Open Client^{TM} Client-Library / C Reference Manual

Open Client Release 10.0

Document ID: 32840-01-1000-04

Change Level: 1

Last Revised: August 5, 1994

Principal authorship: Pegler Swift, Stacia Sambar

Contributing authorship: Mike Allen, Otto Lind

Document ID: 32840-01-1000

Document Orders

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the fax number. All other international customers should contact their Sybase subsidiary or local distributor.

Upgrades are provided only at regularly scheduled software release dates.

Sybase Trademarks

Restricted Rights Legend

Table of Contents

List of Tables

Preface

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

Audience

The *Client-Library Reference Manual* is designed to serve as a reference manual for programmers who are writing Client-Library applications. It is written for application programmers familiar with the C programming language.

How to Use This Book

When writing a Client-Library application, use the *Client-Library Reference Manual* as a source of reference information.

Chapter 1, “Introducing Client-Library,” contains a brief introduction to Client-Library.

Chapter 2, “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.

Although there is some introductory material about application development in this manual, it is highly recommended that applications programmers read the *Client-Library Programmer’s Guide* before designing a Client-Library application.

Related Documents

The *Open Client/Server Installation Guide* explains how to install Client-Library.

The *Client-Library Programmer's 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™
- Client-Library and Server-Library™ bulk copy routines

The *Open Client/Server Supplement* contains platform-specific material for Open Client / Server products. This document includes information about:

- Open Client compatibility with pre-10.0 server releases
- The interfaces file
- Compiling and linking an application
- The example programs that are included on-line with Open Client / Server products
- Routines that have platform-specific behaviors
- Localization

Other Sources of Information

SYBASE® documents include a wide range of user guides and reference manuals which describe all aspects of the SYBASE relational database management system. Because application development can draw on a number of different parts of the SYBASE system, you may encounter most of the SYBASE document set at some time or another. A few manuals, however, will prove to be particularly useful:

- The *SQL Server Reference Manual* describes the Transact-SQL® database language, which an application uses to create and manipulate SYBASE SQL Server™ database objects.
- The *Open Client DB-Library Reference Manual* describes DB-Library™. Like Client-Library, DB-Library is a collection of routines for use in writing client applications.
- The *Open Server Server-Library Reference Manual* contains reference information for Open Server Server-Library, a collection of routines for use in writing Open Server™ applications.

- The *APT Workbench User's Guide* documents APT-Edit™, used to create forms and specify their processing, and APT-SQL™, a fourth-generation language for developing forms-based applications.
- APT Workbench™ reference manuals include:
  - The *APT-Edit Reference Manual*, which contains detailed information about the forms editor
  - The *APT-SQL Reference Manual*, which contains detailed information on the APT-SQL language
  - The *APT-Library/C Reference Manual*, which describes the library of C routines that give an application access to APT-Edit forms
- The *Data Workbench User's Guide* describes Data Workbench, a set of tools that provide forms-based, interactive access to SQL Server. Since Data Workbench uses the SYBASE forms run-time system, it can serve as a valuable example of how a forms-based application looks and feels.

Conventions

Client-Library routine syntax is shown in a bold, monospace font:

```
CS_RETCODE ct_init(context, version)

CS_CONTEXT       *context;
CS_INT           version;
```

Program text and computer output are shown in a monospace font:

```
ct_init(mycontext, CS_VERSION_100);
```

Structure names and symbolic constants are shown in small capital letters:

CS_CONTEXT, CS_SYNC_IO

Routine names and Transact-SQL keywords are written in a narrow, bold font:

ct_init, the **select** statement

Code Fragments

Code fragments in this book are taken from the on-line example programs that are included with Client-Library.

The example programs, and consequently the code fragments in this book, use **EX_***, **Ex_***, and **ex_*** #defines, variables, and routines.

These #defines, variables, and routines are part of the example programs but not a part of Client-Library.

If You Need Help

Help is available for your SYBASE software in the form of documentation, on-line help, and a Technical Support Center.

On-Line Help

If you have access to a 10.0 SQL Server, you can use **sp_syntax**, a SYBASE system procedure, to retrieve the syntax of Client-Library routines.

For information on how to install **sp_syntax**, see the *System Administration Guide Supplement* for your platform. For information on how to run **sp_syntax**, see its manual page in Volume 2 of the *SQL Server Reference Manual*.

Technical Support

Your company has designated someone with the authority to contact Sybase Technical Support. If you cannot resolve a problem using the information in the Sybase documentation, ask that person to contact Sybase Technical Support for you.

Introduction

1 Introducing Client-Library

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.

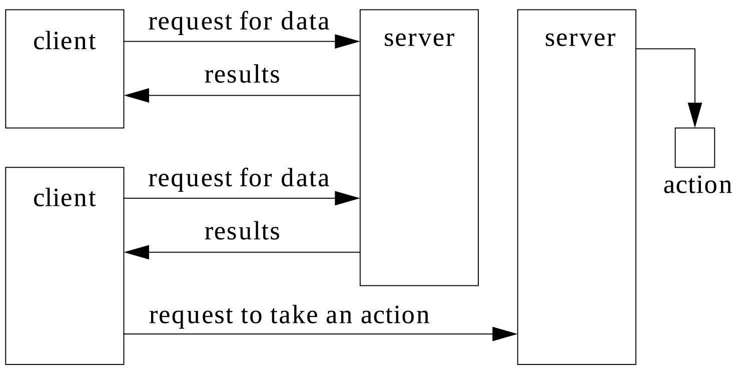


Figure 1-1: Client/Server architecture

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 varied 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:

- SQL Toolset products such as APT-Edit and Data Workbench
- Stand-alone utilities provided with SQL Server, such as **isql** and **bcp**
- Applications written using Open Client libraries
- Applications written using SYBASE Embedded SQL™

Types of Servers

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

- SYBASE SQL Server is a database server. SQL Servers manage information stored in one or more databases.
- SYBASE Open Server provides the tools and interfaces needed to create a custom server, also called an “Open Server application.”

An Open Server application can be any type of server. For example, an Open Server application 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.

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

- SQL Server and Open Server applications are both servers, responding to client requests.
- Clients communicate with both SQL 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. SQL 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. SQL Server is a database server, and understands only Transact-SQL.
- An Open Server application can communicate with "foreign" applications and servers that are not based on SYBASE protocols, as well as SYBASE applications and servers. SQL Server can communicate directly only with SYBASE applications and servers, although SQL Server can communicate with foreign applications and servers by using an Open Server gateway application as an intermediary.

The following diagram illustrates some of the different capabilities of SQL Server and Open Server applications:

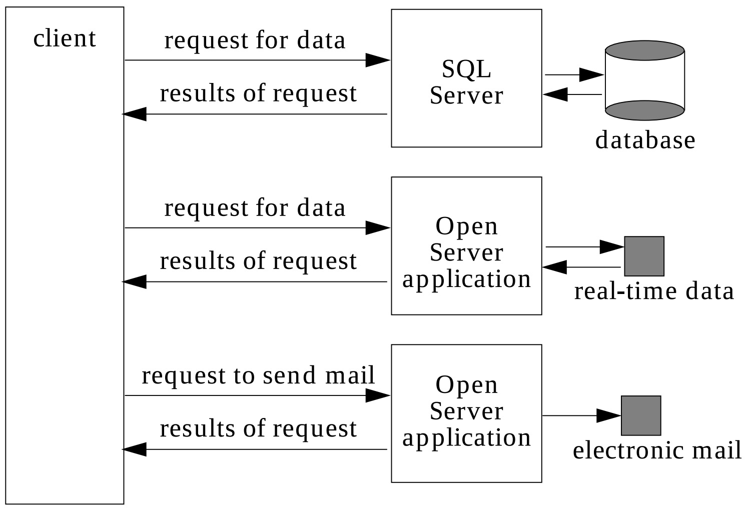


Figure 1-2: SQL Server and Open Server applications

The Open Client and Open Server Products

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

- SYBASE Open Client
- SYBASE Open Server

SYBASE Open Client

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

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

The programming interfaces component of Open Client is made up of libraries designed for use in writing client applications: Client-Library, DB-Library, and CS-Library. (Both Open Client and Open Server include CS-Library, which contains utility routines that are useful to both client and server applications.)

Open Client network services include Net-Library™, which provides support for specific network protocols, such as TCP / IP or DECnet.

SYBASE Open Server

SYBASE Open Server provides the tools and interfaces needed to create custom servers.

Like Open Client, Open Server has a programming interfaces component and a network services component.

The programming interfaces component of Open Server contains Server-Library and CS-Library. (Both Open Client and Open Server include CS-Library, which contains utility routines that are useful to both client and server applications.)

Open Server network services are transparent.

Application Calls to Libraries

The following diagram illustrates the Open Client and Open Server library calls that different types of applications might make. For example, a client application might include calls to Client-Library and CS-Library, while an application that acts as both client and server (for example, a gateway application) might include calls to Client-Library, CS-Library, and Server-Library:

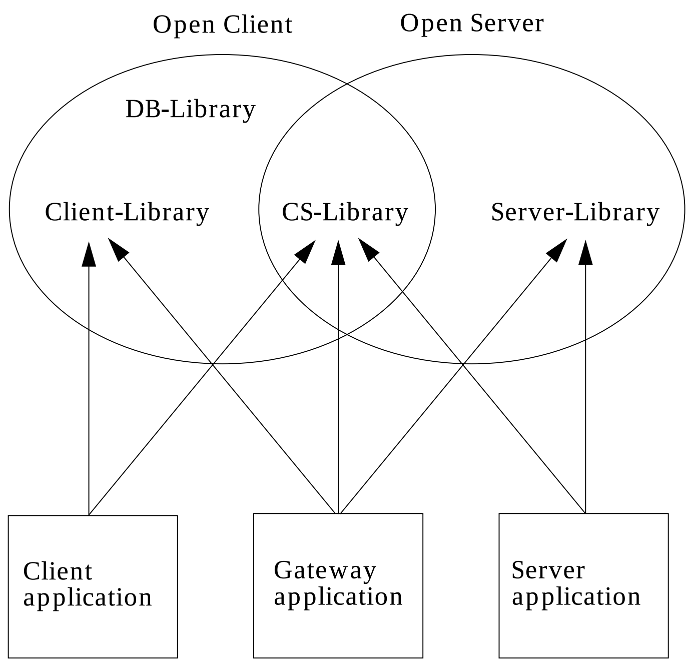


Figure 1-3: Application calls to libraries

The Open Client Libraries

The libraries that make up Open Client programming interfaces are:

- DB-Library, a collection of routines for use in writing client applications. DB-Library includes a bulk copy library and the two-phase commit special library.

- Client-Library, a collection of routines for use in writing client applications. Client-Library is a new library, designed to accommodate cursors and other advanced features in the SYBASE 10.0 product line.
- 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 which is allocated in CS-Library.

What Is in Client-Library?

Client-Library includes routines that send commands to a server and routines that process the results of those commands. Other routines set application properties, handle error conditions, and provide a variety of information about an application's interaction with a server.

Client-Library also contains a header file, *ctpublic.h*, that defines structures, types, and values used by Client-Library routines.

Client-Library is a Generic Interface

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

An Embedded SQL application includes SQL commands in-line. The host language precompiler processes the commands into calls to Open Client libraries. All SYBASE 10.0 precompilers use a run-time library composed solely of documented Client-Library and CS-Library calls.

In a sense, then, the precompiler transforms an Embedded SQL application into an Open Client library application.

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.

Using Client-Library

An application programmer writes a Client-Library program, using calls to Client-Library and CS-Library routines to set up structures, connect to servers, send commands, process results, and clean up. A Client-Library program is compiled and run in the same way as any other C language program.

Basic Control Structures

In order to send commands to a server, a Client-Library application must allocate three types of structures:

- A CS_CONTEXT structure, which defines a particular application "context", or operating environment
- A CS_CONNECTION structure, which defines a particular client / server connection
- A CS_COMMAND structure, which defines a "command space" in which commands are sent to a server

An application allocates these structures by calling the CS-Library and Client-Library routines **cs_ctx_alloc**, **ct_con_alloc**, and **ct_cmd_alloc**.

The general relationship between the three basic control structures is illustrated by the following diagram:

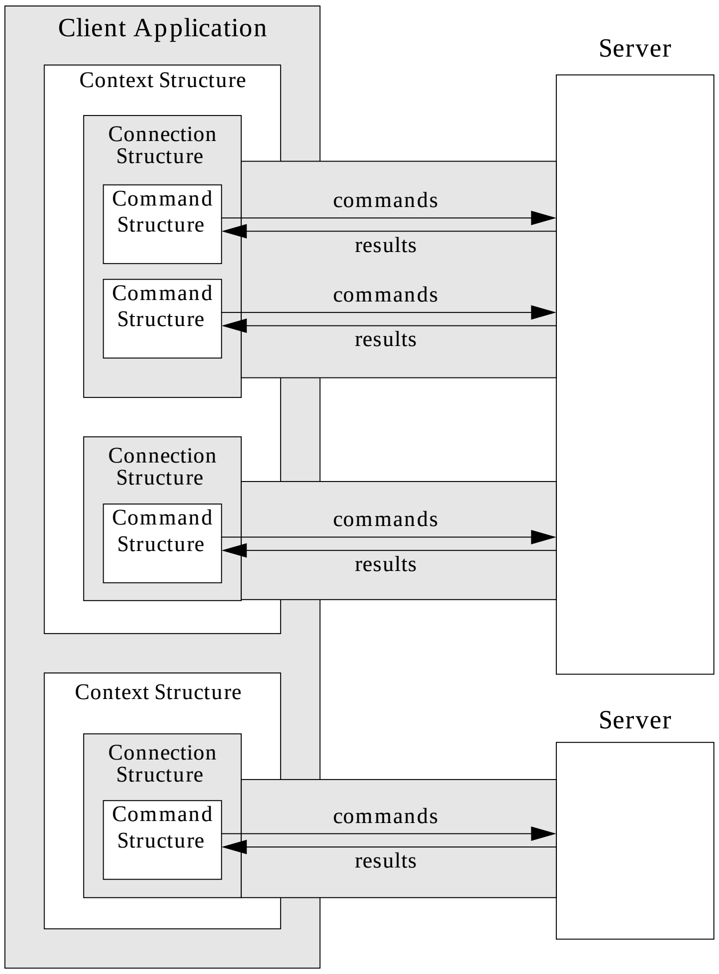


Figure 1-1: Relationship of control structures

Through these structures, an application sets up its environment, connects to servers, sends commands, and processes results.

For more information about these control structures, see **Structures** in Chapter 2, “Topics,” or the *Client-Library Programmer’s Guide*.

Steps in a Simple Program

On most platforms, a simple Client-Library program involves the following steps:

1. Set up the Client-Library programming environment.
2. Define error handling. Most applications will use callback routines to handle Client-Library and server error and informational messages. Some applications, however, will handle messages in-line. For a discussion of error and message handling, see **Error and Message Handling** in Chapter 2, “Topics.”
3. Connect to a server.
4. Send a command to the server.
5. Process the results of the command.
6. Finish up.

The example program in the following section demonstrates these steps.

A Simple Example Program

The following example demonstrates the basic framework of a Client-Library application. The program follows the steps outlined in the previous section, sending a language command to a SQL Server and processing the results of the command. In this case, the language command is a Transact-SQL **select** command.

For brevity’s sake, this program does not include code for the message callback routines that handle Client-Library and server messages. However, message callback routines are included with the on-line example programs.

```
/*
** Language Query Example Program.
*/

#include <stdio.h>
#include <ctpublic.h>

/*
** Define a global context structure to use
*/
CS_CONTEXT     *context;

#define        ERROR_EXIT  (-1)
```

```
#define        MAXCOLUMNS  2
#define        MAXSTRING   40

extern int print_data();

/* Client message and server message callback routines: */
CS_RETCODE clientmsg_callback();
CS_RETCODE servermsg_callback();
void error();

/*
** Main entry point for the program.
*/
main(argc, argv)
int        argc;
char       **argv;
{
     CS_CONNECTION     *connection; /* Connection structure. */
     CS_COMMAND        *cmd;        /* Command structure.    */

     /* Data format structures for column descriptions: */
     CS_DATAFMT        columns[MAXCOLUMNS];

     CS_INT            datalength[MAXCOLUMNS];
     CS_SMALLINT       indicator[MAXCOLUMNS];
     CS_INT            count;
     CS_RETCODE        ret, res_type;
     CS_CHAR           name[MAXSTRING];
     CS_CHAR           city[MAXSTRING];

     /*
     ** Get a context structure to use.
     */
     cs_ctx_alloc(CS_VERSION_100, &context)

     /*
     ** Initialize Open Client.
     */
     ct_init(context, CS_VERSION_100);

     /*
     ** Install message callback routines.
     */
     ct_callback(context, NULL, CS_SET, CS_CLIENTMSG_CB,
               clientmsg_callback);

     ct_callback(context, NULL, CS_SET, CS_SERVERMSG_CB,
               servermsg_callback);
```

```
/*
** Connect to the server:
**    Allocate a connection structure.
**    Set user name and password.
**    Create the connection.
*/
ct_con_alloc(context, &connection);
ct_con_props(connection, CS_SET, CS_USERNAME, “username”,
          CS_NULLTERM, NULL);
ct_con_props(connection, CS_SET, CS_PASSWORD, “password”,
          CS_NULLTERM, NULL);

/*
** This call actually creates the connection:
*/
ct_connect(connection, “servername”, CS_NULLTERM);

/*
** Allocate a command structure.
*/
ct_cmd_alloc(connection, &cmd);

/*
** Initiate a language command.
*/
ct_command(cmd, CS_LANG_CMD,
          “use pubs2 \
           select au_lname, city from pubs2..authors \
              where state = ’CA’”,
          CS_NULLTERM, CS_UNUSED);

/*
** Send the command.
*/
ct_send(cmd);

/*
** Process the results of the command.
*/
while((ret = ct_results(cmd, &res_type))== CS_SUCCEED)
{
     switch (res_type)
     {
     case CS_ROW_RESULT:
         /*
         ** We’re expecting exactly two columns.
         ** For each column, fill in the relevant
** fields in a data format structure, and
** bind the column.
*/
columns[0].datatype = CS_CHAR_TYPE;
columns[0].format = CS_FMT_NULLTERM;
columns[0].maxlength = MAXSTRING;
columns[0].count = 1;
columns[0].locale = NULL;
ct_bind(cmd, 1, &columns[0], name, &datalength[0],
        &indicator[0]);

columns[1].datatype = CS_CHAR_TYPE;
columns[1].format = CS_FMT_NULLTERM;
columns[1].maxlength = MAXSTRING;
columns[1].count = 1;
columns[1].locale = NULL;
ct_bind(cmd, 2, &columns[1], city, &datalength[1],
        &indicator[1]);

/*
** Now fetch and print the rows.
*/
while(((ret = ct_fetch(cmd, CS_UNUSED, CS_UNUSED,
    CS_UNUSED, &count))
    == CS_SUCCEED) || (ret == CS_ROW_FAIL))
{
    /*
    ** Check if we hit a recoverable error.
    */
    if( ret == CS_ROW_FAIL )
    {
        fprintf(stderr,
        "Error on row %d in this fetch batch.",
        count+1);
    }

    /*
    ** We have a row, let's print it.
    */
    fprintf(stdout, "%s: %s\n", name, city);
}

/*
** We're finished processing rows, so check
** ct_fetch's final return value.
*/
if( ret == CS_END_DATA )
{
```

```
                fprintf(stdout,
                "“All done processing rows.”);
            }
            else /* Failure occurred. */
            {
                error(“ct_fetch failed”);
            }

            /*
            ** All done with this result set.
            */
            break;

        case CS_CMD_SUCCEED:
            /*
            ** Executed a command that never returns rows.
            */
            fprintf(stderr, “No rows returned.\n”);
            break;


        case CS_CMD_FAIL:
            /*
            ** The server encountered an error while
            ** processing our command.
            */
            break;

        case CS_CMD_DONE;
            /*
            ** The logical command has been completely
            ** processed.
            */
            break;

        default:
            /*
            ** We got something unexpected.
            */
            error(“ct_result returned unexpected result
                type”);
            break;
        }
    }

    /*
    ** We’ve finished processing results. Let’s check
    ** the return value of ct_results() to see if
```

```
     ** everything went ok.
     */
     switch(ret)
     {
          case CS_END_RESULTS:
              /*
              ** Everything went fine.
              */
              break;

          case CS_FAIL:
              /*
              ** Something terrible happened.
              */
              error(“ct_results() returned FAIL.”);
              break;

          default:
              /*
              ** We got an unexpected return value.
              */
              error(“ct_result returned unexpected return
                  code”);
              break;
     }

     ** All done.
     */
     ct_cmd_drop(cmd);
     ct_close(connection, CS_UNUSED);
     ct_con_drop(connection);
     ct_exit(context, CS_UNUSED);
     cs_ctx_drop(context);
     return 0;
}

/*
** Error occurred, cleanup and exit.
*/
void error(msg)
char  *msg;
{
     fprintf(stderr, “FATAL ERROR: %s\n”, msg);
     exit(ERROR_EXIT);
}
```

Notes on the Example Program

The header file *ctpublic.h* is required in all source files that contain calls to Client-Library / C. It defines symbolic constants used by Client-Library routines and contains typedefs for Client-Library datatypes.

Setting Up the Client-Library Programming Environment

The CS-Library routine **cs_ctx_alloc** allocates a context structure. A context structure is used to store configuration parameters that describe a particular "context," or operating environment, for a set of server connections. On most platforms, an application can have multiple contexts, although a typical application will need just one.

Application properties that can be defined at the context level include the name and location of the interfaces file, the login timeout value, and the maximum number of connections allowed within the context.

ct_init initializes Client-Library. An application calls **ct_init** after calling **cs_ctx_alloc** and before calling any other Client-Library routine.

