns an internal worker thread to handle the request at this point.

  If the request cannot be queued, ct_ds_lookup returns CS_FAIL.

c The connection's directory callback is invoked. On a fully asynchronous connection, Client-Library invokes the callback automatically. On a deferred-asynchronous connection, Client-Library invokes the callback when the application calls ct_poll.

If the search returned objects, then the callback is called repeatedly to pass objects to the application until the application has seen all the objects or the callback returns CS_SUCCEED.

If the lookup found no objects, then the callback is called once with the *numentries* callback argument equal to 0.

If the search failed or was canceled, then the callback is called once with CS_FAIL or CS_CANCELED as the *status* callback argument.

d The connection’s completion callback is invoked. On a fully asynchronous connection, Client-Library invokes the completion callback automatically. On a deferred-asynchronous connection, the application must poll for request completion with ct_poll, and ct_poll invokes the callback. The completion callback receives the final return status for the lookup operation (CS_SUCCEED, CS_FAIL, or CS_CANCELLED).

Note On fully asynchronous connections, the directory and completion callbacks are invoked by an internal Client-Library thread. Make sure that shared data is protected from simultaneous access by mainline code, other application threads, and the callback code executing in the Client-Library thread. The contents of **userdata* must also be protected from simultaneous access.

Canceling a directory lookup (asynchronous connections only)

- If the connection’s network I/O mode (CS_NETIO property) is fully asynchronous or deferred asynchronous, then a lookup operation can be canceled by calling ct_ds_lookup(CS_CLEAR, &*reqid*) before the search completes.
  - ct_ds_lookup(CS_CLEAR) returns immediately with a status of CS_SUCCEED or CS_FAIL. However, the connection remains busy until the directory provider acknowledges the request. At this point, Client-Library invokes the directory callback and the completion callback, in that order, with a status of CS_CANCELED.

- ct_ds_lookup(CS_CLEAR) cannot be called after the connection's completion callback or ct_poll has indicated that the search has completed. At this point, the request has been fulfilled, and ct_ds_lookup(CS_CLEAR) will fail.

- Applications can also truncate the search results simply by returning CS_SUCCEED rather than CS_CONTINUE from the directory callback.
- Lookup requests made on synchronous connections cannot be canceled. However, a time limit for request completion can be set if the underlying directory service provider supports it. See "Directory search time limit" on page 123.

See also

ct_ds_objinfo, ct_ds_dropobj, "Directory services" on page 104, "Server directory object" on page 276

ct_ds_objinfo

Description

Retrieve information associated with a directory object.

Syntax

```
CS_RETCODE ct_ds_objinfo(ds_object, action, infotype, number, buffer,
                         buflen, outlen)

CS_DS_OBJECT      *ds_object;
CS_INT            action;
CS_INT            infotype;
CS_INT            number;
CS_VOID           *buffer;
CS_INT            buflen;
CS_INT            *outlen;
```

Parameters

ds_object

A pointer to a CS_DS_OBJECT structure. An application receives a directory object pointer as an input parameter to its directory callback.

action

Must be CS_GET.

infotype

The type of information to retrieve into **buffer*. For a description of the available types, see Table 3-22 on page 466.

number
When *infotype* is CS_DS_ATTRIBUTE or CS_DS_ATTRVALS, *number* specifies the number of the attribute to retrieve. Attribute numbers start at 1.

For other values of *infotype*, pass *number* as CS_UNUSED.

buffer
The address of the buffer that holds the requested information. Table 3-22 on page 466 lists the **buffer* datatypes for values of *infotype*.

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

outlen
If this argument is supplied, **outlen* is set to the length of the value returned in **buffer*. This argument is optional and can be passed as NULL.

Return value

ct_ds_objinfo returns the following values:

Return value:	Meaning
CS_SUCCEED	The routine completed successfully
CS_FAIL	The routine failed

Examples

For an explanation of the steps in this example, see Chapter 9, “Using Directory Services,” in the Open Client *Client-Library/C Programmer's Guide*.

Usage

The following table summarizes ct_ds_objinfo call syntax when action is CS_GET:

Table 3-22: Summary of ct_ds_objinfo parameters

infotype value	number value	*buffer datatype	Value written to *buffer
CS_DS_CLASSOID	CS_UNUSED	CS_OID structure	The OID of the directory object class.
CS_DS_DIST_NAME	CS_UNUSED	CS_CHAR array	Fully qualified (distinguished) directory name of the object, to 512 bytes. The output name is null-terminated. If *outlen* is not NULL, Client-Library puts the number of bytes written to **buffer* (not including the null-terminator) in **outlen*.
CS_DS_NUMATTR	CS_UNUSED	CS_INT variable	Number of attributes associated with the object.

infotype value	number value	*buffer datatype	Value written to *buffer
CS_DS_ATTRIBUTE	A positive integer	CS_ATTRIBUTE structure.	A CS_ATTRIBUTE structure that contains metadata for the attribute specified by the value of *number*. See "Retrieving object attributes and attribute values" on page 469 for a description of the CS_ATTRVALUE and CS_ATTRIBUTE data structures.
CS_DS_ATTRVALS	A positive integer	An array of CS_ATTRVALUE unions. The array must be long enough for the number of values indicated by the CS_ATTRIBUTE structure.	The values of the attribute specified by the value of *number*. See "Retrieving object attributes and attribute values" on page 469 for a description of the CS_ATTRVALUE and CS_ATTRIBUTE data structures.

- ct_ds_lookup, ct_ds_objinfo, and the connection's directory callback routine provide a mechanism for Client-Library applications to view directory entries. The typical process is as follows:

  a The application installs a directory callback to handle the search results.

  b (Network-based directories only.) The application sets the CS_DS_DITBASE connection property to specify the subtree to be searched.

  c (Network-based directories only.) The application sets any other necessary connection properties to constrain the search.

  d The application calls ct_ds_lookup to initiate the directory search.

  e Client-Library calls the application's directory callback once for each found directory entry. Each invocation of the callback receives a CS_DS_OBJECT pointer, which provides an abstract view of the directory entry's contents.

  f The application examines each object by calling ct_ds_objinfo as many times as necessary. This can be done in the callback, in mainline code, or both.

  g When the application is finished with the directory objects returned by the search, it frees the memory associated with them by calling ct_ds_dropobj.

Structure of directory objects

- The physical structure of a directory varies between directory service providers. Client-Library maps physical directory entries onto the contents of the CS_DS_OBJECT hidden structure. This minimizes an application's dependencies on any particular physical directory structure.
- An application uses ct_ds_objinfo to inspect the contents of the CS_DS_OBJECT hidden structure.
- A typical application calls ct_ds_objinfo several time to inspect the contents of the object. The steps below show a typical call sequence:

  a The application retrieves the OID that gives the object class of the directory entry by calling ct_ds_objinfo with *infotype* as CS_DS_CLASSOID and *buffer* as the address of a CS_OID structure. This step is optional and can be skipped if the application already knows the object class.

  b The application retrieves the fully qualified name of the entry by calling ct_ds_objinfo with *infotype* as CS_DS_DISTNAME and *buffer* as the address of a character string.

  c The application retrieves the number of attributes present in the object by calling ct_ds_objinfo with *infotype* as CS_DS_NUMATTR and *buffer* as the address of a CS_INT variable.

  d The application retrieves the metadata for each attribute present in the object by calling ct_ds_objinfo with *infotype* as CS_DS_NUMATTR and *buffer* as the address of a CS_ATTRIBUTE structure.

  e The application determines if it wants the attribute's values by checking the OID specified by the *attribute.attr_type* field of the CS_ATTRIBUTE structure. If the application wants the values, it allocates an array of CS_ATTRVALUE unions of size *attribute.attr_numvals*. It then retrieves the values by calling ct_ds_objinfo with *infotype* as CS_DS_ATTRVALS and *buffer* as the address of the array.

  f The application repeats steps d and e for each attribute.

Retrieving the object class

- To identify the directory object class being returned, call ct_ds_objinfo with *infotype* as CS_DS_CLASSOID and *buffer* as the address of a CS_OID structure.
  - ct_ds_objinfo sets the fields of the CS_OID to specify the OID of the directory entries object class.

In the returned CS_OID structure, the *oid–>oid_buffer* field contains the OID string for the object class. The *oid–>oid_length* contains the length of the string, not counting any null-terminator.

- The *oid_buffer* field can be compared to the OID string constant for the expected object class.

Retrieving the fully qualified entry name

- To retrieve the fully qualified directory name of the object, call ct_ds_objinfo with *infotype* as CS_DS_DIST_NAME and *buffer* as the address of a CS_CHAR string.
- The name string is null-terminated.
- For server (CS_OID_OBJSERVER) class objects, the application can pass the object's fully qualified name to ct_connect to open a connection to the server represented by the object.

Retrieving object attributes and attribute values

- The attributes of a directory object are available as a numbered set. However, the position of individual attributes within the set may vary depending on the directory service provider, and some directory providers do not guarantee that attribute orders are invariant. Also, Sybase may add new attributes to a directory object class between versions.

  For the above reasons, applications should be coded to work independently of the number and order of object attributes.
- ct_ds_objinfo uses a CS_ATTRIBUTE structure to define the metadata for attribute values, and returns the values themselves in an array of CS_ATTRVALUE unions.

CS_ATTRIBUTE structure

The CS_ATTRIBUTE structure is used with ct_ds_objinfo to describe the attributes of a directory object.

```
typedef struct
 {
     CS_OID  attr_type;
     CS_INT  attr_syntax;
     CS_INT  attr_numvals;
 } CS_ATTRIBUTE;
```

where:

attr_type is a CS_OID structure that uniquely describes the type of the attribute. This field tells the application which of an object's attributes it has received.

The definition of the directory object class determines the attribute types that an object can contain.
attr_syntax is a syntax specifier that tells how the attribute value is expressed. Attribute values are passed within a CS_ATTRVALUE union, and the syntax specifier tells which member of the union to use.
attr_numvals tells how many values the attribute contains. This information can be used to size an array of CS_ATTRVALUE unions to hold the attribute values.

CS_ATTRVALUE union

Attribute values are returned to the application in a CS_ATTRVALUE union. This union contains a members for each possible data type needed to represent attribute values. The declaration looks like this:

```
typedef union _cs_attrvalue
 {
        CS_STRING       value_string;
        CS_BOOL         value_boolean;
        CS_INT          value_enumeration;
        CS_INT          value_integer;
        CS_TRANADDR     value_tranaddr;
        CS_OID          value_oid;
 } CS_ATTRVALUE;
```

Attribute values are retrieved by ct_ds_objinfo into an array of CS_ATTRVALUE unions. The array size should match the *attr_numvals* field of the CS_ATTRIBUTE structure. The value should be taken as the union member designated by the *attr_syntax* field of the CS_ATTRIBUTE structure. Table 3-23 shows the correspondence between attribute syntax specifiers and the members of CS_ATTRVALUE.

Table 3-23: Syntax specifiers for the CS_ATTRVALUE union

Attribute syntax specifier	Union member
CS_ATTR_SYNTAX_STRING	*value_string* String values are represented by a CS_STRING structure, which is described under String values below.
CS_ATTR_SYNTAX_BOOLEAN	*value_boolean* Boolean values are represented as CS_BOOL.
CS_ATTR_SYNTAX_ENUMERATIO N	*value_enumeration* Enumerated values are represented as CS_INT.
CS_ATTR_SYNTAX_INTEGER	*value_integer* Integer values are represented as CS_INT.
CS_ATTR_SYNTAX_TRANADDR	*value_tranaddr* Transport addresses are represented as a CS_TRANADDR structure, which is described under Transport address values below.
CS_ATTR_SYNTAX_OID	*value_oid* OID values are represented as CS_OID structure, which is explained on page 93.

String values

The CS_STRING structure is defined as follows:

```
typedef struct _cs_string
 {
        CS_INT  str_length;
        CS_CHAR str_buffer[CS_MAX_DS_STRING];
 } CS_STRING;
```

The contents of *str_buffer* are null-terminated. *str_length* does not count the null-terminator in the length.

Transport address values

Transport addresses are encoded in a Sybase-specific format within the CS_TRANADDR structure shown below.

```
typedef struct _cs_tranaddr
 {
        CS_INT      addr_accesstype;
        CS_STRING   addr_trantype;
        CS_STRING   addr_tranaddress;
 } CS_TRANADDR;
```

See also

ct_ds_lookup, ct_ds_dropobj, “Directory services” on page 104, “Server directory object” on page 276

ct_dynamic

Description

Initiate a dynamic SQL command.

Syntax

```
CS_RETCODE ct_dynamic(cmd, type, id, idlen, buffer, buflen)

CS_COMMAND      *cmd;
CS_INT           type;
CS_CHAR         *id;
CS_INT           idlen;
CS_CHAR         *buffer;
CS_INT          buflen;
```

Parameters

cmd
A pointer to the CS_COMMAND structure managing a client/server operation.

type
The type of dynamic SQL command to initiate.Table 3-24 lists the symbolic values for *type*.

id
A pointer to the statement identifier. This identifier is defined by the application and must conform to server standards.

idlen
The length, in bytes, of **id*. If **id* is null-terminated, pass *idlen* as CS_NULLTERM. If *id* is NULL, pass *idlen* as CS_UNUSED.

buffer
A pointer to data space.

buflen
The length, in bytes, of **buffer*. If **buffer* is null-terminated, pass *buflen* as CS_NULLTERM. If *buffer* is NULL, pass *buflen* as CS_UNUSED.

Return value

ct_dynamic returns the following values:

Return value:	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Return value:	Meaning
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Usage

Table 3-24 summarizes ct_dynamic usage.

Table 3-24: Summary of ct_dynamic parameters

Value of type	Result	*id is	*buffer is
CS_CURSOR_DECLARE	Declares a cursor on a previously prepared SQL statement.	The prepared statement identifier.	The cursor name.
CS_DEALLOC	Deallocates a prepared SQL statement.	The prepared statement identifier.	NULL
CS_DESCRIBE_INPUT	Retrieves, from the server, a description of the input parameters required to execute a prepared statement. ct_results returns a CS_DESCRIBE_RESULT *result_type* value when the server has sent the description. An application can access this information by calling ct_res_info and ct_describe, ct_dynsqlda, or ct_dyndesc.	The prepared statement identifier.	NULL
CS_DESCRIBE_OUTPUT	Retrieves, from the server, a description of the row format of the result set that would be returned if the prepared statement were executed. ct_results returns a CS_DESCRIBE_RESULT *result_type* value when the server has sent the description. An application can access this information by calling ct_res_info and ct_describe, ct_dynsqlda, or ct_dyndesc.	The prepared statement identifier.	NULL
CS_EXECUTE	Executes a prepared SQL statement that requires zero or more parameters.	The prepared statement identifier.	NULL
CS_EXEC_IMMEDIATE	Executes a literal SQL statement.	NULL	The SQL statement to execute.
CS_PREPARE	Prepares a SQL statement.	The prepared statement identifier.	The SQL statement to prepare.

- ct_dynamic initiates dynamic Adaptive Server commands.
- For an overview of dynamic SQL commands, see Chapter 8, “Using Dynamic SQL Commands,” in the Open Client *Client-Library/C Programmer's Guide*.

- Initiating a command is the first step in sending it to a server. For a client application to execute a server command, Client-Library must convert the command to a symbolic command stream that can be sent to the server. The command stream contains information about the type of the command and the data needed for execution. For example, a dynamic SQL prepare command requires a statement identifier and the text of the statement to prepare. The steps for executing a dynamic SQL command are as follows:

  a Initiate the command by calling ct_dynamic. This routine sets up internal structures that are used in building a command stream to send to the server.

  b Pass parameters for the command, if required. Most applications pass parameters by calling ct_param or ct_setparam once for each parameter that the command requires, but it is also possible to pass parameters for a command by using ct_dyndesc or ct_dynsqlda.

  c Send the command to the server by calling ct_send.

  d Process the results of the command by calling ct_results.

  A dynamic SQL command that executes a prepared statement returns fetchable results. The other dynamic SQL command types do not return fetchable results, but do return command status results. See “Results” on page 241 for a discussion of processing results.

- The following rules apply to the use of ct_dynamic:

  - When a command structure is initiated, an application must either send the initiated command or clear it before a new command can be initiated with ct_command, ct_cursor, ct_dynamic, or ct_sendpassthru.

  - After sending a command, an application must completely process or cancel all results returned by the command’s execution before initiating a new command on the same command structure.

  - An application cannot call ct_dynamic to initiate a command on a command structure that is managing a cursor. The application must deallocate the cursor first or use a different command structure.

- Client-Library allows an application to resend a command by calling ct_send immediately after the application has processed the results from the previous execution. To resend a command, the application updates the contents of any parameter source variables that were specified with ct_setparam, then calls ct_send. The following dynamic SQL commands can be resent successfully:

  - Execute-immediate commands

- Execute commands on a prepared statement
- Describe-output or describe-input commands

If the application resends other dynamic SQL commands, they result in server processing errors. Client-Library allows an application to resend a command as long as a new command has not been initiated with ct_command, ct_cursor, ct_dynamic, or ct_sendpassthru.

Preparing a statement

- To initiate a command to prepare a statement, an application calls ct_dynamic with *type* as CS_PREPARE, *id* as a unique statement identifier, and *buffer* as the statement text.
- A prepared SQL statement is a SQL statement that is compiled and stored by a server. Each prepared statement is associated with a unique identifier.
- An application can prepare an unlimited number of statements, but identifiers for prepared statements must be unique within a connection.
- Although the command structure used to prepare a statement can be different from the one used to execute it, both of the command structures must belong to the same connection.
- A prepared statement can be a Transact-SQL statement containing **placeholders** for values. Placeholders act like variables in the prepared statement. A placeholder is indicated by a question mark (?) in the statement. A placeholder can occur in the following locations:
  - In place of one or more values in an insert statement
  - In the set clause of an update statement
  - In the where clause of a select or update statement

  When building a command to execute the prepared statement, the application substitutes a value for each dynamic parameter marker by calling ct_param, ct_setparam, ct_dyndesc, or ct_dynsqlda.

  Once a statement is prepared, an application can send a dynamic SQL describe-input command to the server to get a description of the input parameters required to execute the statement.
- To initiate a command to prepare a statement that executes a stored procedure, specify “exec sp_name” as the SQL text, where “sp_name” is the name of the stored procedure to be executed:

```
ct_dynamic(cmd, CS_PREPARE, “myid”, CS_NULLTERM,
     “exec sp_2”, CS_NULLTERM);
```

- Once a statement is successfully prepared, the application can execute it repeatedly until it is deallocated.

Declaring a cursor on a prepared statement

- To initiate a command to declare a cursor on a prepared statement, an application calls ct_dynamic with *type* as CS_CURSOR_DECLARE.
- After declaring a cursor on a prepared statement, an application can call ct_cursor(CS_CURSOR_OPTION) to set an option ("readonly" or "'for update") for the cursor-declaration command. This step is necessary only if the select statement does not include a for read only or for update of clause to specify which, if any, columns are to be updatable. The sequence of calls is:
  - ct_dynamic(CS_CURSOR_DECLARE)
  - ct_cursor(CS_CURSOR_OPTION)
  - ct_send
  - ct_results, as many times as necessary
- A ct_dynamic cursor-declare command cannot be batched with subsequent ct_cursor cursor-rows or cursor-open commands.
- After a cursor is declared on a prepared statement, use ct_cursor to initiate additional commands on the cursor.
- An application must declare a cursor on a prepared statement prior to executing the prepared statement.

Getting a description of prepared statement input

- An application typically retrieves a description of prepared statement input parameters before executing the prepared statement for the first time.
- To get a description of prepared statement input:
  - a Call ct_dynamic with *type* as CS_DESCRIBE_INPUT to initiate a command to get the description.
  - b Call ct_send to send the command to the server.
  - c Call ct_results as necessary to process the results of the command. A CS_DESCRIBE_INPUT command generates a result set of type CS_DESCRIBE_RESULT. This result set contains no fetchable data but does contain descriptive information for each of the input values.

d Call ct_res_info to retrieve the number of input values. This assumes that CS_DESCRIBE_RESULT was returned, as does the following step.

e For each input value, call ct_describe.

Alternately, an application can use ct_dyndesc or ct_dynsqlda to retrieve the description. ct_dyndesc requires several calls to obtain the number of inputs and the format of each. ct_dynsqlda can retrieve a description with one call but requires an application-managed SQLDA structure. These alternatives are described in the following sections:

- For a description of the ct_dynsqlda method, see "Sybase SQLDA: Retrieving input formats" on page 492.
- For a description of the ct_dyndesc method, see "Getting descriptions of command inputs or outputs with ct_dyndesc" on page 487.

Getting a description of prepared statement output

- An application typically retrieves a description of prepared statement result columns before executing the prepared statement for the first time.

Note A single dynamic SQL batch may contain multiple SQL statements. The description of the prepared statement output, however, only describes the first resultset. You will receive full descriptions of each resultset only when the dynamic SQL statement is executed.

- To get a description of prepared statement output columns:

a Call ct_dynamic with *type* as CS_DESCRIBE_OUTPUT to initiate a command to get the description.

b Call ct_send to send the command to the server.

c Call ct_results as necessary to process the results of the command. A ct_dynamic(CS_DESCRIBE_OUTPUT) command generates a result set of type CS_DESCRIBE_RESULT. This result set contains no fetchable data but does contain descriptive information for each output column.

d Call ct_res_info to retrieve the number of output columns. This assumes that CS_DESCRIBE_RESULT was returned, as does the following step.

e For each output column, call ct_describe.

Alternately, an application can use ct_dyndesc or ct_dynsqlda to retrieve the description. ct_dyndesc requires several calls to obtain the number of columns and the format of each. ct_dynsqlda can retrieve a description with one call but requires an application-managed SQLDA structure. These alternatives are described in the following sections:

- For a description of the ct_dynsqlda method, see "Sybase SQLDA: Retrieving output formats" on page 493.
- For a description of the ct_dyndesc method, see "Getting descriptions of command inputs or outputs with ct_dyndesc" on page 487.

Executing a prepared statement

- To execute a prepared statement:
  - a Call ct_dynamic with *type* as CS_EXECUTE to initiate a command to execute the statement.
  - b Define the input values to the SQL statement. You can do this by:
    - Calling ct_param once for each parameter. ct_param and ct_setparam offer the best performance. ct_param does not allow the application to change parameter values before resending the command.
    - Calling ct_setparam once for each parameter. ct_setparam takes pointers to parameter source values. This method is the only one that allows parameter values to be changed before resending the command.
    - Calling ct_dyndesc several times to allocate a dynamic descriptor area, populate it with data values, and apply it to the command. See "Passing parameter values with ct_dyndesc" on page 488 for more information. ct_dyndesc(CS_USE_DESC) calls ct_param internally.
    - By calling ct_dynsqlda to apply the contents of a user-allocated SQLDA structure to the command. See "Sybase SQLDA: Passing command input parameters" on page 494 for more information. ct_dynsqlda(CS_SQLDA_PARAM) calls ct_param internally.
  - c Call ct_send to send the command to the server.
  - d Call ct_results as necessary to process the results of the command.

Executing a literal statement

- A dynamic SQL statement can be executed immediately if it meets the following criteria:
  - It does not return data (it is not a select statement).
  - It does not contain placeholders for parameters, which are indicated by a question mark (?) in the text of the statement.
  - Dynamic parameter markers act as placeholders that allow users to specify actual data to be substituted into a SQL statement at runtime.
- To execute a literal statement:
  - a Call ct_dynamic with *type* as CS_EXEC_IMMEDIATE, *id* as NULL, and *buffer* as the statement to execute.
  - b Call ct_send to send the command to the server.
  - c Call ct_results as necessary to process the results of the command.

Deallocating a prepared statement

- To initiate a command to deallocate a prepared statement, an application calls ct_dynamic with *type* as CS_DEALLOC and *id* as the statement identifier.

See also

ct_dyndesc, ct_dynsqlda, ct_param, ct_setparam, ct_send, ct_cursor

ct_dyndesc

Description

Perform operations on a dynamic SQL descriptor area.

Syntax

```
CS_RETCODE ct_dyndesc(cmd, descriptor, desclen, operation, index,
                      datafmt, buffer, buflen, copied, indicator)

CS_COMMAND      *cmd;
CS_CHAR         *descriptor;
CS_INT          desclen;
CS_INT          operation;
CS_INT          index;
CS_DATAFMT      *datafmt;
CS_VOID         *buffer;
CS_INT          buflen;
CS_INT          *copied;
CS_SMALLINT     *indicator;
```

Parameters

cmd
A pointer to a CS_COMMAND structure. Any CS_COMMAND in the same context in which a descriptor is allocated can be used to operate on the descriptor.

descriptor
A pointer to the name of the descriptor. Descriptor names must be unique within a context.

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

operation
The descriptor operation to initiate. The following table lists the values for *operation*:

Table 3-25: Values for ct_dyndesc operation parameter

Value of operation	Result
CS_ALLOC	Allocates a descriptor.
CS_DEALLOC	Deallocates a descriptor.
CS_GETATTR	Retrieves a parameter or result item's attributes.
CS_GETCNT	Retrieves the number of parameters or columns.
CS_SETATTR	Sets a parameter's attributes.
CS_SETCNT	Sets the number of parameters or columns.
CS_USE_DESC	Associates a descriptor with a statement or a command structure.

index
When used, an integer variable.

Depending on the value of *operation*, *index* can be either the 1-based index of a descriptor item or the number of items associated with a descriptor.

datafmt
When used, a pointer to a CS_DATAFMT structure.

buffer
When used, a pointer to data space.

buflen
When used, *buflen* is the length, in bytes, of the **buffer* data.

copied
When used, a pointer to an integer variable. ct_dyndesc sets **copied* to the length, in bytes, of the data placed in **buffer*.

indicator

When used, a pointer to an indicator variable.

Table 3-26: Values for ct_dyndesc indicator parameter

Value of operation	Value of *indicator	Meaning
CS_GETATTR	-1	Truncation of a server value by Client-Library.
	0	No truncation.
	integer value	Truncation of an application value by the server.
CS_SETATTR	-1	The parameter has a null value.

Return value

ct_dyndesc returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_ROW_FAIL	A recoverable error occurred. Recoverable errors include conversion errors that occur while copying values to program variables as well as memory allocation failures.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Usage

- A dynamic SQL descriptor area contains information about the input parameters to a dynamic SQL statement or the result data items generated by the execution of a dynamic SQL statement.
- Although ct_dyndesc takes a CS_COMMAND structure as a parameter, the scope of a dynamic SQL descriptor area is a Client-Library context; that is:
  - Descriptor names must be unique within a context.
  - An application can use any command structure within a context to reference the context’s descriptor areas. For example, a descriptor area allocated through one command structure can be deallocated by another command structure within the same context.
- For more information about dynamic SQL, see Chapter 8, “Using Dynamic SQL Commands,” in the Open Client *Client-Library/C Programmer's Guide*.