Installing Message Callback Routines

ct_callback installs a Client-Library callback routine. Callbacks are custom routines which are called automatically by Client-Library when a triggering event of the appropriate type occurs. For example, a client message callback is called automatically whenever OC-Library generates an error or informational message.

There are several types of callbacks, but the example program installs only two: a client message callback, to handle Client-Library error and informational messages, and a server message callback, to handle server error and informational messages. Code for the callbacks is not supplied with this example.

➤ ***Note***

Callback routines are not supported for all programming language/ platform combinations.If callbacks are not supported for a programming language/platform version of Client-Library, the *Open Client/Server Supplement* for that language and platform will indicate the lack of support.

Connecting to a Server

ct_con_alloc allocates a connection structure. A connection structure contains information about a particular client / server connection.

ct_con_props sets and retrieves the values of a connection's properties. Connection properties include user name and password, which are used in logging into a server; application name, which appears in SQL Server's *sysprocess* table, and packet size, which determines the size of network packets that an application will send and receive. For a complete list of connection properties, see the **Properties** topics page.

The example program sets only the user name and password properties.

ct_connect opens a connection to a server, logging into the server with the connection information specified via **ct_con_props**.

Sending a Command to the Server

ct_cmd_alloc allocates a command structure. A command structure is used to send commands to a server and to process the results of those commands.

ct_command initiates the process of sending a non-cursor command. In this case, the example program initiates a language command.

ct_send sends a command to the server.

Processing the Results of the Command

Almost all Client-Library programs will process results by using a loop controlled by **ct_results**. Inside the loop, a switch takes place on the current type of result. Different types of results require different types of processing.

For row results, typically the number of columns in the result set is determined and then used to control a loop in which result items are bound to program variables. An application can call **ct_res_info** to get the number of result columns, but in the example this is not necessary, because exactly two columns were selected. After the result items are bound, **ct_fetch** is called to fetch data rows until end-of-data.

The results processing model used in the example looks like this:

```
while ct_results returns CS_SUCCEED
       switch on result_type
           case row results
               for each column:
                   ct_bind
               end for
               while ct_fetch is returning rows
                   process each row
               end while
               check ct_fetch’s final return code
           end case row results
           case other result type....
           case other result type....
           ....
       end switch
end while
check ct_results’ final return code
```

ct_results sets up results for processing. **ct_results**’ return parameter *result_type* indicates the type of result data that is available for processing. Because the example program expects only a single result set of type CS_ROW_RESULT, most result types are not included as cases in the switch on *result_type*.

Note that the example program calls **ct_results** in a loop that continues as long as **ct_results** returns CS_SUCCEED, indicating that result sets are available for processing. Although this type of program structure is not strictly necessary in the case of a simple language command, it is highly recommended. In more complex programs, it is not possible to predict the number and type of result sets than an application will receive in response to a command.

ct_bind binds a result item to a program variable. Binding creates an association between a result item and a program data space.

ct_fetch fetches result data. In the example, since binding has been specified and the count field in the CS_DATAFMT structure for each column is set to 1, each **ct_fetch** call copies one row of data into program data space. As each row is fetched, the example program prints it.

After the **ct_fetch** loop terminates, the example program checks its final return code to find out whether we dropped out because of end-of-data, or because of failure.

Finishing Up

ct_cmd_drop de-allocates a command structure.

ct_close closes a server connection.

ct_con_drop de-allocates a connection structure.

ct_exit terminates Client-Library.

The CS-Library routine **cs_ctx_drop** de-allocates a context structure.

More Advanced Programs

Although some Client-Library applications will be as simple as the example program in this chapter, most will be more complex. Client-Library is a rich programming interface that supports a variety of advanced features.

Some of these features are:

- Asynchronous network I/O support. When asynchronous network I/O is enabled, a Client-Library routine that reads from or writes to the network does not block, but instead returns immediately.
- Registered Procedures. For clients connected to a release 2.0 or greater Open Server, registered procedures provide a means of inter-application communication and synchronization. An application can create, wait for, and execute registered procedures.
- Cursor support. Client-Library contains routines to declare, open, and manipulate cursors as supported in 10.0 SQL Server and Open Server.
- International support. Client-Library allows an application to choose a language for Client-Library and SQL Server messages, process datetime, money, and numeric values in local formats, and specify character sets and collating sequences. A Client-Library application can specify localization information for a context, connection, or individual data element.
- Gateway support. A gateway is an application that acts as a “translator” for clients and server that cannot communicate directly. A gateway application passes requests from a client to a server, acting as both client and server itself. Client-Library provides routines specifically for use in building gateway applications.
- Remote procedure calls. A Client-Library application can send remote procedure calls to SQL Servers or Open Server applications.

- Text and image datatype support. Client-Library provides routines to transfer large text or image values to or from a server.

For more information on these and other advanced features, see Chapter 2, “Topics.”

Topics

2 Topics

This chapter contains information on:

- 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.

List of Topics

The following topics are included in this section:

Asynchronous Programming

Asynchronous applications are designed to make constructive use of time that would otherwise be spent waiting for certain types of operations to complete. Typically, reading from and writing to a network or external device is much slower than straightforward program execution.

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 to CS_ASYNC_IO. When asynchronous behavior is enabled, all Client-Library routines that read from or write to the network either:

- Initiate the requested operation and return CS_PENDING immediately
- Return CS_BUSY to indicate that an asynchronous operation is already pending for this connection

Non-asynchronous routines can also return CS_BUSY if called when an asynchronous operation is pending for a connection.

Learning about Completions

An application can learn of an asynchronous routine completion in one of two ways:

- On platforms that support interrupt-driven I/O, Client-Library automatically calls the application's completion callback routine when an asynchronous operation completes.
- On platforms that do not support interrupt-driven I/O, an application can use **ct_poll** to find out if any asynchronous operations have completed. If it finds a completed operation, Client-Library will call an application's completion callback routine from within **ct_poll**.

Asynchronous Routines

The following Client-Library routines can behave asynchronously:

- **ct_cancel**
- **ct_close**
- **ct_connect**

- **ct_fetch**
- **ct_get_data**
- **ct_options**
- **ct_recvpassthru**
- **ct_results**
- **ct_send**
- **ct_send_data**
- **ct_sendpassthru**

Any Client-Library routine that takes a command or connection structure as a parameter can return CS_BUSY. CS_BUSY indicates that a routine is unable to perform because the relevant connection is currently busy, waiting for an asynchronous operation to complete.

An application can call 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**

Client-Library's Interrupt-Level Memory Requirements

Ordinarily, Client-Library routines satisfy their memory requirements by calling **malloc**. However, because not all implementations of **malloc** are re-entrant, it is not safe for Client-Library routines that are called at the interrupt level to use **malloc**. For this reason, 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 can satisfy Client-Library's memory requirements:

- The application can use the CS_MEM_POOL property to provide Client-Library with a memory pool.

- The application can use the CS_USER_ALLOC and CS_USER_FREE properties to install memory allocation routines that Client-Library can safely call at the interrupt level.

If an 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

Layered Applications

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

What's In the Layers?

The higher-level layer typically consists of:

- Main-line code
- Routines that asynchronously perform "larger" operations.

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

The lower-level layer typically consists of:

- The Client-Library routines required to perform a larger 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 useful in layered asynchronous applications:

- A layered application can use CS_DISABLE_POLL to prevent **ct_poll** from reporting asynchronous Client-Library routine completions.

- A layered application can use **ct_wakeup** to let the higher layer know when a "larger" asynchronous operation is complete.

A layered application that is using a routine to perform a "larger" operation typically uses **ct_wakeup** and CS_DISABLE_POLL in the following manner:

1. The application performs any necessary initialization, installs callback routines, opens connections, etc.
2. The application calls the routine performing the larger 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 re-installs 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.

A Brief Example

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

The main application can call **do_update** asynchronously and go on with its other work.

When called, **do_update** replaces the main application's completion callback with its own (so that the main application's callback is not triggered by low-level asynchronous completions), and proceeds with the work of the update. In order to perform the update, **do_update** needs to call 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 re-installs 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 10.0 Client-Library in order to provide compatibility with Open Server applications and older Open Client libraries. Its use in new Open Client Client-Library applications is discouraged, 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 a 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.

Since a row being browsed is not the actual row residing in the database, but is instead a copy residing in program variables, the program must be able to ensure that changes to the variables' values can be reliably used to update the original database row. In particular, in multi-user situations, the program needs to ensure that updates made to the database by one user do not unwittingly overwrite updates recently made by another user. This can be a problem because an application typically selects a number of rows from a database at one time, but the application's users browse and update the database one row at a time. A timestamp column in browsable tables provides the information necessary to regulate this type of multi-user 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 a second for performing the updates.

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

Implementing Browse Mode

Conceptually, 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 multi-user 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 will need the values of these columns to build a qualifier 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 can be 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 qualifier 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 can fetch and process the next row.

Browse-mode Conditions

To use browse mode, the following conditions must be true:

- The **select** command that generated the result set must end with the key words **for browse**.
- The table(s) to be updated must be "browsable" (*i.e.*, 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 **max(colname)**.

Callbacks

What Are Callbacks?

Callbacks are user-supplied routines that are automatically called by Client-Library whenever 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 Are Callbacks Called?

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

In order for Client-Library to recognize some callback events, it must be actively engaged in reading from the network. Most callback events of this type occur when Client-Library is naturally reading from the network, and are handled automatically.

Two types of callback events, however, can occur when Client-Library is not reading from the network. These are:

- The completion callback event, which occurs when an asynchronous Client-Library routine completes.
- The notification callback event, which occurs when an Open Server notification arrives for an application.

If a platform supports interrupt-driven I/O, completion and notification callbacks are called at the interrupt level when a completion or notification callback event occurs. If a platform does not support interrupt-driven I/O, however, an application will need to call **ct_poll** to check for these events if it is not otherwise reading from the network.

If **ct_poll** finds an asynchronous routine completion or an Open Server notification, it automatically calls the appropriate callback routine before returning.

➤ ***Note***

Because some types of callback routines can be executed at interrupt time, a callback routine must take care in accessing data structures that are also used by the application's main-line code.

Types of Callbacks

The following table lists the types of callbacks, when they are called, and whether an application needs to use **ct_poll** to trigger them:

Type of Callback:	When Is it Called?	How Is it 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.
Completion	When an asynchronous Client-Library routine completes.	An asynchronous routine completion can occur at any time. On platforms that support interrupt-driven I/O, the completion callback is called automatically, at the interrupt level, when the completion occurs. On platforms that do not support interrupt-driven I/O, an application can use **ct_poll** to find out if any routines have completed.
Encryption	During the connection process, in response to a server request for an encrypted password.	If a connection's CS_SEC_ENCRYPTION property is set to CS_TRUE, then Client-Library automatically triggers the encryption callback when a server requests an encrypted password during a connection attempt.
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.

Table 2-1: Types of callbacks

Type of Callback:	When Is it Called?	How Is it Called?
Notification	When an Open Server notification arrives.	An Open Server notification can arrive at any time. On platforms that support interrupt-driven I/O, the notification callback is called at the interrupt level when the completion occurs. On platforms that do not support interrupt-driven I/O, an application must be actively reading from the network in order for Client-Library to recognize a notification. If an application is not actively reading from the network, it can use **ct_poll** to find out if a notification has arrived.
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.
Signal	In response to an operating-system signal.	When a signal arrives, Client-Library's own signal handler automatically calls the signal callback that an application has installed.

Table 2-1: Types of callbacks (continued)

Callbacks Are Not Universally Implemented

Callbacks may not be implemented for programming language / 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 in-line, using **ct_diag**.
- Can still use **ct_poll** to check for a completion or notification callback event, but will have to call any routine handling the event directly.

If callbacks are not supported for a programming language / platform version of Client-Library, the *Open Client/Server Supplement* for that language and platform will indicate the lack of support.

Installing a Callback Routine

An application installs a callback routine by calling **ct_callback**, passing a pointer to the callback routine and indicating its type via 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 can override 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 can save this address for re-use at a later time.

To de-install 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** will replace the existing callback with the new callback.

Defining Callback Routines

All callback routines are limited as to which Client-Library routines they can call. The following table lists types of callback routines and the Client-Library routines that they can call:

Type of Callback	Can Call	Under What Circumstances?
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 or to set the CS_USERDATA property only.
	ct_cancel (CS_CANCEL_ATTN)	
Server Message	**ct_bind**, **ct_describe**, **ct_fetch**, **ct_get_data**, **ct_res_info**	The routines must be called with the command structure returned by the callbacks's **ct_con_props**(CS_EED_CMD) call. For more information, see "Extended Error Data" on page 2-79.
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 callbacks's **ct_con_props**(CS_NOTIF_CMD) call. For more information, see "Registered Procedures" on page 2-157.
Completion	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 or free.	

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

The following sections contain information on how to define each type of callback routine:

Client Message Callbacks

An application can handle Client-Library error and informational messages in-line, 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 can:

- Install a different client message callback for the connection.
- Call **ct_diag** to initialize in-line 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 in-line message handling is not enabled, Client-Library discards message information.

If callbacks are not implemented for a particular programming language / platform version of Client-Library, an application must handle Client-Library messages in-line, 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 clientmsg_cb(context, connection, message)

CS_CONTEXT      *context;
CS_CONNECTION   *connection;
CS_CLIENTMSG    *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, if any. *connection* can be NULL.

message is a pointer to a CS_CLIENTMSG structure containing Client-Library message information. For information on the CS_CLIENTMSG structure, see the **CS_CLIENTMSG** topics page.

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, to instruct Client-Library to continue any processing that is currently occurring on this connection.

  If a timeout error occurs, 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***

It is possible, in some cases, that a server will be unable to respond to a client's cancel command. Such a situation could occur, for example, if the server is processing a very complex query and is not in an interruptable state.

- CS_FAIL, to instruct 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. In order to continue using the connection, the application must close the connection and reopen it.

The following table lists the Client-Library routines that a client message callback can call:

A Client Message Callback Can Call:	Under What Circumstances?
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.

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

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 is called whenever an application receives notice that an asynchronous routine has completed.

A context or a connection can be defined to be asynchronous. If a context is asynchronous, then all connections within that context are asynchronous, unless defined otherwise.

When a connection is asynchronous, Client-Library routines that perform network I/O do not block, but instead return CS_PENDING immediately. When a routine completes, Client-Library automatically calls the completion callback.

A completion callback is typically coded to notify the main-line code of the asynchronous routine's completion.

Defining a Completion Callback

A completion callback is defined as follows:

```
CS_RETCODE 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, if any. *cmd* can be NULL.

function indicates which routine has completed. The following table lists the symbolic values possible for *function*:

Value of *function*:	**Indicating:**
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_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.

Table 2-4: Values for function *(Completion Callback)*

status is the return status of the completed routine. To find out what values status can have, see **Returns** on the manual page for the routine.

Because it is regarded as an extension of main-line code, a completion callback can call any Client-Library routine.

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. It is recommended, however, that the completion callback return either 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:

```
/*
** ex_acompletion_cb()
**
** Type of function:
**      Internal 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_INTERNAL
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;
    }
```

Encryption Callbacks

SQL Server uses encrypted password handshakes. Most applications are not aware of this because Client-Library automatically handles it.

Client-Library applications that are acting as gateways, however, need to handle password encryption explicitly, using an encryption callback routine to pass the server's encryption key to the client and to return the encrypted password back to the server.

In order to use an encryption callback, a connection must have its CS_SEC_ENCRYPTION property set to CS_TRUE.

For more information on handling encrypted password security handshakes, see "Encrypted Password Security Handshakes" on page 2-177.

Defining an Encryption Callback

An encryption callback is defined as follows:

```
CS_RETCODE 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 into the server.

pwd is the user password to be encrypted

pwdlen is the length of the password

key is the key to use to encrypt the password.

keylen is the length 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 placed in **buffer.*

An encryption callback should return CS_SUCCEED to indicate that the password was 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.

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 "Security Features" on page 2-175.

Trusted-User Security Handshakes

When logging into a server, a trusted-user security handshake occurs when the server asks the client for identifying security labels, which the client then provides.

A connection can use a negotiation callback to provide these security labels. To do this, the connection installs a negotiation callback routine. At connection time, when Client-Library receives the server request for login security labels, it triggers the negotiation callback.

A connection can also use **ct_labels** to define security labels. For more information, see the manual page for **ct_labels**.

A connection that will be participating in trusted-user security handshakes must set its CS_SEC_NEGOTIATE property to CS_TRUE.

Challenge/Response Security Handshakes

When logging into a server, a challenge / response security handshake occurs when the server issues a challenge, to which the client must respond.

A connection can use 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, it triggers the negotiation callback.

A connection that will be participating in challenge / response security handshakes must set either its CS_SEC_CHALLENGE property or its CS_SEC_APPDEFINED property to CS_TRUE.

Defining a Negotiation Callback

A negotiation callback is defined as follows:

```
CS_RETCODE 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. The following table lists the values that are legal for *inmsgid*:

Value of *inmsgid:*	To Indicate:
CS_MSG_GETLABELS	The server is requesting security labels.
A user-defined value < CS_USER_MSGID	The server is requesting a Sybase-defined value. It is the negotiation callback's responsibility to understand the meaning of *inmsgid*.
A user-defined value >= CS_USER_MSGID and <= CS_USER_MAX_MSGID	The Open Server application is requesting an application-defined value. It is the negotiation callback's responsibility to understand the meaning of *inmsgid*.

Table 2-5: Values for inmsgid *(Negotiation Callback)*

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

Value of *outmsgid:*	To Indicate:
CS_MSG_LABELS	The negotiation callback is returning security labels.

Table 2-6: Values for outmsgid *(Negotiation Callback)*

Value of *outmsgid:*	To Indicate:
A user-defined 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.

Table 2-6: Values for outmsgid *(Negotiation Callback) (continued)*

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.

outbuffmt 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; however, Secure SQL Server requires values for the *name*, *namelen*, and *datatype* fields.

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*→*max-length*.

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 type of procedure that is defined and installed in a running Open Server. A Client-Library application can use a remote procedure call command to execute a registered procedure, and can "watch" for a registered procedure to execute.

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.

When Client-Library receives a notification, it calls an application's notification callback routine.

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.

For more information on registered procedures, see the **Registered Procedures** topics page, 2-157.

Defining a Notification Callback

A notification callback is defined as follows:

```
CS_RETCODE notification_cb(conn, proc_name, pnamelen)

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

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.

pnamelen is the length, in bytes, of **proc_name*.

A notification callback must return CS_SUCCEED.

The following table lists the Client-Library routines that a notification callback can call:

A Notification Callback Can Call:	Under What Circumstances?
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
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 callbacks's **ct_con_props**(CS_NOTIF_CMD) call. For more information, see "Registered Procedures" on page 2-157.

Table 2-7: Routines that a notification callback can call

Server Message Callbacks

An application can handle server error and informational messages in-line, 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 can:

- Install a different server message callback for the connection.
- Call **ct_diag** to initialize in-line message handling for the connection. Note that **ct_diag** automatically de-installs all message callbacks for the connection.

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

If callbacks are not implemented for a particular programming language / platform version of Client-Library, an application must handle server messages in-line, 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 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 the CS_SERVERMSG data structure, see the **CS_SERVERMSG** topics page.

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

A server message callback must return CS_SUCCEED.

The following table lists the Client-Library routines that a server message callback can call:

A Server Message Callback Can Call:	Under What Circumstances?
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.

Table 2-8: Routines that a server message callback can call

A Server Message Callback Can Call:	Under What 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 callbacks's **ct_con_props**(CS_EED_CMD) call. A server message callback can call these routines only while extended error data is available; that is, until **ct_fetch** returns CS_END_DATA. For more information, see "Extended Error Data" on page 2-79.

Table 2-8: Routines that a server message callback can call (continued)

Server Message Callback Example

This 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(connection, cmd, srvmsg)
CS_CONNECTION*  connection;
CS_COMMAND      *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;
}
```

Signal Callbacks

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

An application that needs to handle signals for its own purposes must do so by calling **ct_callback** to install signal callbacks, rather than by making a signal system call to install a signal handler. This is because a signal system call will de-install Client-Library's signal handler. If this occurs, Client-Library's behavior is undefined.

When Client-Library receives a signal, Client-Library's signal handler:

- Performs any internal Client-Library processing that is required.
- Calls the appropriate user-defined signal callback, if any.

An application that plans to install a signal callback must include the header file *sys/signal.h*.

Defining a Signal Callback

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

Installing a Signal Callback

A signal callback can be installed only at the context level.

Signal callbacks are identified by adding the signal number on to the manifest 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
**      signalhandlerSignal 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);
}
```

Capabilities

Capabilities describe features that a client / server connection supports.

For a list of capabilities, see **ct_capability** on page 3-31.

What are Capabilities Good For?

An application can use capabilities to find out what features are supported by a connection's actual TDS version.

In particular, an application can:

- Find out whether a server connection supports a particular type of request.
- Tell a server not to send a particular type of response on a connection.

Types of Capabilities

There are two types of capabilities:

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

Setting and Retrieving Capabilities

Before calling **ct_connect** to open a connection, an application can:

- Retrieve 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.

- Set response capabilities, to indicate that a connection does not wish to receive particular types of responses. For example, an application can set a connection's TDS_RES_NOEED capability to CS_TRUE to indicate that the connection does not wish to receive extended error data.

During the connection process, the client and server negotiate a TDS version level for the connection. The TDS version level determines which capabilities the connection will support.

After a connection is open, an application can:

- Retrieve request capabilities to find out what types of requests the connection will support.
- Retrieve 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_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.