Allocating a descriptor

- To allocate a descriptor, an application calls ct_dyndesc with *operation* as CS_ALLOC.
- Table 3-27 lists parameter values for CS_ALLOC operations:

Table 3-27: Parameter values for ct_dyndesc(CS_ALLOC) operations

descriptor, desclen	index	datafmt	buffer, buflen	copied	indicator
The name of the descriptor to allocate, the length of the name or CS_NULLTERM.	The maximum number of items that the descriptor will accommodate.	NULL	NULL, CS_UNUSED	NULL	NULL

Deallocating a descriptor

- To deallocate a descriptor, an application calls ct_dyndesc with *operation* as CS_DEALLOC.
- Table 3-28 lists parameter values for CS_DEALLOC operations:

Table 3-28: Parameter values for ct_dyndesc(CS_DEALLOC) operations

descriptor, desclen	index	datafmt	buffer, buflen	copied	indicator
The name of the descriptor to deallocate, the length of the name or CS_NULLTERM.	CS_UNUSED	NULL	NULL, CS_UNUSED	NULL	NULL

Retrieving a parameter or result item's attributes

- To retrieve the attributes of a parameter or a result data item, an application calls ct_dyndesc with *operation* as CS_GETATTR.
- Table 3-29 lists parameter values for CS_GETATTR operations:

Table 3-29: Parameter values for ct_dyndesc(CS_GETATTR) operations

descriptor, desclen	index	datafmt	buffer, buflen	copied	indicator
The name of the descriptor of interest, the length of the name or CS_NULLTERM.	The number of the item whose description is being requested. Index numbers start with 1.	As an input parameter, **datafmt* describes **buffer*. ct_dyndesc overwrites **datafmt* with a description of the item.	If supplied, **buffer* is set to the value of the item. If *buffer* is NULL, only a description of the item is returned. *buflen* must be CS_UNUSED. *datafmt->maxlength* describes **buffer*'s length.	If supplied, **copied* is set to the number of bytes placed in **buffer*. Can be NULL.	If supplied, **indicator* is set to the value of the item's indicator. Can be NULL.

- If necessary, ct_dyndesc converts the column's source data to the format described by **datafmt* and places the result in **buffer*. If pointers are supplied for **indicator* and **copied*, they are set accordingly.
- An application needs to set the **datafmt* fields for a CS_GETATTR operation exactly as they would be set for a ct_bind call, except that *datafmt–>count* must be 0 or 1 (only one column value at a time can be retrieved). Table 3-30 lists the CS_DATAFMT fields that are used:

Table 3-30: CS_DATAFMT settings for ct_dyndesc(CS_GETATTR) operations

Field name	Set field to
datatype	The datatype of the *buffer* variable.
format	A bitmask of format symbols.
maxlength	The length of the *buffer* data space.
scale	If *buffer* is a numeric or decimal variable, the maximum number of digits that can be represented to the right of the decimal point; *scale* is ignored for all other datatypes.
precision	If *buffer* is a numeric or decimal variable, the maximum number of decimal digits that can be represented; *precision* is ignored for all other datatypes.
count	0 or 1.
locale	A pointer to a valid CS_LOCALE structure or NULL.

All other fields are ignored.

- ct_dyndesc(CS_GETATTR) sets the **datafmt* fields exactly as ct_describe would set them. Table 3-31 lists the fields in **datafmt* that ct_dyndesc sets:

Table 3-31: CS_DATAFMT fields set by ct_dyndesc(CS_GETATTR) operations

Field name	ct_dyndesc sets field to
name	The null-terminated name of the data item, 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 to indicate a NULL *name*.
datatype	The datatype of the item. All datatypes listed in "Datatypes support" on page 294 are valid, with the exceptions of CS_VARCHAR and CS_VARBINARY.
maxlength	The maximum possible length of the data for the column or parameter.
scale	The maximum number of digits to the right of the decimal point in the result data item.
precision	The maximum number of decimal digits that can be represented in the result data item.
status	A bitmask of the following values: • CS_CANBENULL to indicate that the column can contain NULL values. • CS_HIDDEN to indicate that the column is a "hidden" column that has been exposed. For information on hidden columns, see "Hidden keys" on page 214. • 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. • CS_TIMESTAMP to indicate the column is a timestamp column. • CS_UPDATABLE to indicate that the column is an updatable cursor column. • CS_UPDATECOL to indicate that the column is in the update clause of a cursor declare command. • CS_RETURN to indicate that the column is a return parameter of an RPC command.
count	ct_dyndesc sets *count* to 1.
usertype	The Adaptive Server user-defined datatype of the column or parameter, if any. usertype is set in addition to (not instead of) datatype.

Field name	ct_dyndesc sets field to
locale	A pointer to a CS_LOCALE structure that contains locale information for the data. This pointer can be NULL.

Retrieving the number of parameters or columns

- To retrieve the number of parameters or result items a descriptor can describe, an application calls ct_dyndesc with *operation* as CS_GETCNT.
- ct_dyndesc sets **buffer* to the number of dynamic parameter specifications or the number of columns in the dynamic SQL statement's select list, depending on whether input parameters or output columns are being described.
- The following table lists parameter values for CS_GETCNT operations:

Table 3-32: Parameter values for ct_dyndesc(CS_GETCNT) operations

descriptor, desclen	index	datafmt	buffer, buflen	copied	indicator
The name of the descriptor of interest, the length of the name or CS_NULLTERM.	CS_UNUSED	NULL	A pointer to a CS_INT, CS_UNUSED.	If supplied, **copied* is set to the number of bytes placed in **buffer*. Can be NULL.	NULL

Setting a parameter's attributes

- To set a parameter's attributes, an application calls ct_dyndesc with *operation* as CS_SETATTR.
- Table 3-33 lists parameter values for CS_SETATTR operations:

Table 3-33: Parameter values for ct_dyndesc(CS_SETATTR) operations

descriptor, desclen	index	datafmt	buffer, buflen	copied	indicator
The name of the descriptor of interest, the length of the name or CS_NULLTERM.	The number of the item whose description is being set. Index numbers start with 1.	**datafmt* contains a description of the item.	A pointer to the value of the item, the length of the value. Pass *buflen* as CS_UNUSED if *buffer* points to a fixed-length type.	NULL	If supplied, **indicator* is the value of the item's indicator. If **indicator* is -1, then *buffer* is ignored and the value of the item is set to NULL. *indicator* can be NULL.

- An application needs to set the **datafmt* fields for a CS_SETATTR operation exactly as they would be set for a ct_param call. Table 3-34 lists the fields that are used:

Table 3-34: CS_DATAFMT fields for ct_dyndesc(CS_SETATTR) operations

Field name	Set field to
name	The name of the parameter.
namelen	The length of the name or CS_NULLTERM.
datatype	The datatype of the item being set.
maxlength	For variable-length return parameters, *maxlength* is the maximum number of bytes to be returned for this parameter. *maxlength* is ignored if *status* is CS_INPUTVALUE or if *datatype* represents a fixed-length type.
status	CS_INPUTVALUE, CS_UPDATECOL, or CS_RETURN. CS_UPDATECOL indicates an update column for a cursor-declare command. CS_RETURN indicates a return parameter.
locale	A pointer to a valid CS_LOCALE structure or NULL.

All other fields are ignored.

Setting the number of parameters or columns

- To set the number of parameters or columns a descriptor can describe, an application calls ct_dyndesc with *operation* as CS_SETCNT.
- Table 3-35 lists parameter values for CS_SETCNT operations:

Table 3-35: Parameter values for ct_dyndesc(CS_SETCNT) operations

descriptor, desclen	index	datafmt	buffer, buflen	copied	indicator
The name of the descriptor of interest, the length of the name or CS_NULLTERM.	The new descriptor count	NULL	NULL, CS_UNUSED	NULL	NULL

Associating a descriptor with a statement or command structure

- To associate a descriptor with a prepared statement or command structure, an application calls ct_dyndesc with *operation* as CS_USE_DESC.
- Table 3-36 lists parameter values for CS_USE_DESC operations:

Table 3-36: Parameter values for ct_dyndesc(CS_USE_DESC) operations

descriptor, desclen	index	datafmt	buffer, buflen	copied	indicator
The name of the descriptor of interest, the length of the name or CS_NULLTERM.	CS_UNUSED	NULL	NULL, CS_UNUSED	NULL	NULL

- Descriptor areas are normally associated with a context structure. When a descriptor area is used to describe input to or output from a cursor, however, it must first be associated with the command structure which opened the cursor.
- When using a descriptor to describe cursor input, a typical application's sequence of calls is:

```
ct_dyndesc(CS_ALLOC)
 ct_dyndesc(CS_SETCNT)
 for each input value:
     ct_dyndesc(CS_SETATTR)
 end for
 ct_cursor to open the cursor
 ct_dyndesc(CS_USE_DESC)
 ct_send
```

Getting descriptions of command inputs or outputs with *ct_dyndesc*

- The sequence of calls to retrieve a description of a prepared statement's input parameters or result columns with ct_dyndesc is described below.

  a Call ct_dyndesc with *operation* as CS_ALLOC to allocate a descriptor area.

  b Call ct_dynamic to initiate the command to get the description. Pass the ct_dynamic *type* argument as CS_DESCRIBE_INPUT for input descriptions and as CS_DESCRIBE_OUTPUT for output descriptions.

  c Call ct_send to send the command to the server.

  d Call ct_results as necessary to process the results of the command. A describe command generates a result set of type CS_DESCRIBE_RESULT. This result set contains no fetchable data but does contain descriptive information for each of the input values.

  e Call ct_dyndesc with *operation* as CS_USE_DESC to associate the prepared statement with the descriptor area allocated in step 1. This assumes that CS_DESCRIBE_RESULT was returned as ct_results' current *result_type* value, as do the following two steps.

f Call ct_dyndesc with *operation* as CS_GETCNT to get the number of parameters or columns.

g For each parameter or column, call ct_dyndesc with *operation* as CS_GETATTR to get the value's description.

Passing parameter values with *ct_dyndesc*

- When executing a prepared dynamic SQL statement, an application can supply input parameter values with ct_dyndesc. The sequence of calls is as follows:

  a Call ct_dynamic(CS_EXECUTE) to initiate the command.

  b For each required input parameter, call ct_dyndesc with *operation* as CS_SETATTR to place a parameter value in the descriptor area. If necessary, convert the value with cs_convert first. The CS_SETATTR usage is summarized under "Setting a parameter's attributes" on page 485.

  c Call ct_dyndesc with *operation* as CS_USE_DESC to apply the parameter values to the command.

  d Call ct_send to send the command to the server.

  e Process the results of the command. See "Results" on page 241 if you are unfamiliar with Client-Library's results model.

- Client-Library allows applications to resend a dynamic-SQL execute command by calling ct_send immediately after the application has processed the results of the previous execution. However, parameter values are fixed for all executions of the command if ct_dyndesc is used to supply parameter values. Applications that resend commands with different parameter values should use ct_setparam instead. For more information on this feature, see ct_setparam and "Resending commands" on page 580.

Retrieving result column values with *ct_dyndesc*

- When processing fetchable results, an application can retrieve result column values with ct_dyndesc and ct_fetch. (Fetchable results are indicated by the value of the ct_results *result_type* parameter).

- The sequence of calls is summarized below. This sequence assumes that ct_results has returned a *result_type* value that indicates fetchable data:

  a Call ct_dyndesc with *operation* as CS_USE_DESC to associate the descriptor with the result rows.

b Call ct_fetch to fetch a result row. If ct_fetch returns CS_END_DATA, then all the rows have been retrieved.

c For each column in the result set, call ct_dyndesc with *operation* as CS_GETATTR to get the column's value. CS_GETATTR usage is summarized under "Retrieving a parameter or result item's attributes" on page 482.

d Repeat steps 2–4 until ct_fetch returns CS_END_DATA.

See also

ct_bind, ct_cursor, ct_describe, ct_dynamic, ct_dynsqlda, ct_fetch, ct_param

ct_dynsqlda

Description

Operate on a SQLDA structure.

Syntax

CS_RETCODE ct_dynsqlda(cmd, sqlda_type, dap, operation)

```
CS_COMMAND    *cmd;
CS_INT        sqlda_type;
SQLDA         *dap;
CS_INT        operation;
```

Parameters

cmd

A pointer to a CS_COMMAND structure.

sqlda_type

Symbolic constant describing the type of SQLDA structure pointed at by *dap*. In this version, *sqlda_type* must be CS_SQLDA_SYBASE to indicate a Sybase-style SQLDA structure.

dap

The address of a SQLDA structure. The SQLDA structure is defined in the Sybase *sqlda.h* header file. See "Sybase-style SQLDA structure" on page 490 for the definition of this structure.

operation

The operation to perform. Table 3-37 summarizes the use of ct_dynsqlda:

Table 3-37: Values for ct_dynsqlda operation parameter

Value of operation	Function
CS_GET_IN	Fills **dap* with a description of the input parameters for a prepared dynamic SQL statement.
CS_GET_OUT	Fills **dap* with a description of the columns returned by a prepared dynamic SQL statement.
CS_SQLDA_ PARAM	Uses a SQLDA structure to supply input parameters for the execution of a prepared statement. When executing a prepared dynamic SQL statement, this operation applies the contents of **dap* as input parameters.
CS_SQLDA_ BIND	Uses a SQLDA structure to process results from the execution of a prepared statement. When processing the results returned by the execution of a prepared dynamic SQL statement, this operation binds the contents of **dap* to the result columns.

Return value

ct_dynsqlda returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is pending for this connection. See “Asynchronous programming” on page 12.

Usage

- A SQLDA structure is used with prepared dynamic SQL statements. It contains format descriptions and (optionally) values for command input parameters or result columns.
- For more information about dynamic SQL, see Chapter 8, “Using Dynamic SQL Commands,” in the Open Client *Client-Library/C Programmer's Guide*.
- ct_dynsqlda manages a SQLDA structure. A SQLDA structure contains data areas for the descriptions and values of a command’s input parameters or result values.

Sybase-style SQLDA structure

- The Sybase-style SQLDA is a self-describing, variable-length structure, declared as follows:

```
typedef struct _sqlda
 {
     CS_SMALLINT sd_sqln; /* Actual length of column array    */
     CS_SMALLINT sd_sqld; /* Current number of columns        */
```

```
    /*
    ** The following array is treated as if it were the length
    ** indicated by sd_sqln.
    */
    struct _sd_column
    {
        CS_DATAFMT sd_datafmt; /* Format of column i.       */
        CS_VOID *sd_sqldata;   /* Value buffer for column i. */
        CS_INT sd_sqllen;      /* Length of current value.   */
        CS_SMALLINT sd_sqlind; /* Indicator for column i.    */
        CS_VOID *sd_sqlmore;   /* Reserved for future use.   */
    } sd_column[1];
} sqlda;

#define SYB_SQLDA_SIZE(n) (sizeof(sqlda) \
        - sizeof(struct _sd_column) \
                + (n) * sizeof(struct _sd_column))
```

Allocating SQLDA structures

- The application is responsible for correctly allocating and initializing the structure pointed to by *dap*. The actual size of the structure depends on the number of columns that the structure is to describe. An application can use the SYB_SQLDA_SIZE macro to allocate a SQLDA buffer of the appropriate size. On a system that uses malloc to allocate memory, this can be done as follows:

```
#define MAX_COLUMNS 16
 SQLDA *dap;

 dap = (SQLDA *) malloc(
                  SYB_SQLDA_SIZE(MAX_COLUMNS) );
 if (dap == (SQLDA *) NULL)
   ... out of memory ...

 memset((void *)dap, 0,
        SYB_SQLDA_SIZE(MAX_COLUMNS));
 dap->sd_sqln = MAX_COLUMNS;
```

An application can invoke the SQLDA_DECL macro to declare a static SQLDA structure. The invocation:

```
SQLDA_DECL(name, size);
```

Is equivalent to the declaration:

```
struct {
         CS_SMALLINT sd_sqln;
         CS_SMALLINT sd_sqld;
```

```
                struct {
                        CS_DATAFMT sd_datafmt;
                        CS_VOID *sd_sqldata;
                        CS_SMALLINT sd_sqlind;
                        CS_INT sd_sqllen;
                        CS_VOID *sd_sqlmore;
                } sd_column[(size)];
        } name;
```

- If the structure will be used to pass input parameters or retrieve results, the (using ct_dynsqlda's CS_SQLDA_PARAM or CS_SQLDA_BIND operations), then the application must also allocate buffers for item values and set the buffer lengths in the structure.
- The use of the Sybase-style SQLDA is explained in the following sections.

Sybase SQLDA: Retrieving input formats

- ct_dynsqlda(*cmd*, CS_SQLDA_SYBASE, &*sqlda*, CS_GET_IN) fills the fields of *sqlda* with a description of the input parameters required to execute a prepared statement.
- A prepared dynamic SQL statement can contain parameter markers for values to be supplied at execution time.
- A dynamic SQL statement can contain parameter markers for parameters to be supplied at execution time. After a dynamic SQL statement is prepared, the application can request a description of the format of the statement's parameter. The procedure is:

  a Build and send a ct_dynamic(CS_DESCRIBE_INPUT) command to the server.

  b Handle the results of the command with ct_results. When ct_results returns a *result_type* value of CS_DESCRIBE_RESULT, the parameter formats are available.

  c If necessary, call ct_res_info(CS_NUMDATA) to find out how many parameters the statement requires. The SQLDA structure contains an array of column descriptors. The array must contain at least one entry for each required parameter.

  d Call ct_dynsqlda to retrieve the parameter formats.
- The application is responsible for allocating the SQLDA structure and the memory pointed to by its constituent pointers. The field settings for a CS_GET_IN operation are as follows:

Table 3-38: SQLDA fields for ct_dynsqlda(CS_GET_IN) calls

Field	Description
sqlda->sd_sqln	On input, the number of elements in the array that starts at *sqlda->sd_column*. The SQLDA must be sufficiently large. See "Allocating SQLDA structures" on page 491.
sqlda->sd_sqld	On output, the actual number of items.
sqlda->sd_column[*i*].*sd_sqldata*	Unused (ignored).
sqlda->sd_column[*i*].*sd_sqllen*	Unused (ignored).
sqlda->sd_column[*i*].*sd_datafmt*	On output, the CS_DATAFMT fields for each parameter are set exactly as ct_describe would set them (see Table 3-18 on page 446).
sqlda->sd_column[*i*].*sd_sqlind*	Unused (ignored).

Sybase SQLDA: Retrieving output formats

- ct_dynsqlda(*cmd*, CS_SQLDA_SYBASE, &*sqlda*, CS_GET_OUT) fills the fields of *sqlda* with a description of the results returned by the execution of a prepared statement.
- A dynamic SQL statement can contain a server select command. After a dynamic SQL statement is prepared, the application can request a description of the format of the row data returned by the statement. The procedure is:

  a Build and send a ct_dynamic(CS_DESCRIBE_OUTPUT) command to the server.

  b Handle the results of the command with ct_results. When ct_results returns a *result_type* value of CS_DESCRIBE_RESULT, the output formats are available to the application.

  c If necessary, call ct_res_info(CS_NUMDATA) to find out how many columns the statement returns. The SQLDA structure contains the address of an array of column descriptors. This array must contain at least one entry per column.

  d Call ct_dynsqlda to retrieve the column formats.

- The application is responsible for allocating the SQLDA structure and the memory pointed to by its constituent pointers. The field settings for a CS_GET_OUT operation are as follows:

Table 3-39: SQLDA fields for ct_dynsqlda(CS_GET_OUT) calls

Field	Description
sqlda->sd_sqln	On input, the number of elements in the array that starts at *sqlda->sd_column*. The SQLDA must be sufficiently large. See “Allocating SQLDA structures” on page 491.
sqlda->sd_sqld	On output, the actual number of items.
sqlda->sd_column[*i*].*sd_sqldata*	Unused (ignored).
sqlda->sd_column[*i*].*sd_sqllen*	Unused (ignored).
sqlda->sd_column[*i*].*sd_datafmt*	On output, the CS_DATAFMT fields for each column are set exactly as ct_describe would set them (see Table 3-18 on page 446).
sqlda->sd_column[*i*].*sd_sqlind*	Unused (ignored).

Sybase SQLDA: Passing command input parameters

- ct_dynsqlda(*cmd*, CS_SQLDA_SYBASE, &*sqlda*, CS_SQLDA_PARAM) applies the contents of an SQLDA structure as input parameter values for the execution of a prepared statement.
- The procedure for using ct_dynsqlda to pass parameters for the execution of a prepared statement is as follows:

  a (Optional) Get a description of the command inputs as described by “Sybase SQLDA: Retrieving input formats” on page 492.

  b Call ct_dynamic(CS_EXECUTE) to initiate the command.

  c Fill in the fields of the SQLDA as described in the table below.

  d Call ct_dynsqlda(CS_SQLDA_PARAM) to apply the SQLDA’s contents as input parameter values.

  e Send the command with ct_send.

  f Handle the results of the command.

- The application is responsible for allocating the SQLDA structure and the memory pointed to by its constituent pointers. The field settings for a CS_SQLDA_PARAM operation are as follows:

Table 3-40: SQLDA fields for ct_dynsqlda(CS_SQLDA_PARAM) calls

Field	Description
sqlda->sd_sqln	On input, the number of elements in the array that starts at *sqlda->sd_column*. The array must have as many entries as the number of items requested by the *sd_sqld* field. See "Allocating SQLDA structures" on page 491.
sqlda->sd_sqld	On input, the number of items in the *sqlda->sd_column* array that should be applied as parameter values.
sqlda->sd_column[*i*].*sd_sqldata*	When executing a command, contains the address of a buffer containing a value for parameter *i* (with 0 being the first parameter marker in the statement).
sqlda->sd_column[*i*].*sd_sqllen*	The length, in bytes, of the buffer pointed at by *sd_column*[*i*].*sd_sqldata*.
sqlda->sd_column[*i*].*sd_datafmt*	The CS_DATAFMT fields for each column must be set exactly as required by ct_param (see Table 3-49 on page 538).
sqlda->sd_column[*i*]->*sd_sqlind*	When executing a command, a value of -1 indicates that the value for parameter *i* is NULL.

Sybase SQLDA: Retrieving results

- ct_dynsqlda(*cmd*, CS_SQLDA_SYBASE, &*sqlda*, CS_SQLDA_BIND) binds the contents of an SQLDA structure to columns in the results returned by the execution of a prepared statement.
- The procedure for using ct_dynsqlda for results processing is as follows:
  - a (Optional) Get a description of the command outputs as described in "Sybase SQLDA: Retrieving output formats" on page 493.
  - b Call ct_dynamic(CS_EXECUTE) to initiate the command.
  - c Supply any necessary parameter values for execution.
  - d Send the command with ct_send.
  - e Handle the results of the command. When ct_results returns a *result_type* value of CS_ROW_RESULT, the SQLDA structure can be bound to the result rows.

f Fill in the fields of the SQLDA as described in the table below, then call ct_dynsqlda(CS_SQLDA_BIND) to bind them to the column values in the result rows.

g Process the rows with ct_fetch. Each call to ct_fetch places values, converted if necessary, into the bound fields of the SQLDA.

- The application is responsible for allocating the SQLDA structure and the memory pointed to by its constituent pointers. The field settings for a CS_SQLDA_BIND operation are as follows:

Table 3-41: SQLDA fields for ct_dynsqlda(CS_SQLDA_BIND) calls

Field	Description
sqlda->sd_sqln	On input, the number of elements in the array that starts at *sqlda->sd_column*. The array must be at least as long as the number of items requested by the *sd_sqld* field. See "Allocating SQLDA structures" on page 491.
sqlda->sd_sqld	On input, the number of items in the *sqlda->sd_column* array that should be bound to result columns.
sqlda->sd_column[*i*].*sd_sqldata*	Contains the address of a buffer where ct_fetch will place values for column *i* (with 0 being the first column).
sqlda->sd_column[*i*].*sd_sqllen*	The length, in bytes, of the buffer pointed at by *sd_column*[*i*]->*sd_sqldata*.
sqlda->sd_column[*i*].*sd_datafmt*	The CS_DATAFMT fields for each column must be set exactly as required by ct_bind (see Table 3-1 on page 325).
sqlda->sd_column[*i*].*sd_sqlind*	Subsequent calls to ct_fetch will write indicator values for each column. Indicator values are as follows: • -1 indicates the column value is NULL. • 0 indicates a successful fetch. • Any positive integer indicates truncation. The value is the actual length of the column value before truncation.

See also

ct_bind, ct_cursor, ct_describe, ct_dynamic, ct_dyndesc, ct_fetch, ct_param, ct_res_info

ct_exit

Description

Exit Client-Library.

Syntax

CS_RETCODE ct_exit(context, option)

```
CS_CONTEXT    *context;
CS_INT         option;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

context identifies the Client-Library context being exited.

option

ct_exit can behave in different ways, depending on the value specified for *option*. The following symbolic values are legal for *option*:

Value of option	Result
CS_UNUSED	ct_exit closes all open connections for which no results are pending and terminates Client-Library for this context. If results are pending on one or more connections, ct_exit returns CS_FAIL and does not terminate Client-Library.
CS_FORCE_EXIT	ct_exit closes all open connections for this context, whether or not any results are pending, and terminates Client-Library for this context.

To properly exit Client-Library, wait until all asynchronous operations are complete, then call ct_exit.

If an asynchronous operation is in progress when ct_exit is called, the routine returns CS_FAIL and does not exit Client-Library properly, even when CS_FORCE_EXIT is used.

Return value

ct_exit returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

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;
 }
```

This code excerpt is from the *exutils.c* sample program.

Usage

- ct_exit terminates Client-Library for a specific context. It closes all open connections, deallocates internal data space and cleans up any platform-specific initialization.
- ct_exit must be the last Client-Library routine called within a Client-Library context.
- If an application finds it needs to call Client-Library routines after it has called ct_exit, it can reinitialize Client-Library by calling ct_init again.
- If results are pending on any of the context's connections and *option* is not passed as CS_FORCE_EXIT, ct_exit returns CS_FAIL. This means that Client-Library is not correctly terminated and that the application must call ct_exit again after handling the pending results.
- ct_exit always completes synchronously, even if asynchronous network I/O has been specified for any of the context's connections.

- An application can call ct_close to close a single connection.
- If ct_init is called for a context, it is an error to deallocate the context before calling ct_exit.

See also

ct_close, ct_init

ct_fetch

Description

Fetch result data.

Syntax

```
CS_RETCODE ct_fetch(cmd, type, offset, option,
        rows_read)

 CS_COMMAND *cmd;
 CS_INT      type;
 CS_INT      offset;
 CS_INT      option;
 CS_INT      *rows_read;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

type

This parameter is currently unused and must be passed as CS_UNUSED to ensure compatibility with future versions of Client-Library.

offset

This parameter is currently unused and must be passed as CS_UNUSED to ensure compatibility with future versions of Client-Library.

option

This parameter is currently unused and must be passed as CS_UNUSED to ensure compatibility with future versions of Client-Library.

rows_read

A pointer to an integer variable. ct_fetch sets *rows_read* to the number of rows read by the ct_fetch call.

rows_read is an optional parameter intended for use by applications using array binding.

In asynchronous mode, **rows_read* is not set until ct_fetch completes.

Return value

ct_fetch returns the following values:

Table 3-42: ct_fetch return values

Return value	Meaning
CS_SUCCEED	The routine completed successfully. ct_fetch places the number of rows read in **rows_read*. The application must continue to call ct_fetch, as the result data is not yet completely fetched.
CS_END_DATA	All rows of the current result set have been fetched. The application should call ct_results to get the next result set. Note that this return value does not apply to "ct_scroll_fetch" on page 567. ct_scroll_fetch returning CS_END_DATA is a fatal internal error.
CS_ROW_FAIL	A recoverable error occurred while fetching a row. The application must continue calling ct_fetch to keep retrieving rows, or can call ct_cancel to cancel the remaining results. When using array binding, CS_ROW_FAIL indicates a partial result is available in the bound arrays. ct_fetch 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 ct_fetch 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, ct_fetch sets the corresponding **indicator* variable(s) to a value greater than 0. Indicator variables must have been specified in the application's calls to ct_bind.
CS_FAIL	The routine failed. ct_fetch places the number of rows fetched in **rows_read*. This number includes the failed row. Unless the routine failed due to application error (for example, bad parameters), additional result rows are not available. If ct_fetch returns CS_FAIL, an application must call ct_cancel with *type* as CS_CANCEL_ALL before using the affected command structure to send another command. If ct_cancel returns CS_FAIL, the application must call ct_close(CS_FORCE_CLOSE) to force the connection closed.

Return value	Meaning
CS_CANCELED	The current result set and any additional result sets have been canceled. Data is no longer available. ct_fetch places the number of rows fetched before the cancel occurred in **rows_read*.
CS_PENDING	Asynchronous network I/O is in effect. See “Asynchronous programming” on page 12.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

A common reason for a ct_fetch failure is that a program variable specified through ct_bind is not large enough for a fetched data item.

Examples

```
/* ex_fetch_data()*/

CS_RETCODE CS_PUBLIC
 ex_fetch_data(cmd)
 CS_COMMAND  *cmd;
 {
     CS_RETCODE    retcode;
     CS_INT        num_cols;
     CS_INT        i;
     CS_INT        j;
     CS_INT        row_count = 0;
     CS_INT         rows_read;

    /*
     ** Determine the number of columns in this
     ** result set.
     */
     ...CODE DELETED.....

    /* Get column descriptions and bind columns */
     ...CODE DELETED.....

    /*
     ** Fetch the rows.  Loop while ct_fetch() returns
     ** CS_SUCCEED or CS_ROW_FAIL
     */
     while (((retcode = ct_fetch(cmd, CS_UNUSED,
         CS_UNUSED, CS_UNUSED,&rows_read)) ==
         CS_SUCCEED) || (retcode == CS_ROW_FAIL))
     {
         /*
         ** Increment our row count by the number of
         ** rows just fetched.
```

```
        */
        row_count = row_count + rows_read;

       /* Check if we hit a recoverable error */
        if (retcode == CS_ROW_FAIL)
        {
            fprintf(stdout, "Error on row %d.\n",
                row_count);
        }

       /*
        ** We have a row.  Loop through the columns
        ** displaying the column values.
        */
        for (i = 0; i < num_cols; i++)
        {
            ...CODE DELETED.....
        }
        fprintf(stdout, "\n");
    }

   /* Free allocated space */
    ...CODE DELETED.....

   /*
    ** We're done processing rows.  Let's check the
    ** final return value of ct_fetch().
    */
    switch ((int)retcode)
    {
        case CS_END_DATA:
            /* Everything went fine */
            fprintf(stdout, "All done processing
                rows.\n");
            retcode = CS_SUCCEED;
            break;

       case CS_FAIL:
            /* Something terrible happened */
            ex_error("ex_fetch_data: ct_fetch()
                failed");
            return retcode;
            break;

       default:
            /* We got an unexpected return value */
            ex_error("ex_fetch_data: ct_fetch() \
                returned an unexpected retcode");
            return retcode;
```

```
                break;
        }
        return retcode;
}
```

This code excerpt is from the *exutils.c* sample program.

Usage

- **Result data** is an umbrella term for all the types of data that a server can return to an application. The types of data include:
  - Regular rows
  - Cursor rows
  - Return parameters, such as message parameters, stored procedure return parameters, extended error data, and registered procedure notification parameters.
  - Stored procedure status values
  - Compute rows

  ct_fetch is used to fetch all of these types of data.
- Conceptually, result data is returned to an application in the form of one or more rows that make up a **result set**.

  Regular row and cursor row result sets can contain more than one row. For example, a regular row result set might contain a hundred rows.

  If array binding has been specified for the data items in a regular row or cursor row result set, then multiple rows can be fetched with a single call to ct_fetch.

  Note Asynchronous applications should always specify array binding to fetch multiple rows at a time. This ensures that the application has sufficient time in which to accomplish something before Client-Library calls the application's completion callback routine.

  Return parameter, status number, and compute-row result sets, however, only contain a single "row." For this reason, even if array binding is specified, only a single row of data is fetched.

- ct_results sets **result_type* to indicate the type of result available. ct_results must indicate a result type of CS_ROW_RESULT, CS_CURSOR_RESULT, CS_PARAM_RESULT, CS_STATUS_RESULT, or CS_COMPUTE_RESULT before an application calls ct_fetch.
- After ct_results returns a *result_type* that indicates fetchable results, an application can:
  - Retrieve the result row(s) by binding the result items and fetching the data. A typical application calls ct_res_info to get the number of data items, ct_describe to get data descriptions, ct_bind to bind result items, ct_fetch to fetch result rows, and ct_get_data, if the result set contains large text or image values.
  - Retrieve result rows using ct_dyndesc or ct_dynsqlda with ct_fetch. Typically, only applications that execute dynamic SQL commands use these routines, but ct_dyndesc or ct_dynsqlda can be used to process fetchable data returned by any command type.
  - Discard the result rows using ct_cancel for non-cursor results and ct_cursor(CS_CURSOR_CLOSE) for cursor results.
- If an application does not cancel a result set, it must completely process the result set by calling ct_fetch as long as ct_fetch continues to indicate that rows are available.

  The simplest way to do this is in a loop that terminates when ct_fetch fails to return either CS_SUCCEED or CS_ROW_FAIL. After the loop terminates, an application can use a switch-type statement against ct_fetch's final return code to find out what caused the termination.

  If a result set contains zero rows, an application's first ct_fetch call will return CS_END_DATA.

  Note An application must call ct_fetch in a loop even if a result set contains only a single row. An application must call ct_fetch until it fails to return either CS_SUCCEED or CS_ROW_FAIL.

- If a conversion error occurs when retrieving a result item, the rest of the items in the row are retrieved. If truncation occurs, the indicator variable, if any, provided in the application's ct_bind call for this item is set to the actual length of the result data.

  ct_fetch returns CS_ROW_FAIL if a conversion or truncation error occurs.

Fetching regular rows and cursor rows

- Regular rows and cursor rows can be fetched one row at a time, or several rows at once.
- An application indicates the number of rows to be fetched per ct_fetch call using the *datafmt–>count* field in its ct_bind calls that bind result columns to program variables. If *datafmt–>count* is 0 or 1, each call to ct_fetch fetches one row. If *datafmt–>count* is greater than one, then array binding is considered to be in effect and each call to ct_fetch fetches *datafmt–>count* rows. Note that *datafmt–>count* must have the same value for all ct_bind calls for a result set.
- When fetching multiple rows, if a conversion error occurs on one of the rows, no more rows are retrieved by this ct_fetch call.

Fetching return parameters

- Several types of data can be returned to an application as a parameter result set, including:
  - Stored procedure return parameters
  - Message parameters
- Extended error data and registered procedure notification parameters are also returned as parameter result sets, but since an application does not call ct_results to process these types of data, the application never sees a result type of CS_PARAM_RESULT. Instead, the row of parameters is simply available to be fetched after the application retrieves the CS_COMMAND structure containing the data.
- A return parameter result set consists of a single row with a number of columns equal to the number of return parameters.

Fetching a return status

- A stored procedure return status result set consists of a single row with a single column, the status number.

Fetching compute rows

- Compute rows result from the compute clause of a select statement.
- A compute row result set consists of a single row with a number of columns equal to the number of aggregate operators in the compute clause that generated the row.
- Each compute row is considered to be a distinct result set.

See also

ct_bind, ct_describe, ct_get_data, ct_results, “Results” on page 241, ct_scroll_fetch

ct_get_data

Description

Read a chunk of data from the server.

Syntax

CS_RETCODE ct_get_data(cmd, item, buffer, buflen, outlen)

```
CS_COMMAND    *cmd;
CS_INT         item;
CS_VOID       *buffer;
CS_INT        buflen;
CS_INT        *outlen;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

item

An integer representing the data item of interest. When using ct_get_data to retrieve data for more than one item in a result set, *item* can only be increased by; that is, an application cannot retrieve data for item number 3 after it has retrieved data for item number 4.

When retrieving a column, *item* is the column's column number. The first column in a select-list is column number 1, the second is number 2, and so forth.

When retrieving a compute column, *item* is the column number of the compute column. Compute columns are returned in the order in which they are listed in the compute clause. The first column returned is number 1.

When retrieving a return parameter, *item* is the parameter number. The first parameter returned by a stored procedure is number 1. Stored procedure return parameters are returned in the same order as the parameters were originally specified in the stored procedure's create procedure statement. This is not necessarily the same order as specified in the RPC command that invoked the stored procedure. In determining what number to pass as *item* do not count non-return parameters. For example, if the second parameter in a stored procedure is the only return parameter, pass *item* as 1.

When retrieving a stored procedure return status, *item* must be 1, as there can be only a single status in a return status result set.

buffer

A pointer to data space. ct_get_data fills **buffer* with a *buflen*-sized chunk of the column's value.

buffer cannot be NULL.

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

If *buflen* is 0, ct_get_data updates the I/O descriptor for the item without retrieving any data.

buflen is required even for fixed-length buffers, and cannot be CS_UNUSED.

outlen
A pointer to an integer variable.

If *outlen* is supplied, ct_get_data sets **outlen* to the number of bytes placed in **buffer*.

Return value

ct_get_data returns the following values:

Table 3-43: ct_get_data return values

Return value	Meaning
CS_SUCCEED	ct_get_data successfully retrieved a chunk of data that is not the last chunk of data for this column.
CS_FAIL	The routine failed. Unless the routine failed due to application error (for example, bad parameters), additional result data is not available.
CS_END_ITEM	ct_get_data successfully retrieved the last chunk of data for this column. This is not the last column in the row.
CS_END_DATA	ct_get_data successfully retrieved the last chunk of data for this column. This is the last column in the row.
CS_CANCELED	The operation was canceled. Data for this result set is no longer available.
CS_PENDING	Asynchronous network I/O is in effect. See “Asynchronous programming” on page 12.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

```
/*
 ** FetchResults()
 **
 ** The result set contains four columns: integer, text,
 ** float, and integer.
 */

CS_STATIC CS_RETCODE
 FetchResults(cmd, textdata)
```

```
CS_COMMAND   *cmd;
TEXT_DATA    *textdata;
{
    CS_RETCODE   retcode;
    CS_DATAFMT   fmt;
    CS_INT       firstcol;
    CS_TEXT      *txtptr;
    CS_FLOAT     floatitem;
    CS_INT       count;
    CS_INT       len;

   /*
    ** All binds must be of columns prior to the columns
    ** to be retrieved by ct_get_data().
    ** To demonstrate this, bind the first column returned.
    */
    ...CODE DELETED.....

   /* Retrieve and display the results */
    while(((retcode = ct_fetch(cmd, CS_UNUSED, CS_UNUSED,
        CS_UNUSED,&count)) == CS_SUCCEED) ||
        (retcode == CS_ROW_FAIL) )
    {
        /* Check for a recoverable error */
        ...CODE DELETED.....

       /*
        ** Get the text data item in the second column.
        ** Loop until we have all the data for this item.
        ** The text used for this example could be
        ** retrieved in one ct_get_data call, but data
        ** could be too large for this to be the case.
        ** Instead, the data would have to be retrieved
        ** in chunks. This example will retrieve the text
        ** in 5 byte increments to demonstrate retrieving
        ** data items in chunks.
        */
        txtptr = textdata->textbuf;
        textdata->textlen = 0;
        do
        {
            retcode = ct_get_data(cmd, 2, txtptr, 5,
                &len);
            textdata->textlen += len;
            /*
            ** Protect against overflowing the string
            ** buffer.
            */
```

```
            if ((textdata->textlen + 5) > (EX_MAX_TEXT -
                1))
            {
                break;
            }
            txtptr += len;
        } while (retcode == CS_SUCCEED);

       if (retcode != CS_END_ITEM)
        {
            ex_error("FetchResults: ct_get_data()
                failed");
            return retcode;
        }

       /*
        ** Retrieve the descriptor of the text data. It is
        ** available while retrieving results of a select
        ** query. The information will be needed for
        ** later updates.
        */
         ...CODE DELETED....

       /* Get the float data item in the 3rd column */
        retcode = ct_get_data(cmd, 3, &floatitem,
             sizeof (floatitem), &len);
        if (retcode != CS_END_ITEM)
        {
             ex_error("FetchResults: ct_get_data()
                 failed");
             return(retcode);
        }

       /*
        ** When using ct_get_data to process results,
        ** it is not required to get all the columns
        ** in the row. To illustrate this, the last
        ** column of the result set is not retrieved.
        */
     }

    /*
     ** We're done processing rows. Check the
     ** final return value of ct_fetch().
     */
       ...CODE DELETED.....

    return retcode;
}
```

This code excerpt is from the *getsend.c* sample program.

Usage

- An application typically calls ct_get_data in a loop to retrieve large text or image values, although it can be used on columns of any datatype. Each call to ct_get_data retrieves a *buflen*-sized chunk of data.
- For information about the steps involved in using ct_get_data to retrieve a text or image value, see "Using ct_get_data to fetch text and image values" on page 285.
- ct_get_data retrieves data exactly as it is sent by the server. No conversion is performed. For this reason, care must be taken when interpreting data contained in **buffer*. In particular, CS_CHAR data may not be null-terminated and multibyte character strings may be broken within a byte sequence defining a single character.
- An application calls ct_get_data after calling ct_fetch to fetch the row of interest. If array binding was indicated in an earlier call to ct_bind, the application cannot use ct_get_data.
- Only those columns following the last bound column are available to ct_get_data. Data in unbound columns that precede bound columns is discarded. For example, if an application selects column numbers 1–4 and binds column numbers 1 and 3, the application cannot use ct_get_data to retrieve the data for column 2, but can use ct_get_data to retrieve the data for column 4.
- Once data has been retrieved for a column, it is no longer available.
- If an application reads a text or image column that it will need to update at a later time, it needs to retrieve an I/O descriptor for the column. To do this, an application can call ct_data_info after calling ct_get_data for the column.
- If a column value is null, ct_get_data sets **outlen* to 0 and returns CS_END_ITEM or CS_END_DATA.
- An application cannot retrieve an I/O descriptor for a column before it has called ct_get_data for the column. However, this ct_get_data call does not have to actually retrieve any data. That is, an application can call ct_get_data with a buflen of 0, and then call ct_data_info to retrieve the descriptor. This technique is useful when an application needs to determine the length of a text or image value before retrieving it.

See also

ct_bind, ct_data_info, ct_fetch, ct_send_data, text and image data handling

ct_getformat

Description

Return the server user-defined format string associated with a result column.

Syntax

```
CS_RETCODE ct_getformat (cmd, colnum, buffer, buflen,
        outlen)

CS_COMMAND      *cmd;
 CS_INT         colnum;
 CS_VOID        *buffer;
 CS_INT         buflen;
 CS_INT         *outlen;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

colnum

The number of the column whose user-defined format is desired. The first column in a select list is column number 1, the second is number 2, and so forth.

buffer

A pointer to the space in which ct_getformat will place a null-terminated format string.

buflen

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

outlen

A pointer to an integer variable.

If *outlen* is supplied, ct_getformat sets **outlen* to the length, in bytes, of the format string. This length includes the null terminator.

If the format 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 no format string is associated with the column identified by *colnum*, ct_getformat sets **outlen* to 1 (for the null terminator).

Return value

ct_getformat returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Usage

- An application can call ct_getformat after ct_results indicates results of type CS_ROW_RESULT.
- If no format string is associated with the column identified by *colnum*, ct_getformat sets **outlen* to 1.
- Typical applications will not use ct_getformat, which is provided primarily for gateway applications support.

See also

ct_bind, ct_describe

ct_getloginfo

Description

Transfer TDS login response information from a CS_CONNECTION structure to a newly allocated CS_LOGINFO structure.

Syntax

CS_RETCODE ct_getloginfo (connection, logptr)

```
CS_CONNECTION    *connection;
CS_LOGINFO       **logptr;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

logptr

A pointer to a program variable which ct_getloginfo sets to the address of a newly allocated CS_LOGINFO structure.

Return value

ct_getloginfo returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Usage

- TDS (Tabular Data Stream) is a communications protocol used for the transfer of requests and request results between clients and servers.
- There are two reasons an application might call ct_getloginfo:
  - If it is an Open Server gateway application using TDS passthrough.

- To copy login properties from an open connection to a newly allocated connection structure.

Note Do not call ct_getloginfo from within a completion callback routine. ct_getloginfo calls system-level memory functions that may not be reentrant.

TDS passthrough

- When a client connects directly to a server, the two programs negotiate the TDS format they will use to send and receive data. When a gateway application uses TDS passthrough, the gateway forwards TDS packets between the client and a remote server without examining or processing them. For this reason, the remote server and the client must agree on a TDS format to use.
- ct_getloginfo is the third of four calls, two of them Server Library calls, that allow a client and a remote server to negotiate a TDS format. The calls, which can be made only in an Open Server SRV_CONNECT event handler, are:

  a srv_getloginfo to allocate a CS_LOGINFO structure and fill it with TDS information from a client login request.

  b ct_setloginfo to transfer the TDS information retrieved in step 1 from the CS_LOGINFO structure to a Client-Library CS_CONNECTION structure. The gateway uses this CS_CONNECTION structure in the ct_connect call which establishes its connection with the remote server.

  c ct_getloginfo to transfer the remote server's response to the client's TDS information from the CS_CONNECTION structure into a newly allocated CS_LOGINFO structure.

  d srv_setloginfo to send the remote server's response, retrieved in step c, to the client.

Copying login properties

For information about using ct_getloginfo to copy login properties from an open connection to a newly allocated connection structure, see "Properties" on page 180.

See also

ct_recvpassthru, ct_sendpassthru, ct_setloginfo

ct_init

Description

Initialize Client-Library for an application context.

Syntax

```
CS_RETCODE ct_init(context, version)

 CS_CONTEXT    *context;
 CS_INT         version;
```

Parameters

context

A pointer to a CS_CONTEXT structure. An application must have previously allocated this context structure by calling the CS-Library routine cs_ctx_alloc.

context identifies the Client-Library context being initialized.

version

The version of Client-Library behavior that the application expects. Table 3-44 lists the symbolic values for *version*:

Table 3-44: Values for ct_init version parameter

Value of version	Meaning	Features supported
CS_VERSION_100	10.0 behavior.	Cursors, registered procedures, remote procedure calls. This is the initial version of Client-Library.
CS_VERSION_110	11.0 behavior.	All 10.0 features plus these new version 11.1 features: • Network-based directory and security services. • External configuration of properties, options, and capabilities.
CS_VERSION_120	12.0 behavior	All previous features plus: • High-availability failover • Native thread support for Digital UNIX platforms • Bulk-row inserts • A new property for enabling/disabling sort-merge joins
CS_VERSION_125	12.5 behavior	Added features for version 12.5 include: • LDAP security features • SSL security features • Unichar-16 for 2-byte character support • support for wide columns and wide tables.
CS_VERSION_150	15.0 behavior	BCP partitions, BCP computed columns, large identifiers, Unilib, ASE default packet size, scrollable cursors, and clusters support. Also support for unitext, xml, bigint, usmallint, uint, and ubigint datatypes. Note Sybase library name change.

Return value

ct_init returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_MEM_ERROR	The routine failed due to a memory allocation error.
CS_FAIL	The routine failed for other reasons.

ct_init returns CS_FAIL if Client-Library cannot provide *version*-level behavior.

Note When ct_init returns CS_FAIL due to a Net-Library error, extended error information is sent to standard error (STDERR) and to the sybinit.err file that is created in the current working directory.

A ct_init failure does not typically make **context* unusable. Instead of dropping the context structure, an application can try calling ct_init again with the same *context* pointer.

Examples

```
/*
 ** ex_init() -- Allocate and initialize a CS_CONTEXT
 **    structure.
 **
 ** EX_CTLIB_VERSION is defined in the examples header file
 ** as CS_VERSION_110.
 */

 CS_RETCODE CS_PUBLIC
  ex_init(context)
  CS_CONTEXT  **context;
  {
      CS_RETCODE   retcode;

     /* Get a context handle to use */
      retcode = cs_ctx_alloc(EX_CTLIB_VERSION, context);
           ... error checking code deleted ...

     /* Initialize Open Client */
      retcode = ct_init(*context, EX_CTLIB_VERSION);
      if (retcode != CS_SUCCEED)
      {
          ex_error("ex_init: ct_init() failed");
          cs_ctx_drop(*context);
          *context = NULL;
          return retcode;
      }

     /* Install client and server message handlers */
      ... ct_callback calls deleted .....

     /* Call ct_config to set context properties */
      ... ct_config calls deleted ...

     /* Exit from Client-Library */
```

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

       return retcode;
    }
```

This code excerpt is from the *exutils.c* sample program.

Usage

- ct_init sets up internal control structures and defines the version of Client-Library behavior that the application expects.
- ct_init must be the first Client-Library routine called in a Client-Library application context. Other Client-Library routines will fail if they are called before ct_init.

Note A Client-Library application can call CS-Library routines before calling ct_init (and, in fact, must call the CS-Library routine cs_ctx_alloc before calling ct_init).

- If ct_init returns CS_SUCCEED, Client-Library will provide the requested behavior, regardless of the actual version of Client-Library in use. If Client-Library cannot provide the requested behavior, ct_init returns CS_FAIL. Generally speaking, higher-level versions of Client-Library can provide lower-level behavior, but lower-level versions cannot provide higher-level behavior.
- Because an application calls ct_init before it sets up error handling, an application must check ct_init's return code to detect failure.
- It is not an error for an application to call ct_init multiple times for the same context. If this occurs, only the first call has any effect. Client-Library provides this functionality because some applications cannot guarantee which of several modules will execute first. In such a case, each module needs to contain a call to ct_init.
- *version* is the version of Client-Library behavior that the application expects. *version* determines the value of the context's CS_VERSION property. Connections allocated within a context use default CS_TDS_VERSION values based on their parent context's CS_VERSION level.

Configuring context properties externally

- Client-Library reads the Open Client and Open Server configuration file to get default context property values if the application requests external configuration by calling cs_config to set the CS_CONFIG_FILE context property before calling ct_init.
- External configuration can eliminate several ct_config calls in an application. Also, if an application is coded to request external configuration, it allows the application's runtime property settings to be changed without recompiling. For more information on this feature, see "Using the runtime configuration file" on page 305.

See also

cs_ctx_alloc, ct_exit, ct_config

ct_keydata

Description

Specify or extract the contents of a key column.

Syntax

```
CS_RETCODE ct_keydata (cmd, action, colnum, buffer,
        buflen, outlen)

CS_COMMAND    *cmd;
CS_INT        action;
CS_INT        colnum;
CS_VOID       *buffer;
CS_INT        buflen;
CS_INT        *outlen;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server cursor operation.

action

One of the following symbolic values:

Value of action	Result
CS_SET	Sets the contents of the key column
CS_GET	Retrieves the contents of the key column

colnum

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

colnum must represent a CS_KEY or CS_VERSION_KEY column. ct_describe sets its *datafmt–>status* field to indicate whether or not a column is a CS_KEY or CS_VERSION_KEY column.

buffer

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

If a key column value is being retrieved, *buffer* points to the space in which ct_keydata will place the requested information.

buflen

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

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

If a key column value is being retrieved and *buflen* indicates that **buffer* is not large enough to hold the requested information, ct_keydata sets **outlen* to the length of the requested information and returns CS_FAIL.

buflen is required even for fixed-length buffers, and cannot be passed as CS_UNUSED.

outlen

A pointer to an integer variable.

If a key column value is being set, *outlen* is unused and must be passed as NULL.

If a key column value is being retrieved, ct_keydata 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.

If an application is setting a key column value or does not care about return length information, it can pass *outlen* as NULL.

Return value

ct_keydata returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Return value	Meaning
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

ct_keydata returns CS_FAIL if *colnum* does not represent a key column.

Usage

- An application can use ct_keydata to redefine the current cursor position before performing a cursor update or delete.
- ct_keydata has two primary uses:
  - In gateway applications that buffer cursor rows between a client and a server. In this case, the client’s notion of cursor position can differ from the gateway’s. If the client sends a positioned update or delete request, the gateway can use ct_keydata to correctly identify the target row to the server.
  - In applications that allow users to browse through data rows, altering or deleting them in random order. In this case, a user may ask the application to alter or delete a row that is not the current cursor row. The application can use ct_keydata to redefine the target row as the current row.
- Because a key can span multiple columns, an application may need to call ct_keydata multiple times to specify a row’s entire key.
- Calling ct_fetch wipes out any key column values that an application has specified.
- An application can call ct_keydata only under the following circumstances:
  - The current result type is CS_CURSOR_RESULT.
  - The command structure which is supporting the cursor has CS_HIDDEN_KEYS property set to CS_TRUE.
  - The cursor has been fetched at least once.
- When updating a key, all key columns must be updated. If a positioned update or delete is attempted when the row’s entire key has not been redefined, ct_cursor returns CS_FAIL.
- An application can set a key column’s value to NULL by calling ct_keydata with *buffer* as NULL and *buflen* as 0 or CS_UNUSED. If the column does not allow null values, ct_keydata returns CS_FAIL.

See also

ct_cursor, ct_describe, ct_res_info, ct_results

ct_labels

Description

Define a security label or clear security labels for a connection.

Syntax

```
CS_RETCODE ct_labels(connection, action,
            labelname, namelen, labelvalue,
            valuelen, outlen)

CS_CONNECTION    *connection;
CS_INT           action;
CS_CHAR          *labelname;
CS_INT           namelen;
CS_CHAR          *labelvalue;
CS_INT           valuelen;
CS_INT           *outlen;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

**connection* must represent a closed connection.

action

One of the following symbolic values:

Value of action	Result
CS_SET	Sets a security label
CS_CLEAR	Clears all security labels previously specified for this connection

labelname

If *action* is CS_SET, *labelname* points to the name of the security label being set.

If *action* is CS_CLEAR, *labelname* must be NULL.

namelen

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

Security label names must be at least 1 byte long and no more than CS_MAX_NAME bytes long.

If *action* is CS_CLEAR, pass *namelen* as CS_UNUSED.

labelvalue

If *action* is CS_SET, *labelvalue* points to the value of the security label being set.

If *action* is CS_CLEAR, *labelvalue* must be NULL.

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

Security label values must be at least 1-byte long.

If *action* is CS_CLEAR, pass *valuelen* as CS_UNUSED.

outlen
This parameter is currently unused and must be passed as NULL.

Return value

ct_labels returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Usage

- An application needs to define security labels if it will be connecting to a server that uses trusted-user security handshakes.
- There are two ways for an application to define security labels. An application can use either, or both, of these methods:
  - The application can call ct_labels one time for each label it wants to define.
  - The application can call ct_callback to install a user-supplied negotiation callback to generate security labels. At connection time, Client-Library automatically triggers the callback in response to a request for security labels.

  If an application uses both methods, the labels defined using ct_labels and the labels generated by the negotiation callback are sent to the server at the same time.
- A connection that will be participating in trusted-user security handshakes must set the CS_SEC_NEGOTIATE property to CS_TRUE.
- There is no limit on the number of security labels that can be defined for a connection.
- ct_labels does not perform any type of checking on security labels, but simply passes the label name and label value combinations on to the server.

  For example, ct_labels does not raise an error if an application supplies two label values for the same label name.

See also ct_callback, ct_con_props, ct_connect

ct_options

Description Set, retrieve, or clear the values of server query-processing options.

Syntax

```
CS_RETCODE ct_options(connection, action, option,
                      param, paramlen, outlen)

CS_CONNECTION   *connection;
CS_INT          action;
CS_INT          option;
CS_VOID         *param;
CS_INT          paramlen;
CS_INT          *outlen;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

connection is the server connection for which the option is set, retrieved, or cleared.

action

One of the following symbolic values:

Value of action	Result
CS_SET	Sets the option.
CS_GET	Retrieves the option.
CS_CLEAR	Clears the option by resetting it to its default value. Default values are determined by the server to which an application is connected.

option

The server option of interest. Table 3-45 on page 525 lists the symbolic values for *option*. For more information about these options, see “Options” on page 174.

param

All options take parameters.