Client-Library Messages

Client-Library message numbers are four bytes long. Each byte contains a separate piece of information.

What the Bytes Represent

The first (high-order) byte represents the Client-Library layer that is reporting the message. A typical application will not examine this byte except to provide information for SYBASE Technical Support.

The second byte represents the message's origin. A typical application will not examine this byte except to provide information for SYBASE Technical support.

The third byte represents the message's severity. For a list of possible severities, see *Table 2-9: Client-Library message severities*, on page 2-36.

The fourth (low-order) byte represents a layer-specific message number.

Decoding a Message Number

Client-Library provides the following macros to help an application decode a message number:

- **CS_LAYER**
- **CS_ORIGIN**
- **CS_SEVERITY**
- **CS_NUMBER**

These macros are defined in the header file *cstypes.h*.

A typical application uses these macros to split a message number into four parts, which it then displays separately.

The client message callback example on page 2-18 demonstrates the use of these macros.

Message Severities

The following table lists Client-Library message severities:

Severity:	Explanation:	User Action:
CS_SV_INFORM	No error has occurred. The message is informational.	No action is required.
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 can 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 re-open it, if desired, 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 salvageable.	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. Note that 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 salvageable.	Call **ct_close**(CS_FORCE_CLOSE) to close the server connection and then re-open it, if desired, by calling **ct_connect**. Note that it is illegal to make these calls from within a client message callback routine.

Table 2-9: Client-Library message severities

Severity:	**Explanation:**	**User Action:**
CS_SV_COMM_FAIL	An unrecoverable error in the server communication channel has occurred. The server connection is not salvageable.	Call **ct_close**(CS_FORCE_CLOSE) to close the server connection and then re-open it, if desired, by calling **ct_connect**. Note that 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. Note that it is illegal to call **ct_exit** from within a client message callback routine.
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. Note that it is illegal to call **ct_exit** from within a client message callback routine.

Table 2-9: Client-Library message severities (continued)

Commands

In the client / server model, a server accepts commands from multiple clients and responds by returning data and other information to the clients. SYBASE Open Client applications use Client-Library routines to communicate commands to servers. For example, an application might send a cursor open command to a server, directing it to execute a SQL **select** statement.

The following table lists Client-Library routines that initiate commands, together with the types of commands that each routine initiates:

Client-Library routine:	Initiates:
ct_command	Miscellaneous commands, including language, message, package, remote procedure call (RPC), and send-data commands.
ct_cursor	Cursor commands, including cursor declare, cursor options, cursor rows, cursor open, cursor update, cursor delete, cursor close, and cursor de-allocate.
ct_dynamic	Dynamic commands, including cursor declare, prepare, describe input, execute, describe output, and de-allocate.

Table 2-10: Client-Library routines that initiate commands

Sending a Command to a Server

In general, sending a command to a server is a four step process. To send a command to a server, an application must:

1. Initiate the command by calling **ct_command**, **ct_cursor**, or **ct_dynamic**. These routines set up internal structures that are used in building a command stream to send to the server.
2. Pass parameters for the command (if required). Most applications pass parameters by calling **ct_param** once for each parameter that the command requires, but it is also possible to pass parameters for a command by using **ct_dyndesc**.

   Not all commands require parameters. For example, an RPC command may or may not require parameters, depending on the stored procedure being called. For information about which commands require parameters, see the **ct_param** and **ct_dyndesc** manual pages.

3. Send the command to the server by calling **ct_send**.
4. Verify the success of the command by calling **ct_results**.

➤ ***Note***

Step 4 does not imply that an application need only call **ct_results** once. If the value of **ct_results**' *result_type* parameter indicates that there are fetchable results, the application will most likely process the results using a loop controlled by **ct_results**. See the *Open Client Client-Library/C Programmer's Guide* for a discussion of processing results.

Deciding Which Type of Command to Use

In some cases, different Client-Library commands actually do the same thing. For example, an application might send an RPC command to a server to direct it to execute a stored procedure, or it might send a cursor command.

The following table lists common tasks as well as various Client-Library routines that an application can use to accomplish them.

Task:	Routines:	Notes
Execute a statement (with no variables)	**ct_command**(CS_LANG_CMD) **ct_send**	
	ct_dynamic(CS_EXEC_IMMEDIATE) **ct_send**	A **select** statement is not allowed.
Execute a statement (with variables)	**ct_command**(CS_LANG_CMD) **ct_param** **ct_send**	
	ct_command(CS_RPC_CMD) **ct_param** **ct_send**	
	ct_cursor(CS_CURSOR_DECLARE) **ct_param** **ct_cursor**(CS_CURSOR_OPEN) **ct_param** **ct_send**	

Table 2-11: Different Client-Library commands that accomplish the same tasks

Task:	Routines:	Notes
	ct_dynamic(CS_PREPARE) **ct_dynamic**(CS_EXECUTE) **ct_param** **ct_send**	
Execute a SQL Server stored procedure	**ct_command**(CS_RPC_CMD) **ct_send**	**buffer* is the name of the stored procedure.
	ct_command(CS_LANG_CMD) **ct_send**	**buffer* is an "**execute** st_proc_name" statement.
Declare a cursor	**ct_command**(CS_LANG_CMD) **ct_send**	Declares a (language based) cursor on a SQL statement.
	ct_cursor(CS_CURSOR_DECLARE) **ct_send**	Declares a (Client-Library based) cursor on a SQL statement.
	ct_dynamic(CS_PREPARE) **ct_dynamic**(CS_CURSOR_DECLARE) **ct_send**	Declares a (language based) cursor on a prepared statement.
	ct_dynamic(CS_PREPARE) **ct_dynamic**(CS_CURSOR_DECLARE) **ct_cursor**(CS_CURSOR_OPTION) **ct_send**	Declares a (Client-Library based) cursor on a prepared statement.
	ct_cursor(CS_CURSOR_DECLARE) **ct_send**	Declares a (Client-Library) cursor on a SQL Server stored procedure. **text* is an "**execute** st_proc_name" statement.
	ct_dynamic(CS_CURSOR_DECLARE) **ct_send**	Declares a (language based) cursor on a SQL Server stored procedure. **id* is the identifier for the "**execute** st_proc_name" statement that was previously prepared by a call to **ct_dynamic**(CS_PREPARE).

Table 2-11: Different Client-Library commands that accomplish the same tasks

Because an application can use different Client-Library commands to accomplish the same task, it is not always easy to choose which command type to use in an application.

The following sections contain information about each of the Client-Library routines that initiate commands.

ct_command

This routine is unique for a number of reasons:

- It is the only one of the three command initiation routines that initiates more than one kind of command.
- It is the only command initiation routine that accepts multiple language statements at one time.
- It has the ability to send a command to execute a SQL Server stored procedure either as a language command or as an RPC command.
- It provides the only means of inserting text and image data.

For information about RPC commands, see "Remote Procedure Calls"on page 2-160 and "RPC (remote procedure call) Commands" on page 3-56.

For information about text and image data, see "Text and Image" on page 2-188 and "Send-Data Commands" on page 3-56.

ct_cursor

This routine allows you to create and use Client-Library-based cursors. These are different from language-based cursors in that a single server connection can support multiple open cursors, each simultaneously processing its own result set.

In addition, an application can send commands to update or delete rows in the underlying table(s) while actively fetching rows of a cursor result set.

For information about Client-Library-based cursors, see "Cursors" on page 2-59.

For information on using Client-Library to update previously-fetched cursor rows, see the **ct_keydata** manual page.

ct_dynamic

This routine was designed for precompiler use, but it can offer a Client-Library application the following advantages:

- The ability to send a command to execute a prepared statement and reference the statement with a unique identifier.

A prepared statement is a statement that has been compiled and stored with an identifier as a result of a **ct_dynamic**(CS_PREPARE) call and a **ct_send** call.

An application typically prepares a statement if it plans to execute the statement multiple times. Variables are particularly useful in dynamic commands because they allow an application to compile a statement once and change the values of the statement's variables each time it executes the statement.

- The ability to describe (with CS_DESCRIBE_OUTPUT) prepared statement output before sending a command to execute the statement.
- Less overhead and faster performance than **ct_command**, if the statement is executed more than once. This benefit is specific to the execution of SQL statements on a SQL Server.

All of the above advantages can also be realized using a stored procedure and either language or RPC commands. Because of the limitations of dynamic SQL, its use is discouraged. For a discussion of the limitations of dynamic SQL functionality, see "Dynamic SQL" on page 2-63.

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_NAME];
    CS_INT      orignlen;
    CS_INT      tablenum;
    CS_CHAR     tablename[CS_OBJ_NAME];
    CS_INT      tabnlen;
} CS_BROWSEDESC;
```

where:

status is a bit mask of the following symbols, or-ed together:

CS_EXPRESSION to indicate the column is the result of an expression, for example, “sum*2” in the query “select sum*2 from areas”.

CS_HIDDEN to indicate that the column is a “hidden” column that has been exposed. For more information, see CS_HIDDEN_KEYS on the **Properties** topics page.

CS_KEY to indicate that the column is a key column. For more information, see the **ct_keydata** manual page.

CS_RENAMED to indicate 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 can 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 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 in-line, **ct_diag** can return information in a CS_CLIENTMSG.

For information on error and message handling, see "Error and Message Handling" on page 2-74.

For information on Client-Library messages, see "Client-Library Messages" on page 2-35.

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. The following table lists the legal values for *severity*:

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 the operation 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 salvageable.
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 salvageable.
CS_SV_COMM_FAIL	An unrecoverable error in the server communication channel has occurred. The server connection is not salvageable.
CS_SV_INTERNAL_FAIL	An internal Client-Library error has occurred.
CS_SV_FATAL	A serious error has occurred. All server connections are unusable.

Table 2-12: Values for severity *(CS_CLIENTMSG)*

msgnumber is the Client-Library message number. For information on how to interpret this number, see the **Client-Library Messages** topics page, 2-35.

msgstring is the null-terminated Client-Library message string.

msgstring is the 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.

For more information on sequenced messages, see "Sequencing Long Messages" on page 2-77.

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 used to indicate various types of information, such as whether or not this is the first, a middle, or the last chunk of an error message. The following table lists the values that can be present in *status*:

Symbolic Value:	To Indicate:
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. For more information on sequenced messages, see "Sequencing Long Messages" on page 2-77.

Table 2-13: Values for status *(CS_CLIENTMSG)*

Symbolic Value:	**To Indicate:**
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. For more information on sequenced messages, see "Sequencing Long Messages" on page 2-77.

Table 2-13: Values for status *(CS_CLIENTMSG) (continued)*

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.

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** requires a CS_DATAFMT to describe an input parameter.
- **cs_convert** requires CS_DATAFMT structures to describe source and destination data.

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 manual page for the routine.

A CS_DATAFMT structure is defined as follows:

```
typedef struct _cs_datafmt
{
    CS_CHAR name[CS_MAX_NAME];   /* Name of data */
    CS_INT  namelen;             /* Length of name */
    CS_INT  datatype;            /* Data’s 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 SQL 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 listed on the **Types** topics page, or an Open Client user-defined datatype. For information on user-defined datatypes, see the **Types** topics page.

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 only used if the data has a SQL Server user-defined datatype in addition to an Open Client datatype.

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

```
sp_addtype birthday, datetime
```

and this command creates a table containing a column of the new 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 SQL Server id for the type *birthday*.

format describes the destination format of character or binary data. *format* is a bit mask of the following symbols, or'ed together:

Symbol:	To Indicate:	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.

Table 2-14: Values for format *(CS_DATAFMT)*

maxlength can represent various lengths, depending on which Open Client routine is using the CS_DATAFMT. The following table lists the meanings of *maxlength*:

Open Client routine:	***maxlength* is:**
ct_bind	The length of the bind variable.
ct_describe	The maximum possible length of the column or parameter being described.
ct_param	The maximum desired length of return parameter data.
cs_convert	The length of the source data and the length of the destination buffer space.

Table 2-15: Meaning of maxlength *(CS_DATAFMT)*

scale is the scale of the data. *scale* is used only with decimal or numeric datatypes.

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_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 precision of the data. *precision* is used only with decimal or numeric datatypes.

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.

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 bit mask used to indicate various types of information. The following table lists the values that can make up *status*:

Symbolic Value:	**To Indicate:**	**Value is Legal For:**
CS_CANBENULL	The column can contain NULL values.	**ct_describe**, **ct_dyndesc**
CS_HIDDEN	The column is a "hidden" column that has been exposed. For more information, see the CS_HIDDEN_KEYS on the **Properties** topics page.	**ct_describe**, **ct_dyndesc**
CS_IDENTITY	The column is an identity column.	**ct_describe**, **ct_dyndesc**
CS_KEY	The column is a key column. For more information, see the manual page for **ct_keydata**.	**ct_describe**, **ct_dyndesc**
CS_UPDATABLE	The column is an updatable cursor column.	**ct_describe**, **ct_dyndesc**
CS_VERSION_KEY	The column is part of the version key for the row. SQL Server uses version keys for positioning cursors. For more information, see the manual page for **ct_keydata**.	**ct_describe**, **ct_dyndesc**

Table 2-16: Values for status *(CS_DATAFMT)*

Symbolic Value:	**To Indicate:**	**Value is Legal For:**
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_dyndesc**
CS_INPUTVALUE	The parameter is an input parameter value for a Client-Library command.	**ct_param**, **ct_dyndesc**
CS_RETURN	The parameter is a return parameter to an RPC command.	**ct_param**, **ct_dyndesc**

Table 2-16: Values for status *(CS_DATAFMT) (continued)*

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 the **Types** topics page.

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, also called an "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.

Once it has a valid CS_IODESC, a typical application will change 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];   /* SQL Server id.  */
     CS_INT       timestamplen;       /* Length of timestamp.*/
     CS_BYTE      textptr[CS_TP_SIZE];     /* SQL Server ptr. */
     CS_INT       textptrlen;         /* Length of textptr.  */
} CS_IODESC;
```

where:

iotype indicates the type of I/O to perform. For text and image operations, *iotype* always has the value CS_IODATA.

datatype is the datatype of the data object. The only legal 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 either by setting it to NULL or to the address of a valid CS_LOCALE structure.

usertype is the SQL 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_textlen is the total length, in bytes, of the text or image value.

offset is reserved for future use.

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 can set *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*.

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 in-line, **ct_diag** can return information in a CS_SERVERMSG.

For information on error and message handling, see "Error and Message Handling" on page 2-74.

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_NAME];
    CS_INT      svrnlen;

    /*
    ** If the error involved a stored procedure,
    ** the following fields contain information
    ** about the procedure:
    */
    CS_CHAR     proc[CS_MAX_NAME];
    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 SQL 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 SQL 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 2-77.

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 which 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* can 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. The following table lists the values that can be present in *status*:

Symbolic Value:	To Indicate:
CS_HASEED	Extended error data is included with the message. For more information on extended error data, see "Extended Error Data" on page 2-79.
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. For more information on sequenced messages, see "Sequencing Long Messages" on page 2-77.
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. For more information on sequenced messages, see "Sequencing Long Messages" on page 2-77.

Table 2-17: Values for status *(CS_SERVERMSG)*

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.

Cursors

A cursor is a symbolic name that is associated with one of the following SQL statements:

- A SQL **select** statement
- A Transact-SQL **execute** statement

  The stored procedure being executed can contain only a single SQL **select** statement.
- A dynamic SQL prepared statement where the statement can be either:
  - A SQL **select** statement
  - A Transact-SQL **execute** statement

    The stored procedure being executed can contain only a single SQL **select** statement.

The SQL statement associated with a cursor is called the **body** of the cursor.

Client-Library allows an application to declare and manipulate a cursor as either a language cursor (using **ct_command**) or a Client-Library cursor (using **ct_cursor**).

Both language cursors and Client-Library cursors can be used to send commands to SYBASE SQL Servers and Open Server applications.

A language cursor cannot be manipulated using **ct_cursor**; likewise, a Client-Library cursor cannot be manipulated using **ct_command**.

Language Cursors

Declaring Language Cursors

An application creates a language cursor by initiating a call to **ct_command**(CS_LANG_CMD) and specifying a cursor declare statement. For an application accessing SQL Server, the cursor declare statement would be a **declare cursor** command.

An application can declare one or more language cursors using the same or different command structures. The association between a language cursor and a command structure is short-lived: from the time at which a command is initiated (**ct_command**(CS_LANG_CMD)) until the command results are fully processed.

The following commands use Transact-SQL to declare two language cursors, specify their cursor rows settings, and open them, all in the same command structure space.

```
strcpy(buf, “declare A cursor for select * from \
                A_Table \
            set cursor rows 10 for A \
            declare B cursor for select * from \
                B_Table \
            set cursor rows 5 for B \
            open A \
            open B”);
```

```
ct_command(cmd, CS_LANG_CMD, buf, CS_NULLTERM,
    CS_UNUSED);
```

```
ct_send(cmd);
```

For detailed information about **ct_command**, see the **ct_command** manual page.

Regular Row Result Sets

Language cursors generate regular row result sets when an application fetches against them:

```
strcpy(buf2, “fetch A \
             fetch B”);
```

```
ct_command(cmd, CS_LANG_CMD, buf2, CS_NULLTERM,
    CS_UNUSED);
```

```
ct_send(cmd);
```

The number of rows returned to Client-Library per **fetch** command is equal to the current “cursor rows” setting for the cursor: 10 for cursor A and 5 for cursor B, in this case. It is useful to refer to this portion of the result set as a buffer’s worth of rows. The application fetches the rows from Client-Library by making calls to **ct_fetch**.

Fetching From Regular Row Result Sets

An application can simultaneously fetch from multiple regular row result sets. It must, however, completely process the buffer’s worth of rows returned by a cursor’s **fetch** command before it attempts to fetch rows from another cursor’s result set. A buffer’s worth of rows has been completely processed when the value returned from **ct_fetch** is CS_END_DATA.

An application can update or delete the most recently fetched row while fetching from a regular row result set. The modification is propagated back to the underlying database table.

Client-Library Cursors

Declaring Client-Library Cursors

An application creates a Client-Library cursor by initiating a call to **ct_cursor**(CS_CURSOR_DECLARE).

Unlike language cursors, each Client-Library cursor must be declared using a different command structure. All operations on the same Client-Library cursor, from its declaration to its de-allocation, must reference the same unique command structure.

The following commands use Transact-SQL to declare two Client-Library cursors and specify their cursor rows settings. The commands for each cursor use different command structures.

```
ct_cursor(cmd, CS_CURSOR_DECLARE, A, CS_NULLTERM,
    cursor_body, CS_NULLTERM, CS_FOR_UPDATE);

ct_cursor(cmd, CS_CURSOR_ROWS, NULL, CS_UNUSED,
    NULL, CS_UNUSED, 5);

ct_send(cmd);

ct_cursor(cmd2, CS_CURSOR_DECLARE, B, CS_NULLTERM,
    cursor_body2, CS_NULLTERM, CS_FOR_UPDATE);

ct_cursor(cmd2, CS_CURSOR_ROWS, NULL, CS_UNUSED,
    NULL, CS_UNUSED, 10);

ct_send(cmd2);
```

For detailed information about **ct_cursor**, see the **ct_cursor** manual page.

Cursor Result Sets

Client-Library cursors generate cursor result sets when an application makes calls to **ct_cursor**(CS_CURSOR_OPEN) to initiate commands to open the cursors.