When setting an option, *param* can point to a symbolic value, a Boolean value, an integer value, or a character string.

For example:

- The CS_OPT_DATEFIRST option takes a symbolic value as a parameter:

```
CS_INT       parmvalue;
 parmamvalue = CS_OPT_TUESDAY;
 ct_options(conn, CS_SET, CS_OPT_DATEFIRST,
      &paramvalue, CS_UNUSED, NULL);
```

- The CS_OPT_CHAINXACTS option takes a Boolean value as a parameter:

```
CS_BOOL    parmvalue;
 parmamvalue = CS_TRUE;
 ct_options(conn, CS_SET, CS_OPT_CHAINXACTS,
      &paramvalue, CS_UNUSED, NULL);
```

- The CS_OPT_ROWCOUNT option takes an integer as a parameter:

```
CS_INT       parmvalue;
 paramvalue = 50;
 oc_options(conn, CS_SET, CS_OPT_ROWCOUNT,
      &paramvalue, CS_UNUSED, NULL);
```

- The CS_OPT_IDENTITYOFF option takes a character string as a parameter:

```
ct_options(conn, CS_SET, CS_OPT_IDENTITYOFF,
      "authors", CS_NULLTERM, NULL);
```

When retrieving an option, *param* points to the space in which ct_options places the value of the option.

If *paramlen* indicates that **param* is not large enough to hold the option's value, ct_option sets **outlen* to the length of the value and returns CS_FAIL.

When clearing an option, *param* must be NULL.

paramlen

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

When setting or retrieving an option that takes a fixed-length parameter, pass *paramlen* as CS_UNUSED.

When setting an option that takes a character string parameter, if the value in **param* is null-terminated, pass *paramlen* as CS_NULLTERM.

When retrieving an option, if *paramlen* indicates that **param* is not large enough to hold the requested information, ct_options sets **outlen* to the length of the requested information and returns CS_FAIL.

When clearing an option, *paramlen* must be CS_UNUSED.

outlen

A pointer to an integer variable.

If an option is being set or cleared, *outlen* is not used and must be passed as NULL.

If an option is being retrieved, ct_options sets **outlen* to the length, in bytes, of the option's value. This length includes a null terminator, if applicable.

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

Return value

ct_options returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed. If ct_options returns CS_FAIL, **param* remains untouched.
CS_CANCELED	The operation was canceled.
CS_PENDING	Asynchronous network I/O is in effect. See "Asynchronous programming" on page 12.
CS_BUSY	An asynchronous operation is already pending for this connection. See "Asynchronous programming" on page 12.

Usage

Table 3-45: Summary of ct_options parameters

Value of option	Value of *param	Legal Values for *param	Default
CS_OPT_ANSINULL	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_ANSIPERM	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_ARITHABORT	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_ARITHIGNORE	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE

Value of option	Value of *param	Legal Values for *param	Default
CS_OPT_AUTHOFF	A string value representing an authority level.	A string value. Possible values include “sa”, “sso”, and “oper.”	Not applicable
CS_OPT_AUTHON	A string value representing an authority level.	A string value. Possible values include “sa”, “sso”, and “oper.”	Not applicable
CS_OPT_CHAINXACTS	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_CHARSET	The name of a language that is supported and found on the *locale.dat* file. Used to set the language or character set on an open connection.	A value representing a language, for your platform, on the *locales.dat* file.	NULL
CS_OPT_CURCLOSEONXACT	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_DATEFIRST	A symbolic value representing the day to use as the first day of the week.	CS_OPT_SUNDAY, CS_OPT_MONDAY, CS_OPT_TUESDAY, CS_OPT_WEDNESDAY, CS_OPT_THURSDAY, CS_OPT_FRIDAY, CS_OPT_SATURDAY	For us_english, the default is CS_OPT_SUNDAY.
CS_OPT_DATEFORMAT	A symbolic value representing the order of year, month, and day to be used in datetime values.	CS_OPT_FMTMDY, CS_OPT_FMTDMY, CS_OPT_FMTYMD, CS_OPT_FMTYDM, CS_OPT_FMTMYD, CS_OPT_FMTDYM	For us_english, the default is CS_OPT_FMTMDY.
CS_OPT_FIPSFLAG	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_FORCEPLAN	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_FORMATONLY	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_GETDATA	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_HIDE_VCC	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_IDENTITYOFF	A string value representing a table name.	A string value.	NULL
CS_OPT_IDENTITYON	A string value representing a table name.	A string value.	NULL
CS_OPT_IDENTITYUPD_OFF	Disable the identity update option.	A string value.	NULL
CS_OPT_IDENTITYUPD_ON	Enable the identity update option.	A string value.	NULL

Value of option	Value of *param	Legal Values for *param	Default
CS_OPT_ISOLATION	A symbolic value representing the transaction isolation level.	CS_OPT_LEVEL1, CS_OPT_LEVEL0, CS_OPT_LEVEL3 CS_OPT_LEVEL0 requires Adaptive Server version 11.0 or later.	CS_OPT_ LEVEL1
CS_OPT_NATLANG	The name of a language that is supported and found on the *locale.dat* file. Used to set the language or character set on an open connection.	A value representing a language, for your platform, on the *locales.dat* file.	NULL
CS_OPT_NOCOUNT	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_NOEXEC	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_PARSEONLY	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_QUOTED_ IDENT	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_RESTREES	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_ROWCOUNT	The maximum number of rows that can be affected by a query. Limits the number of regular rows returned by a select or the number of rows changed by an update or delete.	An integer value. 0 means there is no limit.	0, no limit
CS_OPT_SHOW_FI	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_SHOWPLAN	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_STATS_IO	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_STATS_TIME	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_STR_RTRUNC	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_TEXTSIZE	The length, in bytes, of the longest text or image value the server should return.	An integer value.	32,768 bytes
CS_OPT_TRUNCIGNORE	A Boolean value.	CS_TRUE, CS_FALSE	CS_FALSE

- Although query-processing options can be set and cleared through the Transact-SQL set command, it is recommended that Client-Library applications use ct_options instead. This is because ct_options allows an application to check the status of an option, which cannot be done through the set command.

- An application can use ct_options to change server options only for a single connection at a time. The connection must be open and must have no active commands or pending results, but can have an open cursor.
- The routine ct_connect optionally reads a section from the Open Client and Open Server runtime configuration file to set server options for a newly opened connection. For a description of this feature, see “Using the runtime configuration file” on page 305.

See also

ct_capability, ct_con_props, “Options” on page 174

ct_param

Description

Supplies values for a server command’s input parameters.

Syntax

```
CS_RETCODE ct_param(cmd, datafmt, data, datalen, indicator);

CS_COMMAND      *cmd;
CS_DATAFMT      *datafmt;
CS_VOID         *data;
CS_INT          datalen;
CS_SMALLINT     indicator;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

datafmt

A pointer to a CS_DATAFMT structure that describes the parameter.

For information about how to set these fields for specific uses of ct_param, see “Usage” on page 476.

data

The address of the parameter data.

There are two ways to indicate a parameter with a null value:

- Pass *indicator* as -1. In this case, *data* and *datalen* are ignored.
- Pass *data* as NULL and *datalen* as 0 or CS_UNUSED.

datalen
The length, in bytes, of the parameter data.

If *datafmt–>datatype* indicates that the parameter is a fixed-length type, *datalen* is ignored. CS_VARBINARY and CS_VARCHAR are considered to be fixed-length types.

indicator
An integer variable used to indicate a parameter with a null value. To indicate a parameter with a null value, pass *indicator* as -1. If *indicator* is -1, *data* and *datalen* are ignored.

Return value

ct_param returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

This code excerpt is from the *rpc.c* sample program.

```
/*
 ** BuildRpcCommand()
 **
 ** Purpose:
 **     Builds an RPC command but does not send it.
 **
 */

CS_STATIC CS_RETCODE
 BuildRpcCommand(cmd)
 CS_COMMAND     *cmd;
 {
      CS_CONNECTION   *connection;
      CS_CONTEXT      *context;
      CS_RETCODE      retcode;
      CS_DATAFMT      datafmt;
      CS_DATAFMT      srcfmt;
      CS_DATAFMT      destfmt;
      CS_INT          intvar;
      CS_SMALLINT     smallintvar;
      CS_FLOAT        floatvar;
      CS_MONEY        moneyvar;
      CS_BINARY       binaryvar;
      char            moneystring[10];
      char            rpc_name[15];
```

```
 CS_INT           destlen;

/*
 ** Assign values to the variables used for
 ** parameter passing.
 */
 intvar = 2;
 smallintvar = 234;
 floatvar = 0.12;
 binaryvar = (CS_BINARY)0xff;
 strcpy(rpc_name, "sample_rpc");
 strcpy(moneystring, "300.90");
 /*
 ** Clear and setup the CS_DATAFMT structures used
 ** to convert datatypes.
 */
 memset(&srcfmt, 0, sizeof (CS_DATAFMT));
 srcfmt.datatype = CS_CHAR_TYPE;
 srcfmt.maxlength = strlen(moneystring);
 srcfmt.precision = 5;
 srcfmt.scale = 2;
 srcfmt.locale = NULL;

memset(&destfmt, 0, sizeof (CS_DATAFMT));
 destfmt.datatype = CS_MONEY_TYPE;
 destfmt.maxlength = sizeof(CS_MONEY);
 destfmt.precision = 5;
 destfmt.scale = 2;
 destfmt.locale = NULL;

/*
 ** Convert the string representing the money value
 ** to a CS_MONEY variable. Since this routine
 ** does not have the context handle, we use the
 ** property functions to get it.
 */
 if ((retcode = ct_cmd_props(cmd, CS_GET,
      CS_PARENT_HANDLE, &connection, CS_UNUSED,
      NULL)) != CS_SUCCEED)
         ...error checking deleted ...
 if ((retcode = ct_con_props(connection, CS_GET,
      CS_PARENT_HANDLE, &context, CS_UNUSED,
      NULL)) != CS_SUCCEED)
        ...error checking deleted ...

retcode = cs_convert(context, &srcfmt,
      (CS_VOID *)moneystring, &destfmt, &moneyvar,
      &destlen);
```

```
 if (retcode != CS_SUCCEED)
        ...error checking deleted ...
 /*
 ** Initiate the RPC command for our stored
 ** procedure.
 */
 if ((retcode = (cmd, CS_RPC_CMD,
      rpc_name, CS_NULLTERM, CS_NO_RECOMPILE)) !=
      CS_SUCCEED)
          ...error checking deleted ...

/*
 ** Clear and set up the CS_DATAFMT structure, then
 ** pass each of the parameters for the RPC.
 */
 memset(&datafmt, 0, sizeof (datafmt));
 strcpy(datafmt.name, "@intparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_INT_TYPE;
 datafmt.maxlength = CS_UNUSED;
 datafmt.status = CS_INPUTVALUE;
 datafmt.locale = NULL;

if ((retcode = ct_param(cmd, &datafmt,
      (CS_VOID *)&intvar, sizeof(CS_INT),0))
     != CS_SUCCEED)
       ...error checking deleted ...

strcpy(datafmt.name, "@sintparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_SMALLINT_TYPE;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 if ((retcode = ct_param(cmd, &datafmt,
      (CS_VOID *)&smallintvar,
      sizeof(CS_SMALLINT), 0))
     != CS_SUCCEED)
 ...error checking deleted ...

strcpy(datafmt.name, "@floatparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_FLOAT_TYPE;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 if((retcode = ct_param(cmd, &datafmt,
      (CS_VOID *)&floatvar,sizeof(CS_FLOAT),0))
     != CS_SUCCEED)
        ...error checking deleted ...
```

```
strcpy(datafmt.name, "@moneyparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_MONEY_TYPE;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 if((retcode = ct_param(cmd, &datafmt,
      (CS_VOID *)&moneyvar, sizeof(CS_MONEY),0))
     != CS_SUCCEED)
     ...error checking deleted ...

 strcpy(datafmt.name, "@dateparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_DATETIME4_TYPE;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 /*
 ** The datetime variable is filled in by the RPC
 ** so pass NULL for the data, 0 for data length,
 ** and -l for the indicator arguments.
 */
 if((retcode = ct_param(cmd, &datafmt, NULL, 0,
      -1)) != CS_SUCCEED)
            ...error checking deleted ...

strcpy(datafmt.name, "@charparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_CHAR_TYPE;
 datafmt.maxlength = EX_MAXSTRINGLEN;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 /*
 ** The character string variable is filled in by
 ** the RPC so pass NULL for the data 0 for data
 ** length, and -l for the indicator arguments.
 */
 if((retcode = ct_param(cmd, &datafmt, NULL, 0,
      -1)) != CS_SUCCEED)
         ...error checking deleted ...

strcpy(datafmt.name, "@binaryparam");
 datafmt.namelen = CS_NULLTERM;
 datafmt.datatype = CS_BINARY_TYPE;
 datafmt.maxlength = EX_MAXSTRINGLEN;
 datafmt.status = CS_RETURN;
 datafmt.locale = NULL;
 if((retcode = ct_param(cmd, &datafmt,
      (CS_VOID *)&binaryvar,
```

```
                    sizeof(CS_BINARY), 0))
                   != CS_SUCCEED)
                     ...error checking deleted ...

            return retcode;
        }
```

Usage

Table 3-46 summarizes ct_param usage.

Table 3-46: Summary of ct_param parameters

Type of command	Purpose of ct_param call	datafmt->status is	*data, datalen are
Cursor declare	To identify update columns	CS_UPDATECOL	The name of the update column and the name's length
Cursor declare	To define host variable formats	CS_INPUTVALUE	NULL and CS_UNUSED
Cursor open	To pass parameter values	CS_INPUTVALUE	The parameter value and length
Cursor update	To pass parameter values	CS_INPUTVALUE	The parameter value and length
Dynamic SQL execute	To pass parameter values	CS_INPUTVALUE	The parameter value and length
Language	To pass parameter values	CS_INPUTVALUE	The parameter value and length
Message	To pass parameter values	CS_INPUTVALUE	The parameter value and length
RPC	To pass parameter values	CS_RETURN to pass a return parameter; CS_INPUTVALUE to pass a non-return parameter.	The parameter value and length

- ct_param supplies parameter values for an initiated command.
- Initiating a command is the first step in executing it. Some commands require the application to define input parameters with ct_param or ct_setparam before calling ct_send to send the command to the server. For a description of this feature, see "Resending commands" on page 580.
- ct_setparam and ct_param perform the same function, with the following exceptions:
  - ct_param copies the contents of program variables.

- ct_setparam copies the address of program variables, and subsequent calls to ct_send read the contents of the variables. ct_setparam allows the application to change parameter values when resending a command.

Calls to ct_param and ct_setparam can be mixed.

- An application may need to call ct_param:
  - To identify update columns for a cursor declare command.
  - To define host variable formats for a cursor declare command.
  - To pass input parameter values for a cursor open, cursor update, dynamic SQL execute, language, message, or RPC command.

  An application calls ct_command to initiate a language, RPC or message command, calls ct_cursor to initiate a cursor declare or cursor open command, and calls ct_dynamic to initiate a Dynamic SQL execute command.

  For specific information about these uses, see the following sections:

  - "Passing input parameter values" on page 536
  - "Defining host variable formats" on page 535
  - "Identifying update columns for a cursor declare command" on page 534
- Client-Library does not perform any conversion on parameters before passing them to the server. The application must supply parameters in the datatype required by the server. If necessary, the application can call cs_convert to convert parameter values into the required datatype.

Identifying update columns for a cursor declare command

- Some servers require a client application to identify update columns for a cursor declare command if the cursor is updatable, but not all of the columns are "for update." Update columns can be used to change values in underlying database tables.

- Adaptive Server does not require the application to specify update columns with additional ct_param/ct_setparam calls as described in this section. In fact, Adaptive Server ignores requests to identify update columns as described here. The application must use the Transact-SQL for read only or for update of syntax in the select statement to specify which columns are updatable (see the Adaptive Server Enterprise for a description of this syntax). Depending on its design, an Open Server application may require clients to specify a cursor's update columns as described in this section.
- If all of the cursor's columns are "for update," an application does not need to call ct_param to specify them individually.
- To identify an update column for a cursor declare command, an application calls ct_param with *datafmt–>status* as CS_UPDATECOL and **data* as the name of the column.
- The following table lists the fields in **datafmt* that are used when identifying update columns for a cursor declare command:

Table 3-47: CS_DATAFMT fields for identifying update columns

Field name	Set to
status	CS_UPDATECOL

All other fields are ignored.

Defining host variable formats

- An application needs to define host variable formats for cursor declare commands when the text of the cursor being declared is a SQL string that contains host variables.
- To define the format of a host variable, an application calls ct_param with *datafmt–>status* as CS_INPUTVALUE, *datafmt–>datatype* as the datatype of the host variable, *data* as NULL and *datalen* as CS_UNUSED.
- An application defines host variable formats during a cursor declare command but does not pass data values for the variables until cursor open time.
- When defining host variable formats, the variables can either be named or unnamed. If one variable is named, all variables must be named. If variables are not named, they are interpreted positionally.
- The following table lists the fields in **datafmt* that are used when defining host variable formats:

Table 3-48: CS_DATAFMT fields for defining host variable formats

Name	Set To
name	The name of the host variable.
namelen	The length, in bytes, of *name*, or 0 to indicate an unnamed parameter.
datatype	The datatype of the host variable. All standard Client-Library types are valid except for CS_TEXT_TYPE, CS_UNITEXT_TYPE, CS_IMAGE_TYPE, CS_XML_TYPE, and Client-Library user-defined types. If *datatype* is CS_VARCHAR_TYPE or CS_VARBINARY_TYPE then *data* must point to a CS_VARCHAR or CS_VARBINARY structure.
status	CS_INPUTVALUE

All other fields are ignored.

Passing input parameter values

- An application may need to pass input parameter values for:
  - Client-Library cursor open commands
  - Client-Library cursor update commands
  - Dynamic SQL execute commands
  - Language commands
  - Message commands
  - Package commands
  - RPC commands
- When passing input parameter values, parameters can either be named or unnamed. If one parameter is named, all parameters must be named. If parameters are not named, they are interpreted positionally.
- In some cases, an application may need to pass a parameter that has a null value. For example, an application might pass parameters with null values to a stored procedure that assigns default values to null input parameters.

  There are two ways to indicate a parameter with a null value:
  - Pass *indicator* as -1. ct_param ignores *data* and *datalen*.
  - Pass *data* as NULL and *datalen* as 0 or CS_UNUSED.

- Client-Library cursor open commands require input parameter values when:
  - The body of the cursor is a SQL text string containing host variables.
  - The body of the cursor is a stored procedure that requires parameters. In this case, *datafmt–>status* should be CS_INPUTVALUE.
  - The cursor is declared on a prepared dynamic SQL statement that contains placeholders (indicated by the ? character).
- Client-Library cursor update commands require input parameter values when the SQL text representing the update command contains host variables.
- Dynamic SQL execute commands require input parameter values when the prepared statement being executed contains dynamic parameter markers.
- Language commands require input parameter values when the text of the language command contains host variables.
- Message commands require input parameters values when the message takes parameters.
- RPC and package commands require input parameter values when the stored procedure or package being executed takes parameters.
- Message, package, and RPC commands can take return parameters, indicated by passing *datafmt–>status* as CS_RETURN.
- A command that takes return parameters may generate a parameter result set that contains the return parameter values. See ct_results for a description of how an application retrieves values from a parameter result set.
- The following table lists the fields in **datafmt* that are used when passing input parameter values:

Table 3-49: CS_DATAFMT fields for passing input parameter values

Name	Set to
name	The name of the parameter. *name* is ignored for dynamic SQL execute commands.
namelen	The length, in bytes, of *name*, or 0 to indicate an unnamed parameter. *namelen* is ignored for dynamic SQL execute commands.
datatype	The datatype of the input parameter value. All standard Client-Library types are valid except for CS_TEXT_TYPE, CS_UNITEXT_TYPE, CS_IMAGE_TYPE, CS_XML_TYPE, and Client-Library user-defined types. If *datatype* is CS_VARCHAR_TYPE or CS_VARBINARY_TYPE then *data* must point to a CS_VARCHAR or CS_VARBINARY structure.
maxlength	When passing return parameters for RPC commands, *maxlength* represents the maximum length, in bytes, of data to be returned for this parameter. *maxlength* is not used when passing input parameter values for other types of commands.
status	CS_RETURN when passing return parameters for RPC commands; otherwise CS_INPUTVALUE.

All other fields are ignored.

See also

ct_command, ct_cursor, ct_dynamic, ct_send, ct_setparam

ct_poll

Description

Poll connections for asynchronous operation completions and registered procedure notifications.

Syntax

```
CS_RETCODE ct_poll (context, connection, milliseconds, compconn,
                    compcmd, compid, compstatus)

CS_CONTEXT        *context;
CS_CONNECTION     *connection;
CS_INT            milliseconds;
CS_CONNECTION     **compconn;
CS_COMMAND        **compcmd;
```

```
CS_INT          *compid;
CS_RETCODE      *compstatus;
```

Parameters

context

A pointer to a CS_CONTEXT structure.

Either *context* or *connection* must be NULL. If *context* is NULL, ct_poll checks only a single connection.

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

Either *context* or *connection* must be NULL. If *connection* is NULL, ct_poll checks all open connections within the context.

milliseconds

The length of time, in milliseconds, to wait for pending operations to complete.

If *milliseconds* is 0, ct_poll returns immediately. To check for operation completions without blocking, pass *milliseconds* as 0.

If *milliseconds* is CS_NO_LIMIT, ct_poll does not return until any of the following is true:

- A server response arrives. This can be a registered procedure notification or the data needed to complete a call to an asynchronous routine.
- No asynchronous-routine completions are pending. If no completions are pending when ct_poll is called, then it returns CS_QUIET (see the Return value section for more information).
- A system interrupt occurs.

Note ct_poll does not wait for the arrival of notification events. However, ct_poll does trigger the notification callback for notification events that are present when it is called or that arrive while ct_poll is waiting for asynchronous routine completions.

---

compconn

The address of a pointer variable. If *connection* is NULL, all connections are polled and ct_poll sets **compconn* to point to the connection structure owning the first completed operation it finds.

If no operation has completed by the time ct_poll returns, ct_poll sets **compconn* to NULL.

If *connection* is supplied, *compconn* must be NULL.

compcmd

The address of a pointer variable. ct_poll sets **compcmd* to point to the command structure owning the first completed operation it finds. If no operation has completed by the time ct_poll returns, ct_poll sets **compcmd* to NULL.

compid

The address of an integer variable. ct_poll sets **compid* to one of the following symbolic values to indicate what has completed:

Table 3-50: Values for ct_poll *compid parameter

Value of compid	Meaning
BLK_DONE	blk_done has completed.
BLK_INIT	blk_init has completed.
BLK_ROWXFER	blk_rowxfer has completed.
BLK_SENDROW	blk_sendrow has completed.
BLK_SENDTEXT	blk_sendtext has completed.
BLK_TEXTXFER	blk_textxfer has completed
CT_CANCEL	ct_cancel has completed.
CT_CLOSE	ct_close has completed.
CT_CONNECT	ct_connect has completed.
CT_DS_LOOKUP	ct_ds_lookup has completed.
CT_FETCH	ct_fetch has completed.
CT_GET_DATA	ct_get_data has completed.
CT_NOTIFICATION	A notification has been received.
CT_OPTIONS	ct_options has completed.
CT_RECVPASSTHRU	ct_recvpassthru has completed.
CT_RESULTS	ct_results has completed.
CT_SEND	ct_send has completed.
CT_SEND_DATA	ct_send_data has completed.
CT_SENDPASSTHRU	ct_sendpassthru has completed.
A user-defined value. This value must be greater than or equal to CT_USER_FUNC.	A user-defined function has completed.

compstatus

A pointer to a variable of type CS_RETCODE. ct_poll sets **compstatus* to indicate the final return code of the completed operation. This value corresponds to the value that would be returned by a synchronous call to the routine under the same conditions. This can be any of the return codes listed for the routine, with the exception of CS_PENDING.

Return value

ct_poll returns the following values:

Table 3-51: ct_poll return values

Return value	Meaning
CS_SUCCEED	An operation has completed.
CS_FAIL	An error occurred.
CS_TIMED_OUT	The timeout value specified by *milliseconds* elapsed before any operation completed. Asynchronous operations may be in progress.
CS_QUIET	ct_poll was called with *milliseconds* as 0 (to indicate that it should return immediately). No asynchronous operations are in progress, and no completed operations or registered procedure notifications were found.
CS_INTERRUPT	A system interrupt has occurred.

ct_poll returns CS_FAIL if it polls a connection that has died.

Examples

```
/*
 ** BusyWait()
 **
 ** Type of function:
 **   async example program api
 **
 ** Purpose:
 **   Silly routine that prints out dots while waiting
 **   for an async operation to complete. It demonstrates
 **   the ability to do other work while an async
 **   operation is pending.
 **
 ** Returns:
 **   completion status.
 **
 ** Side Effects:
 **   None
 */

 CS_STATIC CS_RETCODE CS_INTERNAL
 BusyWait(connection, where)
 CS_CONNECTION   *connection;
 char            *where;
 {
    CS_COMMAND      *compcmd;
    CS_INT          compid;
    CS_RETCODE      compstat;
```

```
    CS_RETCODE      retstat;

    fprintf(stdout, "\nWaiting [%s]", where);
    fflush(stdout);
    do
    {
      fprintf(stdout, ".");
      fflush(stdout);
      retstat = ct_poll(NULL, connection, 100, NULL, &compcmd,
                        &compid, &compstat);
      if (retstat != CS_SUCCEED
          && retstat != CS_TIMED_OUT
          && retstat != CS_INTERRUPT)
      {
        fprintf(stdout,
                "\nct_poll returned unexpected status of %d\n",
                retstat);
        fflush(stdout);
        break;
      }
    } while (retstat != CS_SUCCEED);

    if (retstat == CS_SUCCEED)
    {
      fprintf(stdout,
        "\nct_poll completed: compid = %ld, compstat = %ld\n",
              compid, compstat);
      fflush(stdout);
    }

    return compstat;
  }
```

This code excerpt is from the *ex_amain.c* sample program.

Usage

Table 3-52 summarizes ct_poll usage.

Table 3-52: Summary of ct_poll parameters

context	connection	compconn	Result
NULL	Must have a value.	Must be NULL.	Checks the single connection specified by *connection*.
Has a value	Must be NULL.	Must have a value.	Checks all connections within this context. Sets **compconn* to point to the connection owning the first completed operation it finds.

- ct_poll polls either a specific connection or all connections within a specific context.

  Note On platforms where Client-Library uses signals to implement asynchronous network I/O, the application's callback routines can execute at the system interrupt level.

  Do not call ct_poll from within any Client-Library callback function or within any other function that can execute at the system interrupt level.

  Calling ct_poll at the system-interrupt level can corrupt Open Client and Open Server internal resources and cause recursion in the application.

- If a platform does not provide interrupt- or thread-driven I/O, then an application must periodically read from the network to recognize asynchronous operation completions and registered procedure notifications.

  All routines that can return CS_PENDING read from the network. If an application is not actively using any of these routines, it must call ct_poll to recognize asynchronous operation completions and registered procedure notifications.

  - ct_poll must be called periodically to recognize asynchronous operation completions. ct_poll reports which routine has completed and the completion status of the asynchronous operation. If a completion callback is installed for the connection on which the completion occurred, then the completion callback is invoked by ct_poll.

- For registered procedure notifications, the application can be reading from the connection (as part of the normal process of sending commands processing results) or call ct_poll to cause Client-Library to recognize notification events. The notification callback can be invoked by ct_poll or by any routine which is reading from the connection. If the application is not actively sending commands and processing results on the connection, it should poll the connection with ct_poll to receive the notification event.

- If CS_ASYNC_NOTIFS is CS_FALSE, ct_poll does not read from the network. This means that an application must be reading results for ct_poll to report a registered procedure notification.
- If a platform allows the use of callback functions, ct_poll automatically calls the proper callback routine, if one is installed, when it finds a completed operation or a notification.
- ct_poll does not check for asynchronous operation completions if the CS_DISABLE_POLL property is set to CS_TRUE.
- If there are no pending asynchronous operations, ct_poll returns immediately, regardless of the value of *milliseconds*.

See also

"Asynchronous programming" on page 12, "Callbacks" on page 24, ct_callback, ct_wakeup

ct_recvpassthru

Description

Receive a TDS (Tabular Data Stream) packet from a server.

Syntax

CS_RETCODE ct_recvpassthru (cmd, recvptr)

```
CS_COMMAND    *cmd;
CS_VOID       **recvptr;
```

Parameters

cmd

A pointer to a CS_COMMAND structure.

recvptr

The address of a pointer variable. ct_recvpassthru sets the variable to the address of a buffer containing the most recently received TDS packet. The application is not responsible for allocating this buffer.

Return value

ct_recvpassthru returns the following values:

Table 3-53: ct_recvpassthru return values

Return value	Meaning
CS_PASSTHRU_MORE	Packet received successfully; more packets are available.
CS_PASSTHRU_EOM	Packet received successfully; no more packets are available.
CS_FAIL	The routine failed.
CS_CANCELED	The passthrough operation was canceled.
CS_PENDING	Asynchronous network I/O is in effect. See "Asynchronous programming" on page 12.
CS_BUSY	An asynchronous operation is already pending for this connection. See "Asynchronous programming" on page 12.

Usage

- TDS is a communications protocol used for the transfer of requests and request results between clients and servers. Under ordinary circumstances, non-gateway applications do not usually have to deal with TDS, because Client-Library manages the data stream.
- ct_recvpassthru and ct_sendpassthru are useful in gateway applications. When an application serves as the intermediary between two parties (such as a client and a remote server, or two servers), it can use these routines to pass the TDS stream from one server to the other, eliminating the process of interpreting the information and re-encoding it.
- ct_recvpassthru reads a packet of bytes from a server connection and sets **recvptr* to point to the buffer containing the bytes.
- Default packet sizes vary by platform. On most platforms, a packet has a default size of 512 bytes. A connection can change its packet size through ct_con_props.
- ct_recvpassthru returns CS_PASSTHRU_EOM if the TDS packet has been marked by the server as EOM (End Of Message). If the TDS packet is not marked EOM, ct_recvpassthru returns CS_PASSTHRU_MORE.
- A connection which is being used for a passthrough operation cannot be used for any other Client-Library function until CS_PASSTHRU_EOM has been received.

See also

ct_getloginfo, ct_sendpassthru, ct_setloginfo

ct_remote_pwd

Description

Define or clear passwords to be used for server-to-server connections.

Syntax

```
CS_RETCODE ct_remote_pwd(connection, action,
            server_name, snamelen, password,
            pwdlen)

CS_CONNECTION   *connection;
CS_INT          action;
CS_CHAR         *server_name;
CS_INT          snamelen;
CS_CHAR         *password;
CS_INT          pwdlen;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

It is illegal to define remote passwords for a connection that is open.

action

One of the following symbolic values:

Value of action	Result
CS_SET	Sets the remote password
CS_CLEAR	Clears all remote passwords specified for this connection by setting them to NULL.

server_name

A pointer to the name of the server for which the password is being defined. **server_name* is the name given to the server in an *interfaces* file.