```
ct_cursor(cmd, CS_CURSOR_OPEN, NULL, CS_UNUSED,
    NULL, CS_UNUSED, CS_UNUSED);

ct_send(cmd);

ct_cursor(cmd2, CS_CURSOR_OPEN, NULL, CS_UNUSED,
    NULL, CS_UNUSED, CS_UNUSED);
```

```
ct_send(cmd2);
```

The number of rows returned to Client-Library per internal fetch command is equal to the current "cursor rows" setting for the cursor: 10 for cursor A and 5 for cursor B, in this case. The application fetches the rows from Client-Library by making calls to **ct_fetch**.

Fetching From Cursor Result Sets

An application can simultaneously fetch from multiple cursor result sets. Unlike language cursors, an application does not need to completely process the buffer's worth of rows returned on a Client-Library cursor before it fetches rows from another cursor's result set.

Also, while a language cursor can only be used to delete or update the most recently fetched row, a Client-Library cursor can be used to modify **any** previously fetched row. This modification is propagated back to the underlying database table.

For information about using Client-Library to modify previously fetched cursor rows, see the **ct_keydata** manual page.

Language Cursor and Client-Library Cursor Interaction

Language cursors and Client-Library cursors can share the same connection structure. The implication of this is that whenever a language cursor sends a **fetch** command, all other cursors (including the Client-Library cursors) sharing the same connection are blocked until the application processes the buffer's worth of regular rows returned.

Where to Go for More Information

For detailed information on the routines that initiate language and Client-Library cursor commands, see the manual pages for **ct_command** and **ct_cursor.**

For information on how to declare and manipulate Client-Library cursors , see the *Open Client Client-Library/C Programmer's Guide*.

For information on how cursors are implemented in 10.0 SQL Server, see the *SQL Server Reference Manual*.

For information on how cursors are supported by 10.0 Open Server, see the *Open Server Server-Library Reference Manual*.

Dynamic SQL

Dynamic SQL is primarily useful for precompiler support, but it can also be used by interactive applications that do either of the following:

- Generate SQL statements based on information provided by an end-user
- Allow end-users to create whole or partial SQL statements

What is Dynamic SQL?

Dynamic SQL is the process of generating, preparing, and executing SQL statements at run-time.

Dynamic SQL addresses a number of issues, namely:

- The need to execute SQL statements whose text is not known prior to run-time.

  It would be difficult to anticipate all of the SQL statements that an end user might want to execute. An application can benefit from dynamically constructing SQL statements, binding variable values, and executing the statements, all at run-time.
- The efficiency of preparing a 'generic' SQL statement once and executing it multiple times, each time changing the values of its host variables.

  Preparing a SQL statement is analogous to compiling an application; the syntax of the SQL statement is checked and the DBMS has an opportunity to optimize the SQL statements, deciding on query plans and storing those plans in anticipation of later application execution.
- The advantage of referencing a prepared SQL statement with an identifier.
- The need to create one or more cursors at run-time to handle multiple-row access.

Limitations of Dynamic SQL

Dynamic SQL has some significant limitations.

Performance

Dynamic SQL generally performs poorer than static SQL, the term for SQL when it is embedded into an application. When you compile an embedded SQL application, SQL statement preparation and optimization is performed as well. The overhead incurred by these impacts the application developer, not the end-user.

A dynamic SQL application, on the other hand, incurs the overhead of SQL statement preparation and optimization at run-time, which affects the end-user.

ANSI Cursor Restriction

A dynamic SQL application using Transact-SQL cursors or Client-Library cursors (as opposed to language cursors) is subject to the following restriction.

By ANSI definition, a cursor is associated with a *single* result set, and thus, a single SQL statement. This means that a dynamic SQL prepared statement can only be either:

- A SQL **select** statement

or a:

- A Transact-SQL **execute** statement

  The stored procedure being executed can contain only a single SQL **select** statement.

SQL Server Restrictions

SQL Server implements dynamic SQL using temporary stored procedures. A temporary stored procedure is created when a SQL statement is prepared, and destroyed when that prepared statement is de-allocated. De-allocation can occur either explicitly with a **ct_dynamic**(CS_DEALLOC) call or implicitly when a connection is closed.

As a consequence of this implementation, an application accessing SQL Server and using dynamic SQL is subject to the restrictions of SQL Server stored procedures. Some of the implications of this are:

- Temporary tables are destroyed when the prepared statement is de-allocated.
- Parameters of text and image datatypes are not supported.
- The maximum number of parameters supported is 255.

See the *Transact-SQL User's Guide* for a complete discussion of stored procedures.

Dynamic SQL Implementation

There are two ways to dynamically execute SQL statements. One is to perform the prepare and execute operations in one step, and the other is to perform the prepare and execute operations separately.

Preparing a SQL statement consists of:

- Parsing - checking the statement's syntax and verifying the names of the specified columns and tables against the system catalog
- Optimizing - determining the data access path (execution plan), if possible
- Generating execution code

Executing a SQL statement is what actually makes things happen: rows are added by an **insert** statement, removed by a **delete** statement, changed by an **update** statement, or retrieved by a **select** statement.

The following sections discuss the two methods, and the circumstances in which an application would choose one method over the other.

Execute Immediate

The execute immediate method performs the prepare and execute operations in one step. 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 dynamic parameter markers (?'s).

  Dynamic parameter markers act as placeholders that allow users to specify actual data to be substituted into a SQL statement at run-time.
- The application will execute it only once.

  Using the execute immediate method, an application can execute a literal SQL statement more than once, but this is discouraged since it incurs the overhead associated with statement prepa-ration each time it executes the statement.

Prepare and Execute

The execute and prepare method performs the prepare and execute operations separately. An application *must* use this method if the dynamic SQL statement meets any of the following criteria:

- It returns data.

- It contains dynamic parameter markers (?'s).

and it *should* use this method if:

- The application will execute it multiple times.

  Using the prepare and execute method, an application incurs the overhead associated with statement preparation only once: when it prepares the statement. Each execution of the statement thereafter costs nothing in terms of overhead.

Separating the prepare and execute operations offers an application the following advantages over the execute immediate method:

- It allows **select** statements to be executed.
- It increases the performance of statements which are executed more than once.
- It provides the application with an opportunity to describe prepared statement input values.

Execute Immediate

To execute a literal, non-query, dynamic SQL statement:

1. Store the text of the dynamic SQL statement in a character string host variable.
2. Call **ct_dynamic** with *type* as CS_EXEC_IMMEDIATE to initiate a command to execute the statement.
3. Call **ct_send** to send the command to the server.
4. Call **ct_results** and examine the value of the **result_type* parameter to determine whether the command succeeded (CS_CMD_SUCCEED) or failed (CS_CMD_FAIL).

Prepare and Execute

Preparing and executing a dynamic SQL statement is more complex than performing an execute immediate operation. The steps are:

- Prepare the dynamic SQL statement.
- Get a description of prepared statement input, if necessary.
- Get a description of prepared statement output, if necessary.
- Execute the prepared statement or declare and open a cursor on the prepared statement.

- Process results, if necessary.
- De-allocate the prepared statement.

Preparing a Statement

When an application prepares a dynamic SQL statement separately from executing it, these additional tasks are performed during the prepare operation:

- The statement is associated with an identifier for easy access.
- The compiled statement is stored on the server for later execution.

To prepare a dynamic SQL statement:

1. Store the text of the statement in a character string host variable, for example:

   ```
   char    *query = “select type, title, price
                       from titles
                       where title_id = ?”
   ```

   The SQL statement may include one or more dynamic parameter markers that act as placeholders. A placeholder is represented by a “?” character. Placeholders can be specified:

   - For one or more columns in a **select** list
   - For one or more values in an **insert** statement
   - In the **set** clause of an **update** statement
   - In a **where** clause of a **select** or **update** statement

   At execution time, the application must substitute a value for each dynamic parameter marker.

2. Call **ct_dynamic** with *type* as CS_PREPARE to initiate a command to prepare the statement.

   To initiate a command to prepare the above SQL statement:

   ```
   ct_dynamic(cmd, CS_PREPARE, “myid”, CS_NULLTERM,
       query, CS_NULLTERM);
   ```

   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 actual name of the stored procedure to be executed:

   ```
   ct_dynamic(cmd, CS_PREPARE, “myid”, CS_NULLTERM,
       “exec sp_2”, CS_NULLTERM);
   ```

3. Call **ct_send** to send the command to the server.

4. Call **ct_results** as necessary to process the results of the command. A successful CS_PREPARE command will generate a CS_CMD_SUCCEED result.

Getting a Description of Prepared Statement Input

If a SQL statement contains any dynamic parameter markers, it is often useful for an application to get a description from the server of the values to be input. This description includes the number of input values, as well as their data types, lengths, and so on. The application can then use this information to prompt the end user for input values. After prompting for input values, it can pass those values to the prepared statement just prior to executing the statement.

To get a description of prepared statement input:

1. Call **ct_dynamic** with *type* as CS_DESCRIBE_INPUT to initiate a command to get the description.
2. Call **ct_send** to send the command to the server.
3. 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.
4. Call **ct_res_info** to get the number of input values. This assumes that CS_DESCRIBE_RESULT was returned, as does the following step.
5. For each input value, call **ct_describe**. Increment the value of the *item* parameter by 1 with each call.

Alternately, an application can use a dynamic SQL descriptor area to hold the description of the prepared statement input. If this is the case, the following steps replace those specified above:

1. Call **ct_dyndesc** with *operation* as CS_ALLOC to allocate a descriptor area.
2. Call **ct_dynamic** with *type* as CS_DESCRIBE_INPUT to initiate the command to get the description.
3. Call **ct_send** to send the command to the server.
4. 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.

5. 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 do the following two steps.
6. Call **ct_dyndesc** with *operation* as CS_GETCNT to get the number of input values.
7. For each input value, call **ct_dyndesc** with *operation* as CS_GETATT to get the value's description.

The first method (not using a dynamic SQL descriptor area) is recommended since performance is better and it is consistent with the way in which results are processed for non-dynamic SQL statements as well.

Getting a Description of Prepared Statement Output

If the dynamic SQL statement is a **select** statement and the **select** list was not known prior to run-time, the application must get a description of the prepared statement output before processing the results.

For example, a forms-based application that processes interactive SQL queries needs to determine result column types and lengths in order to display output.

To get a description of prepared statement output columns:

1. Call **ct_dynamic** with *type* as CS_DESCRIBE_OUTPUT to initiate a command to get the description.
2. Call **ct_send** to send the command to the server.
3. 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 of the output columns.
4. Call **ct_res_info** to get the number of output columns. This assumes that CS_DESCRIBE_RESULT was returned, as does the following step.
5. For each output column, call **ct_describe**. Increment the value of the *item* parameter by 1 with each call.

Alternately, an application can use a dynamic SQL descriptor area to hold the prepared statement output. If this is the case, the following steps replace those specified above:

1. Call **ct_dyndesc** with *operation* as CS_ALLOC to allocate a descriptor area.
2. Call **ct_dynamic** with *type* as CS_DESCRIBE_OUTPUT to initiate the command to get the description.
3. Call **ct_send** to send the command to the server.
4. Call **ct_results** as necessary to process the results of the command. A 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 of the output columns.
5. Call **ct_dyndesc** with *operation* as CS_USE_DESC to associate the results with the descriptor area allocated in step 1. This assumes that CS_DESCRIBE_RESULT was returned, as do the following two steps.
6. Call **ct_dyndesc** with *operation* as CS_GETCNT to get the number of output columns.
7. For each output column, call **ct_dyndesc** with *operation* as CS_GETATT to get the value's description.

Executing a Prepared Statement

To execute a previously-prepared statement:

1. Call **ct_dynamic** with *type* as CS_EXECUTE to initiate a command to execute the statement.
2. Define the input values to the SQL statement by performing the following steps for each input value:
   - Prompt the end-user for an input value.
   - Call **ct_param** to pass the input value to the SQL statement.

   Alternately, if the application is using a dynamic SQL descriptor area, perform these steps for each input value:
   - Prompt the end-user for an input value.
   - Call **ct_dyndesc** with *operation* as CS_SETATT to put the value into the descriptor area.

   If the application is using a dynamic SQL descriptor area, then after all the input values have been defined, associate the dynamic SQL descriptor area with the prepared statement:
   - Call **ct_dyndesc** with *operation* as CS_USE_DESC.

The input values are substituted for the dynamic parameter markers.

3. Call **ct_send** to send the command to the server.
4. Call **ct_results** as necessary to process the results of the command.

Declaring and Opening a Cursor on a Prepared Statement

Instead of executing a prepared statement, an application can declare and open a cursor on it. The prepared statement serves as the body of the cursor. When the application opens the cursor, the prepared statement is executed. Any results generated are available to the application as a cursor row result set.

To declare and open a cursor on a prepared statement:

1. Call **ct_dynamic** with *type* as CS_CURSOR_DECLARE to initiate a command to declare a cursor.

   Note that **ct_dynamic**, not **ct_cursor**, is used to declare a cursor on a prepared statement.
2. Call **ct_cursor** with *type* as CS_CURSOR_OPTION to set options for the cursor.
3. Call **ct_send** to send the command to the server.
4. Call **ct_results** as necessary to process the results of the command.
5. Call **ct_cursor** with *type* as CS_CURSOR_OPEN to initiate a command to open the cursor.
6. Define the input values to the SQL statement by performing the following two steps for each input value:
   - Prompt the end-user for an input value.
   - Call **ct_param** to pass the input value to the SQL statement.

   Alternately, if the application is using a dynamic SQL descriptor area, perform the following two steps for each input value:
   - Prompt the end-user for an input value.
   - Call **ct_dyndesc** with *operation* as CS_SETATT to put the value into the descriptor area.

   To then associate the dynamic SQL descriptor area with the prepared statement:
   - Call **ct_dyndesc** with *operation* as CS_USE_DESC.

   The input values are substituted for the dynamic parameter markers.

7. Call **ct_send** to send the command to the server.
8. Call **ct_results** as necessary to process the results of the command.

If desired, an application can batch together the commands to declare and open the cursor by eliminating the **ct_send** and **ct_results** calls that follow the **ct_cursor**(CS_CURSOR_OPTION) call.

Processing Results

Processing the results of a dynamic SQL statement is the same as processing the results of any other SQL statement.

See the **Results** topics page, the **ct_results** manual page, and the "Step 5: Processing Results" section of the *Open Client Client-Library/C Programmer's Guide* for detailed information about processing results.

De-allocating a Prepared Statement

When an application is done with a prepared statement, it can de-allocate the statement and free any resources associated with it.

To de-allocate a prepared statement:

1. If the application used descriptor areas for the prepared statement input and output, de-allocate the descriptor areas by calling **dt_dyndesc** with *operation* as CS_DEALLOC once for each descriptor area.
2. Call **ct_dynamic** with *type* as CS_DEALLOC to initiate a command to de-allocate the prepared statement.
3. If the application declared and opened a cursor on the prepared statement, call **ct_cursor** with *type* as CS_CURSOR_CLOSE and *option* as CS_DEALLOC to initiate a command both to close and de-allocate the cursor.
4. Call **ct_send** to send the command to de-allocate the statement.
5. Call **ct_results** as necessary to process the results of the command.

Alternatives to Dynamic SQL

Because of the numerous restrictions of dynamic SQL, we recommend that applications use stored procedures to accomplish the same tasks. Stored procedures offer identical functionality to dynamic SQL except for the ability to get a description of prepared statement input: creating a stored procedure is analogous to preparing a SQL statement, a stored procedure’s input parameters serve the same purpose as do dynamic parameter markers, and executing a stored procedure is equivalent to executing a prepared statement.

Error and Message Handling

All Client-Library routines return success or failure indications. It is highly recommended that applications check these return codes.

In addition, Client-Library applications must handle two types of error and informational messages:

- Client-Library messages, also known as "'client messages", are generated by Client-Library. They range in severity from informational messages to fatal errors.
- Server messages are generated by the server. Server messages also range in severity from informational messages to fatal errors. SQL Server messages can be listed by executing the Transact-SQL command "select * from sysmessages". See the *Open Server Server-Library Reference Manual* for a list of Open Server messages.

➤ ***Note***

Don't 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 in-line, using **ct_diag**. A message result set, on the other hand, is one of several types of result sets a server can 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 can handle Client-Library and server messages in one of two ways:

- By installing callback routines to handle messages
- In-line, 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 main-line error-handling logic may not successfully trap errors that have not been anticipated.

In-line 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 will use the callback method to handle messages. However, an application that is running on a platform / language combination that does not support callbacks must use the in-line method.

An application indicates which method it will use by calling **ct_callback** to install message callbacks or by calling **ct_diag** to initialize in-line message handling.

An application can use different methods on different connections. For example, an application can install message callbacks at the context level, allocate two connections, and then call **ct_diag** to initialize in-line 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 can switch back and forth between the in-line and the callback methods:

- Installing either a client message callback or a server message callback turns off in-line message handling. Any saved messages are discarded.
- Likewise, calling **ct_diag** to initialize in-line 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 in-line 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 on using callbacks to handle Client-Library and server messages, see "Callbacks" on page 2-11 and the manual page for **ct_callback**, page 3- 21.

In-Line Message Handling

An application calls **ct_diag** to initialize in-line 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 in-line 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," below, 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 on the in-line method of handling Client-Library and server messages, see the manual page for **ct_diag**, 3-114.

Client-Library's Message Structures

Client-Library uses the following structures to return message information:

- The CS_CLIENTMSG structure, documented on page 2-45
- The CS_SERVERMSG structure, documented on page 2-56
- The SQLCA structure, documented on page 2-181
- The SQLCODE structure, documented on page 2-183

- The SQLSTATE structure, documented on page 2-184

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, a loss of information will occur.

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 - 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 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.

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.

➤ ***Note***

Operating system messages, if any, are reported via 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 table lists *status* values that are related to sequenced messages:

Symbolic Value:	To Indicate:
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.

Table 2-18: status *values for sequenced messages*

  - 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 always 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 on extended error data, see "Extended Error Data" on page 2-79.

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) call with *index* i returns the i'th message chunk, not the i'th message.
- 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 SQL Server messages, the additional information is most typically 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 can be named, and can be of any datatype.

An application can retrieve extended error data but is not required to do so.

What's Extended Error Data Good For?

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.

How Can an Application Tell if Extended Error Data is Available?

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.

Once 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**.

In-Line Error Handling and Extended Error Data

An application that is handling server messages in-line 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.

Once 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:

Symbolic Value:	To Indicate:
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.

Table 2-19: Transaction states

Retrieving Transaction States in Main-Line Code

In main-line 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 2-19: Transaction states* on page 2-81.

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 can retrieve 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 can retrieve the current transaction state by:

1. Retrieving the CS_COMMAND with the extended error data by calling **ct_con_props** with *property* as CS_EED_CMD.
2. Calling **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 2-19: Transaction states*.

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.
  - Typedefs for common client / server structures.
  - Declarations for CS-Library routines.
  - *cstypes.h*, which contains typedefs for Client-Library datatypes.
  - *sqlca.h*, which contains a typedef for the SQLCA structure.
  - *csconfig.h*, which contains platform-dependent datatypes and definitions.

International Support

Client-Library provides support for international applications by allowing them to **localize**. An application that is localized typically:

- Uses a local language for Client-Library and SQL Server messages
- Uses local 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 than an application will use. If these default values meet an application's needs, it does not have to localize further.

If the default values don't meet an application's needs, it can use a CS_LOCALE structure to set custom localization values at the context, connection, or data element levels. For information on how to use a CS_LOCALE structure, see "Using a CS_LOCALE Structure" on page 2-85.

◆ ***WARNING!***

Platform-specific localization issues are discussed in the International Support chapter of the *Open Client/Server Supplement*. You must read this chapter in order to understand Client-Library's localization mechanism and how environment variables affect localization.

When Does an Application Need to Use a CS_LOCALE?

An application needs to use a CS_LOCALE structure if Client-Library's default localization values don't meet its needs. For information on how Client-Library sets up default localization values, see the *Open Client/Server Supplement* for your platform.

Typically, an application needs to use a CS_LOCALE if it will be working in a language or character set that differs from the predominant local language and character set.

For example:

- An application running in a German environment might need to use a CS_LOCALE in order to receive Client-Library error messages in French.

- An English gateway application that supports Danish clients might need to use a CS_LOCALE.

Using a CS_LOCALE Structure

A CS_LOCALE structure defines localization values. An application can use a 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 can:
   - Call **cs_config** with *property* as CS_LOC_PROP to copy the custom localization values into a context structure.
   - Call **ct_con_props** with *property* as CS_LOC_PROP to copy the custom localization values into a connection structure. Note that because CS_LOC_PROP is a login property, an application cannot change its value after a connection is open.
   - Supply the CS_LOCALE as a parameter to a routine that accepts custom localization values (**cs_strcmp**, **cs_time**).
   - Include the CS_LOCALE in a CS_DATAFMT structure describing a destination program variable (**cs_convert**, **ct_bind**).
4. Calls **cs_loc_drop** to de-allocate 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 on how default localization values are assigned on your platform, see the *Open Client/Server Supplement*.

Because default localization values are always defined, an application only needs to define new context-level localization values 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.

◆ ***WARNING!***

This functionality is very different from that of DB-Library, where a connection uses SQL Server's default national language unless the application calls DBSETLNATLANG to set the national language name.

Data-Element-Level Localization

A data-element-level CS_LOCALE defines localization values for a specific data element, for example, a routine parameter or 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 an English locale (us_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 can:

- Load a CS_LOCALE with French datetime formats.
- Call **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 day and month names and copied into the bound variable.

Where Does Client-Library Look for Localization Information?

When determining which localization values to use, Client-Library starts at the data element level and proceeds up. 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 can 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 / Server products use the locales file when loading localization information.

The locales file directs Open Client / Server products to language, character set, and sort order names, but does not contain actual localized messages or character set information.

For more information on the locales file, see the *Open Client/Server Supplement*.

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 = C.japanese, us_english, eucjis
```

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 “C.japanese” is specified, a language of “us_english”, a character set of “eucjis”, and a sort order of “binary” (the default sort order) should be used.

Predefined Locale Names

SYBASE pre-defines some locale names by including entries for them in the locales file. If these entries don’t meet your needs, you can either modify them or add additional 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, **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 on this search, see the *Open Client/Server Supplement* for your platform.
2. Looks the locale name up 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 the 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.

Logical Sequence of Calls

Client-Library uses state machines to define and enforce the order in which applications call Client-Library routines. This defined order is known as ‘a logical sequence’. For example, an application must send a SQL query statement to a server before it can execute the statement, and it must execute a statement before it can fetch rows from the statement’s result set.

Client-Library State Machines

The application programming interface (API) layer of Client-Library consists of three state machines, each corresponding to one of the three basic control structures: CS_CONTEXT, CS_CONNECTION, or CS_COMMAND. See chapter 3 of the *Open Client Client-Library/C Programmer’s Guide* for a discussion of the basic control structures.

At the context level, an application sets up its environment: allocating one or more context structures, setting CS-Library properties for the contexts, initializing Client-Library, and setting Client-Library properties for the contexts.

At the connection level, an application connects to a server: allocating one or more connection structures, setting properties for the connections, opening the connections, and setting any server options for the connections. An application can allocate a connection structure only after a context structure has been allocated.

At the command level, an application allocates one or more command structures, sends commands, and processes results. An application can allocate a command structure only after a connection structure has been allocated.

Command Level Sequence of Calls

It is at the command level that the logical sequence of calls becomes complex. This is due to the number of routines that are managed at the command level, compared to the number managed at the context and connection levels.

Client-Library’s command state machine gets help from two other state tables when it attempts to verify that a call to a particular routine is permitted. These are the initiated commands state table and the result types state table.

Commands State Table

The commands table defines 'states' in which an application can be. For example, it defines a Command Sent state which indicates that the last thing an application did was send a command to a server (via **ct_send**).

The commands table also maps each state to valid Client-Library routines that an application can call while in that state. For example, in the Command Sent state, an application can cancel the command or the result set, get or set command structure properties, perform operations on a dynamic SQL descriptor area, receive a TDS packet from the server, or set up results for processing.

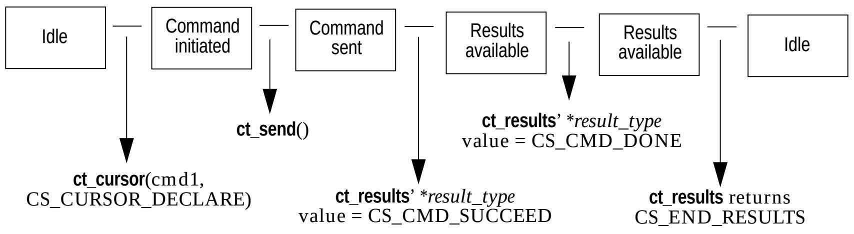
See "Command States" on page 2-92 for a detailed description of each of the command states. See "Callable Routines in Each Command State" on page 2-95 for a mapping of command states with Client-Library routines.

Initiated Commands State Table

The initiated commands table focuses on routines that initiate and set up commands to be sent to a server (**ct_command**, **ct_cursor**, **ct_dynamic**, **ct_param**, and so on). It provides a finer level of enforcement than is possible with the commands table.

For example, the command state machine ensures that **ct_param** is called only after a command has been initiated. However, it cannot prevent an application from calling **ct_param** when the initiated command does not take parameters (as in the case of a **ct_cursor**(CS_CURSOR_CLOSE)). It is in cases like these that the initiated commands table enforces the logical sequence of calls.

As another example, assume that a Client-Library cursor is declared using the *cmd1* CS_COMMAND structure. After the cursor declare command is sent to the server and the results processed, the state machine is back in an Idle state:


Idle
Command initiated
Command sent
Results available
Results available
Idle
ct_cursor(cmd1, CS_CURSOR_DECLARE)
ct_send()
ct_results' *result_type value = CS_CMD_SUCCEED
ct_results' *result_type value = CS_CMD_DONE
ct_results returns CS_END_RESULTS

From an Idle state, the command state machine permits an application to initiate a new command. This means that it cannot prevent an application from declaring a second cursor using the same CS_COMMAND structure that it used to declare the first cursor (*cmd1*).

The initiated commands table, however, keeps track of the state of a cursor on a command handle. It recognizes that if a cursor has been previously declared using a particular CS_COMMAND structure, a second attempt to declare a cursor using the same CS_COMMAND structure is illegal.

See "Initiated Commands" on page 2-104 for a detailed description of each of the initiated command states. See "Callable Routines for Initiated Commands" on page 2-107 for a mapping of initiated command states with Client-Library routines.

Result Types State Table

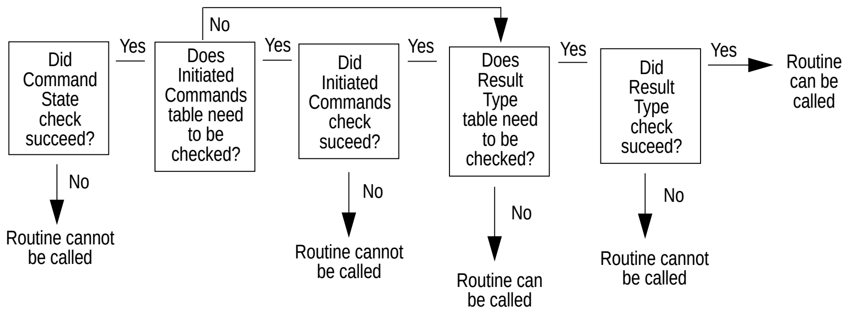
The result types table focuses on routines that return information about result sets. The command state machine defines states (like Fetchable Results and Fetchable Cursor Results) which indicate that results are available. The result types table goes a step farther by indicating the type of available results.

This information is important because certain routines only make sense for certain result types. For example, calling **ct_compute_info** is only logical when compute results are available, and calling **ct_br_column** is only logical when regular row results are available. It is in cases like these that the result types table enforces the logical sequence of calls.

See "Result Types" on page 2-108 for a detailed description of each of the result type states. See "Callable Routines for Each Result Type" on page 2-110 for a mapping of result type states with Client-Library routines.

Summary

This diagram shows how the state tables work together:


Did Command State check succeed?
Yes
No
Routine cannot be called
Does Initiated Commands table need to be checked?
No
Yes
Did Initiated Commands check suceed?
No
Routine cannot be called
Yes
Does Result Type table need to be checked?
No
Routine can be called
Yes
Did Result Type check suceed?
No
Routine cannot be called
Yes
Routine can be called

Be aware that if there are multiple command structures sharing the same connection,

The information that follows is intended to serve as a reference for valid Client-Library application behavior. Use it when you want to verify that a particular sequence of routine calls is valid or when you need to know 'where to go from here.'

➤ ***Note***

The important point to remember is that Client-Library enforces the logical sequence of calls *for* you. It returns descriptive error messages at run-time if an application has not called routines in a logical sequence.

Command States

Client-Library keeps track of a command's current state. A command can be in any one of the following states.

Command State:	What it Indicates:
Idle	The application either: 1) hasn't yet initiated a command, 2) has completely processed the results of the last command, 3) has fetched all cursor rows but has not closed the Client-Library cursor, or 4) has closed a Client-Library cursor that is still associated with unprocessed results.

Table 2-20: Command states

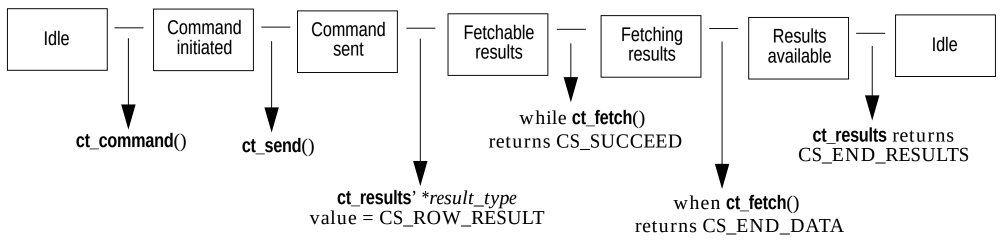
Command State:	What it Indicates:
Command initiated	The application called **ct_command**, **ct_cursor**, or **ct_dynamic** to initiate a command, but it hasn't yet sent it to the server.
Command sent	The application called **ct_send** to send a command to the server, but it hasn't yet called **ct_results** to set up result data for processing.
Non-fetchable results available	The application called **ct_results** and the result set contains no actual result data. Additional calls to **ct_results** are necessary. or: The application called **ct_fetch** which returned CS_END_DATA.
ANSI-style cursor end-data	The application called **ct_fetch** which returned CS_END_DATA and the CS_ANSI_BINDS property is set.
Fetchable results	The application called **ct_results** and the result set contains fetchable results (compute results, return parameter results, regular row results, and stored procedure return status results). **ct_fetch** has not been called yet.
Fetchable cursor results	The application called **ct_results** and the result set contains fetchable cursor results. **ct_fetch** has not been called yet.
Fetchable nested command	The application initiated a cursor close command (**ct_cursor**(CS_CURSOR_CLOSE)) before fetching from a result set that contains fetchable cursor results.
Sent fetchable nested command	The application called **ct_send** to send the cursor close command to the server before fetching from a result set that contains fetchable cursor results.
Processing fetchable nested command	The application called **ct_results** to process the results of the cursor close command before fetching from a result set that contains fetchable cursor results.
Fetching results	The application called **ct_fetch** at least once and is currently in the process of fetching results (compute results, return parameter results, regular row results, and stored procedure return status results).
Fetching cursor results	The application called **ct_fetch** at least once and is currently in the process of fetching cursor row results.

Table 2-20: Command states (continued)

Command State:	What it Indicates:
Fetching nested command	The application initiated a cursor close (**ct_cursor**(CS_CURSOR_CLOSE)), cursor update (**ct_cursor**(CS_CURSOR_UPDATE)), or cursor delete (**ct_cursor**(CS_CURSOR_DELETE)) command while fetching from a result set that contains cursor results.
Sent fetching nested command	The application called **ct_send** to send the cursor close, cursor update, or cursor delete command to the server while fetching from a result set that contains cursor results.
Processing fetching nested command	The application called **ct_results** to process the results of the cursor close, cursor update, or cursor delete command while fetching from a result set that contains cursor results.
Result set canceled	The application canceled the current command (**ct_cancel**(CS_CANCEL_ALL)). An application can call **ct_results** once more to return the command to an Idle state.
Undefined	The command structure is in an undefined state. Call **ct_cancel**(CS_CANCEL_ALL).
In receive passthrough	The application called **ct_recvpassthru** and CS_PASSTHRU_MORE was returned.
In send passthrough	The application called **ct_sendpassthru** and CS_PASSTHRU_MORE was returned.

Table 2-20: Command states (continued)

This diagram shows a command transitioning through several command states.


Idle
Command initiated
Command sent
Fetchable results
Fetching results
Results available
Idle
ct_command()
ct_send()
ct_results' *result_type value = CS_ROW_RESULT
while **ct_fetch**() returns CS_SUCCEED
when **ct_fetch**() returns CS_END_DATA
ct_results returns CS_END_RESULTS

Command-level Routines

These Client-Library routines are managed at the command level:

ct_bind	**ct_command**	**ct_dyndesc**	**ct_res_info**
ct_br_column	**ct_compute_info**	**ct_fetch**	**ct_results**
ct_br_table	**ct_cursor**	**ct_get_data**	**ct_send**
ct_cancel	**ct_data_info**	**ct_getformat**	**ct_send_data**
ct_cmd_drop	**ct_describe**	**ct_keydata**	**ct_recvpassthru**
ct_cmd_props	**ct_dynamic**	**ct_param**	**ct_sendpassthru**

Callable Routines in Each Command State

This table maps each command state to the Client-Library routines that an application can legally call while in that state. It also identifies the state of the command after the routine completes.

Beginning State:	Callable Routines:	Resulting Command State:
Idle	**ct_cancel**(CS_CANCEL_ALL) **ct_cancel**(CS_CANCEL_ATTN)	Idle, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_drop	Idle.
	ct_cmd_props	Idle.
	ct_command	Command initiated, if CS_SUCCEED. Idle, if CS_FAIL.
	ct_cursor	Command initiated, if CS_SUCCEED. Idle, if CS_FAIL.
	ct_dynamic	Command initiated, if CS_SUCCEED. Idle, if CS_FAIL.
	ct_dyndesc	Idle.
	ct_sendpassthru	In send passthrough, if CS_PASSTHRU_MORE. Command sent, if CS_PASSTHRU_EOM. Undefined, if CS_FAIL.
Command initiated	**ct_cancel**(CS_CANCEL_ALL)	Idle, if CS_SUCCEED. Command initiated, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Command initiated.
	ct_cmd_props	Command initiated.
	ct_cursor	Command initiated.
	ct_data_info(CS_SET)	Command initiated.
	ct_dyndesc	Command initiated.
	ct_param	Command initiated.

Table 2-21: Callable routines at each command state

Beginning State:	Callable Routines:	Resulting Command State:
	ct_send	Command sent, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_send_data	Command initiated, if CS_SUCCEED. Undefined, if CS_FAIL.
Command sent	**ct_cancel**(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Command sent, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	Command sent
	ct_dyndesc	Command sent
	ct_recvpassthru	In receive passthrough, if CS_PASSTHRU_MORE. Idle, if CS_PASSTHRU_EOM, CS_CANCELED. Undefined, if CS_FAIL.
	ct_results	Non-fetchable results available, if CS_SUCCEED and **result_type* equals CS_MSG_RESULT, CS_CMD_SUCCEED, CS_CMD_FAIL, CS_CMD_DONE, CS_ROWFMT_RESULT, CS_COMPUTEFMT_RESULT, or CS_DESCRIBE_RESULT. Fetchable results, if CS_SUCCEED and **result_type* equals CS_ROW_RESULT, CS_COMPUTE_RESULT, CS_PARAM_RESULT, or CS_STATUS_RESULT. Fetchable cursor results, if CS_SUCCEED and **result_type* equals CS_CURSOR_RESULT. Idle, if CS_CANCELED or CS_END_RESULTS. Undefined, if CS_SUCCEED and **result_type* equals CS_CMD_FAIL.
Non-fetchable results available	**ct_br_column**	Non-fetchable results available.
	ct_br_table	Non-fetchable results available.

Table 2-21: Callable routines at each command state

Beginning State:	Callable Routines:	Resulting Command State:
	ct_cancel(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Non-fetchable results available, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_CURRENT)	Non-fetchable results available, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	Non-fetchable results available.
	ct_compute_info	Non-fetchable results available.
	ct_describe	Non-fetchable results available.
	ct_dyndesc	Non-fetchable results available.
	ct_getformat	Non-fetchable results available.
	ct_res_info	Non-fetchable results available.
	ct_results	Fetchable results, if CS_SUCCEED and **result_type* equals CS_ROW_RESULT, CS_COMPUTE_RESULT, CS_PARAM_RESULT, or CS_STATUS_RESULT. Fetchable cursor results, if CS_SUCCEED and **result_type* equals CS_CURSOR_RESULT. Idle, if CS_CANCELED or CS_END_RESULTS. Undefined, if CS_FAIL.
ANSI-style cursor end-data	**ct_bind**	ANSI-style cursor end-data.
	ct_br_column	ANSI-style cursor end-data.
	ct_br_table	ANSI-style cursor end-data.
	ct_cancel(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	ANSI-style cursor end-data if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_CURRENT)	ANSI-style cursor end-data.
	ct_cmd_props	ANSI-style cursor end-data.

Table 2-21: Callable routines at each command state

Beginning State:	**Callable Routines:**	**Resulting Command State:**
	ct_compute_info	ANSI-style cursor end-data.
	ct_describe	ANSI-style cursor end-data.
	ct_dyndesc	ANSI-style cursor end-data.
	ct_fetch	ANSI-style cursor end-data, if CS_END_DATA. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_getformat	ANSI-style cursor end-data.
	ct_res_info	ANSI-style cursor end-data.
	ct_results	Non-fetchable results available, if CS_SUCCEED and **result_type* equals CS_MSG_RESULT or CS_CMD_DONE. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
Fetchable results	**ct_bind**	Fetchable results.
	ct_br_column	Fetchable results.
	ct_br_table	Fetchable results.
	ct_cancel(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Fetchable results, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_CURRENT)	Non-fetchable results available, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_cmd_props	Fetchable results.
	ct_compute_info	Fetchable results.
	ct_describe	Fetchable results.
	ct_dyndesc	Fetchable results.
	ct_fetch	Fetching results, if CS_SUCCEED or CS_ROW_FAIL. Non-fetchable results available, if CS_END_DATA. Idle, if CS_CANCELED. Undefined, if CS_FAIL.

Table 2-21: Callable routines at each command state

Beginning State:	Callable Routines:	Resulting Command State:
	ct_getformat	Fetchable results.
	ct_res_info	Fetchable results.
Fetchable cursor results	**ct_bind**	Fetchable cursor results.
	ct_cancel(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Fetchable cursor results, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_CURRENT)	Non-fetchable results available, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_cmd_props	Fetchable cursor results.
	ct_cursor	Fetchable nested command, if CS_SUCCEED. Fetchable cursor results, if CS_FAIL.
	ct_describe	Fetchable cursor results.
	ct_dyndesc	Fetchable cursor results.
	ct_fetch	Fetching cursor results, if CS_SUCCEED or CS_ROW_FAIL. Idle, if CS_CANCELED. Non-fetchable results available, if CS_END_DATA. Ansi-style cursor end-data, if CS_END_DATA and CS_ANSI_BINDS property is set. Undefined, if CS_FAIL.
	ct_getformat	Fetchable cursor results.
	ct_res_info	Fetchable cursor results.
Fetchable nested command	**ct_cancel**(CS_CANCEL_ALL)	Fetchable cursor results, if CS_SUCCEED. Fetchable nested command, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Fetchable nested command, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	Fetchable nested command.
	ct_dyndesc	Fetchable nested command.

Table 2-21: Callable routines at each command state

Beginning State:	Callable Routines:	Resulting Command State:
	ct_param	Fetchable nested command.
	ct_send	Sent fetchable nested, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
Sent fetchable nested	**ct_cancel**(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Sent fetchable nested, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	Sent fetchable nested.
	ct_results	Processing fetchable nested command, if CS_CMD_SUCCEED or CS_CMD_FAIL. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
Processing fetchable nested command	**ct_cancel**(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Processing fetchable nested command, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_CURRENT)	Processing fetchable nested command, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	Processing fetchable nested command.
	ct_dyndesc	Processing fetchable nested command.
	ct_res_info	Processing fetchable nested command.
	ct_results	Fetchable cursor results, if CS_END_RESULTS. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
Fetching results	**ct_bind**	Fetching results.
	ct_br_column	Fetching results.
	ct_br_table	Fetching results.
	ct_cancel(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.

Table 2-21: Callable routines at each command state

Beginning State:	Callable Routines:	Resulting Command State:
	ct_cancel(CS_CANCEL_ATTN)	Fetching results, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_CURRENT)	Non-fetchable results available, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_cmd_props	Fetching results.
	ct_compute_info	Fetching results.
	ct_data_info(CS_GET)	Fetching results.
	ct_describe	Fetching results.
	ct_dyndesc	Fetching results, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_fetch	Fetching results, if CS_SUCCEED. Non-fetchable results available, if CS_END_DATA. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_get_data	Fetching results, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_getformat	Fetching results.
	ct_res_info	Fetching results.
Fetching cursor results	**ct_bind**	Fetching cursor results.
	ct_cancel(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Fetching cursor results, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_CURRENT)	Non-fetchable results available, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_cmd_props	Fetching cursor results.

Table 2-21: Callable routines at each command state

Beginning State:	Callable Routines:	Resulting Command State:
	ct_cursor	Fetching nested command, if CS_SUCCEED. Fetching cursor results, if CS_FAIL.
	ct_describe	Fetching cursor results.
	ct_dyndesc	Fetching cursor results, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_fetch	Fetching cursor results, if CS_SUCCEED. Non-fetchable results available, if CS_END_DATA. Ansi-style cursor end-data, if CS_END_DATA and CS_ANSI_BINDS property is set. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_get_data	Fetching cursor results, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
	ct_getformat	Fetching cursor results.
	ct_keydata	Fetching cursor results.
	ct_res_info	Fetching cursor results.
Fetching nested command	**ct_cancel**(CS_CANCEL_ALL)	Fetching cursor results, if CS_SUCCEED. Fetching nested command, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Fetching nested command, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	Fetching nested command.
	ct_dyndesc	Fetching nested command.
	ct_param	Fetching nested command.
	ct_send	Sent fetching nested command, if CS_SUCCEED. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
Sent fetching nested command	**ct_cancel**(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.

Table 2-21: Callable routines at each command state

Beginning State:	Callable Routines:	Resulting Command State:
	ct_cancel(CS_CANCEL_ATTN)	Sent fetching nested command, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	Sent fetching nested command.
	ct_results	Processing fetching nested command, if CS_CMD_SUCCEED or CS_CMD_FAIL. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
Processing fetching nested command	**ct_cancel**(CS_CANCEL_ALL)	Result set canceled, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Processing fetching nested command, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_CURRENT)	Processing fetching nested command, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	Processing fetching nested command.
	ct_dyndesc	Processing fetching nested command.
	ct_keydata	Processing fetching nested command.
	ct_res_info	Processing fetching nested command.
	ct_results	Processing fetching nested command, if CS_SUCCEED. Fetching cursor results, if CS_END_RESULTS. Idle, if CS_CANCELED. Undefined, if CS_FAIL.
Result set canceled	**ct_cancel**(CS_CANCEL_ALL)	Idle, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Idle, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_drop	Idle.
	ct_cmd_props	Idle.
	ct_command	Command initiated, if CS_SUCCEED. Idle, if CS_FAIL.
	ct_cursor	Command initiated, if CS_SUCCEED. Idle, if CS_FAIL.

Table 2-21: Callable routines at each command state

Beginning State:	Callable Routines:	Resulting Command State:
	ct_dynamic	Command initiated, if CS_SUCCEED. Idle, if CS_FAIL.
	ct_dyndesc	Idle, if CS_SUCCEED, CS_ROW_FAIL, or CS_CANCELED. Undefined, if CS_FAIL.
	ct_results	Result set canceled, if CS_SUCCEED or CS_FAIL. Idle, if CS_CANCELED.
	ct_sendpassthru	Result set canceled.
Undefined	**ct_cancel**(CS_CANCEL_ALL)	Idle, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	Undefined.
	ct_cmd_props	Undefined.
	ct_dyndesc	Undefined.
In receive passthrough	**ct_cancel**(CS_CANCEL_ALL)	Idle, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	In receive passthrough, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	In receive passthrough.
	ct_recvpassthru	Idle, if CS_PASSTHRU_EOM or CS_CANCELED. Undefined, if CS_FAIL.
In send passthrough	**ct_cancel**(CS_CANCEL_ALL)	Idle, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cancel(CS_CANCEL_ATTN)	In send passthrough, if CS_SUCCEED. Undefined, if CS_FAIL.
	ct_cmd_props	In send passthrough.
	ct_sendpassthru	Command sent, if CS_PASSTHRU_EOM. Idle, if CS_CANCELED. Undefined, if CS_FAIL.

Table 2-21: Callable routines at each command state

Initiated Commands

In addition to command states, Client-Library keeps track of initiated commands. An initiated command can be in any one of the following states.