If *server_name* is NULL, the specified password will be considered a universal password, to be used with any server that does not have a password explicitly specified.

If *action* is CS_CLEAR, *server_name* must be NULL.

snamelen

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

If *action* is CS_SET and *server_name* is NULL, pass *snamelen* as 0 or CS_UNUSED.

If *action* is CS_CLEAR, *snamelen* must be CS_UNUSED.

password
A pointer to the password being installed for remote logins to the **server_name* server.

If *action* is CS_CLEAR, *password* must be NULL.

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

If *action* is CS_SET and *password* is NULL, pass *pwdlen* as 0 or CS_UNUSED.

If *action* is CS_CLEAR, *pwdlen* must be CS_UNUSED.

Return value

ct_remote_pwd returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Usage

- ct_remote_pwd defines the password that a server will use when logging into another server.
- A Transact-SQL language command or stored procedure running on one server can execute a stored procedure located on another server. To accomplish this server-to-server communication, the first server, to which an application has connected through ct_connect, actually logs into the second, remote server, performing a server-to-server remote procedure call.

  ct_remote_pwd allows an application to specify the password to be used when the first server logs into the remote server.
- Multiple passwords may be specified, one for each server that a server might need to log in to. Each password must be defined with a separate call to ct_remote_pwd.
- An application can specify a universal password for server-to-server communication by calling ct_remote_pwd with a NULL *server_name* and the *password* value. Once the connection is open, the connection’s server uses this password to log in to any remote server for which a server-name/password pair was not specified with ct_remote_pwd.

If an application does not specify any remote server passwords, then Client-Library sends the connection password as the default universal password for server-to-server communication. The connection password is set through ct_con_props(CS_PASSWORD) and defaults to NULL. So, if an application user has the same password on different servers, the application need not call ct_remote_pwd.

However, if the application specifies a password for any particular server, then the application must explicitly define a universal password. For example, the following code specifies “tigger2” as the password for the “honey_tree” server and specifies “christopher” as the universal password to be used with any other remote server:

```
/*
 ** User’s password is "tigger2" on the "honey_tree" server.
 */
 retcode = ct_remote_pwd(conn, CS_SET, "honey_tree", CS_NULLTERM,
                         "tigger2", CS_NULLTERM);
 if (retcode != CS_SUCCEED)
       ... handle the error ...

 /*
 ** User’s password is "christopher" everywhere else.
 */
 retcode = ct_remote_pwd(conn, CS_SET, (CS_CHAR *) NULL, 0
                         "christopher", CS_NULLTERM);
 if (retcode != CS_SUCCEED)
       ... handle the error ...
```

- Remote passwords are stored in an internal buffer which is only 255 bytes long. Each password’s entry in the buffer consists of the password itself, the associated server name, and two extra bytes. If the addition of a password to this buffer would cause overflow, ct_remote_pwd returns CS_FAIL and generates a Client-Library error message that indicates the problem.
- It is an error to call ct_remote_pwd to define a remote password for a connection that is already open. Define remote passwords before calling ct_connect to create an active connection.
- An application can call ct_remote_pwd to clear remote passwords for a connection at any time.

See also ct_con_props, ct_connect

ct_res_info

Description

Retrieve current result set or command information.

Syntax

```
CS_RETCODE ct_res_info(cmd, type, buffer, buflen,
            outlen)

CS_COMMAND     *cmd;
CS_INT         type;
CS_VOID        *buffer;
CS_INT         buflen;
CS_INT         *outlen;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server command.

type

The type of information to return. Table 3-54 lists the symbolic values for *type*.

buffer

A pointer to the space in which ct_res_info will place the requested information.

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

buflen

The length, in bytes, of the **buffer* data space, or CS_UNUSED if **buffer* represents a fixed-length or symbolic value.

outlen

A pointer to an integer variable.

ct_res_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

ct_res_info returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Return value	Meaning
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

ct_res_info returns CS_FAIL if the requested information is larger than *buflen* bytes, or if there is no current result set.

Examples

This fragment from the *rpc.c* sample program retrieves the number of columns in a fetchable result set:

```
CS_INT   num_cols;

/*
 ** Determine the number of columns in this result
 ** set.
 */
 retcode = ct_res_info(cmd, CS_NUMDATA, #_cols,
      CS_UNUSED, NULL);
 if (retcode != CS_SUCCEED)
 {
      ...CODE DELETED...
 }
```

This fragment from the *rpc.c* sample program retrieves the message identifier from a message result.

```
CS_SMALLINT msg_id;

... ct_results has returned with a CS_MSG_RESULT
     result type ...

case CS_MSG_RESULT:
      retcode = ct_res_info(cmd, CS_MSGTYPE,
           (CS_VOID *)&msg_id, CS_UNUSED, NULL);
      if (retcode != CS_SUCCEED)
      {
           ...CODE DELETED...
      }
      fprintf(stdout, "ct_result returned \
           CS_MSG_RESULT where msg id = %d.\n", msg_id);
      break;
```

Usage

Table 3-54 summarizes ct_res_info usage.

Table 3-54: Summary of ct_res_info parameters

Value of type	Returned by ct_res_info into *buffer	Information available after ct_results sets its *result_type parameter to	*buffer Datatype
CS_BROWSE_INFO	CS_TRUE if browse-mode information is available; CS_FALSE if browse-mode information is not available.	CS_ROW_RESULT	CS_BOOL
CS_CMD_NUMBER	The number of the command that generated the current result set.	Any value	CS_INT
CS_MSGTYPE	An integer representing the ID of the message that makes up the current result set.	CS_MSG_RESULT	CS_USHORT
CS_NUM_COMPUTES	The number of compute clauses in the current command.	CS_COMPUTE_RESULT	CS_INT
CS_NUMDATA	The number of items in the current result set.	CS_COMPUTE_RESULT, CS_COMPUTEFMT_RESULT, CS_CURSOR_RESULT, CS_DESCRIBE_RESULT, CS_PARAM_RESULT, CS_ROW_RESULT, CS_ROWFMT_RESULT, CS_STATUS_RESULT	CS_INT
CS_NUMORDERCOLS	The number of columns specified in the order-by clause of the current command.	CS_ROW_RESULT	CS_INT
CS_ORDERBY_COLS	The select list ID numbers of columns specified in a the order by clause of the current command.	CS_ROW_RESULT	Array of CS_INT
CS_ROW_COUNT	The number of rows affected by the current command.	CS_CMD_DONE, CS_CMD_FAIL CS_CMD_SUCCEED	CS_INT
CS_TRANS_STATE	The current server transaction state.	Any value. CT_RESULTS must have returned CS_SUCCEED.	CS_INT

- ct_res_info returns information about the current result set or the current command. The current command is defined as the command that generated the current result set.

- A result set is a collection of a single type of result data. Result sets are generated by commands. For more information about result sets, see the ct_results reference page and “Results” on page 241.
- Most typically, an application calls ct_res_info with *type* as CS_NUMDATA, to determine the number of items in a result set.

Determining whether Browse-mode information is available

- To determine whether browse-mode information is available, call ct_res_info with *type* as CS_BROWSE_INFO.
- If browse-mode information is available, an application can call ct_br_column and ct_br_table to retrieve the information. If browse-mode information is not available, calling ct_br_column or ct_br_table will result in a Client-Library error.
- For more information about browse mode, see “Browse mode” on page 21.

Retrieving the command number for current results

- To determine the number of the command that generated the current results, call ct_res_info with *type* as CS_CMD_NUMBER.
- Client-Library keeps track of the command number by counting the number of times ct_results returns CS_CMD_DONE.

  An application’s first call to ct_results following a ct_send call sets the command number to 1. After that, the command number is incremented each time ct_results is called after returning CS_CMD_DONE.
- CS_CMD_NUMBER is useful in the following cases:
  - To find out which Transact-SQL command within a language command generated the current result set
  - To find out which cursor command, in a batch of cursor commands, generated the current result set
  - To find out which select command in a stored procedure generated the current result set
- A language command contains a string of Transact-SQL text. This text represents one or more Transact-SQL commands. When used with a language command, “command number” refers to the number of the Transact-SQL command in the language command.

  For example, the string:

```
select * from authors
```

```
select * from titles
insert newauthors
     select *
     from authors
     where city = "San Francisco"
```

represents three Transact-SQL commands, two of which can generate result sets. In this case, the command number that ct_res_info returns can be from 1 to 3, depending on when ct_res_info is called.

- Inside stored procedures, only select statements cause the command number to be incremented. If a stored procedure contains seven Transact-SQL commands, three of which are selects, the command number that ct_res_info returns can be any integer from 1 to 3, depending on which select generated the current result set.
- ct_cursor is used to initiate a cursor command. Several cursor commands can be defined as a batch before they are sent to a server. When used with a cursor command batch, “command number” refers to the number of the cursor command in the command batch.

  For example, an application can make the following calls:

  ```
  ct_cursor(...CS_CURSOR_DECLARE...);
   ct_cursor(...CS_CURSOR_ROWS...);
   ct_cursor(...CS_CURSOR_OPEN...);
   ct_send();
  ```

  The command number that ct_res_info returns can be from 1 to 3 depending on which cursor command generated the current result type.

Retrieving a message ID

- To retrieve a message ID, call ct_res_info with *type* as CS_MSGTYPE.
- Servers can send messages to client applications. Messages are received in the form of “message result sets.” Message result sets contain no fetchable data, but rather have an ID number.
- Messages can also have parameters. Message parameters are returned to an application as a parameter result set, immediately following the message result set.

Retrieving the number of *compute* clauses

- To determine the number of compute clauses in the command that generated the current result set, call ct_res_info with *type* as CS_NUM_COMPUTES.

- A Transact-SQL select statement can contain compute clauses that generate compute result sets.

Retrieving the number of result data items

- To determine the number of result data items in the current result set, call ct_res_info with *type* as CS_NUMDATA.
- Results sets contain result data items. Row, cursor, and compute result sets contain columns, a parameter result set contains parameters, and a status result set contains a status. The columns, parameters, and status are known as **result data items**.
- A message result set does not contain any data items.

Retrieving the number of columns in an *order by* clause

- To determine the number of columns in a Transact-SQL select statement's order by clause, call ct_res_info with *type* as CS_NUMORDERCOLS.
- A Transact-SQL select statement can contain an order by clause, which determines how the rows resulting from the select statement are ordered on presentation.

Retrieving the column IDs of order-by columns

- To get the select list column IDs of order-by columns, call ct_res_info with *type* as CS_ORDERBY_COLS.
- Columns named in an order by clause must also be named in the select list of the select statement. Columns in a select list have a select list ID, which is the number in which they appear in the list. For example, in the following query, au_lname and au_fname have select list IDs of 1 and 2, respectively:

```
select au_lname, au_fname from authors
      order by au_fname, au_lname
```

- Given the preceding query, the call:

```
ct_res_info(cmd, CS_ORDERBY_COLS, myspace, 8,
      outlength)
```

  sets **myspace* to an array of two CS_INT values containing the integers 2 and 1.

Retrieving the number of rows for the current command

- To determine the number of rows affected by or returned by the current command, call ct_res_info with *type* as CS_ROW_COUNT.

- An application can retrieve a row count after ct_results sets its **result_type* parameter to CS_CMD_SUCCEED, CS_CMD_DONE, or CS_CMD_FAIL. A row count is guaranteed to be accurate if ct_results has just set **result_type* to CS_CMD_DONE.
- Applications that allow ad-hoc query entry may need to print a rows-affected message (as done by the isql client application) when processing results. To do this, the application should do the following when ct_results indicates a CS_CMD_DONE *result_type* value:

  a Retrieve the row count with ct_res_info(CS_ROW_COUNT).

  b If the count is not CS_NO_COUNT, print it.

  If the application only needs row counts for commands that modify data (such as insert or update statements), it performs the above steps when ct_results indicates a CS_CMD_SUCCEED *result_type* value.
- If the command is one that executes a stored procedure, for example a Transact-SQL exec language command or a remote procedure call command, ct_res_info sets **buffer* to the number of rows affected by the last statement in the stored procedure that affects rows.
- ct_res_info sets **buffer* to CS_NO_COUNT if any of the following are true:
  - The Transact-SQL command fails for any reason, such as a syntax error.
  - The command is one that *never* affects rows, such as a Transact-SQL print command.
  - The command executes a stored procedure that does not affect any rows.
  - The CS_OPT_NOCOUNT option is on.

Retrieving the current server transaction state

- To determine the current server transaction state, call ct_res_info with *type* as CS_TRANS_STATE.

See also

ct_cmd_props, ct_con_props, ct_results, “Options” on page 174, “Server transaction states” on page 131

ct_results

Description

Set up result data to be processed.

Syntax

```
CS_RETCODE ct_results(cmd, result_type)

CS_COMMAND      *cmd;
CS_INT          *result_type;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

result_type

A pointer to an integer variable which ct_results sets to indicate the current type of result.

The following table lists the possible values of **result_type*:

Table 3-55: Values for ct_results *result_type parameter

Result category	Value of *result_type	Meaning	Contents of result set
Values that indicate command status	CS_CMD_DONE	The results of a logical command have been completely processed.	Not applicable.
	CS_CMD_FAIL	The server encountered an error while executing a command.	No results.
	CS_CMD_SUCCEED	The success of a command that returns no data, such as a language command containing a Transact-SQL insert statement.	No results.
Values that indicate fetchable results	CS_COMPUTE_RESULT	Compute row results.	A single row of compute results.
	CS_CURSOR_RESULT	Cursor row results from a ct_cursor cursor-open command.	Zero or more rows of tabular data.
	CS_PARAM_RESULT	Return parameter results.	A single row of return parameters.
	CS_ROW_RESULT	Regular row results.	Zero or more rows of tabular data.
	CS_STATUS_RESULT	Stored procedure return status results.	A single row containing a single status.

Result category	Value of *result_type	Meaning	Contents of result set
Values that indicate information is available.	CS_COMPUTEFMT_RESULT	Compute format information.	No fetchable results. The application can retrieve the format of forthcoming compute results for the current command. An application can call ct_res_info, ct_describe, and ct_compute_info to retrieve compute format information.
	CS_ROWFMT_RESULT	Row format information.	No fetchable results. An application can call ct_describe and ct_res_info to retrieve row format information.
	CS_MSG_RESULT	Message arrival.	No fetchable results. An application can call ct_res_info to get the message's ID. Parameters associated with the message, if any, are returned as a separate parameter result set.
	CS_DESCRIBE_RESULT	Dynamic SQL descriptive information from a describe-input or describe-output command.	No fetchable results, but the description of command inputs or outputs. The application can retrieve the results by any of the following methods: • Call ct_res_info to get the number of items and ct_describe to get item descriptions. • Call ct_dyndesc several times to get the number of items and a description of each. • Call ct_res_info to get the number of items, and call ct_dynsqlda once to get item descriptions.

Return value

ct_results returns the following values:

Table 3-56: ct_results return values

Return value	Meaning
CS_SUCCEED	A result set is available for processing.
CS_END_RESULTS	All results have been completely processed.
CS_FAIL	The routine failed; any remaining results are no longer available. If ct_results returns CS_FAIL, an application must call ct_cancel with *type* as CS_CANCEL_ALL before using the affected command structure to send another command. If ct_cancel returns CS_FAIL, the application must call ct_close(CS_FORCE_CLOSE) to force the connection closed.
CS_CANCELED	Results have been canceled.
CS_PENDING	Asynchronous network I/O is in effect. See “Asynchronous programming” on page 12.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

This code excerpt is from the *compute.c* sample program.

```
/*
** DoCompute(connection)
*/

CS_STATIC CS_RETCODE
DoCompute(connection)
CS_CONNECTION    *connection;
{

    CS_RETCODE   retcode;
    CS_COMMAND   *cmd;
    /* Result type from ct_results */
    CS_INT       res_type;

    /* Use the pubs2 database */
        ...CODE DELETED.....

    /*
    ** Allocate a command handle to send the compute
    ** query with.
    */
        ...CODE DELETED.....

    /*
    ** Define a language command that contains a
    ** compute clause.  SELECT is a select statment
```

```
** defined in the header file.
*/
    ...CODE DELETED.....

/* Send the command to the server */
    ...CODE DELETED.....

/*
** Process the results.
** Loop while ct_results() returns CS_SUCCEED.
*/
while ((retcode = ct_results(cmd, &res_type)) == CS_SUCCEED)
{
    switch ((int)res_type)
    {
        case CS_CMD_SUCCEED:
            /*
            ** Command returning no rows completed successfully.
            */
            break;

        case CS_CMD_DONE:
            /*
            ** This means we're done with one result set.
            */
            break;

        case CS_CMD_FAIL:
            /*
            ** This means that the server encountered
            ** an error while processing our command.
            */
            ex_error("DoCompute: ct_results() \ returned CMD_FAIL");
            break;

        case CS_ROW_RESULT:
            retcode = ex_fetch_data(cmd);
            if (retcode != CS_SUCCEED)
            {
                ex_error("DoCompute: ex_fetch_data()\ failed");
                return retcode;
            }
            break;

        case CS_COMPUTE_RESULT:
            retcode = FetchComputeResults(cmd);
            if (retcode != CS_SUCCEED)
            {
                ex_error("DoCompute: \ FetchComputeResults() failed");
```

```
                    return retcode;
                }
                break;

            default:
                /* We got an unexpected result type */
                ex_error("DoCompute: ct_results() \ returned unexpected
                            returned unexpected result type");
                return CS_FAIL;
        }
    }

    /*
    ** We've finished processing results. Let's check the return value
    ** of ct_results() to see if everything went ok.
    */
    switch ((int)retcode)
    {
        case CS_END_RESULTS:
        /* Everything went fine */
        break;

        case CS_FAIL:
            /* Something went wrong */
            ex_error("DoCompute: ct_results() \ failed”);
            return retcode;

        default:
            /* We got an unexpected return value */
            ex_error("DoCompute: ct_results() \
                        returned unexpected result code");
            return retcode;
    }

    /* Drop our command structure */
        ...CODE DELETED.....

    return retcode;
}
```

Usage

- An application calls ct_results as many times as necessary after sending a command to the server using ct_send.
- If a command returns fetchable result data, then ct_results prepares the server connection so that the application can read the result data returned by the command using ct_fetch or ct_res_info.
- **Result data** is an umbrella term for all the types of data that a server can return to an application. The types of data include:

- Regular rows
- Cursor rows
- Return parameters
- Stored procedure return status numbers
- Compute rows
- Dynamic SQL descriptive information
- Regular row and compute row format information
- Messages

ct_results is used to set up all of these types of results for processing.

Note Don't confuse message results with server error and informational messages. See "Error handling" on page 123 for a discussion of error and informational messages.

- Result data is returned to an application in the form of a **result set**. A result set includes only a single type of result data. For example, a regular row result set contains only regular rows, and a return parameter result set contains only return parameters.

The *ct_results* loop

- Because a command can generate multiple result sets, an application must call ct_results as long as it continues to return CS_SUCCEED, indicating that results are available.
- The simplest way to read results is in a loop that terminates when ct_results does not return CS_SUCCEED. After the loop, an application can use a case-type statement to test ct_results' final return code and determine why the loop terminated. The following rules apply to the logic of a results handling loop:
  - ct_results returns CS_SUCCEED as long as results are still available to the application.
  - When ct_results sets *result_type* to a value that indicates fetchable result data, the application must fetch or cancel the data before continuing.

- ct_results sets the value of *result_type* to CS_CMD_DONE to indicate that the results of a logical command have been completely processed. Logical commands are explained in the following section titled "ct_results and logical commands."
- ct_results returns CS_END_RESULTS when all results have been processed successfully.
- ct_results returns CS_CANCELED if the application cancels the results with ct_cancel(CS_CANCEL_ALL) or ct_cancel(CS_CANCEL_ATTN).

- Results are returned to an application in the order in which they are produced. However, this order is not always easy to predict. For example, when an application calls a stored procedure that in turn calls another stored procedure, the application might receive a number of regular row and compute row result sets, as well as a return parameter and a return status result set. The order in which these results are returned depends on how the stored procedures are written.

  For this reason, Sybase recommended that an application's ct_results loop be coded so that control drops into a case-type statement that handles all types of results that can be received. The return parameter *result_type* indicates symbolically what type of result data the result set contains.

- A connection has **pending results** if it has not processed all of the results generated by a Client-Library command. Usually, an application cannot send a new command on a connection with pending results. An exception to this rule occurs for CS_CURSOR_RESULT results. For more information about this exception, see Chapter 7, "Using Client-Library Cursors," in the Open Client *Client-Library/C Programmers Guide*.

ct_results and logical commands

- ct_results sets **result_type* to CS_CMD_DONE to indicate that the results of a "logical command" have been completely processed.
- A **logical command** is defined as any command defined through ct_command, ct_dynamic, or ct_cursor, with the following exceptions:
  - Each Transact-SQL select statement that returns columns inside a stored procedure is a logical command. Other Transact-SQL statements inside stored procedures do not count as logical commands (including select statements that assign values to local variables).
  - Each Transact-SQL statement executed by a dynamic SQL command is a distinct logical command.

  - Each Transact-SQL statement in a language command is a logical command.

- A command sent by a client application can execute multiple logical commands on the server.
- A logical command can generate one or more result sets.
- For example, suppose a Client-Library language command contains the following Transact-SQL statements:

```
select type, price
      from titles
      order by type, price
      compute sum(price) by type

select type, price, advance
      from titles
      order by type, advance
      compute sum(price), max(advance) by type
```

ct_results**result_types*

```
CS_ROW_RESULT        Row and compute results from
 CS_COMPUTE_RESULT    the first select,
 ...                  repeated as many times as the
                      value of the type column
                      changes.
 CS_CMD_DONE          Indicates that the results
                      of the first query have been
                      processed.


CS_ROW_RESULT         Row and compute results from
 CS_COMPUTE_RESULT     the second select,
 ...                  repeated as many times as the
                      value of the type column
                      changes.
 CS_CMD_DONE           Indicates that the results of
                       the second query have been
                       processed.
```

When calling to process the results of this language command, an application would see the following :

- A **result_type* of CS_CMD_SUCCEED or CS_CMD_FAIL is immediately followed by a **result_type* of CS_CMD_DONE.

Canceling results

- To cancel all remaining results from a command (and eliminate the need to continue calling ct_results until it fails to return CS_SUCCEED), call ct_cancel with *type* as CS_CANCEL_ALL.
- To cancel only the current results, call ct_cancel with *type* as CS_CANCEL_CURRENT.
- Unwanted cursor results from a ct_cursor cursor-open command should not be canceled. Instead, close the cursor with a ct_cursor cursor-close command.

Special kinds of result sets

- A **message result set** contains no actual result data. Rather, a message has a ID. An application can call ct_res_info to retrieve a message ID. In addition to an ID, messages can have parameters. Message parameters are returned to an application as a parameter result set immediately following the message result set.
- **Row format** and **compute format** result sets contains no actual result data. Instead, format result sets contain formatting information for the regular row or compute row result sets with which they are associated.

  This type of format information is of use primarily in gateway applications, which need to repackage Adaptive Server format information before sending it to a foreign client. After ct_results indicates format results, a gateway application can retrieve format information by calling:

  - ct_res_info, for the number of columns;
  - ct_describe, for a description of each column; and
  - ct_compute_info, for information on the compute clause that generated the compute rows.

  All format information for a command is returned before any data. That is, the row format and compute format result sets for a command precede the regular row and compute row result sets generated by the command.

  An application will not receive format results unless the Client-Library CS_EXPOSE_FMTS property is set to CS_TRUE.

- A **describe result set** contains no actual result data. Instead, a describe result set contains descriptive information generated by a dynamic SQL describe input or describe output command. After ct_results indicates describe results, an application can retrieve the description with any of these techniques:
  - Call ct_res_info to get the number of items and ct_describe to get a description of each item.
  - Call ct_dyndesc several times to get the number of items and a description of each.
  - Call ct_res_info to get the number of items, and call ct_dynsqlda once to get item descriptions.

ct_results and stored procedures

- A runtime error on a language command containing an execute statement will generate a **result_type* of CS_CMD_FAIL. For example, this occurs if the procedure named in the execute statement cannot be found.