Initiated Command:	What it Indicates:
Idle	The application either hasn't yet initiated a command or has completely processed the results of the last command.
Idle, with declared cursor	The application initiated a cursor declare command (**ct_cursor**(CS_CURSOR_DECLARE)), sent the command to the server, and completely processed the results.
Idle, with opened cursor	The application initiated a cursor open command (**ct_cursor**(CS_CURSOR_OPEN)) and fetched all the results (**ct_results** returned CS_END_RESULTS), but has not yet closed the cursor.
Opened cursor, no rows fetched	The application called **ct_results**, but hasn't yet processed any of the results.
Opened cursor, fetching rows	The application called **ct_fetch** at least once and is currently in the process of fetching results.
Command initiated	The application initiated a language, message, package, or RPC command via **ct_command**.
Sent data	The application initiated a send-data or send-bulk-data command via **ct_command**.
Declared cursor	The application initiated a cursor declare command (**ct_cursor**(CS_CURSOR_DECLARE)), but hasn't yet sent it to a server via **ct_send**.
Set cursor rows	The application initiated a cursor rows command via **ct_cursor**(CS_CURSOR_ROWS).
Opened cursor	The application initiated a cursor open command (**ct_cursor**(CS_CURSOR_OPEN)), but hasn't yet sent it to a server.
Closed cursor	The application initiated a cursor close command (**ct_cursor**(CS_CURSOR_CLOSE)) but hasn't yet sent it to a server.
De-allocated cursor	The application initiated a cursor de-allocate command (**ct_cursor**(CS_CURSOR_DEALLOC)) but hasn't yet sent it to a server.
Updated cursor row	The application initiated a cursor update command (**ct_cursor**(CS_CURSOR_UPDATE)) but hasn't yet sent it to a server.

Table 2-22: Initiated command states

Initiated Command:	**What it Indicates:**
Deleted cursor row	The application initiated a cursor delete command (**ct_cursor**(CS_CURSOR_DELETE)) but hasn't yet sent it to a server.
Dynamic cursor declared	The application initiated a cursor declare command on a dynamically prepared SQL statement (**ct_dynamic**(CS_CURSOR_DECLARE)) but hasn't yet sent it to a server.
Dynamic de-allocated	The application initiated a command to de-allocate a prepared SQL statement (**ct_dynamic**(CS_DEALLOC)) but hasn't yet sent it to a server.
Dynamic described	The application initiated a command to retrieve input parameter information (**ct_dynamic**(CS_DESCRIBE_INPUT)) or column list information (**ct_dynamic**(CS_DESCRIBE_OUTPUT)) but hasn't yet sent it to a server.
Dynamic executed	The application initiated a command to execute a prepared SQL statement (**ct_dynamic**(CS_EXECUTE)) but hasn't yet sent it to a server.
Dynamic execute immediate	The application initiated a command to execute a literal SQL statement (**ct_dynamic**(CS_EXEC_IMMEDIATE)) but hasn't yet sent it to a server.
Dynamic prepare	The application initiated a command to prepare a SQL statement (**ct_dynamic**(CS_PREPARE)) but hasn't yet sent it to a server.
ct_send_data succeeded	The application successfully called **ct_send_data** at least once.
Bulk copy	The application initiated a send-bulk-data command (**ct_command**(CS_SEND_BULK_CMD)) but hasn't yet sent it to a server.

Table 2-22: Initiated command states (continued)

Initiated Command Routines

The following Client-Library routines are useful for processing initiated commands:

ct_cmd_drop	**ct_dyndesc**
ct_command	**ct_param**
ct_cursor	**ct_send_data**
ct_data_info	**ct_sendpassthru**
ct_dynamic	

Callable Routines for Initiated Commands

This table maps each initiated command state to the Client-Library routines that an application can legally call while in that state.

Where (none) is specified, it indicates that an application can call none of the routines listed above. From within those states that map to a (none) value in the Callable Routines column, an application's only options are to either send (**ct_send**) or cancel (**ct_cancel**) the initiated command.

Initiated Command:	Callable Routines:
Idle	**ct_cmd_drop** **ct_command**(CS_LANG_CMD) **ct_command**(CS_MSG_CMD) **ct_command**(CS_PACKAGE_CMD) **ct_command**(CS_RPC_CMD) **ct_command**(CS_SEND_BULK_CMD) **ct_command**(CS_SEND_DATA_CMD) **ct_cursor**(CS_CURSOR_DECLARE) **ct_dynamic**(CS_CURSOR_DECLARE) **ct_dynamic**(CS_DEALLOC) **ct_dynamic**(CS_DESCRIBE_INPUT) **ct_dynamic**(CS_DESCRIBE_OUTPUT) **ct_dynamic**(CS_EXECUTE) **ct_dynamic**(CS_EXEC_IMMEDIATE) **ct_dynamic**(CS_PREPARE) **ct_sendpassthru**
Idle, with declared cursor	**ct_cursor**(CS_CURSOR_ROWS) **ct_cursor**(CS_CURSOR_OPEN) **ct_cursor**(CS_CURSOR_CLOSE, CS_DEALLOC) **ct_cursor**(CS_CURSOR_DEALLOC) **ct_dynamic**(CS_DEALLOC)
Idle, with opened cursor	**ct_cursor**(CS_CURSOR_CLOSE) **ct_cursor**(CS_CURSOR_CLOSE, CS_DEALLOC) **ct_dynamic**(CS_DEALLOC)
Opened cursor, no rows fetched	**ct_cursor**(CS_CURSOR_CLOSE) **ct_cursor**(CS_CURSOR_CLOSE, CS_DEALLOC)

Table 2-23: Callable routines for initiated commands

Initiated Command:	Callable Routines:
Opened cursor, fetching rows	**ct_cursor**(CS_CURSOR_CLOSE) **ct_cursor**(CS_CURSOR_CLOSE, CS_DEALLOC) **ct_cursor**(CS_CURSOR_UPDATE) **ct_cursor**(CS_CURSOR_DELETE)
Command initialized	**ct_dyndesc**(CS_USE_DESC) **ct_param**
Sent data	**ct_data_info**(CS_SET) **ct_send_data**
Declared cursor	**ct_cursor**(CS_CURSOR_ROWS) **ct_cursor**(CS_CURSOR_OPEN) **ct_cursor**(CS_CURSOR_OPTION) **ct_dyndesc**(CS_USE_DESC) **ct_param**
Set cursor rows	**ct_cursor**(CS_CURSOR_OPEN)
Opened cursor	**ct_dyndesc**(CS_USE_DESC) **ct_param**
Closed cursor	(None)
De-allocated cursor	(None)
Updated cursor row	**ct_dyndesc**(CS_USE_DESC) **ct_param**
Deleted cursor row	(None)
Dynamic declared	(None)
Dynamic de-allocated	(None)
Dynamic described	(None)
Dynamic executed	**ct_dyndesc**(CS_USE_DESC) **ct_param**
Dynamic execute immediate	(None)
Dynamic prepare	**ct_dyndesc**(CS_USE_DESC) **ct_param**
ct_send_data succeeded	**ct_send_data**
Bulk copy	**ct_send_data**

Table 2-23: Callable routines for initiated commands (continued)

Result Types

If a command is in one of the following states:

Results available
Fetchable results
Fetchable cursor results
Fetchable nested command
Sent fetchable nested command
Processing fetchable nested command
Fetching results
Fetching cursor results

Fetching nested command
Sent fetching nested command
Processing fetching nested command

Client-Library pre-defines which routines can be called, based on the result type.

The following table briefly describes the different result types:

Result Type:	Description:
Regular row results	Zero or more rows of tabular data generated by the execution of a Transact-SQL **select** statement.
Cursor row results	Zero or more rows of tabular data generated when an application executes a Client-Library cursor open command.
Parameter results	A single row of message parameters or stored procedure return parameters.
Stored procedure return status results	A single row containing a single value (a return status).
Message results	No data is available, but an application can call **ct_res_info** to get the message's id.
Compute row results	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.
CS_CMD_DONE	The results of a command have been completely processed.
CS_CMD_SUCCEED	A command that returns no data (such as a language command containing a Transact-SQL **insert** statement) was successful.
CS_CMD_FAIL	The server encountered an error while executing a command.

Table 2-24: Result types

Result Type:	Description:
Regular row format results	Format information for an associated regular row result set.
Compute row format results	Format information for an associated compute row result set.
Describe results	Descriptive information returned as the result of a dynamic SQL describe input or output command.
Extended error data results	A single row of extended error data.
Notification results	A single row of arguments with which a registered procedure was called.

Table 2-24: Result types

See the **Results** topics page for detailed information about the various types of results.

Result Type Processing Routines

The following Client-Library routines are useful for processing various types of results:

ct_bind	**ct_describe**
ct_br_column	**ct_dyndesc**
ct_br_table	**ct_getformat**
ct_compute_info	**ct_keydata**
ct_data_info	**ct_res_info**

Callable Routines for Each Result Type

When an application calls **ct_results** to find out what kind of results are available, Client-Library defines which routines are callable based on the value of **ct_results**' **result_type* parameter.

This table maps each result type to the Client-Library routines that an application can legally call to process that result type.

Result Type:	Callable Routines:
Regular row results	**ct_bind** **ct_br_column** **ct_br_table** **ct_data_info**(CS_GET) **ct_describe** **ct_getformat** **ct_res_info**(CS_BROWSE_INFO) **ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_NUMDATA) **ct_res_info**(CS_NUMORDERCOLS) **ct_res_info**(CS_ORDERBY_COLS) **ct_res_info**(CS_TRANS_STATE) **ct_dyndesc**(CS_USE_DESC)
Cursor row results	**ct_bind** **ct_describe** **ct_getformat** **ct_keydata** **ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_CMD_NUMDATA) **ct_res_info**(CS_TRANS_STATE) **ct_dyndesc**(CS_USE_DESC)
Parameter results	**ct_bind** **ct_describe** **ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_NUMDATA) **ct_res_info**(CS_TRANS_STATE) **ct_dyndesc**(CS_USE_DESC)
Stored procedure return status results	**ct_bind** **ct_describe** **ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_CMD_NUMDATA) **ct_res_info**(CS_TRANS_STATE) **ct_dyndesc**(CS_USE_DESC)
Message results	**ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_MSGTYPE) **ct_res_info**(CS_TRANS_STATE)

Table 2-25: Callable routines for each result type

Result Type:	Callable Routines:
Compute row results	**ct_bind** **ct_compute_info** **ct_describe** **ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_NUM_COMPUTES) **ct_res_info**(CS_NUMDATA) **ct_res_info**(CS_TRANS_STATE) **ct_dyndesc**(CS_USE_DESC)
CS_CMD_DONE	**ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_ROW_COUNT) **ct_res_info**(CS_TRANS_STATE)
CS_CMD_SUCCEED	**ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_ROW_COUNT) **ct_res_info**(CS_TRANS_STATE)
CS_CMD_FAIL	**ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_ROW_COUNT) **ct_res_info**(CS_TRANS_STATE)
Regular row format results	**ct_describe** **ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_CMD_NUMDATA) **ct_res_info**(CS_TRANS_STATE) **ct_dyndesc**(CS_USE_DESC)
Compute row format results	**ct_compute_info** **ct_describe** **ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_NUM_COMPUTES) **ct_res_info**(CS_NUMDATA) **ct_res_info**(CS_TRANS_STATE) **ct_dyndesc**(CS_USE_DESC)
Describe results	**ct_describe** **ct_res_info**(CS_CMD_NUMBER) **ct_res_info**(CS_NUMDATA) **ct_res_info**(CS_TRANS_STATE) **ct_dyndesc**(CS_USE_DESC)
Extended error data results	**ct_bind** **ct_describe** **ct_res_info**(CS_NUMDATA) **ct_res_info**(CS_TRANS_STATE)
Notification results	**ct_bind** **ct_describe** **ct_res_info**(CS_NUMDATA) **ct_res_info**(CS_TRANS_STATE)

Table 2-25: Callable routines for each result type (continued)

Pending Results

Multiple command structures sharing the same connection can block one another when results are pending on the connection. ‘Pending results’ is a term that indicates that the results of a command have not yet been completely processed. For example, assume two CS_COMMAND structures (A and B) sharing the same CS_CONNECTION structure. If A is in the Results Available state, B is blocked from sending a command to the server because there are results pending on the connection. B will remain blocked until A processes all the results of the current command and transitions into a state that indicates that no results are pending.

States that indicate pending results are:

Command sent
Results available
ANSI-style cursor end-data
Fetchable results
Sent fetchable nested command
Processing fetchable nested command
Fetching results
Sent fetching nested command
Undefined
In receive passthrough
In send passthrough

States that do *not* indicate pending results are:

Idle
Command initiated
Fetchable cursor results
Fetchable nested command
Fetching cursor results
Fetching nested command
Processing fetching nested command
Result set canceled

Message Commands and Results

Message commands and results provide a way for clients and servers to communicate specialized information to one another.

For example, if the CS_OPT_GETDATA option is enabled, then on every **insert**, **delete**, or **update** SQL Server returns a message with parameters that allows a client application to construct the name of the temporary table that SQL Server is using for the operation.

Message Commands

To send a message command:

1. Call **ct_command** to initiate the command.
2. Call **ct_param** once for each parameter that the message requires.
3. Call **ct_send** to send the message command.

Message Results

ct_results sets its **result_type* parameter to CS_MSG_RESULT to indicate a message result set.

A message result set contains no fetchable data. Rather, a message has an "id," which an application can retrieve by calling **ct_res_info**.

Any parameters associated with a message are returned in the form of a parameter result set following the message result set.

Legal Message Ids

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.

Open Client Macros

Macros are C language definitions that typically take one or more arguments and expand into inline C code when the source file is pre-processed. The following sections introduce you to the Open Client macros by presenting them in their functional contexts.

Decoding a Message Number

Client-Library and CS-Library message numbers consist of four components: layer, origin, severity, and number.

Open Client provides the following macros to help an application decode a Client-Library or CS-Library message number and break it into its four parts so that each component can be displayed separately:

- **CS_LAYER**(*msg_number*) - identifies the layer reporting the error.
- **CS_ORIGIN**(*msg_number*) - indicates where the error manifested itself.
- **CS_SEVERITY**(*msg_number*) - indicates the severity of the error.
- **CS_NUMBER**(*msg_number*) - identifies the actual layer-specific error number being reported.

These macros are defined in the header file *cstypes.h*.

See the **Client-Library Messages** topics page for more information about Client-Library message numbers.

For information on CS-Library error handling, see the *Open Client and Open Server Common Libraries Reference Manual*.

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.

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

- **CS_CLR_CAPMASK**(*mask, capability*) - clears bits in a CS_CAP_TYPE structure.
- **CS_SET_CAPMASK**(*mask, capability*) - sets bits in a CS_CAP_TYPE structure.

- **CS_TST_CAPMASK**(*mask, capability*) **-** tests bits in a CS_CAP_TYPE structure.

where *mask* is a pointer to a CS_CAP_TYPE structure and *capability* is the capability of interest.

These macros are defined in the header file *cspublic.h*.

See the **Capabilities** topic page for more information about capabilities.

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, using `sizeof` can be problematic for Client-Library applications. In particular, specifying `sizeof` as an argument to a Client-Library routine may result in a compiler error or warning if the type returned is not a CS_INT.

Client-Library provides the following macro to enable an application to use the `sizeof` function with Client-Library:

- **CS_SIZEOF -** casts a value to a CS_INT.

This macro is defined in the header file *cstypes.h*.

Options

A Client-Library application can set and clear SQL Server query-processing options in one of two ways:

- Through a Transact-SQL language command (**set**)
- By calling **ct_options**

An application must use only one of these methods, because otherwise Client-Library / server communications can become confused.

The **ct_options** method is recommended, because it has the advantage of allowing an application to check the status of an option, which cannot be done through the Transact-SQL **set** command.

For more information on SQL Server query-processing options, see the **set** command in the *SQL Server Reference Manual.*

Symbolic Constants for Server Options

The following table lists the symbolic constants that are used with **ct_options**:

Symbolic Constant:	What the Option Does:	Default Value:
CS_OPT_ANSINULL	Determines whether or not evaluation of NULL-valued operands in SQL equality (=) or inequality (!=) comparisons is ANSI-compliant. If CS_TRUE, SQL 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 or not SQL Server is ANSI-compliant with respect to permissions checks on **update** and **delete** statements. If CS_TRUE, SQL Server is ANSI-compliant.	CS_FALSE

Table 2-26: Symbolic constants for server options

Symbolic Constant:	What the Option Does:	Default Value:
CS_OPT_ARITHABORT	Determines how SQL Server behaves when an arithmetic error occurs. ** If CS_TRUE, both the **arith_overflow** and **numeric_truncation** options are set 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 during an implicit datatype conversion occurs, 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 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 SQL Server displays a message after a divide-by-zero error or a loss of precision. If CS_TRUE, warning message are suppressed after these errors. If CS_FALSE, warning messages are displayed after these errors.	CS_FALSE
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

Table 2-26: Symbolic constants for server options (continued)

Symbolic Constant:	What the Option Does:	Default Value:
CS_OPT_CHAINXACTS	If CS_TRUE, SQL Server uses chained transaction behavior. Chained transaction behavior means that each server command is considered to be a distinct transaction. SQL 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_CURREAD	Sets a security label specifying the current read level.	NULL
CS_OPT_CURWRITE	Sets a security label specifying the current write level.	NULL
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 *datetime* or *smalldatetime* data.	For us_english, the default is CS_OPT_FMTMDY.
CS_OPT_FIPSFLAG	Determines whether SQL Server displayss a warning message when SQL extensions are used. If CS_TRUE, SQL Server flags any non-standard SQL commands that are sent. If CS_FALSE, SQL Server does not flag non-ANSI SQL.	CS_FALSE
CS_OPT_FORCEPLAN	If CS_TRUE, SQL 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, SQL Server sends back a description of the data, rather than the data itself, in response to a **select** query. If CS_FALSE, SQL Server sends back data in response to a **select** query.	CS_FALSE

Table 2-26: Symbolic constants for server options (continued)

Symbolic Constant:	What the Option Does:	Default Value:
CS_OPT_GETDATA	If CS_TRUE, on every **insert**, **delete**, or **update** statement, SQL Server returns information (in the form of a message result set and parameters) that an application can use to construct the name of the temporary table that will contain the rows to be inserted and / or deleted. Note that an update consists of insertions and deletions.	CS_FALSE
CS_OPT_IDENTITYOFF	Disables inserts into a table's IDENTITY column. For more information, see the **set** command (**identity_insert** option) in the SQL Server documentation.	Not applicable
CS_OPT_IDENTITYON	Enables inserts into a table's IDENTITY column. For more information, see the **set** command (**identity_insert** option) in the SQL Server documentation.	Not applicable
CS_OPT_ISOLATION	Specifies a transaction isolation level. Possible levels are CS_OPT_LEVEL1 and CS_OPT_LEVEL3. If CS_OPT_LEVEL1, shared locks are placed on all accessed pages of tables specified in a **select** query's **from** clause. The locks are held for the duration of a transaction. If CS_OPT_LEVEL3, hold locks are placed on all accessed pages of tables specified in a **select** query's **from** clause. The locks are held for the duration of the transaction.	CS_OPT_LEVEL1
CS_OPT_NOCOUNT	Turns off the display of the number of rows affected by each SQL statement. An application can ordinarily obtain this information by calling **ct_res_info**.	CS_FALSE
CS_OPT_NOEXEC	If CS_TRUE, SQL Server compiles each query but does not execute it. Use this option in conjunction with CS_OPT_SHOWPLAN.	CS_FALSE

Table 2-26: Symbolic constants for server options (continued)

Symbolic Constant:	What the Option Does:	Default Value:
CS_OPT_PARSEONLY	If CS_TRUE, SQL 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, SQL Server treats all strings enclosed in double quotes (“) as identifiers.	CS_FALSE
CS_OPT_RESTREES	If CS_TRUE, SQL 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
CS_OPT_ROWCOUNT	If set to 0, all rows generated by a **select** statement are returned. If set to a value greater than 0, SQL Server returns up to the specified number of regular rows for each **select** statement. This option is always on, never off. This option does not limit the number of compute rows returned.	0.
CS_OPT_SHOWPLAN	Detemines whether a description of each query’s processing plan is returned between its compilation and execution. If CS_TRUE, SQL Server compiles a query, generates a description of its processing plan, and then executes the query.	CS_FALSE
CS_OPT_STATS_IO	Determines whether SQL 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. These statistics are returned to Client-Library in the form of informational server messages. Application programs can access them through the user-supplied server message handler.	CS_FALSE

Table 2-26: Symbolic constants for server options (continued)

Symbolic Constant:	What the Option Does:	Default Value:
CS_OPT_STATS_TIME	Determines whether SQL Server parsing, compilation, and execution time statistics are returned for each query. If CS_TRUE, statistics are returned. These statistics are returned to Client-Library in the form of informational server messages. Application programs can access them through the user-supplied server message handler.	CS_FALSE
CS_OPT_STR_RTRUNC	If CS_TRUE, SQL Server is ANSI-compliant with respect to right truncation of character data.	CS_FALSE
CS_OPT_TEXTSIZE	Specifies the value of the SQL Server global variable *@@textsize*, which limits the size of text or image values that SQL Server returns. When setting this option, supply a parameter which is the length, in bytes, of the longest text or image value that SQL Server should return. In programs that allow application users to run ad hoc queries, the user may override this option with the Transact-SQL **set textsize** command. To set a text limit that the user cannot override, use the Client-Library CS_TEXTLIMIT property instead.	32,768 bytes
CS_OPT_TRUNCIGNORE	If CS_TRUE, SQL Server ignores truncation errors. This is standard ANSI behavior. If CS_FALSE, SQL Server raises an error when conversion results in truncation.	CS_FALSE

Table 2-26: Symbolic constants for server options (continued)

Parameter Conventions

This topics page contains information on Client-Library parameter conventions.

Exceptions to these conventions are documented on the manual pages for the routines for which the exceptions occur.

NULL and Unused Parameters

This section contains information on NULL and unused parameters.

Pointer Parameters

A pointer parameter can:

- Have a non-NULL value
- Have a value of NULL
- Be unused

Pass NULL and unused pointer parameters as NULL.

If the parameter has a NULL value, the length variable associated with the parameter, if any, must be 0 or CS_UNUSED.

If the parameter is unused, the length variable associated with the parameter, if any, must be CS_UNUSED.

Client-Library uses current programming context information to determine whether to interpret the parameter as NULL or unused.

Non-Pointer Parameters

Pass non-pointer unused parameters as CS_UNUSED.

Input Parameter Strings

Most string parameters are associated with a parameter that indicates the length of the string.

When passing a null-terminated string, an application can pass the length parameter as CS_NULLTERM.

When passing a string that is not null-terminated, an application must set the associated length parameter to the length, in bytes, of the string.

If a string parameter is NULL the associated length parameter must be 0 or CS_UNUSED.

Output Parameter Strings

An application indicates the length of a string buffer by setting an associated length parameter.

If the length parameter indicates that the buffer is not large enough to hold a null-terminated output string, Client-Library routines return CS_FAIL.

Pointers to Basic Structures

All Client-Library routines take a pointer to a CS_CONTEXT structure, a CS_CONNECTION structure, or a CS_COMMAND structure as a parameter.

An application must allocate these structures (via **cs_ctx_alloc**, **ct_con_alloc**, or **ct_cmd_alloc**) before using them as parameters.

If an application passes an invalid structure pointer to a Client-Library routine, the routine returns CS_FAIL but Client-Library does not call the application‘s client message callback routine. This is because Client-Library stores the location of the client message callback in the CS_CONTEXT, CS_CONNECTION, and CS_COMMAND structures.

Item Numbers

Many Client-Library routines that process results or return information about results take an “item number” as a parameter. An item number identifies a result item in a result set, and can be a column number, a compute column number, a parameter number, or a return status number.

Item numbers start at 1 and never exceed the number of items in the current result set. An application can call **ct_res_info** with *type* as CS_NUMDATA to get the number of items in the current result set.

When the result set contains columns, *item* is a column number. Columns are returned to an application in select-list order.

When the result set contains compute columns, *item* is the column number of a compute column. Compute columns are returned in the order in which they are listed in the compute clause.

When the result set contains parameters, *item* is a parameter number. 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 the result set contains a return status, *item* is always 1, as there can be only a single status in a return status result set.

action, *buffer*, *buflen*, and *outlen*

Many Client-Library routines use some combination of the parameters *action, buffer, buflen,* and *outlen.*

action describes whether to set or retrieve information. For most routines, *action* can take the symbolic values CS_GET, CS_SET, and CS_CLEAR.

If *action* is CS_CLEAR, *buffer* must be NULL and *buflen* must be CS_UNUSED.

buffer is typically a pointer to program data space.

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

If information is being retrieved, *buffer* points to the space in which the Client-Library routine will place the requested information.

If information is being cleared, *buffer* must be NULL.

If the Client-Library routine returns CS_FAIL, **buffer* remains unchanged.

buflen is the length, in bytes, of the *buffer* data space.

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

If **buffer* is a fixed-length value, a symbolic value, or a function, *buflen* must be CS_UNUSED.

If *buffer* is NULL, *buflen* must be 0 or CS_UNUSED.

outlen is a pointer to an integer variable.

outlen must be NULL if information is being set.

When information is being retrieved, *outlen* is an optional parameter. If supplied, Client-Library sets the variable to the length, in bytes, of the requested information (including a null terminator, if applicable).

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

The following table summarizes the interaction between *action*, *buffer*, *buflen*, and *outlen*:

action:	buffer:	buflen:	outlen:	What Happens
CS_CLEAR	NULL	CS_UNUSED	NULL	The Client-Library information is cleared by resetting it to its default value.
CS_SET	A pointer to a null-terminated character string.	CS_NULLTERM or the length of the string, not including the null terminator.	NULL	The Client-Library information is set to the value of the **buffer* character string.
CS_SET	A pointer to a character string that is not null-terminated.	The length of the string.	NULL	The Client-Library information is set to the value of the **buffer* character string.
CS_SET	A pointer to a variable-length, non-character value. (For example, binary data.)	The length of the data.	NULL	The Client-Library information is set to the value of the **buffer* data.
CS_SET	A pointer to a fixed-length or symbolic value.	CS_UNUSED	NULL	The Client-Library information is set to the value of the integer or symbolic value.
CS_SET	NULL	0 or CS_UNUSED	NULL	The Client-Library information is set to NULL.
CS_GET	A pointer to space large enough for the return character string plus a null terminator.	The length of **buffer*.	Supplied or NULL	The return value is copied to **buffer*. A null-terminator is appended. If supplied, **outlen* is set to the length of the return value, including the null terminator.

Table 2-27: Interaction between action, buffer, buflen, *and* outlen *parameters*

action:	buffer:	buflen:	outlen:	What Happens
CS_GET	A pointer to space that is not large enough for the return character string plus a null terminator.	The length of **buffer*.	Supplied or NULL	No data is copied to **buffer*. If supplied, **outlen* is set to the length of the return value, including the null terminator. The routine returns CS_FAIL.
CS_GET	A pointer to space that is large enough for the return variable-length, non-character data.	The length of **buffer*.	Supplied or NULL	The return value is copied to **buffer*. If supplied, **outlen* is set to the length of the return value.
CS_GET	A pointer to space that is not large enough for the return variable-length, non-character data.	The length of **buffer*.	Supplied or NULL	No data is copied to **buffer*. If supplied, **outlen* is set to the length of the return value. The routine returns CS_FAIL.
CS_GET	A pointer to space that is assumed to be large enough for a fixed-length or symbolic value.	CS_UNUSED	Supplied or NULL	The return value is copied to **buffer*. If supplied, **outlen* is set to the length of the return value.

Table 2-27: Interaction between action, buffer, buflen, *and* outlen *parameters (continued)*

Properties

Properties define aspects of Client-Library behavior. For example, the CS_NETIO property determines whether a connection is synchronous or asynchronous, and the CS_HIDDEN_KEYS property determines whether or not hidden keys returned as part of a result set are exposed.

Login properties are used when logging into a server. Login properties include CS_USERNAME, CS_PASSWORD, and CS_PACKETSIZE.

A server can change the values of some login properties during the log-in process. For example, if an application sets CS_PACKETSIZE to 2048 bytes and then logs into a server that cannot support this packet size, the server will overwrite 2048 with a packet size it can support. These types of properties are called **negotiated properties**.

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 can override 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 can be either set or retrieved by an application, but some properties are "retrieve only."

Three Kinds of Context Properties

There are actually 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 via **cs_config** affect only CS-Library. CS-Library-specific context properties are listed on the manual page for **cs_config** in the *Common Libraries Reference Manual*.

ct_config sets and retrieves the values of Client-Library-specific context properties. Properties set via **ct_config** affect only Client-Library. Client-Library-specific context properties are listed in *Table 2-28: Client-Library properties*.

srv_props sets and retrieves the values of Server-Library-specific context properties. Properties set via **srv_props** affect only Server-Library.

Copying Login Properties

An application can copy 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** de-allocates the CS_LOGINFO structure.
7. Customizes any non-login properties in the second connection (**ct_con_props**).
8. Opens the second connection (**ct_connect**).

Summary of Properties

The following table lists Client-Library properties. The context properties in this table are set via **ct_config**. For a list of context properties set via **cs_config**, see the manual page for **cs_config** in the *Open Client and Open Server Common Libraries Reference Manual*.

Property name:	What it is:	Possible values:	Applicable at what level?	Notes
CS_ANSI_BINDS	Whether or not to use ANSI-style binds.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection.	
CS_APPNAME	The application name used when logging into the server.	A character string. The default is NULL.	Connection.	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. The default is CS_FALSE.	Connection.	
CS_BULK_LOGIN	Whether or not a connection is enabled to perform bulk copy "in" operations.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Login property. Cannot be set after connection is established.
CS_CHARSETCNV	Whether or not character set conversion is taking place.	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.	A pointer value. The default is NULL.	Connection.	Cannot be set after connection is established.

Table 2-28: Client-Library properties

Property name:	What it is:	Possible values:	Applicable at what level?	Notes
CS_CON_STATUS	The connection's status.	A CS_INT-sized bit-mask. For a list of possible values, see "Connection Status" on page 2-138.	Connection.	Retrieve only.
CS_CUR_ID	The cursor's identification number.	An integer value. A default is not applicable.	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_CURSOR _DECLARE) call.	A character string. A default is not applicable.	Command.	Retrieve only, after **ct_cursor** (CS_CURSOR_ DECLARE) returns CS_SUCCEED.
CS_CUR_ROW COUNT	The current value of cursor rows. Cursor rows is the number of rows returned to Client-Library per internal fetch request.	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.	A CS_INT-sized bit-mask. For a list of possible values, see "Cursor Status" on page 2-140.	Command.	Retrieve only.
CS_DIAG_ TIMEOUT	When in-line error handling is in effect, whether Client-Library should fail or retry on timeout errors.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means Client-Library should retry.	Connection.	

Table 2-28: Client-Library properties (continued)

Property name:	What it is:	Possible values:	Applicable at what level?	Notes
CS_DISABLE_ POLL	Whether or not to disable polling. If polling is disabled, **ct_poll** does not report asynchronous operation completions.	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_EED_CMD	A pointer to a command structure containing extended error data.	A pointer value. A default is not applicable.	Connection.	Retrieve only.
CS_ENDPOINT	The file descriptor for a connection.	An integer value. A default is not applicable.	Connection.	Retrieve only, after connection is established.
CS_EXPOSE_FMTS	Whether or not to expose results of type CS_ROWFMT_ RESULT and CS_COM PUTEFMT_RESULT.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	Cannot be set after connection is established.
CS_EXTRA_INF	Whether or not to return the extra information that's required when processing Client-Library messages in-line using a SQLCA, SQLCODE, or SQLSTATE.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Context, connection	
CS_HIDDEN_KEYS	Whether or not to expose hidden keys.	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.	A character string. The default is NULL.	Connection.	Login property. Cannot be set after connection is established.

Table 2-28: Client-Library properties (continued)

Property name:	What it is:	Possible values:	Applicable at what level?	Notes
CS_IFILE	The path and name of the interfaces file.	A character string. The default varies by platform. On UNIX platforms, the default is *$SYBASE/interfaces*.	Context.	
CS_LOC_PROP	A CS_LOCALE structure that defines localization information.	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 or not the connection is open.	CS_TRUE or CS_FALSE. A default is not applicable.	Connection.	Retrieve only.
CS_LOGIN_TIMEOUT	The login timeout value.	An integer value. The default is 60 seconds. A value of CS_NO_LIMIT represents an infinite timeout period.	Context.	
CS_MAX_CONNECT	The maximum number of connections for this context.	An integer value. The default varies by platform. On UNIX platforms, the default is 25.	Context.	
CS_MEM_POOL	A memory pool that Client-Library will use to satisfy interrupt-level memory requirements.	A pointer value.	Context.	Useful in asynchronous applications. Cannot be set or cleared when context has connections.

Table 2-28: Client-Library properties (continued)

Property name:	What it is:	Possible values:	Applicable at what level?	Notes
CS_NETIO	Whether network I/O is synchronous, fully asynchronous, or deferred asynchronous.	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_NO_TRUNCATE	Whether Client-Library should truncate or sequence messages that are longer than CS_MAX_MSG.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means that Client-Library truncates long messages.	Context.	
CS_NOINTERRUPT	Whether or not the application can be interrupted.	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.	An integer value. The default varies by platform. On most platforms, the default is 512 bytes.	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.	A pointer value.	Connection, command.	Retrieve only.
CS_PASSWORD	The password used to log into the server.	A character string. The default is NULL.	Connection.	Login property.

Table 2-28: Client-Library properties (continued)

Property name:	What it is:	Possible values:	Applicable at what level?	Notes
CS_SEC_ APPDEFINED	Whether or not the connection will use application-defined challenge / response security handshaking.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_ CHALLENGE	Whether or not the connection will use Sybase-defined challenge / response security handshaking.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_ ENCRYPTION	Whether or not the connection will use encrypted password security handshaking.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_ NEGOTIATE	Whether or not the connection will use trusted-user security handshaking.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SERVERNAME	The name of the server to which a connection is connected.	A string value. A default is not applicable.	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. CS_TDS_VERSION 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.	An integer value. The default is CS_NO_LIMIT.	Context, connection.	
CS_TIMEOUT	The timeout value.	An integer value. The default is CS_NO_LIMIT.	Context.	
CS_TRANS ACTION_NAME	A transaction name.	A string value. The default is NULL.	Connection.	

Table 2-28: Client-Library properties (continued)

Property name:	What it is:	Possible values:	Applicable at what level?	Notes
CS_USER_ALLOC	A user-defined memory allocation routine.	A user-defined function. A default is not applicable.	Context.	Useful in asynchronous application.
CS_USER_FREE	A user-defined memory free routine.	A user-defined function. A default is not applicable.	Context.	Useful in asynchronous application.
CS_USERDATA	User-allocated data.	User-allocated data.	Connection, command. To set CS_USER DATA at the context level, call **cs_config**.	
CS_USERNAME	The name used to log into the server.	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.	A character string. A default is not applicable.	Context.	Retrieve only.
CS_VERSION	The version of Client-Library in use by this context.	A symbolic version level. CS_VERSION gets its value from a context's **ct_init** call. Currently, the only possible value is CS_VERSION_100.	Context.	Retrieve only.

Table 2-28: Client-Library properties (continued)

About the Properties

ANSI-Style Binds

- CS_ANSI_BINDS defines whether or not Client-Library will use

ANSI-style binds.

- When ANSI-style binds are in effect:
  - It is considered an error to bind some, but not all, items in a result set. An application must either bind none of the items or bind all of the items.
  - **ct_fetch** raises an error when copying a NULL or truncated character string value into a variable with which no indicator is associated.

  In both of these cases, **ct_fetch** returns CS_ROW_FAIL.

Application Name

- CS_APPNAME defines the application name that a connection will use when connecting to a server.
- SQL Server uses application names to identify connection processes in the *sysprocesses* table of the *master* database.

Asynchronous Notifications

- CS_ASYNC_NOTIFS determines whether a connection will receive registered procedure notifications asynchronously.
- If CS_ASYNC_NOTIFS is set to CS_TRUE, then Client-Library interrupts the application to report an arriving registered procedure notification. When Client-Library reports the notification, the application's notification callback is automatically triggered.
- If CS_ASYNC_NOTIFS is CS_FALSE, then the application must be reading from the network in order for Client-Library to report a registered procedure notification. When Client-Library reports the notification, the application's notification callback is automatically called.

  Likewise, if CS_ASYNC_NOTIFS is CS_FALSE, **ct_poll** will not read from the network. This means that an application must be reading results in order for **ct_poll** to report a registered procedure notification. When **ct_poll** reports the notification, the application's notification callback is automatically called.
- Setting CS_ASYNC_NOTIFS to CS_FALSE does not immediately turn asynchronous notifications off. In order to truly turn asynchronous notifications off, 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 can receive asynchronous notifications.
  - An asynchronous connection will not receive notifications asynchronously unless it sets CS_ASYNC_NOTIFS to CS_TRUE.
- For information on registered procedures, see the **Registered Procedures** topics page, 2-157.

Bulk Copy Operations

- CS_BULK_LOGIN describes whether or not a connection can perform bulk copy operations into a database.
- 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 via SQL commands. Once a bulk copy sequence is begun, it cannot be stopped with an ordinary SQL command.
- For information on Bulk Copy, see the *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 IBM-370 systems and is ignored by all other platforms.

Connection Status

- CS_CON_STATUS is a CS_INT-sized bit-mask that reflects a connection’s current status.

- The following table lists the symbolic values that can make up CS_CON_STATUS:

Symbolic Value:	To Indicate:
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. 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 and drop the connection.

Table 2-29: Bit values for the CS_CON_STATUS property

Cursor ID

- CS_CUR_ID is the server identification number assigned to a cursor.
- An application can retrieve 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.

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 can retrieve 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.

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 per internal fetch request. Note that this is not the number of rows returned to an application per **ct_fetch** call. For more information on cursor rows, see "Dynamic SQL Cursor Option" on page 3-95.
- An application can retrieve 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 bit-mask that reflects a cursor's current status.
- The following table lists the symbolic values that can make up CS_CUR_STATUS:

Symbolic Value:	**To Indicate:**
CS_CURSTAT_CLOSED	A closed cursor exists in the command space. An application can re-open a closed cursor.
CS_CURSTAT_DECLARED	A cursor is currently declared in this command space.
CS_CURSTAT_NONE	No cursor is declared in this command space.
CS_CURSTAT_OPEN	An open cursor exists in the command space.
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.

Table 2-30: Bit values for the CS_CUR_STATUS property

- 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** can cause a connection's cursors to enter an undefined state. An application can use 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.

Diagnostic Timeout Fail

- When in-line 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 can 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 on CS_DISABLE_POLL, see "Layered Applications" on page 2-5.

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.
- For more information on extended error data, see "Extended Error Data" on page 2-79.

Endpoint Polling

- CS_ENDPOINT allows an application to get a file descriptor, the number associated with a connection to a remote server. This can 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 can be 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. This type of format result set contains format information for a regular row result set.
  - CS_COMPUTEFMT_RESULT. This type of format result set 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 can only retrieve format information while it is processing a result set. For example, after **ct_results** returns CS_ROW_RESULT the application can call **ct_res_info** to determine the number of columns in the result set, **ct_describe** to get a description of each column, etc.

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 SQL Server results before sending them on to a foreign client.
- An application can expose format result sets by setting the CS_EXPOSE_FMTS property to CS_TRUE.
- For more information on format results, see "Format Results" on page 2-166.

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.
- If an application is not retrieving messages into a SQLCA, SQLCODE, or SQLSTATE, the extra information is returned as ordinary Client-Library messages.

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 can expose 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 can bind and fetch the row values of exposed columns.
- If a column is an exposed hidden key, **ct_describe** includes CS_HIDDEN in the *status* field bit mask describing the column.

- An application can use **ct_keydata** with a table's keys to change a cursor's position. For information on how to do this, see the **ct_keydata** manual page.
- 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 in to a server.
- SQL Server lists a process' host name in the *sysprocesses* table of the *master* database.

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 can call **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 can de-allocate 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.
- A connection picks up default localization information from its parent context.
- An application can call **cs_loc_alloc** to allocate a CS_LOCALE structure.
- An application can call **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 on how default localization values are assigned on your platform, see the *Open Client/Server Supplement*.

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 via **ct_config**, **ct_connect** attempts to use a file named *interfaces* in the directory named by the SYBASE environment variable or logical name. If SYBASE has not been set, **ct_connect** attempts to use a file named *interfaces* in the home directory of the user named “sybase”
- For more information on the interfaces file, see the *SYBASE Installation Guide*.

➤ ***Note***

Not all platforms use an interfaces file. If you do not know whether your platform uses an interfaces file, consult your SYBASE System Administrator or see the *SYBASE SQL Server Installation Guide* for your platform.

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.

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**.
- The default timeout value is 60 seconds. A timeout value of CS_NO_LIMIT represents an infinite timeout period.

  Note that a timeout value of CS_NO_LIMIT does not apply to asynchronous connections. **ct_connect** calls on asynchronous connections return immediately.

Maximum Number of Connections

- CS_MAX_CONNECT defines the maximum number of simultaneously open connections that a context can 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 which 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 can use to satisfy its memory requirements.
- Ordinarily, Client-Library routines satisfy their memory requirements by calling **malloc**. However, because not all implementations of **malloc** are re-entrant, it is not safe for Client-Library routines that are called at the interrupt level to use **malloc**. For this reason, 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 can satisfy Client-Library's memory requirements:

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

  If an asynchronous application fails to provide Client-Library with a safe way to satisfy memory requirements, 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 bytes 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 can clear 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 calls the connection's completion callback automatically, at the interrupt level.
  - On a deferred asynchronous connection, a routine that requires a server response returns CS_PENDING immediately. The connection must call **ct_poll** in order to find out if the routine has completed.
- 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 all asynchronous connections within a context must be either fully asynchronous or 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 on asynchronous Client-Library programming, see the **Asynchronous Programming** topics page.

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 on sequenced messages, see "Sequencing Long Messages" on page 2-77.

No Interrupt

- CS_NOINTERRUPT determines whether an application can be interrupted by Client-Library.
- Examples of interrupt situations include:
  - A Client-Library routine running on an asynchronous connection completes, generating an interrupt.
  - A registered procedure notification arrives for an application, generating an interrupt.
- When CS_NOINTERRUPT is CS_TRUE, interruptions are deferred until CS_NOINTERRUPT is reset to CS_FALSE.
- An application can use the CS_NOINTERRUPT property to protect critical sections of code.

Notification Parameters

- The CS_NOTIF_CMD property defines a pointer to a CS_COMMAND structure containing registered procedure notification parameters.
- For more information on registered procedures, see the **Registered Procedures** topics page.

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 text, image, or bulk data, a larger packet size can improve efficiency.

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.

Password

- CS_PASSWORD defines the password that a connection uses when logging in to a server.

Security Application-Defined

- CS_SEC_APPDEFINED determines whether or not a connection will use Open Server application-defined challenge / response security handshaking.
- If a connection is using challenge / response security handshaking, then Client-Library calls the connection's negotiation callback routine when it receives a server challenge.
- For more information on server challenges, see "Challenge / Response Security Handshakes" on page 2-176.

Security Challenge

- CS_SEC_CHALLENGE determines whether or not a connection will use Sybase-defined challenge / response security handshaking.
- If a connection is using challenge / response security handshaking, then Client-Library calls the connection's negotiation callback routine when it receives a server challenge.
- For more information on server challenges, see "Challenge / Response Security Handshakes" on page 2-176.