  A runtime error on a statement inside a stored procedure will *not* generate a CS_CMD_FAIL, however. For example, if the stored procedure contains an insert statement and the user does not have insert permission on the database table, the insert statement will fail, but ct_results will still return CS_SUCCEED. To check for runtime errors inside stored procedures, examine the procedure's return status number, which is returned as a return status result set immediately following the row and parameter results, if any, from the stored procedure. If the error generates a server message, it is also available to the application.

ct_results and the CS_STICKY_BINDS property

- Applications that repeatedly execute the same command can set the CS_STICKY_BINDS property to cause Client-Library to save result bindings established during the original execution of the command. See "Persistent result bindings" on page 224 for a description of this property.
- When CS_STICKY_BINDS is enabled, ct_results compares the format of the current result set with the format that applied when the binds were established. A command's result format information consists of a sequence of the following result set characteristics:
  - The result type, indicated to the application by the ct_results *result_type* parameter
  - (For fetchable results only.) The number of columns, available to the application through ct_res_info.

- (For fetchable results only.) The format of each column, available to the application through ct_describe for each column

- If ct_results detects a format mismatch, it clears all saved bindings for all result sets in the original result sequence. When this happens, ct_results raises an informational error and returns CS_SUCCEED. Note that a format mismatch can only occur when executing a command that contains conditional logic (for example, a stored procedure containing an if or a while clause).

See also

ct_bind, ct_command, ct_cursor, ct_describe, ct_dynamic, ct_fetch, ct_send, "Results" on page 241

ct_scroll_fetch

Description

Used for scrollable fetching after a supported scrollable cursor has been declared and successfully opened.

The capability properties of ct_scroll_fetch detect if the corresponding ASE server supports scrollable cursors. If scrollable cursors is not supported, a fatal error is generated and ct_scroll_fetch cannot be used. If this happens, use ct_fetch instead.

Syntax

```
CS_RETCODE ct_scroll_fetch(md, type, offset, option, rows_read)

CS_COMMAND  *cmd;
 CS_INT      type;
 CS_INT      offset;
 CS_INT      option;
 CS_INT      *rows_read;
```

Parameters

cmd

Command handle that holds the scrollable cursor definition.

type

Fetch orientation, with valid entries listed in Table 3-57:

Table 3-57: Values for ct_scroll_fetch type

Type	Offset Value	Meaning
CS_NEXT	Ignored	Returns CS_CURSOR_ROWS at each call. If the number of rows in CS_CURSOR_ROWS is greater than the number of rows in the cursor result set, the array may have undefined values. This also happens if the last fetch produces fewer rows than CS_CURSOR_ROWS. To find the number of rows returned, view rows_read. This allows you to validate the application array entries. A repeated sequence of CS_NEXT also causes the cursor to move beyond the last table row. If this occurs, zero rows is returned and ct_scroll_fetch returns CS_CURSOR_AFTER_LAST. A warning message is also generated, informing you that the scrollable cursor has moved beyond the resultset boundary. Please note that this is a warning, and does not indicate that an error has occurred. Where the number of rows in CS_CURSOR_ROWS is greater than the number of rows in the cursor result set, a subsequent call to CS_NEXT positions the cursor beyond the last row.
CS_FIRST	Ignored	Setting CS_FIRST returns CS_CURSOR_ROWS, starting at the first row. If followed by CS_PREV, zero rows is returned and ct_scroll_fetch returns CS_CURSOR_BEFORE_FIRST.

Type	Offset Value	Meaning
CS_PREV	Ignored	CS_PREV positions the cursor to the row before the current position. If repeated, CS_PREV calls bring the cursor back to the row 1, the next CS_PREV call will return 0 rows, and CS_CURSOR_BEFORE_FIRST is returned. A warning message is also generated, informing you that the scrollable cursor has moved beyond the resultset boundary. Please note that this is a warning, and does not indicate that an error has occurred.
CS_LAST	Ignored	Returns the last rows in CS_CURSOR_ROWS. If CS_LAST is followed by CS_NEXT, zero rows are returned and ct_scroll_fetch returns CS_CURSOR_AFTER_LAST.
CS_RELATIVE	Positive or negative (if zero, client generates a warning).	The offset value is treated as a signed integer (CS_INT), and is either positive or negative. This indicates a relative jump from the current cursor position.
CS_ABSOLUTE	Positive or negative (if zero, client generates a warning).	The absolute row number must be supplied. This is a signed integer (CS_INT).

offset
Passed as a signed integer, and valid only if type is CS_RELATIVE or CS_ABSOLUTE. In other cases, offset is CS_UNUSED.

option
Used to continue scrolling or to stop. If *option* is CS_TRUE, ct_scroll_fetch continues, based on (new) values given to *type* and *offset*. If *option* is CS_FALSE, the cursor stops scrolling and CS_SCROLL_CURSOR_ENDS is returned.

rows_read
Returns the number of rows per ct_scroll_fetch call.

Return value

ct_scroll_fetch returns the following values, in addition to those provided in “ct_fetch return values” on page 500 (except CS_END_DATA):

Table 3-58: ct_scroll_fetch return values

Return value	Meaning
CS_SCROLL_ CURSOR_ENDS	CS_SCROLL_CURSOR_ENDS is returned when ct_scroll_fetch receives a CS_FALSE value. The return value may be used to signify that no more data is to be fetched from ASE. Typically, ct_scroll_fetch runs under the control of ct_results. Each call to ct_scroll_fetch returns the maximum number of rows indicated in CS_CURSOR_ROW. The application issues a new ct_scroll_fetch call or stops fetching. If the application stops fetching, CS_SCROLL_CURSOR_ENDS is returned by ct_scroll_fetch, ct_results is processed and an internal cleanup is performed.
CS_CURSOR_ BEFORE_FIRST	CS_CURSOR_BEFORE_FIRST is returned when a call to ct_scroll_fetch causes the cursor to move before the first row in the ASE result-set. No rows are returned, and rows_read is zero. CS_CURSOR_BEFORE_FIRST may generate a warning if an error handler is installed.
CS_CURSOR_ AFTER_LAST	CS_CURSOR_AFTER_LAST is returned when a call to ct_scroll_fetch causes the cursor to move after the last row in ASE result set. No rows are returned, and rows_read is zero. CS_CURSOR_BEFORE_FIRST may generate a warning if an error handler is installed.

Examples

The following fragment illustrates the call sequence to scroll fetch:

```
CS_RETCODE CS_PUBLIC
ex_scroll_fetch_1(CS_COMMAND *cmd)
{
    CS_RETCODE       retcode;
    CS_INT           num_cols;
    CS_INT           i;
    CS_INT           j;
    CS_INT           k;
    CS_INT           row_count = 0;
    CS_INT           rows_read;
    CS_INT           disp_len;
    CS_INT           sc_type;
    CS_INT           sc_offset;
    CS_INT           sc_option;
    CS_DATAFMT       *datafmt;
    EX_COLUMN_DATA   *coldata;

/*
```

```
** Find out how many columns there are in this result set.
*/
retcode = ct_res_info(cmd, CS_NUMDATA, &num_cols, CS_UNUSED, NULL);
if (retcode != CS_SUCCEED)
{
    ex_error("ex_scroll_fetch_data: ct_res_info() failed");
    return retcode;
}

/*
** Make sure we have at least one column
*/
if (num_cols <= 0)
{
    ex_error("ex_scroll_fetch_data: ct_res_info() returned zero columns");
    return CS_FAIL;
}

/*
** Our program variable, called 'coldata', is an array of
** EX_COLUMN_DATA structures. Each array element represents
** one column.  Each array element will re-used for each row.
**
** First, allocate memory for the data element to process.
*/
coldata = (EX_COLUMN_DATA *)malloc(num_cols * sizeof (EX_COLUMN_DATA));
if (coldata == NULL)
{
    ex_error("ex_scroll_fetch_data: malloc() failed");
    return CS_MEM_ERROR;
}

datafmt = (CS_DATAFMT *)malloc(num_cols * sizeof (CS_DATAFMT));
if (datafmt == NULL)
{
    ex_error("ex_scroll_fetch_data: malloc() failed");
    free(coldata);
    return CS_MEM_ERROR;
}

/*
** Loop through the columns getting a description of each one
** and binding each one to a program variable.
**
** We're going to bind each column to a character string;
** this will show how conversions from server native datatypes
```

```
** to strings can occur via bind.
**
** We're going to use the same datafmt structure for both the describe
** and the subsequent bind.
**
** If an error occurs within the for loop, a break is used to get out
** of the loop and the data that was allocated is free'd before
** returning.
*/
for (i = 0; i < num_cols; i++)
{
    /*
    ** Get the column description.  ct_describe() fills the
    ** datafmt parameter with a description of the column.
    */
    retcode = ct_describe(cmd, (i + 1), &datafmt[i]);
    if (retcode != CS_SUCCEED)
    {
        ex_error("ex_scroll_fetch_data: ct_describe() failed");
        break;
    }

    /*
    ** update the datafmt structure to indicate that we want the
    ** results in a null terminated character string.
    **
    ** First, update datafmt.maxlength to contain the maximum
    ** possible length of the column. To do this, call
    ** ex_display_len() to determine the number of bytes needed
    ** for the character string representation, given the
    ** datatype described above.  Add one for the null
    ** termination character.
    */
    datafmt[i].maxlength = ex_display_dlen(&datafmt[i]) + 1;

    /*
    ** Set datatype and format to tell bind we want things
    ** converted to null terminated strings
    */
    datafmt[i].datatype = CS_CHAR_TYPE;
    datafmt[i].format   = CS_FMT_NULLTERM;

    /*
    ** Allocate memory for the column string
    */
    coldata[i].value = (CS_CHAR *)malloc(datafmt[i].maxlength);
```

```
        if (coldata[i].value == NULL)
        {
            ex_error("ex_scroll_fetch_data: malloc() failed");
            retcode = CS_MEM_ERROR;
            break;
        }

        /*
        ** Now bind.
        */
        retcode = ct_bind(cmd, (i + 1), &datafmt[i],
                    coldata[i].value, &coldata[i].valuelen,
                   (CS_SMALLINT *)&coldata[i].indicator);
        if (retcode != CS_SUCCEED)
        {
            ex_error("ex_scroll_fetch_data: ct_bind() failed");
            break;
        }
    }
    if (retcode != CS_SUCCEED)
    {
            for (j = 0; j < i; j++)
            {
                free(coldata[j].value);
            }
        free(coldata);
        free(datafmt);
        return retcode;
    }

    /*
    ** Display column header
    */
    ex_display_header(num_cols, datafmt);

    /*
    ** Fetch the rows.  Call ct_scroll_fetch() as long as it returns
    ** CS_SUCCEED, CS_ROW_FAIL, CS_CURSOR_BEFORE_FIRST or
    ** CS_CURSOR_AFTER_LAST.
    ** These are recoverable or "row" producing conditions, e.g. non-fatal.
    ** All other terminate the loop, either by error or choice.
    */
    for (i = 0; i < EX_MAX_ARR; i++)
    {
        sc_type = scroll_index(type_list0[i], scroll_arrmap);
        sc_offset = offset_list0[i];
```

```
if (type_list0[i] != EX_BADVAL)
{
    sc_option = CS_TRUE;
}
else
{
    /*
    ** Since EX_BADVAL is not valid to pass into
    ** either sc_type or sc_offset we set these
    ** to CS_UNUSED respectively.
    */
    sc_type = CS_UNUSED;
    sc_offset = CS_UNUSED;
    sc_option = CS_FALSE;
}

retcode = ct_scroll_fetch(cmd, sc_type, sc_offset, sc_option,
            &rows_read);
switch ((int)retcode)
{
    case CS_ROW_FAIL:

        fprintf(stdout, "Error on row %d.\n", row_count);
        fflush(stdout);
        break;

    case CS_CURSOR_BEFORE_FIRST:

        fprintf(stdout, " Cursor before first row\n");
        fflush(stdout);
        break;

    case CS_CURSOR_AFTER_LAST:

        fprintf(stdout, " Cursor after last row\n");
        fflush(stdout);
        break;

    case CS_SUCCEED:

        /*
        ** Increment our row count by the number of
        ** rows just fetched.
        */
        row_count = row_count + rows_read;
```

```
        /*
        ** Assume we have a row. Loop through the
        ** columns displaying the column values.
        */
        for (k = 0; k < num_cols; k++)
        {
            /*
            ** Display the column value
            */
            fprintf(stdout, "%s", coldata[k].value);
            fflush(stdout);

            /*
            ** If not last column, Print out spaces between
            ** this column and next one.
            */
            if (k != num_cols - 1)
            {
                disp_len = ex_display_dlen(&datafmt[k]);
                disp_len -= coldata[k].valuelen - 1;
                for (j = 0; j < disp_len; j++)
                {
                    fputc(' ', stdout);
                }
            }
        }
        fprintf(stdout, "\n");
        fflush(stdout);
        break;

    case CS_FAIL:

        /*
        ** Free allocated space.
        */
        for (k = 0; k < num_cols; k++)
        {
            free(coldata[k].value);
        }
        free(coldata);
        free(datafmt);
        return retcode;

    case CS_SCROLL_CURSOR_ENDS:
```

```
                /*
                ** User signalled ct_scroll_fetch() to stop
                ** scrolling, we are done with this result set.
                ** Free allocated space.
                */
                for (k = 0; k < num_cols; k++)
                {
                    free(coldata[k].value);
                }
                free(coldata);
                free(datafmt);
                return retcode;

            default:

                fprintf(stdout, "Hit default, this should not happen.
                        Exiting program.\n");
                fflush(stdout);
                exit(0);
        } /* end switch */
      } /* end for */
      return CS_SUCCEED;
    }
```

Usage

- The first row in the result set is numbered 1.
- rows_read returns the number of actual rows fetched in each call to ct_scroll_fetch. Use rows_read to determine the real number of rows versus potentially non-useful information in the application arrays.

See also

ct_fetch.

ct_send

Description

Send a command to the server.

Syntax

CS_RETCODE ct_send(cmd)

CS_COMMAND *cmd;

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

Return value

ct_send returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed. For less serious failures, the application can call ct_cancel(CS_CANCEL_ALL) to clean up the command structure. For more serious failures, the application must call ct_close(CS_FORCE_CLOSE) to force the connection closed.
CS_CANCELED	The routine was canceled.
CS_PENDING	Asynchronous network I/O is in effect. See “Asynchronous programming” on page 12.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Common causes of ct_send failure include results-pending errors and attempts to send a command that has not been initiated.

- Results-pending errors

  ct_send raises results-pending errors if Client-Library is reading results when ct_send is called. This can occur if the application is retrieving non-cursor results associated with another command structure that belongs to the same parent connection. Sometimes this problem can be solved by rewriting the application to use Client-Library cursors (see ct_cursor for details). If the application cannot use cursors, then separate connections are necessary.

- Attempt to send a command that has not been initiated

  ct_send fails if a command has not been defined with ct_command, ct_cursor, or ct_dynamic.

Examples

The following fragment declares a function that sends a language command and processes the results. The code assumes that the command returns no fetchable results.

```
/*
 ** ex_execute_cmd()
 */

CS_RETCODE      CS_PUBLIC
 ex_execute_cmd(connection, cmdbuf)
 CS_CONNECTION   *connection;
 CS_CHAR         *cmdbuf;
 {
```

```
 CS_RETCODE   retcode;
 CS_INT       restype;
 CS_COMMAND   *cmd;
 CS_RETCODE   query_code;

/*
 ** Get a command handle, store the command string
 ** in it, and send it to the server.
 */
 if ((retcode = ct_cmd_alloc(connection, &cmd)) !=
       CS_SUCCEED)
 {
      ex_error("ex_execute_cmd: ct_cmd_alloc() \
           failed");
      return retcode;
 }

if ((retcode = ct_command(cmd, CS_LANG_CMD,
      cmdbuf, CS_NULLTERM, CS_UNUSED)) !=
      CS_SUCCEED)
 {
      ex_error("ex_execute_cmd: ct_command() \
           failed");
      (void)ct_cmd_drop(cmd);
      return retcode;
 }

if ((retcode = ct_send(cmd)) != CS_SUCCEED)
 {
      ex_error("ex_execute_cmd: ct_send() failed");
      (void)ct_cmd_drop(cmd);
      return retcode;
 }

/*
 ** Examine the results coming back. If any errors
 ** are seen, the query result code (which we will
 ** return from this function) will be set to FAIL.
 */
 ...CODE DELETED.....

/* Clean up the command handle used */
 if (retcode == CS_END_RESULTS)
 {
      retcode = ct_cmd_drop(cmd);
      if (retcode != CS_SUCCEED)
      {
           query_code = CS_FAIL;
      }
```

```
        }
    else
        {
            (void)ct_cmd_drop(cmd);
            query_code = CS_FAIL;
        }
    return query_code;
}
```

This code excerpt is from the *exutils.c* sample program.

Usage

- ct_send sends a command over the network for the server to execute.
- Before calling ct_send, the application must specify the type of command and the data needed for its execution. Once this step is done, the command structure is said to be **initiated**. The routines ct_command, ct_cursor, or ct_dynamic initiate a command.

  See Chapter 5, “Choosing Command Types,” in the Open Client *Client-Library/C Programmer's Guide* for a description of the server commands available to a Client-Library application.
- For most command types, ct_send can be called to resend a previously executed command after all the results from the previous execution have been handled. Exceptions are noted in the following section titled “Resending commands.”

First-time sends

- Sending a command to a server for first-time execution is a multi-step process:

  a Initiate the command by calling ct_command, ct_cursor, or ct_dynamic. These routines set up internal structures that are used in building a symbolic command stream to send to the server.

  b Pass parameters for the command (if required) by calling ct_param or ct_setparam once for each parameter that the command requires.

  Not all commands require parameters. For example, a remote procedure call command may or may not require parameters, depending on the stored procedure being called.

  c Send the command to the server by calling ct_send. ct_send writes the symbolic command stream onto the command structure’s parent connection.

d Handle the results of command execution by calling ct_results repeatedly until it no longer returns CS_SUCCEED. See “Results” on page 241 for a discussion of processing results.

- An application can call ct_cancel(CS_CANCEL_ALL) to cancel a command that has been initiated but not yet sent. But after an application has sent a command, it must call ct_results before calling ct_cancel to cancel the results of command execution.
- ct_send uses an asynchronous write and does not wait for a response from the server. An application must call ct_results to verify the success of the command and to set up the command results for processing.

Resending commands

- Most types of commands can be resent immediately after all the results of the previous command have been handled. The exceptions are:
  - Send-data commands initiated by ct_command(CS_SEND_DATA_CMD).
  - Send-bulk commands initiated by ct_command(CS_SEND_BULK_CMD)

  For all other types of commands, the application can resend the command with ct_send immediately after the application has processed all the results of the previous execution. Client-Library does not discard the initiated command information until the application initiates a new command with ct_command, ct_cursor, ct_dynamic, or ct_sendpassthru.
- In general, applications that resend commands should supply command parameters with ct_setparam rather than ct_param.
  - ct_setparam allows the application to change parameter values before resending the command. ct_setparam accepts pointers to program variables that contain parameter values. The variables’ contents are read by subsequent calls to ct_send. The binding between command parameters and program variables persists until the application initiates a new command with ct_command, ct_cursor, ct_dynamic, or ct_sendpassthru.
  - ct_param copies data from program variables before it returns. If ct_param is called to supply command parameters, the parameter values cannot be changed when the command is resent.

  If a parameter is effectively a literal value (that is, it will not change), then it is appropriate for the application to define the parameter with ct_param even if the command will be resent.

- An application can check the CS_HAVE_CMD property to see if a resendable command exists for the command structure. See “Have resendable command” on page 213 for a description of this property.
- Applications which resend commands may be able to use the CS_STICKY_BINDS property to eliminate redundant ct_bind calls. See “Persistent result bindings” on page 224 for a description of this property.

See also

ct_command, ct_cursor, ct_dynamic, ct_fetch, ct_param, ct_results, ct_setparam

ct_send_data

Description

Send a chunk of text or image data to the server.

Syntax

CS_RETCODE ct_send_data(cmd, buffer, buflen)

```
CS_COMMAND    *cmd;
CS_VOID       *buffer;
CS_INT        buflen;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

buffer

A pointer to the value to write to the server.

buflen

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

CS_NULLTERM is not a legal value for *buflen*.

Return value

ct_send_data returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_CANCELED	The send data operation was canceled.
CS_PENDING	Asynchronous network I/O is in effect. See “Asynchronous programming” on page 12.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

Example 1 The following fragment illustrates the call sequence to build and send a send-data command:

```
/*
 ** UpdateTextData()
 */

CS_STATIC CS_RETCODE
 UpdateTextData(connection, textdata, newdata)
 CS_CONNECTION connection;
 TEXT_DATA     textdata;
 char          *newdata;
 {
      CS_RETCODE  retcode;
      CS_INT      res_type;
      CS_COMMAND  *cmd;
      CS_INT      i;
      CS_TEXT     *txtptr;
      CS_INT       txtlen;

     /*
      ** Allocate a command handle to send the text with
      */
      ...CODE DELETED.....

     /*
      ** Inform Client-Library the next data sent will
      ** be used for a text or image update.
      */
      if ((retcode = ct_command(cmd, CS_SEND_DATA_CMD,
           NULL, CS_UNUSED, CS_COLUMN_DATA)) !=
           CS_SUCCEED)
      {
           ex_error("UpdateTextData: ct_command() \
                failed");
           return retcode;
      }

     /*
      ** Fill in the description information for the
      ** update and send it to Client-Library.
      */
      txtptr = (CS_TEXT *)newdata;
      txtlen = strlen(newdata);
      textdata->iodesc.total_txtlen = txtlen;
      textdata->iodesc.log_on_update = CS_TRUE;
      retcode = ct_data_info(cmd, CS_SET, CS_UNUSED,
           &textdata->iodesc);
      if (retcode != CS_SUCCEED)
```

```
 {
      ex_error("UpdateTextData: ct_data_info() \
           failed");
      return retcode;
 }
/*
 ** Send the text one byte at a time. This is not
 ** the best thing to do for performance reasons,
 ** but does demonstrate that ct_send_data()
 ** can handle arbitrary amounts of data.
 */
 for (i = 0; i < txtlen; i++, txtptr++)
 {
      retcode = ct_send_data(cmd, txtptr,
           (CS_INT)1);
      if (retcode != CS_SUCCEED)
      {
         ex_error("UpdateTextData: ct_send_data() \
                failed");
           return retcode;
      }
 }
/*
 ** ct_send_data() writes to internal network
 ** buffers. To insure that all the data is
 ** flushed to the server, a ct_send() is done.
 */
 if ((retcode = ct_send(cmd)) != CS_SUCCEED)
 {
      ex_error("UpdateTextData: ct_send() failed");
      return retcode;
 }
/* Process the results of the command */
 while ((retcode = ct_results(cmd, &res_type)) ==
      CS_SUCCEED)
 {
      switch ((int)res_type)
      {
         case CS_PARAM_RESULT:
         /*
         ** Retrieve a description of the
         ** parameter data. Only timestamp data is
         ** expected in this example.
         */
         retcode = ProcessTimestamp(cmd, textdata);
```

```
                if (retcode != CS_SUCCEED)
                {
                     ex_error("UpdateTextData: \
                          ProcessTimestamp() failed");
                     /*
                     ** Something failed, so cancel all
                     ** results.
                     */
                    ct_cancel(NULL, cmd, CS_CANCEL_ALL);
                     return retcode;
                }
                break;

               case CS_CMD_SUCCEED:
                case CS_CMD_DONE:
                /*
                ** This means that the command succeeded
                ** or is finished.
                */
                break;

               case CS_CMD_FAIL:
                /*
                ** The server encountered an error while
                ** processing our command.
                */
                ex_error("UpdateTextData: ct_results() \
                returned CS_CMD_FAIL");
                break;

               default:
                /*
                ** We got something unexpected.
                */
                ex_error("UpdateTextData: ct_results() \
                     returned unexpected result type”);
                /* Cancel all results */
                ct_cancel(NULL, cmd, CS_CANCEL_ALL);
                break;
           }
      }

     /*
      ** We're done processing results. Let's check the
      ** return value of ct_results() to see if
      ** everything went ok.
      */
      ...CODE DELETED.....
```

```
    return retcode;
}
```

This code excerpt is from the *getsend.c* sample program.

Example 2 The following fragment illustrates the call sequence to send partial update data:

```
/*
** UpdateTextData()
*/
CS_STATIC CS_RETCODE
UpdateTextData(connection, textdata, newdata)
CS_CONNECTION connection;
TEXT_DATA textdata;
char *newdata;
{
   CS_RETCODE retcode;
   CS_INT res_type;
   CS_COMMAND  *cmd;
   CS_INT      i;
   CS_TEXT     *txtptr;
   CS_INT       txtlen;
   /*
   ** Allocate a command handle to send the text with
   */
   ...CODE DELETED.....
   /*
   ** Inform Client-Library the next data sent will
   ** be used for a text or image update.
   */
   if ((retcode = ct_command(cmd, CS_SEND_DATA_CMD,
   NULL, CS_UNUSED, CS_COLUMN_DATA)) != CS_SUCCEED)
   {
      ex_error("UpdateTextData: ct_command() \
         failed");
      return retcode;
   }
   /*
   ** Fill in the description information for the
   ** update and send it to Client-Library.
   */
   txtptr = (CS_TEXT *)newdata;
   txtlen = strlen(newdata);
   textdata->iodesc.total_txtlen = txtlen;
   textdata->iodesc.log_on_update = CS_TRUE;
   /*
```

```
**  Insert newdata at offset 20.
*/
textdata->iodesc.iotype = CS_IOPARTIAL;
textdata->iodesc.offset = 20;
textdata->iodesc.delete_length = 0;
retcode = ct_data_info(cmd, CS_SET, CS_UNUSED,
&textdata->iodesc);
if (retcode != CS_SUCCEED)
{
   ex_error("UpdateTextData: ct_data_info() \
      failed");
   return retcode;
}
/*
** Send the text one byte at a time. This is not
** the best thing to do for performance reasons,
** but does demonstrate that ct_send_data()
** can handle arbitrary amounts of data.
*/
for (i = 0; i < txtlen; i++, txtptr++)
{
   retcode = ct_send_data(cmd, txtptr,(CS_INT)1);
   if (retcode != CS_SUCCEED)
   {
      ex_error("UpdateTextData: ct_send_data() \
         failed");
      return retcode;
   }
}
/*
** ct_send_data() writes to internal network
** buffers. To insure that all the data is
** flushed to the server, a ct_send() is done.
*/
if ((retcode = ct_send(cmd)) != CS_SUCCEED)
{
   ex_error("UpdateTextData: ct_send() failed");
   return retcode;
}
/* Process the results of the command */
while ((retcode = ct_results(cmd, &res_type)) ==
   CS_SUCCEED)
{
   switch ((int)res_type)
   {
      case CS_PARAM_RESULT:
```

```
                /*
                ** Retrieve a description of the
                ** parameter data. Only timestamp data is
                ** expected in this example.
                */
                retcode = ProcessTimestamp(cmd, textdata);
                if (retcode != CS_SUCCEED)
                {
                   ex_error("UpdateTextData: \
                     ProcessTimestamp() failed");
                   /*
                   ** Something failed, so cancel all
                   ** results.
                   */
                   ct_cancel(NULL, cmd, CS_CANCEL_ALL);
                   return retcode;
                }
                break;
             case CS_CMD_SUCCEED:
             case CS_CMD_DONE:
                /*
                ** This means that the command succeeded
                ** or is finished.
                */
                break;
             case CS_CMD_FAIL:
                /*
                ** The server encountered an error while
                ** processing our command.
                */
                ex_error("UpdateTextData: ct_results() \
                   returned CS_CMD_FAIL");
                break;
             default:
                /*
                ** We got something unexpected.
                */
                ex_error("UpdateTextData: ct_results() \
                   returned unexpected result type”);
                /* Cancel all results */
                ct_cancel(NULL, cmd, CS_CANCEL_ALL);
                break;
          }
       }
       /*
       ** We're done processing results. Let's check the
```

```
    ** return value of ct_results() to see if
    ** everything went ok.
    */
    ...CODE DELETED.....
    return retcode;
}
```

This code excerpt is from the *uctext.c* sample program.

Usage

- An application can use ct_send_data to write a text or image value to a database column providing the user has update privileges granted for the underlying table, which may be in a different database and not in the view. This writing operation is actually an update; that is, the column must have a value when ct_send_data is called to write a new value.

  This is because ct_send_data uses text timestamp information when writing to the column, and a column does not have a valid text timestamp until it contains a value. The value contained in the text or image column can be NULL, but the NULL must be entered explicitly with the SQL update statement.

- For information on the steps involved in using ct_send_data to update a text or image column, see “Updating a text or image column” on page 286. For information about sending partial updates with ct_send_data, see “Sending partial updates with ct_send_data” on page 290.

- To perform a send-data operation, an application must have a current I/O descriptor, or CS_IODESC structure, describing the column value that will be updated:

  - The *textptr* field of the CS_IODESC identifies the target column.
  - The *timestamp* field of the CS_IODESC is the text timestamp of the column value. If *timestamp* does not match the current database text timestamp for the value, the update operation will fail.
  - The *total_txtlen* field of the CS_IODESC indicates the total length, in bytes, of the column’s new value. An application must call ct_send_data in a loop to write exactly this number of bytes before calling ct_send to indicate the end of the text or image update operation.
  - The *log_on_update* of the CS_IODESC tells the server whether or not to log the update operation.
  - The *locale* field of the CS_IODESC points to a CS_LOCALE structure that contains localization information for the new value, if any.

A typical application will change only the values of the *locale*, *total_txtlen*, and *log_on_update* fields before using an I/O descriptor in an update operation, but an application that is updating the same column value multiple times will need to change the value of the *timestamp* field as well.

- A successful text or image update generates a parameter result set that contains the new text timestamp for the text or image value. If an application plans to update the text or image value a second time, it must save this new text timestamp and copy it into the CS_IODESC for the value before calling ct_data_info to define the CS_IODESC for the update operation.
- A text or image update operation is equivalent to a language command containing a Transact-SQL update statement.
- The command space identified by *cmd* must be idle before a text or image update operation is initiated. A command space is idle if there are no active commands, pending results, or open cursors in the space.

See also

ct_data_info, ct_get_data, “text and image data handling” on page 284

ct_sendpassthru

Description

Send a Tabular Data Stream (TDS) packet to a server.

Syntax

```
CS_RETCODE ct_sendpassthru (cmd, sendptr)

CS_COMMAND    *cmd;
CS_VOID       *sendptr;
```

Parameters

cmd

A pointer to a CS_COMMAND structure.

sendptr

A pointer to a buffer containing the TDS packet to be sent to the server.

Return value

ct_sendpassthru returns the following values:

Table 3-59: ct_sendpassthru return values

Return value	Meaning
CS_PASSTHRU_MORE	Packet sent successfully; more packets are available.
CS_PASSTHRU_EOM	Packet sent successfully; no more packets are available.
CS_FAIL	The routine failed.
CS_CANCELLED	The routine was cancelled.
CS_PENDING	Asynchronous network I/O is in effect.See "Asynchronous programming" on page 12.
CS_BUSY	An asynchronous operation is already pending for this connection. See "Asynchronous programming" on page 12.

Usage

- TDS is a communications protocol used for the transfer of requests and request results between clients and servers. Under ordinary circumstances, non-gateway applications do not have to deal with TDS, because Client-Library manages the data stream.
- ct_recvpassthru and ct_sendpassthru are useful in gateway applications. When an application serves as the intermediary between two parties (such as a client and a remote server, or two servers), it can use these routines to pass the TDS stream from one server to the other, eliminating the process of interpreting the information and re-encoding it.
- ct_sendpassthru sends a packet of bytes from the **sendptr* buffer. Most commonly, *sendptr* will be **recvptr* as returned by srv_recvpassthru. *sendptr* can also be the address of a user-allocated buffer containing the packet to send.
- Default packet sizes vary by platform. On most platforms, a packet has a default size of 512 bytes. A connection can change its packet size through ct_con_props.
- ct_sendpassthru returns CS_PASSTHRU_EOM if the TDS packet in the buffer is marked EOM (End Of Message). If the TDS packet is not marked EOM, ct_sendpassthru returns CS_PASSTHRU_MORE.
- A connection which is being used for a passthrough operation cannot be used for any other Client-Library function until CS_PASSTHRU_EOM has been received.

See also

ct_getloginfo, ct_recvpassthru, ct_setloginfo

ct_setloginfo

Description

Transfer TDS login response information from a CS_LOGINFO structure to a CS_CONNECTION structure.

Syntax

```
CS_RETCODE ct_setloginfo (connection, loginfo)

 CS_CONNECTION      *connection;
 CS_LOGINFO          *loginfo;
```

Parameters

connection

A pointer to a CS_CONNECTION structure. A CS_CONNECTION structure contains information about a particular client/server connection.

loginfo

A pointer to a CS_LOGINFO structure.

Return value

ct_setloginfo returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Usage

- TDS (Tabular Data Stream) is a communications protocol used for the transfer of requests and request results between Sybase clients and servers.
- Because ct_setloginfo frees the CS_LOGINFO structure after transferring the TDS information, an application cannot use the CS_LOGINFO again. An application can get a new CS_LOGINFO by calling ct_getloginfo.
- There are two reasons an application might call ct_setloginfo:
  - If it is an Open Server gateway application using TDS passthrough.
  - To copy login properties from an open connection to a newly allocated connection structure.

Note Do not call ct_setloginfo from within a completion callback routine. ct_setloginfo calls system-level memory functions which may not be reentrant.

TDS passthrough

- When a client connects directly to a server, the two programs negotiate the TDS format they will use to send and receive data. When a gateway application uses TDS passthrough, the gateway forwards TDS packets between the client and a remote server without examining or processing them. For this reason, the remote server and the client must agree on a TDS format to use.
- ct_setloginfo is the second of four calls, two of them Server Library calls, that allow a client and a remote server to negotiate a TDS format. The calls, which can be made only in an Open Server SRV_CONNECT event handler, are:
  - a srv_getloginfo to allocate a CS_LOGINFO structure and fill it with TDS information from a client login request.
  - b ct_setloginfo to transfer the TDS information retrieved in step 1 from the CS_LOGINFO structure to a Client-Library CS_CONNECTION structure. The gateway uses this CS_CONNECTION structure in the ct_connect call which establishes its connection with the remote server.
  - c ct_getloginfo to transfer the remote server's response to the client's TDS information from the CS_CONNECTION structure into a newly allocated CS_LOGINFO structure.
  - d srv_setloginfo to send the remote server's response, retrieved in step 3, to the client.

Copying login properties

For information on using ct_setloginfo to copy login properties from an open connection to a newly allocated connection structure, see "Copying login properties" on page 183.

See also

ct_getloginfo, ct_recvpassthru, ct_sendpassthru

ct_setparam

Description

Specify source variables from which ct_send reads input parameter values for a server command.

Syntax

```
CS_RETCODE ct_setparam(cmd, datafmt, data,
        datalenp, indp)
```

```
CS_COMMAND *cmd;
CS_DATAFMT    *datafmt;
CS_VOID       *data;
CS_INT        *datalenp;
CS_SMALLINT   *indp;
```

Parameters

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

datafmt

A pointer to a CS_DATAFMT structure that describes the parameter. ct_setparam copies the contents of **datafmt* before returning. Client-Library does not reference *datafmt* afterwards.

data

The address of a value buffer. Client-Library reads the parameter's current value from **data* during subsequent calls to ct_send.

There are three ways to indicate a parameter with a null value:

- Set * *indp* as -1 before calling ct_send. In this case, ct_send ignores **data* and .
- Set **datalenp* to 0 before calling ct_send.
- Call ct_setparam with *data*, *datalenp*, and *indp* as NULL.

datalenp

The address of an integer variable that specifies the length, in bytes, of parameter values in **data*, or NULL if values for this parameter do not vary in length.

If *datalenp* is not NULL, subsequent ct_send calls read the current value's length from **datalenp*. A length of 0 indicates a null value.

If *datalenp* is NULL and *data* is not, *datafmt–>maxlength* specifies the length of all non-null values for this parameter. When *datalenp* is NULL, an indicator variable must be used to indicate null parameter values for subsequent calls to ct_send.

indp

The address of a CS_SMALLINT variable whose value indicates whether the parameter's current value is NULL. To indicate a parameter with a null value, set **indp* as -1. If **indp* is -1, ct_send ignores **data* and **datalenp*.

Return value

ct_setparam returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.

Return value	Meaning
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

The example below shows ct_setparam being used in code that declares, opens, and reopens a cursor that takes parameters.

Example: ct_setparam *for reopening a cursor*

```
/*
 ** Data structures to describe a parameter and a cursor.
 */
 typedef struct _langparam
 {
         CS_CHAR         *name;
         CS_INT          type;
         CS_INT          len;
         CS_INT          maxlen;
         CS_VOID         *data;
         CS_SMALLINT     indicator;
 } LANGPARAM;

typedef struct _cur_control
 {
         CS_CHAR         *name;
         CS_CHAR         *query;
         LANGPARAM       *params;
         CS_INT          numparams;
 } CUR_CONTROL;

/*
 ** Static data for a parameterized cursor body.
 */
 CS_STATIC CS_MONEY  PriceVal;
 CS_STATIC CS_INT    SalesVal;
 CS_STATIC LANGPARAM Params [] =
 {
         { "@price_val", CS_MONEY_TYPE,
            CS_SIZEOF(CS_MONEY), CS_SIZEOF(CS_MONEY),
            (CS_VOID *)&PriceVal, 0
         },
         { "@sales_val", CS_INT_TYPE,
            CS_SIZEOF(CS_INT), CS_SIZEOF(CS_INT),
            (CS_VOID *)&SalesVal, 0
         },
 };
```

```
#define NUMPARAMS  (CS_SIZEOF(Params) / CS_SIZEOF(LANGPARAM))

#define QUERY \
"select title_id, title, price, total_sales from titles \
 where price > @price_val and total_sales > @sales_val \
 for read only"

CS_STATIC CUR_CONTROL Cursor_Control =
{ "curly", QUERY, Params, NUMPARAMS };

/*
 ** OpenCursor() -- Declare and open a new cursor or reopen
 **         an existing cursor (which must have been originally
 **         declared and opened using this function).
 **
 **         If the open is successful, this function processes the cursor
 **         results up to the CS_CURSOR_RESULT result type value. In
 **         other words, the command handle is ready for
 **         ct_bind/ct_fetch/etc.
 **
 ** Parameters
 **   cmd -- CS_COMMAND handle for the new cursor.
 **   cur_control -- address of a CUR_CONTROL structure that contains
 **     the cursor body statement plus parameter formats and value
 **     areas.
 **
 **     If a first-time open is successful, OpenCursor() can be used to
 **     reopen the cursor with new parameter values.
 **
 **     For later opens, the cursor must be closed.
 **
 ** Returns
 **   CS_SUCCEED or CS_FAIL
 */

CS_RETCODE
 OpenCursor(cmd, cur_control)
 CS_COMMAND         *cmd;
 CUR_CONTROL        *cur_control;
 {
   CS_RETCODE          ret;
   CS_INT              i;
   CS_DATAFMT          dfmt;
   LANGPARAM          *params;
   CS_BOOL             have_restorable_cursor;

  /*
   ** Check whether a cursor-open command can be restored with this
```

```
 ** command handle.
 */
 ret = ct_cmd_props(cmd, CS_GET, CS_HAVE_CUROPEN,
                    &have_restorable_cursor, CS_UNUSED,
                    (CS_INT *)NULL);
 if (ret != CS_SUCCEED)
 {
   ex_error("OpenCursor: ct_cmd_props() failed!");
   return CS_FAIL;
 }
/*
 ** If CS_HAVE_CUROPEN is CS_FALSE, then this is a first-time open. So,
 ** we initiate a new declare command and bind to the parameter source
 ** variables in the CUR_CONTROL structure.
 */
 if (have_restorable_cursor != CS_TRUE)
 {
  /*
   ** Initiate the declare command.
   */
   ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
                   cur_control->name, CS_NULLTERM,
                   cur_control->query, CS_NULLTERM,
                   CS_UNUSED);
   if (ret != CS_SUCCEED)
   {
     ex_error("OpenCursor: Initiate-declare failed");
     return CS_FAIL;
   }
  /*
   ** Specify formats for the host language parameters in the cursor
   ** declare command.
   */
   params = cur_control->params;
   (CS_VOID *)memset(&dfmt, 0, sizeof(dfmt));
   dfmt.status = CS_INPUTVALUE;
  for (i = 0; i < cur_control->numparams; i++)
   {
     dfmt.datatype = params[i].type;
     dfmt.maxlength = params[i].maxlen;
     strcpy(dfmt.name, params[i].name);
     dfmt.namelen = strlen(dfmt.name);
    ret = ct_setparam(cmd, &dfmt,
                      (CS_VOID *)NULL, (CS_INT *)NULL,
```

```
                        (CS_SMALLINT *)NULL);
        if (ret != CS_SUCCEED)
        {
          ex_error("OpenCursor: ct_setparam() failed");
          return CS_FAIL;
        }
      }
    }

  /*
   ** Initiate or restore the cursor-open command.
   **
   ** The first time we open the cursor, this call initiates an
   ** open-cursor command which gets batched with the declare command.
   ** Since there is no cursor to restore, ct_cursor ignores the
   ** CS_RESTORE_OPEN option.
   **
   ** The second (and later) times we open the cursor, this call
   ** restores the cursor-open command so that we can send it again.
   ** The declare-cursor command (originally batched with the open
   ** command) is not restored.
   */
   ret = ct_cursor(cmd, CS_CURSOR_OPEN,
                   (CS_CHAR *)NULL, CS_UNUSED,
                   (CS_CHAR *)NULL, CS_UNUSED,
                   CS_RESTORE_OPEN);
   if (ret != CS_SUCCEED)
   {
     ex_error("OpenCursor: Initiate-open failed.");
     return CS_FAIL;
   }

  /*
   ** For the first-time open, supply the address of variables that have
   ** values for the cursor parameters. These variables will be read by
   ** ct_send.
   **
   ** The second (and later) times we open the cursor, we don't have to
   ** call ct_setparam here -- the parameter bindings were restored by
   ** ct_cursor(OPEN, RESTORE_OPEN).
   **
   ** In either case, we assume that our caller has already set the
   ** desired values, lengths, and indicators.
   */
   for (i = 0;
        ((have_restorable_cursor != CS_TRUE) &&
         (i < cur_control->numparams));
```

```
        i++)
    {
      dfmt.datatype = params[i].type;
      dfmt.maxlength = params[i].maxlen;
      strcpy(dfmt.name, params[i].name);
      dfmt.namelen = strlen(dfmt.name);

     ret = ct_setparam(cmd, &dfmt,
                        params[i].data, &params[i].len,
                        &params[i].indicator);
      if (ret != CS_SUCCEED)
      {
        ex_error("OpenCursor: ct_setparam() failed");
        return CS_FAIL;
      }
    }

   /*
    ** Send the command batch.
    */
    ret = ct_send(cmd);
    if (ret != CS_SUCCEED)
    {
      ex_error("OpenCursor: ct_send() failed.");
      return CS_FAIL;
    }

   /*
    ** GetToCursorRows() calls ct_results() until cursor rows are
    ** fetchable on the command structure. GetToCursorRows() fails if
    ** the declare or open command fails on the server.
    */
    ret = GetToCursorRows(cmd);
    if (ret != CS_SUCCEED)
    {
      ex_error("OpenCursor: Cursor could not be opened.");
      return CS_FAIL;
    }

   return CS_SUCCEED;
  } /* OpenCursor() */

 /*
  ** GetToCursorRows() -- Flush results from a cursor-open command
  **   batch until ct_results returns a CS_CURSOR_RESULT result type.
  **
  ** Parameters
  **   cmd -- The command handle to read results from.
  **
```

```
 ** Returns
 **   CS_SUCCEED -- Cursor rows are ready to be fetched.
 **   CS_FAIL -- Failure. Could be due to any of the following:
 **     - No cursor results in the results stream.
 **     - Other kinds of fetchable results in the results stream.
 **     - ct_results failure.
 */

CS_STATIC CS_RETCODE
 GetToCursorRows(cmd)
 CS_COMMAND         *cmd;
 {
   CS_RETCODE          results_ret;
   CS_RETCODE          ret;
   CS_INT              result_type = CS_END_RESULTS;
   CS_BOOL             failing = CS_FALSE;
   CS_INT              intval;
   CS_CHAR             scratch[512];

  while (((results_ret = ct_results(cmd, &result_type)) == CS_SUCCEED)
          && (result_type != CS_CURSOR_RESULT))
   {
     switch ((int)result_type)
     {
     case CS_CMD_SUCCEED:
     case CS_CMD_DONE:
       break;

    case CS_CMD_FAIL:
       /*
       ** Declare or open failed on the server.
       */
       ret = ct_res_info(cmd, CS_CMD_NUMBER, (CS_VOID *)&intval,
                         CS_UNUSED, (CS_INT *)NULL);
       if (ret == CS_SUCCEED)
       {
         sprintf(scratch, “Command %ld failed”, (long)intval);
         ex_error(scratch);
       }
       failing = CS_TRUE;
       break;

    default:
       /*
       ** Nothing else is expected. Just return fail and let the caller
       ** decide how to clean up.
       */
       ex_error(
```

```
            "Unexpected result types received for cursor declare/open.");
        return CS_FAIL;

    }
    }

   /*
    ** We are leaving the cursor results pending on the connection.
    */
    if (results_ret == CS_CANCELED)
    {

     /*
      ** Could happen if the connection has a timeout and the error
      ** handler did ct_cancel(CS_CANCEL_ATTN);
      */
      ex_error("Cursor declare/open was canceled.");
      failing = CS_TRUE;
    }
    else if (results_ret != CS_SUCCEED)
    {
      ex_error("Cursor declare/open: ct_results failed.");
      failing = CS_TRUE;
    }

   return (failing == CS_TRUE) ? CS_FAIL : CS_SUCCEED;
  } /* GetToCursorRows() */
```

Usage

- ct_setparam specifies program source variables for a server command's input parameter values.
- Initiating a command is the first step in executing it. Some commands require the application to define input parameters with ct_param or ct_setparam before calling ct_send to send the command to the server.
- ct_setparam and ct_param perform the same function, except:
  - ct_param copies the contents of program variables.
  - ct_setparam copies the address of program variables, and subsequent calls to ct_send read the contents of the variables. ct_setparam allows the application to change parameter values when resending a command. For a description of this feature, see "Resending commands" on page 580.

  Calls to ct_param and ct_setparam can be mixed.
- ct_setparam may be required:

- To supply input parameter values for a cursor-open or cursor-update command that was initiated with ct_cursor, a language, message, or RPC command that was initiated with ct_command, or a dynamic-SQL execute command that was initiated with ct_dynamic. This use of ct_setparam is described under "Using ct_setparam to define input parameter sources" on page 601.
- To define the formats of host language variable formats for a cursor-declare command that was initiated with ct_cursor or ct_dynamic. This use of ct_setparam is described under "Using ct_setparam to define cursor parameter formats" on page 603. Cursor-declare commands cannot be resent, so there is no advantage to using ct_setparam rather than ct_param to define parameter formats.
- To define update columns for a cursor-declare command (initiated with ct_cursor or ct_dynamic). This use of ct_setparam is described under "Using ct_setparam to identify updatable cursor columns" on page 604. Note that cursor-declare commands can not be resent, so there is no advantage to using ct_setparam rather than ct_param to define update columns.

- Client-Library does not perform any conversion on parameters before passing them to the server. The application must supply parameters in the datatype required by the server. If necessary, the application can call cs_convert to convert parameter values into the required datatype.

Using *ct_setparam* to define input parameter sources

- An application may need to supply input parameter values for:
  - Client-Library cursor open commands
  - Client-Library cursor update commands
  - Dynamic SQL execute commands
  - Language commands
  - Message commands
  - Package commands
  - RPC commands
- ct_setparam creates a binding between the variables passed as **data*, **datalenp*, and **indp* and one command parameter. Subsequent calls to ct_send read the contents of these variables to determine whether the parameter value is null, and (if not null) the current value and length. A value is considered null if

- **datalen* is 0,
- **indp* is -1, or
- *data*, *datalenp*, and *indp* were all passed as NULL in the call to ct_setparam.

- The command parameter associated with each ct_setparam call is specified either by name or by position.
  - To specify by name, set *datafmt–>name* to the name of the parameter and *datafmt–>namelen* to the length of the name.
  - To specify by position, call ct_setparam in the order that the parameters occur in the SQL statement or stored procedure definition, with *datafmt–>namelen* as 0 for each call.

  All parameters must be specified by name, or all parameters must be specified by position.
- Client-Library cursor open commands require input parameter values when:
  - The cursor is declared with a Transact-SQL select statement containing host-language variables.
  - The cursor is declared with a Transact-SQL execute statement, and the called stored procedure requires parameters. In this case, **datafmt–>status* should be CS_INPUTVALUE to indicate an input parameter.
  - The cursor is declared on a prepared dynamic SQL statement that contains placeholders (indicated by the ? character).
- Client-Library cursor-update commands require input parameter values when the SQL text representing the update command contains host variables.
- Dynamic SQL execute commands require input parameter values when the prepared statement being executed contains dynamic parameter markers (indicated by the ? character).
- Language commands require input parameter values when the text of the language command contains host variables.
- Message commands require input parameters values when the message takes parameters.
- RPC and package commands require input parameter values when the stored procedure or package being executed takes parameters.

- Message, RPC, and package commands can take return parameters, indicated by passing *datafmt–>status* as CS_RETURN.
- A command that takes return parameters may generate a parameter result set that contains the return parameter values. See ct_results for a description of how an application retrieves values from a parameter result set.
- Table 3-60 lists the fields in **datafmt* that are used when passing input parameter values. A parameter's format cannot be changed after ct_setparam returns:

Table 3-60: CS_DATAFMT fields for passing input parameter values

Field	Description
name	The name of the parameter. *name* is ignored for dynamic SQL execute commands.
namelen	The length, in bytes, of *name*, or 0 to indicate an unnamed parameter. *namelen* is ignored for dynamic SQL execute commands.
datatype	The datatype of the input parameter value. All standard Client-Library types are valid except for CS_TEXT_TYPE, CS_IMAGE_TYPE, and Client-Library user-defined types. If *datatype* is CS_VARCHAR_TYPE or CS_VARBINARY_TYPE then *data* must point to a CS_VARCHAR or CS_VARBINARY structure.
maxlength	When passing return parameters for RPC commands, *maxlength* represents the maximum length, in bytes, of data to be returned for this parameter. If the ct_setparam *datalenp* parameter is passed as NULL, *maxlength* also specifies the length of all input values for the parameter. In this case, the maximum length for the corresponding return parameter data must agree with the length of input values.
status	Set to CS_RETURN when passing return parameters for RPC commands; otherwise set to CS_INPUTVALUE.

All other fields are ignored.

Using *ct_setparam* to define cursor parameter formats

- An application needs to define host variable formats for cursor declare commands when the cursor is declared with a select statement that contains host-language variables.

- Host variable formats are defined with ct_param or ct_setparam after calling ct_cursor(CS_CURSOR_DECLARE) to initiate the cursor-declare command. Cursor-declare commands cannot be resent, so ct_setparam offers no advantage over ct_param in this situation.
- To define the format of a host variable with ct_setparam, an application passes *datafmt–>status* as CS_INPUTVALUE, *datafmt–>datatype* as the datatype of the host variable, and *data*, *datalenp*, and *indp* as NULL.
- An application defines host variable formats as part of a cursor-declare command but does not specify data values for the variables until after initiating a cursor-open command for the cursor.
- When defining host variable formats, the host-language variables associated with each ct_setparam call can be specified either by name (with *datafmt –>name* and *datafmt –> namelen* set accordingly) or by the order of ct_setparam and ct_param calls (with *datafmt-> namelen* as 0). If one variable is named, all variables must be named.
- The following table lists the fields in **datafmt* that are used when defining host variable formats:

Table 3-61: CS_DATAFMT fields for defining host variable formats

Field	Description
name	The name of the host variable.
namelen	The length, in bytes, of *name*, or 0 to indicate an unnamed parameter.
datatype	The datatype of the host variable. All standard Client-Library types are valid except for CS_TEXT_TYPE, CS_IMAGE_TYPE, and Client-Library user-defined types.
status	CS_INPUTVALUE.

All other fields are ignored.

Using *ct_setparam* to identify updatable cursor columns

- Some servers require a client application to identify update columns for a cursor-declare command if some, but not all, of the columns are updatable. Update columns can be used to change values in underlying database tables.

- Adaptive Server does not require the application to specify update columns with additional ct_param/ct_setparam calls as described in this section. In fact, Adaptive Server ignores requests to identify update columns as described here. The application must use the Transact-SQL for read only or for update of syntax in the select statement to specify which (if any) columns are updatable (see the Adaptive Server Enterprise for a description of this syntax). Depending on its design, an Open Server application may require clients to specify a cursor's update columns as described in this section.
- If all of the cursor's columns are updatable, an application does not need to call ct_param or ct_setparam to specify them individually.
- To identify an update column for a cursor declare command, an application calls ct_param or ct_setparam with *datafmt–>status* as CS_UPDATECOL and **data* as the name of the column.
- The following table lists the fields in **datafmt* that are used when ct_setparam is called to identify update columns for a cursor-declare command:

Table 3-62: CS_DATAFMT fields for identifying update columns

Field name	Set to
status	CS_UPDATECOL

All other fields are ignored.

See also

ct_command, ct_cursor, ct_dynamic, ct_param, ct_send

ct_wakeup

Description

Call a connection's completion callback.

Syntax

```
CS_RETCODE ct_wakeup(connection, cmd, function,
            status)

CS_CONNECTION   *connection;
CS_COMMAND      *cmd;
CS_INT          function;
CS_RETCODE       status;
```

Parameters

connection

A pointer to the CS_CONNECTION structure whose completion callback will be called. A CS_CONNECTION structure contains information about a particular client/server connection.

Either *connection* or *cmd* must be non-NULL.

If *connection* is supplied, its completion callback is called. If *connection* is NULL, *cmd*'s parent connection's completion callback is called.

If *connection* is supplied, it is passed as the *connection* parameter to the completion callback. If *connection* is NULL, *cmd*'s parent connection is passed as the *connection* parameter to the completion callback.

cmd

A pointer to the CS_COMMAND structure managing a client/server operation.

Either *connection* or *cmd* must be non-NULL.

If *connection* is NULL, *cmd*'s parent connection's completion callback is called.

cmd is passed as the *command* parameter to the completion callback. If *cmd* is NULL then NULL is passed for the *command* parameter.

function

A symbolic value indicating which routine has completed. *function* can be a user-defined value. *function* is passed as the *function* parameter to the completion callback. Table 3-63 lists the symbolic values that are legal for *function*:

Table 3-63: Values for ct_wakeup function parameter

Value of function	Meaning
BLK_ROWXFER	blk_rowxfer has completed.
BLK_SENDROW	blk_sendrow has completed.
BLK_SENDTEXT	blk_sendtext has completed.
BLK_TEXTXFER	blk_textxfer has completed
CT_CANCEL	ct_cancel has completed.
CT_CLOSE	ct_close has completed.
CT_CONNECT	ct_connect has completed.
CT_DS_LOOKUP	ct_ds_lookup has completed.
CT_FETCH	ct_fetch has completed.
CT_GET_DATA	ct_get_data has completed.
CT_OPTIONS	ct_options has completed.
CT_RECVPASSTHRU	ct_recvpassthru has completed.
CT_RESULTS	ct_results has completed.
CT_SEND	ct_send has completed.
CT_SEND_DATA	ct_send_data has completed.
CT_SENDPASSTHRU	ct_sendpassthru has completed.
A user-defined value. This value must be greater than or equal to CT_USER_FUNC.	A user-defined function has completed.

status

The return status of the completed routine. This value is passed as the *status* parameter to the completion callback.

Return value

ct_wakeup returns the following values:

Return value	Meaning
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_BUSY	An asynchronous operation is already pending for this connection. See “Asynchronous programming” on page 12.

Examples

```
...CODE DELETED.....
/* Force a wakeup on the connection handle */
 retstat = ct_wakeup(connection, NULL,
     EX_ASYNC_QUERY, status);
 if (retstat != CS_SUCCEED)
    {
```

```
    return retstat;
}

...CODE DELETED.....
```

This code excerpt is from the *ex_alib.c* sample program.

Usage

- ct_wakeup is intended for use in applications that create an asynchronous layer on top of Client-Library.
- An application cannot call ct_wakeup if the CS_DISABLE_POLL property is set to CS_TRUE.

See also

"Asynchronous programming" on page 12, "Callbacks" on page 24, ct_callback, ct_poll

Glossary

Adaptive Server Enterprise

A server in Sybase's client/server architecture. Adaptive Server manages multiple databases and multiple users, keeps track of the actual location of data on disks, maintains mapping of logical data description to physical data storage, and maintains data and procedure caches in memory.

Adaptive Server Enterprise was called "SQL Server" in versions prior to 11.5.

array

A structure composed of multiple identical variables that can be individually addressed.

array binding

The process of binding a result column to an array variable. At fetch time, multiple rows' worth of the column are copied into the variable.

asynchronous

Occurring at any time without regard to the main control flow of a program. Compare to **synchronous**. Client-Library has two asynchronous modes of operation, **deferred-asynchronous** and **fully asynchronous**.

asynchronous routine

In Client-Library, any routine that interacts with the network. The asynchronous routines are those that can return CS_PENDING.

batch

A group of commands or statements.

A Client-Library command batch is one or more Client-Library commands terminated by an application's call to ct_send. For example, an application can batch together commands to declare, set rows for, and open a cursor.

A Transact-SQL statement batch is one or more Transact-SQL statements submitted to Adaptive Server by means of a single Client-Library command or Embedded SQL statement.

browse mode

A method that DB-Library and Client-Library applications can use to browse through database rows, updating their values one row at a time. Cursors provide similar functionality and are generally more portable and flexible.

bulk copy

A network interface provided by Adaptive Server for high-speed transfer of data into database tables. The bcp utility allows administrators to copy data in and out of databases through the bulk copy interface. Client-Library and Server-Library programmers can use **Bulk-Library** to access this interface. (DB-Library programs must use the Bulk-Copy special library built into DB-Library.

bulk descriptor structure

A hidden control structure (CS_BLKDESC) used by Bulk-Library to manage bulk copy operations. See also **Bulk-Library**, **bulk copy**.

Bulk-Library

A collection of routines that allow Client-Library and Server-Library applications to access the Adaptive Server bulk copy interface. See also **bulk copy**.

bylist

A result set sorted into subgroups. A bylist is generated by using the compute clause with the keyword by, followed by a list of columns.

callback

A routine that Open Client or Open Server calls in response to a triggering event, known as a callback event.

callback error handling

In Client-Library applications, a method of handling errors where the program installs **callback** functions to be called when Client-Library or CS-Library detects an error or when the server has sent a server message. See also **client message callback**, **server message callback**, **CS-Library error handler**, **inline error handling**.

callback event

In Open Client and Open Server, an occurrence that triggers a callback routine.

capabilities

The set of features that are supported by the version of the **TDS** communication protocol that is used for a client/server connection. Client-Library applications can call ct_capability to check whether a connection supports a particular type of client request or server response. A client application can also prevent certain types of server responses by calling ct_capability before a connection is opened. Capabilities are determined when a connection is opened and cannot be changed afterwards. See also **options**, **properties**.

character set

A set of specific (usually standardized) characters with an encoding scheme that uniquely defines each character. ASCII and ISO 8859-1 (Latin 1) are two common character sets. See also **character set conversion**, **client character set**.

character set conversion

Changing the encoding scheme of a set of characters on the way into or out of a server. Conversion is used when a server and a client communicating with it use a different **character set**. For example, if a server uses ISO 8859-1 and a client uses Code Page 850, character set conversion must be turned on so that both server and client interpret the data passing back and forth in the same way.

client

In client/server systems, the client is the part of the system that sends requests to servers and processes the results of those requests.

client character set

In a client/server system, the **character set** used by the client application. The client character set can differ from the character set used by the server. See also **character set conversion**.

Client-Library

Part of Open Client, a collection of routines for use in writing client applications. Client-Library is new in System 10 and designed to accommodate cursors, distributed network services, and other advanced features in the Sybase System 10 and later product lines. See also **CS-Library**, **Bulk-Library**.

client message

An error or informational message generated by Client-Library to inform the application of an error or exceptional condition. See also **client message callback**, **inline error handling**, **callback error handling**.

client message callback

An application routine that is called each time Client-Library generates a client message to describe an error or unusual condition. See also **callback error handling.**

code set

See **character set**.

collating sequence

See **sort order**.

command

In Client-Library, a server request initiated by an application's call to ct_command, ct_dynamic, or ct_cursor and terminated by the application's call to ct_send.

command structure

A hidden Client-Library control structure that Client-Library applications use to send commands and process results.

common name

A name that is unique only among entries that have the same parent node. See **fully qualified name**.

completion callback

In Client-Library applications, a type of application **callback** routine. On **fully asynchronous** connections, Client-Library automatically invokes the application's completion callback to communicate the **completion status** of each call to an **asynchronous routine**.

completion status

In Client-Library applications, the final return status of a call to an **asynchronous routine**. On synchronous connections, asynchronous routines block until all required network interaction is complete, then return the completion status directly. On asynchronous connections, asynchronous routines return CS_PENDING immediately, and the application must determine the completion status by polling or through a **completion callback**. See also **synchronous**, **deferred-asynchronous**, **fully asynchronous**.

connection structure

A hidden Client-Library control structure that defines a client/server connection within a context. See also **command structure**.

context structure

A CS-Library hidden structure that defines an application "context," or operating environment, within a Client-Library or Open Server application. The **CS-Library** routines cs_ctx_alloc and cs_ctx_drop allocate and drop a context structure.

conversion

The act of converting a data value from one representation to another. Conversion can yield a new value with a different datatype or, for character-to-character conversions, a new value with a different format or in a different **character set**. See also **character set conversion**.

credential token

A network-based authentication, in which users prove their identity to the network security system before the connection attempt is made. Client-Library then obtains a credential token from the security mechanism and sends it to the server in lieu of a password.

critical section

In a multithreaded application, sections of code that cannot execute simultaneously in multiple threads. Typically, a critical section is code that accesses a resource shared by multiple threads. Critical sections that access the same shared resource are said to be "related." See also **thread serialization**, **thread-safe**.

CS-Library

Included with both the Open Client and Open Server products, a collection of utility routines that are useful to both Client-Library and Server-Library applications. See also **context structure**.

CS-Library error handler

An application routine that is called each time CS-Library generates an error message to describe a error or exceptional condition. A CS-Library error handler is required in a **Server-Library** application and recommended in a **Client-Library** application. See also **callback error handling**.

current row

A row to which a cursor points. A fetch against a cursor retrieves the current row. See also **cursor**.

cursor

A symbolic name that is associated with a select statement. The cursor associates a “row pointer” with the set of rows matching the select conditions.

In Transact-SQL, a language cursor is a cursor declared with a declare cursor language command and scrolled with fetch language commands.

In Client-Library, a Client-Library cursor is a server object created with ct_cursor(CS_CURSOR_DECLARE). An application scrolls a Client-Library cursor with ct_fetch.

In Embedded SQL, a cursor is a data selector that passes multiple rows of data to the host program, one row at a time.

See also **scrollable cursor**.

database

A set of related data tables and other database objects that are organized to serve a specific purpose.

datatype

A defining attribute that describes the values and operations that are legal for a variable.

DB-Library

Part of Open Client, DB-Library is a self-contained collection of routines for use in writing client applications. DB-Library provides source-code compatibility for older Open Client applications that are written in DB-Library.

deadlock

1. In Adaptive Server, a situation that arises when two users, each having a lock on one piece of data, attempt to acquire a lock on the other’s piece of data. Adaptive Server detects deadlocks and resolves them by killing one user’s process.

2. In a multithreaded application, a situation where two threads, each having control of a serialization primitive, attempt to lock the serialization primitive held by the other. Deadlock can freeze a multithreaded application.

default

1. In an Open Client or Open Server application, the value, option, or behavior that Open Client and Open Server products use when none is specified.

2. In Transact-SQL, the value inserted for a column when an insert statement does not specify a value.

default database

The database that a user is in by default when he or she logs into a database server.

default language

1. The language that Open Client and Open Server products use when an application does no localization. The default language is determined by the “default” entry in the locales file.

2. The language that Adaptive Server uses for messages and prompts when a user has not chosen a language.

deferred-asynchronous

An asynchronous mode of operation for Client-Library connections where an application must poll for the completion status of each call to an asynchronous routine. Compare to **fully asynchronous**.

descriptor area

The area that a database management system (DBMS) uses to store information about dynamic parameters in a dynamic SQL statement.

directory

A dictionary that associates unique names with stored information about network entities such as servers, printers, or users. Directory access requires a **directory service provider**. See also **interfaces file**.

directory driver

Converts directory entries from their native storage format into the Server Directory Object format.

directory entry

Contains stored information associated with a given **fully qualified name**.

directory object class

A specification for the set of attributes (data) stored in a **directory entry**.

directory object structure

In a Client-Library application, a hidden structure (datatype CS_DS_OBJECT) that contains a copy of a **directory entry** that was read through a call to the ct_ds_lookup routine.

directory service

Sometimes called a naming service, manages creation, modification, and retrieval of directory entries.

directory service provider

System software that provides access to a **directory** for applications. For some platforms such as Windows, a directory service provider is built into the operating system. On other platforms, the system can be configured to use a third-party provider such as DCE.

DIT base

In directories that have an inverted-tree structure, the name of an interior node. The DIT base name is combined with partially qualified names to create a fully qualified name. See also **directory**, **fully qualified name**.

Dynamic SQL

Allows an Embedded SQL or Client-Library application to associate a name with a prepared SQL statement. Once prepared, the SQL statement can be executed repeatedly by name and can contain variables whose values are determined at execution time. In Adaptive Server, prepared dynamic SQL statements are dropped automatically when a user disconnects. Compare to **stored procedure**.

error message

A message that an Open Client and Open Server product issues when it detects an error condition.

event

An occurrence that prompts a Server-Library application to take certain actions. Client commands and certain commands within Open Server application code can trigger events. When an event occurs, Server-Library calls either the appropriate event-handling routine in the application code or the appropriate default event handler. See also **event handler**, **event-driven programming**.

event-driven programming

The programming style for Open Server applications. The application provides event handlers for each class of event, and the Open Server thread scheduler "dispatches" events by calling the application's event handlers. See also **event**, **event handler**, **Open Server thread**.

event handler

In Open Server, a routine that processes an event. An Open Server application can use the default handlers Open Server provides or can install custom event handlers. See also **event**, **event-driven programming**.

execute cursor

A cursor declared with a stored procedure.

exposed structure

A structure whose internals are exposed to Open Client and Open Server programmers. Open Client and Open Server programmers can declare, manipulate, and deallocate exposed structures directly. The CS_DATAFMT structure is an example of an exposed structure.

extended transaction

In Embedded SQL, a transaction composed of multiple Embedded SQL statements.

FIPS

An acronym for Federal Information Processing Standards. If FIPS flagging is enabled, Adaptive Server or the Embedded SQL precompiler issues warnings when a statement using a non-standard extension to SQL is encountered.

fully asynchronous

An asynchronous mode of operation for Client-Library connections where the application is automatically notified when each call to an **asynchronous routine** completes. See also **deferred-asynchronous**, **signal-driven I/O**, **thread-driven I/O**.

fully qualified name

A name that uniquely and unambiguously identifies a **directory entry**. An entry's fully qualified name provides all the information that a **directory service provider** requires to find the entry.

gateway

An application that acts as an intermediary for clients and servers that cannot communicate directly. Acting as both client and server, a gateway application passes requests from a client to a server and returns results from the server to the client.

global name

An OID functions as a symbolic global name that means the same to all applications in a distributed environment. See **object identifier**.

hidden structure

A structure whose internals are hidden from Open Client and Open Server programmers. Open Client and Open Server programmers must use Open Client and Open Server routines to allocate, manipulate, and deallocate hidden structures. The CS_CONTEXT structure is an example of a hidden structure.

host language

The programming language in which an application is written.

host program

In Embedded SQL, the application program that contains the Embedded SQL code.

host variable

In Embedded SQL, a variable that enables data transfer between Adaptive Server and the application program. See also **indicator variable**, **input variable**, **output variable**, **result variable**, **status variable**.

indicator variable

A variable whose value indicates special conditions about another variable's value or about fetched data.

When used with an Embedded SQL host variable, an indicator variable indicates when a database value is null.

initiated command

In Client-Library applications, a command initiated if the type of command has been defined by ct_command, ct_cursor, or ct_dynamic, but the command has not yet been sent to the server with ct_send. See also **command**.

inline error handling

In Client-Library applications, a method of handling errors where the program tests for the occurrence of **client messages**, CS-Library errors, or **server messages** after each call to a CS-Library or Client-Library routine. Compare to **callback error handling**.

input variable

A variable that is used to pass information to a routine, a stored procedure, or Adaptive Server.

interfaces file

A file that maps server names to transport addresses. When a client application calls ct_connect or dbopen to connect to a server, Client-Library or DB-Library searches the *interfaces* file for the server's address. Client-Library can also use a **directory service** for this purpose instead of the *interfaces* file. Some platforms do not use the *interfaces* file. On these platforms, an alternate mechanism directs clients to server addresses.

interrupt-driven I/O

See **signal-driven I/O**.

isql script file

In Embedded SQL, one of the three files the precompiler can generate. An isql script file contains precompiler-generated stored procedures that are written in Transact-SQL.

key

A subset of row data that uniquely identifies a row. Key data uniquely describes the **current row** in an open cursor.

keytab file

The name and path to an operating system file.

keyword

A word or phrase that is reserved for exclusive use in Transact-SQL or Embedded SQL. Also called a “reserved word.”

listener

In an **Open Server** application, the internal Server-Library system thread that waits for client connection attempts and creates new threads to handle each client connection. A call to srv_init starts the listener.

listing file

In Embedded SQL, one of the three files the precompiler can generate. A listing file contains the input file’s source statements and informational, warning, and error messages.

locale name

A character string that represents a language and character set pair. Locale names are listed in the **locales file**. Sybase predefines some locale names and the **System Administrator** can define additional locale names and add them to the locales file.

locale structure

A CS-Library hidden structure that defines custom localization values for a Client-Library or Open Server application. An application can use a CS_LOCALE structure to define the language, character set, datepart ordering, and sort order it will use. The CS-Library routines cs_loc_alloc and cs_loc_drop allocate and drop a locale structure.

locales file

A file that maps locale names to language and character set pairs. Open Client and Open Server products search the locales file when loading localization information.

localization

The process of setting up an application to run in a particular native language environment. An application that is localized typically generates messages in a local language and character set and uses local datetime formats.

logical command

In a Client-Library application, defined as any command defined through ct_command, ct_dynamic, or ct_cursor, with the following exceptions:

- Each Transact-SQL select statement inside a stored procedure is a logical command. Other Transact-SQL statements inside stored procedures do not count as logical commands.
- Each Transact-SQL statement executed by a dynamic SQL command is a distinct logical command.
- Each Transact-SQL statement in a language command is a logical command.

login authentication

A security service that confirms that users are who they say they are by use of user names and passwords.

login name

The name a user uses to log in to an Adaptive Server. An Adaptive Server login name is valid if Adaptive Server has an entry for that user in the system table *syslogins*.

message number

A number that uniquely identifies an error message.

message queue

In Open Server, a linked list of message pointers through which threads communicate. Threads can write messages into and read messages from the queue.

multibyte character set

A character set that includes characters encoded using more than one byte. EUC JIS and Shift-JIS are examples of multibyte character sets.

multithreaded

A property of program code. Multithreaded code can execute concurrently on two or more threads and, therefore, must be **thread-safe**. See also **thread**.

mutex

A mutual exclusion semaphore. A mutex is a **serialization primitive** provided by Server-Library or an operating system thread interface. A mutex provides a method for multithreaded applications to serialize access to a resource shared by two or more threads. See also **native thread**, **Open Server thread**.

naming service

See **directory service provider**.

native thread

A **thread** whose existence and scheduling are managed by the host operating system. See also **thread scheduling**, **Open Server thread**.

negotiated properties

Certain login properties that the server can change during the login process, such as TDS version support properties.

null

1. With regard to data values, having no explicitly assigned value. NULL is not equivalent to zero nor to blank. A value of NULL is not considered to be greater than, less than, or equivalent to any other value, including another NULL value.

2. With regard to C language pointers, the special NULL address value that refers to no memory address.

object identifier

A string of decimal digits that uniquely names an object in a multi-vendor, multi-platform environment. OIDs provide a means to identify an item that might have different names in different environments. For example, the same character set can be named differently by different operating systems. See **global name**.

Object identifiers are encoded according to the Basic Encoding Rules (BER) defined by ISO 8825. All Sybase-defined OIDs begin with this prefix:

```
1.3.6.1.4.1.897
```

OID
See **object identifier**.

OID string
A character string that contains an **object identifier**. Client-Library and Server-Library applications use OID strings to represent object identifiers. The *cspublic.h* header file defines Sybase-specific OID strings.

Open Server
A Sybase product that provides tools and interfaces for creating custom servers. See also **Server-Library**.

Open Server application
A custom server constructed with **Open Server**. See also **Open Server thread**, **event-driven programming**.

Open Server thread
A path of execution through an **Open Server application** and library code and the path's associated stack space, state information, and event handlers. An Open Server thread is a thread whose existence and scheduling is managed by **Server-Library**. See also **thread**, **thread scheduling**, **native thread**.

options
Software that controls how Adaptive Server processes commands. Applications set, retrieve, or clear options by calling the ct_options Client-Library routine after a connection to a server has been opened. See also **capabilities**, **properties**.

output variable
In Embedded SQL, a variable that passes data from a stored procedure to an application program.

parameter
1. A variable that is used to pass data to and retrieve data from a routine.

2. An argument to a stored procedure.

passthrough mode
A mode in which a gateway relays Tabular Data Stream (TDS) packets between a client and a remote data source without unpacking the packets' contents.

placeholder
An indicator identified by a question mark (?), that acts like a variable in a prepared statement.

properties
Named values stored in a hidden structures. Context, connection, thread, and command structures have properties. A structure's properties determine how CS-Library, Client-Library, or Server-Library responds to calls that pass a pointer to the structure as a parameter. See also **capabilities**, **options**.

query
1. A data retrieval request; usually a select statement.

2. Any SQL statement that manipulates data.

registered procedure

In an Open Server application, an executable entity on the server that can be remotely called by clients. A registered procedure can be a C function in the Open Server application code, an internal Server-Library routine made available as a system registered procedure, or a "bodiless" registered procedure created by a client's call to the sp_regcreate system registered procedure. See also **registered procedure notifications**, **system registered procedure**, **remote procedure call**.

registered procedure notifications

An Open Server feature that allows a client to monitor the execution of a given **registered procedure**. A client requests to "watch" a registered procedure on the Open Server, and thereafter, the Open Server notifies the client when the procedure executes. Registered procedure notifications allow synchronization of clients in a distributed application. See also **system registered procedure**.

remote procedure call

1. One of two ways in which a client application can execute an Adaptive Server stored procedure. (The other is with a Transact-SQL execute statement.) A Client-Library application initiates a remote procedure call command by calling ct_command. A DB-Library application initiates a remote procedure call command by calling dbrpcinit.

2. A type of request that a client can make of an Open Server application. In response, Open Server either executes the corresponding registered procedure or calls the Open Server application's RPC event handler.

3. In Transact-SQL, a stored procedure that is executed on a different server from the server to which the user is connected.

request capabilities

Used by an application to determine what kinds of requests a server connection supports.

reserved word

See **keyword**.

response capabilities

Used by an application to prevent the server from sending a type of response that the application cannot process.

result data

An umbrella term for all the types of data that a server can return to an application.

result set

The form in which results are returned to the application. A result set contains only a single type of result data. Regular row and cursor-row result sets contain multiple rows of data, but other types of result sets contain at most a single row of data

result variable

In Embedded SQL, a variable that receives the results of a select or fetch statement.

security mechanism External software that provides security services for a connection.

select list The list of columns selected by a Transact-SQL select statement.

select-list id A numeric identifier for a column in the results of a Transact-SQL select statement. The first column in the select list has id 1, the second has id 2, and so forth. For example, in the query below, the select-list id of the *title* column is 1 and the select-list id of the *Units Sold* column is 3:

```
select title, price, "Units Sold" = total_sales from titles
```

See also **select list**.

serialization primitive A logical object and associated routines that allow serialization of access to shared resources. A **mutex** is an example of a serialization primitive. See also **native thread**, **Open Server thread**.

server In client/server systems, the part of the system that processes client requests and returns results to clients. A server can be an **Adaptive Server** or an **Open Server application**.

server directory object A generalized description of the logical content of directory entries that describe Sybase servers.

Server-Library A collection of routines for use in writing an **Open Server application**.

server message An error or informational message sent by a server to the client. The server may send server messages to the client to describe errors or unusual conditions that occur when the server is processing a command sent from the client. See also **server message callback**, **callback error handling**, **inline error handling**.

server message callback In a Client-Library application, a **callback** function installed to receive each **server message** sent by the server. See also **callback error handling**.

signal-driven I/O A platform specific method used by Client-Library to allow non-blocking network reads and writes. Internally, Client-Library installs its own internal system interrupt handler and interacts with the network using non-blocking system calls. Compare **thread-driven I/O**.

sort order Used to determine the order in which character data is sorted. Also called **collating sequence**.

SQLCA 1. In an Embedded SQL application, a SQLCA structure provides a communication path between Adaptive Server and the application program. After executing each SQL statement, the precompiler-generated source code stores return codes in the SQLCA.

2. In a Client-Library application, a SQLCA structure can be used by an application to retrieve Client-Library and server error and informational messages.

SQLCODE

1. In an Embedded SQL application, a SQLCODE structure provides a communication path between Adaptive Server and the application program. After executing each SQL statement, the precompiler-generated source code stores return codes in the SQLCODE. A SQLCODE can exist independently or as a variable within a SQLCA structure.

2. In a Client-Library application, a SQLCODE structure can be used by an application to retrieve Client-Library and server error and informational message codes.

scrollable cursor

Allows a current cursor position to be set anywhere in a result set. See also **cursor**.

statement

In Transact-SQL or Embedded SQL, an instruction that begins with a keyword. The keyword names the basic operation or command to be performed.

status variable

In Embedded SQL, a variable that receives the return status value of a stored procedure, thereby indicating the procedure's success or failure.

stored procedure

In Adaptive Server, a collection of SQL statements and optional control-of-flow statements stored under a name. Adaptive Server-supplied stored procedures are called **system stored procedures**.

synchronization primitive

A logical object and associated routines that allow synchronization of dependent actions performed by multiple threads. Synchronization primitives for native threads are provided by the host operating system (for example, a condition variable or barrier). Synchronization primitives for Open Server threads are provided by Server-Library (for example, a **message queue**). See also **native thread**, **Open Server thread**.

synchronous

Occurring at a predictable point in time determined wholly by the logic of main-line program code. Compare to **asynchronous**.

System Administrator

The user in charge of server system administration, including creating user accounts, assigning permissions, and creating new databases. In Adaptive Server, the System Administrator's login name is "sa".

system descriptor

In Embedded SQL, an area of memory that holds a description of variables used in Dynamic SQL statements.

system registered procedure

An internal **registered procedure** that Open Server supplies for managing registered procedure notifications and monitoring the server status. See also **Open Server**.

system stored procedures

Stored procedures that Adaptive Server supplies for use in system administration. These procedures are provided as shortcuts for retrieving information from system tables or as mechanisms for accomplishing database administration and other tasks that involve updating system tables.

target file

In Embedded SQL, one of the three files the precompiler can generate. A target file is similar to the original input file, except that all SQL statements are converted to Client-Library function calls.

TDS

(Tabular Data Stream) An application-level protocol that Sybase clients and servers use to communicate. It allows the transfer of requests and results between clients and servers. See also **capabilities**.

text pointer

A pointer, stored in database tables, in lieu of large text and image values.

text timestamp

A timestamp that is associated with each *text* or *image* column, to prevent competing applications from overwriting one another's modifications to the database.

thread

A path of execution through a program. See also **thread scheduling**, **multithreaded**, **native thread**, **Open Server thread**.

thread-driven I/O

A platform specific method used by Client-Library to allow non-blocking network reads and writes. Internally, Client-Library creates worker threads that interact with the network. The internal worker thread may block for reads or writes, but the user-application thread does not. Compare to **signal-driven I/O**.

thread-safe

A property of a routine or collection of routines that allows them to be safely executed by different threads in a multithreaded application. See also **thread serialization**, **thread synchronization**, **thread-unsafe**.

thread scheduling

The act of managing the concurrent execution of multiple threads. A thread scheduler uses a well-defined algorithm to periodically suspend one thread, save its state, and resume (or begin) execution of another thread. See also **native thread**, **Open Server thread**.

thread serialization

The use of serialization primitives in multithreaded code to ensure that the execution of related critical sections by different threads is mutually exclusive. Serialization consists of using serialization primitives to “protect” critical sections from simultaneous execution of related critical sections in different threads. In other words, serialization guarantees that once a critical section begins execution, its execution is not interleaved with the execution of a related critical section in a different thread. Serialization is commonly used to make code that accesses shared resources **thread-safe**. See also **critical section**, **serialization primitive**.

thread synchronization

The use of synchronization primitives to guarantee a specific order of execution of code executed by two or more threads. Synchronization ensures that dependent actions executed by separate threads are performed in the correct order. See also **synchronization primitive**, **native thread**, **Open Server thread**.

thread-unsafe

Not **thread-safe**. Thread-unsafe describes a property of program code that prohibits its execution by multiple threads. If thread-unsafe code executes simultaneously from multiple threads, it yields unpredictable behavior.

Transact-SQL

An enhanced version of the SQL database language. Applications can use Transact-SQL to communicate with Adaptive Server.

transaction

One or more server commands that are treated as a single unit. Commands within a transaction are committed as a group; that is, either all of them are committed or all of them are rolled back.

transaction mode

The manner in which Adaptive Server manages transactions. Adaptive Server supports two transaction modes: Transact-SQL mode (also called *unchained transactions*) and ANSI mode (also called *chained transactions*).

throughput

A measure of work done per unit of time. For example, the throughput of a **bulk copy** operation might be measured as the number of rows transferred per second.

updatable

Description of a cursor that an application intends to update using ct_cursor update commands.

user name

See **login name**.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X