Security Encryption

- CS_SEC_ENCRYPTION determines whether or not a connection will use encrypted password security handshaking.
- If a connection is using encrypted password security handshaking, then Client-Library calls the connection's encryption callback routine when it receives a request for an encrypted password.
- Typical applications do not need to set CS_SEC_ENCRYPTION, which is provided for gateway support.
- For more information on encrypted passwords, see "Encrypted Password Security Handshakes" on page 2-177.

Security Negotiation

- CS_SEC_NEGOTIATE determines whether or not a connection will use trusted-user security handshaking.
- Trusted-user security handshaking requires a client application to provide identifying security labels to pass to the server during the connection process.
- 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.

• For more information on login security labels, see "Trusted-User Security Handshakes" on page 2-175.

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 can change during the login process. An application can set 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 can retrieve the value of CS_TDS_VERSION after a connection is established to determine the actual version of TDS in use.

• The following table lists the symbolic values that CS_TDS_VERSION can have:

Symbolic Value:	To Indicate:	Features Supported:
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, multi-byte character sets.
CS_TDS_50	5.0 TDS	Cursors.

Table 2-31: Values for CS_TDS_VERSION

• If not otherwise set, CS_TDS_VERSION defaults to a value based on the CS_VERSION level that an application requested via **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_100. If an application calls **ct_init** with *version* as CS_VERSION_100 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 can take some time for an entire text value to be returned over the network. To keep a SQL 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 controls the length of time, in seconds, that Client-Library waits for a server response when making a request.
- The default timeout value is CS_NO_LIMIT, which represents an infinite timeout period. Negative and zero values are not allowed for CS_TIMEOUT.
- **ct_config** can be called to set the timeout value at any time during an application — before or after a call to **ct_connect** creates an open connection. It takes effect for all open connections immediately upon being called.

Transaction Name

- CS_TRANSACTION_NAME defines a transaction name.

- SYBASE Open Server for CICS uses transaction names to identify executables running under CICS. For more information on SYBASE Open Server for CICS, see the Open Server for CICS documentation.
- 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 will use for memory management.
- 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, because not all implementations of **malloc** are re-entrant, it is not safe for Client-Library routines that are called at the interrupt level to use **malloc**. For this reason, 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 can satisfy Client-Library's memory requirements:

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

  If an 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 can 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 can clear a memory routine 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.

User Free Function

- CS_USER_FREE identifies a user-supplied memory free routine that Client-Library will use for 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 free routine must be defined as:

```
void user_free(ptr)
void        *ptr;
```

- For more information, see "User Allocation Function" on page 2-153.

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.
- CS_USERDATA is useful when a callback routine and the main-line application need to share information without using global variables.
- When an application uses CS_USERDATA to store data, Client-Library copies the actual data, not a pointer to the data, into internal data space.
- To associate user data with a context structure, an application can call **cs_config**.
- 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 into a server.

Version String for Client-Library

- CS_VER_STRING defines a character string that represents the true version of Client-Library that an application is using. This property can only be retrieved.
- CS_VER_STRING and CS_VERSION can indicate different version levels because higher-level versions of Client-Library can emulate the behavior of lower-level versions.

  CS_VER_STRING represents the actual version of Client-Library that is in use. CS_VERSION represents the version of Client-Library behavior than an application has requested.

Version of Client-Library

- The CS_VERSION property represents the version of Client-Library behavior than an application has requested via **ct_init**. The value of this property can only be retrieved.
- Currently, the only value that is legal for CS_VERSION is CS_VERSION_100.
- Connections allocated within a context use default CS_TDS_VERSION values that are based on their parent context's CS_VERSION level.
- 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. Release 2.0 is the first Open Server release to support registered procedures. At this time, registered procedures are not supported by SQL 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 can "watch" 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 which puts stock up for sale, has requested to 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 can result 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 can 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 in order to:

- Create a registered procedure in Open Server.

  (Note that a Client--Library application can only create 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.
- Execute a registered procedure.

For more information on Open Server system registered procedures, see the *Open Server Server-Library Reference Manual*.

An application calls Client-Library routines in order to:

- Install a user-supplied 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.

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

For more information on callback routines, see the **Callbacks** topics page.

Receiving Notifications Asynchronously

The CS_ASYNC_NOTIFS property determines whether a connection receives notifications asynchronously:

- An otherwise synchronous connection can receive asynchronous notifications by setting CS_ASYNC_NOTIFS to CS_TRUE.
- An asynchronous connection will not receive notifications asynchronously unless it sets CS_ASYNC_NOTIFS to CS_TRUE.

CS_ASYNC_NOTIFS defaults to CS_FALSE, which means that the application must be reading from the network in order to receive a registered procedure notification.

Remote Procedure Calls

A Client-Library application can call a SQL Server stored procedure in two ways: by executing a Transact-SQL language command (“execute myproc”) or by executing an RPC (remote procedure call) command.

A Client-Library application can call an Open Server registered procedure by executing an RPC command.

Comparing RPCs and Execute Statements

Remote procedure calls have a few advantages over **execute** statements:

- An RPC command can be used to execute a SQL Server stored procedure or an Open Server registered procedure.

  A Transact-SQL language command can be used only to execute a SQL Server stored procedure (unless the Open Server application understands Transact-SQL).
- An RPC command passes the stored procedure’s parameters in their native datatypes, in contrast to the **execute** statement, which passes parameters as ASCII characters. This difference means that the RPC method is faster and more efficient than the **execute** method, because it does not require either the application program or the server to convert between native datatypes and their ASCII equivalents.
- It is simpler and faster to accommodate stored procedure return parameters if the procedure is invoked with an RPC command instead of a language command.

  With an RPC command, the return parameter values automatically become available to the application as a parameter result set. (Note, however, that a return parameter must be specified as such when it is originally added to the RPC command stream with **ct_param**.)

  With an **execute** statement, on the other hand, the return parameter values are available only if the language command declares local variables and passes these variables (not constants) for the return parameters. Because the language command contains more than one SQL statement, this technique involves additional parsing each time the language command is executed. Further, the language command must explicitly select the local variables after the RPC is executed. Their values are then returned to the application as a regular row result set.

Servers Can Execute Remote Procedures

A server can execute a procedure residing on another server. For example, this might occur when a stored procedure being executed on one server contains an **execute** statement for a stored procedure on another server. The **execute** command causes the first server to log into the second server and execute the remote procedure. This is called a "server-to-server remote procedure call," and happens without any intervention from the application, although the application can specify the remote password which the first server uses to log in to the second.

A server-to-server remote procedure call also occurs when an application sends a request to execute a procedure that does not reside on the server to which it is directly connected. For example, if an application is connected to *server1*, the following language command results in a server-to-server remote procedure call:

```
ct_command(cmd, CS_LANG_CMD,
    "execute server2...procedure1",
    CS_NULLTERM, CS_UNUSED);
```

Transact-SQL commands contained in a stored procedure that is executed as the result of a server-to-server remote procedure call cannot be rolled back.

Remote Procedure Call Routines

The following Client-Library routines are related to remote procedure calls:

- **ct_remote_pwd** sets and clears the passwords that are used when logging into a remote server.
- **ct_command** initiates an RPC command.
- **ct_param** defines parameters for an RPC command.
- **ct_send** sends an RPC command.
- **ct_results**, **ct_bind**, and **ct_fetch** are used to process remote procedure results.

Remote Procedure Call Results

In addition to results generated by the Transact-SQL statements they contain, SQL Server stored procedures that are executed via an RPC command can generate return parameter and return status results.

Open Server procedures can generate row, cursor, return parameter and return status results.

All of these types of results can be processed using **ct_results**, **ct_bind**, and **ct_fetch**.

Return Parameters

SQL Server and Open Server procedures can return values for specified "return parameters." Changes made to the value of a return parameter inside the stored procedure are then available to the program that called the procedure. This is analogous to the "pass by reference" facility available in some programming languages.

In order for a parameter to function as a return parameter, it must be declared as such within the stored procedure. The **execute** statement or RPC command that invokes the stored procedure must also indicate that the parameter is a return parameter. In the case of an RPC command, it is the **ct_param** routine that specifies whether a parameter is a return parameter.

Processing Return Parameters

As mentioned in the preceding section, "Comparing RPCs and Execute Statements," return parameter values are available to an application as a parameter result set only if the application invoked the stored procedure using an RPC command.

ct_results sets its **result_type* parameter to CS_PARAM_RESULT if a parameter result set is available to be processed.

An application processes a CS_PARAM_RESULT result set in the same way as it would processes a regular row result set; that is, by binding result items and fetching rows of data. Because stored procedure parameters are returned to an application as a single row, one call to **ct_fetch** will copy all of a procedure's return parameters into the program variables designated via **ct_bind**. However, an application must still call **ct_fetch** in a loop until it returns CS_END_DATA.

Return Status

Stored procedures can return a status number.

All stored procedures that run on a SQL Server version 4.0 or greater return a status number. Stored procedures usually return 0 to indicate normal completion. For a list of SQL Server default return status numbers, see **return** in the *SQL Server Reference Manual*.

Because return status numbers are a feature of stored procedures, only an RPC command or a language command containing an **execute** statement can generate a return status.

Processing an RPC Command Return Status

ct_results sets its **result_type* parameter to CS_STATUS_RESULT if a return status result set is available to be processed.

Because a return status result set contains only a single value, one call to **ct_fetch** will copy the status into the program variable designated via **ct_bind**. However, an application should always call **ct_fetch** in a loop until it returns CS_END_DATA.

Results

When a Client-Library command executes on a server, it can generate various types of results which are returned to the application that sent the command:

They are:

- 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 can 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 Open Client command defined via **ct_command**, **ct_dynamic**, or **ct_cursor**. Exceptions to this rule are documented in ''When are the Results of a Command Completely Processed?'' on page 3-205.

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.

Types of 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 **open** statement. For more information, see "Language Cursors" on page 2-59.

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 can use **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 can be returned as a parameter result set, including:

- Message parameters. For more information, see the **Message Commands and Results** topics page, 2-114.
- Stored procedure return parameters. For more information, see the **Remote Procedure Calls** topics page, 2-160.

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 on extended error data, see the **Error and Message Handling** topics page, 2-74. For information on registered procedure notification parameters, see the **Registered Procedures** topics page, 2-157.

Stored Procedure Return Status Results

A status result set consists of a single row which contains a single value, a return status. For more information on a stored procedure return status, see the **Remote Procedure Calls** topics page.

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 on compute rows, see **compute clause** in the *SQL Server 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 can call **ct_res_info** after **ct_results** returns 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.

For more information on message results, see the **Message Commands and Results** topics page, 2-114.

Describe Results

A describe result set does not contain fetchable data, but rather indicates the existence of descriptive information returned as the result of a dynamic SQL describe input or describe output command.

An application can retrieve this descriptive information by calling **ct_describe** or **ct_dyndesc**.

For more information on dynamic SQL, see the **Dynamic SQL** topics page, 2-63.

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 SQL 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 on via 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 pseudo-code fragment 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
            if CS_SUCCEED
                process the row
            else if CS_ROW_FAIL
                handle the row failure
            end if
            /* For update or delete only: */
            if target row is not the row just fetched
                ct_keydata to specify the target row
                    key
            end if
            /* End for update or delete only */

            /* To send another cursor command: */
            ct_cursor to initiate the cursor command
            ct_param if command is update of some
                columns only
            ct_send to send the command
            while ct_results returns CS_SUCCEED
                (...process results...)
            end while
            /* End to send another cursor command */
        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 final return code
            case CS_END_DATA...
            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 or ct_dyndesc 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 three ways for an application to retrieve a result item's value:

- It can call **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 copied into the associated program variable. Most applications will use this method for all result items except large text or image values.
- It can call **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 application will use **ct_get_data** only to retrieve large text or image values.
- It can call **ct_dyndesc** to retrieve result item descriptions and values. An application calls **ct_dyndesc** once for each result item, after calling **ct_fetch** to fetch the row. Typical applications will not use **ct_dyndesc**, which is intended for precompiler support.

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.

Sample Program	Description
blktxt.c	Uses the bulk copy routines to copy static data to a table.
compute.c	Shows how to process compute results.
csr_disp.c	Demonstrates the use of a read-only cursor.
ex_alib.c *ex_amain.c*	A collection of routines which form an example of how to write an asynchronous layer on top of Client-Library.
example.h	A header file for the Client-Library example programs.
exasync.h	A header file for the constants and data structures in *ex_alib.c* and *ex_amain.c*.
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.
exutils.h	A header file for the utility functions in *exutils.c*.
getsend.c	Shows how to retrieve and update text data.
i18n.c	Demonstrates some of the international features available in Client-Library.
rpc.c	Illustrates sending an RPC command to a server and then processing the row, parameter, and status results returned from the remote procedure.

Table 2-32: Client-Library sample programs and associated header files

Before running a sample program:

- Set the $SYBASE environment variable, or the SYBASE logical for the VMS platform, to indicate the *sybase* directory
- Set the $SYBPLATFORM environment variable to indicate the platform on which the example will run (for example, sun4, sun_svr4, hp800, rs6000, ncr, axposf)
- Set the DSQUERY environment variable to indicate the server to which the program will connect.

- Assign valid username and password values in the *example.h* header file for all but the asynchronous example programs.
- Install the *pubs2* database on the server, for those sample programs which require access to *pubs2* objects.

  Refer to the appropriate sample program's header to find out which database and table(s) the program will be accessing.

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*
blk_bind	*blktxt.c*
blk_done	*blktxt.c*
blk_drop	*blktxt.c*
blk_init	*blktxt.c*
blk_rowxfer	*blktxt.c*
blk_textxfer	*blktxt.c*
cs_config	*i18n.c*
cs_convert	*exutils.c, i18n.c, rpc.c*
cs_ctx_alloc	*ex_amain.c, exutils.c*
cs_ctx_drop	*ex_amain.c, exutils.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*
ct_bind	*compute.c, ex_alib.c, exutils.c, getsend.c, i18n.c*
ct_callback	*ex_alib.c, ex_amain.c, exutils.c*

Table 2-33: Client-Library routines in sample programs

Routine	Sample Program(s)
ct_cancel	*ex_alib.c, ex_amain.c, exutils.c, getsend.c*
ct_close	*ex_amain.c, exutils.c*
ct_cmd_alloc	*compute.c, csr_disp.c, ex_alib.c, exutils.c, getsend.c, i18n.c, rpc.c*
ct_cmd_drop	*compute.c, csr_disp.c, ex_alib.c, exutils.c, i18n.c*
ct_cmd_props	*ex_alib.c, rpc.c*
ct_command	*compute.c, ex_alib.c, exutils.c, getsend.c, i18n.c, rpc.c*
ct_compute_info	*compute.c*
ct_con_alloc	*blktxt.c, ex_amain.c, exutils.c*
ct_con_drop	*blktxt.c, ex_amain.c, exutils.c*
ct_con_props	*blktxt.c, ex_alib.c, ex_amain.c, exutils.c, rpc.c*
ct_config	*exutils.c*
ct_connect	*blktxt.c, ex_amain.c, exutils.c*
ct_cursor	*csr_disp.c*
ct_debug	*ex_alib.c, ex_amain.c, exutils.c*
ct_describe	*compute.c, ex_alib.c, exutils.c, getsend.c, i18n.c*
ct_exit	*ex_amain.c, exutils.c*
ct_fetch	*compute.c, ex_alib.c, exutils.c, getsend.c, i18n.c*
ct_get_data	*getsend.c*
ct_init	*ex_amain.c, exutils.c*
ct_param	*rpc.c*
ct_poll	*ex_amain.c*
ct_res_info	*compute.c, ex_alib.c, exutils.c, i18n.c, rpc.c*
ct_results	*compute.c, csr_disp.c, ex_alib.c, exutils.c, getsend.c, i18n.c, rpc.c*
ct_send	*compute.c, csr_disp.c, ex_alib.c, exutils.c, getsend.c, i18n.c, rpc.c*
ct_send_data	*getsend.c*
ct_wakeup	*ex_alib.c*

Table 2-33: Client-Library routines in sample programs

Security Features

Client-Library's security-related features include:

- Support for various types of security handshakes
- Support for Secure SQL Server's *sensitivity* and *sensitivity_boundary* datatypes.
- Support for bulk copies in to and out from Secure SQL Servers.

Security Handshakes

Client-Library supports three types of security handshakes:

- Trusted-user security handshakes

  In this type of handshake, the server asks the client for identifying security labels, which the client then provides.

  Secure SQL Server uses trusted-user security handshaking. On Secure SQL Server, security labels are known as "sensitivity labels."
- Challenge / response security handshakes

  In this type of handshake, the server issues a challenge to which the client must correctly respond.
- Encrypted password security handshakes

  In this type of handshake, the server provides the client with a key. The client uses the key to encrypt a password, which it then returns to the server.

Trusted-User Security Handshakes

To provide the response that a trusted-user security handshake requires, an application must:

- Call **ct_con_props** to set the CS_SEC_NEGOTIATE property to CS_TRUE.
- Define security labels to pass to the server at connection time.

  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 via **ct_labels** and the labels generated by the negotiation callback are sent to the server at the same time.

When the application calls **ct_connect** to connect to the server, the server responds with a request for security labels. In response, Client-Library generates a list of labels and sends it to the server:

- If the application has called **ct_labels** to define labels, Client-Library includes these labels in the list.
- If the application has installed a negotiation callback to generate security labels, Client-Library triggers the callback and includes the labels that it generates in the list.

  When it is called, the negotiation callback generates a single security label and returns either CS_CONTINUE, CS_SUCCEED, or CS_FAIL.

  If the callback returns CS_CONTINUE, Client-Library calls the negotiation callback again, to get an additional security label.

  If the callback returns CS_SUCCEED, Client-Library sends the list of security labels to the server.

  If the callback returns CS_FAIL, Client-Library aborts the connection process, causing **ct_connect** to return CS_FAIL.

Challenge/Response Security Handshakes

Servers can use challenge / response security handshakes to provide an additional level of login security checking.

To provide the response that this handshake method requires, an application must:

- Call **ct_con_props** to set the CS_SEC_CHALLENGE or CS_SEC_APPDEFINED property to CS_TRUE. CS_SEC_CHALLENGE turns on Sybase-defined challenge / response security handshaking; CS_SEC_APPDEFINED turns on Open Server application-defined challenge / response security handshaking.
- Write a negotiation callback that will return the required response.

- Call **ct_callback** to install the callback either at the context level or for a specific connection.

When the application calls **ct_connect** to connect to the server:

- If the server replies with a challenge, then Client-Library calls the connection's negotiation callback.
- The negotiation callback generates the response and returns either CS_CONTINUE, CS_SUCCEED, or CS_FAIL.
- If the callback 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.

Encrypted Password Security Handshakes

SQL Server uses encrypted password handshakes.

Most applications are not aware of SQL Server's password encryption because Client-Library automatically handles it.

Client-Library applications that are acting as gateways, however, need to handle password encryption explicitly, passing the server's encryption key on to the client and then returning the encrypted password back to the server.

To do this, a gateway application must:

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

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

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

  If the callback returns CS_SUCCEED, Client-Library sends the encrypted password to the server.

If the callback returns CS_FAIL, Client-Library aborts the connection process, causing **ct_connect** to return CS_FAIL.

Security Datatypes

Secure SQL Server uses sensitivity labels, of datatype *sensitivity*, to control access to data. Each row in a Secure SQL Server table has a sensitivity label.

Secure SQL Server uses boundary labels, of datatype *sensitivity_boundary*, to specify either an upper or lower bound for the sensitivity labels that a process can use or access. Secure SQL Server uses boundary labels internally, in system tables.

Client-Library supports these two datatypes by providing the type constants CS_SENSITIVITY_TYPE and CS_BOUNDARY_TYPE.

These type constants differ from other Open Client type constants in that they do not correspond to similarly-named typedefs. Instead, they correspond to CS_CHAR.

This means that although Open Client routines accept and return CS_BOUNDARY_TYPE and CS_SENSITIVITY_TYPE to describe a column or variable's datatype, any corresponding program variable must be of type CS_CHAR.

For example, if an application calls **ct_bind** with the *datatype* field of the CS_DATAFMT structure set to CS_SENSITIVITY_TYPE, the program variable to which the data is being bound must be of type CS_CHAR.

Secure Bulk Copies

For information on how to bulk copy in to or out from a Secure SQL Server, see:

- Chapter 3, "Introducing Bulk Copy," in the *Open Client and Open Server Common Libraries Reference Manual*
- *Secure SQL Server Utility Programs*

Server Restrictions

SYBASE Open Client is a generic programming interface. This means that it is functionally independent of the servers to which it interfaces. Such independence allows Open Client applications to communicate with not only SYBASE SQL Server and SYBASE Open Server applications, but if the Open Server application is a gateway, with non-Sybase servers as well.

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 SQL Server restrictions. This means that communication between Open Client applications and Open Server applications is not constrained by rules that govern SQL Server's behavior.

Communication *is* constrained, however, by any restrictions built into an Open Server application. For example, an Open Server application may decide not to support remote procedure calls (RPCs) by not installing the SRV_RPC event handler. This is a constraint of which an Open Client application must be aware.

An important point to note is that Open Client and Open Server are mirror images of each other. Open Server is capable of receiving anything that Open Client is capable of sending, 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.

SQL Server Restrictions

It is only when an Open Client application accesses SQL Server that the application must be aware of SQL Server restrictions. For example, SQL Server has login name requirements: the login name must follow the rules for SQL Server identifiers and it must be unique. When an Open Client application accesses a SQL Server, it must adhere to such requirements.

What follows are some important SQL Server restrictions:

- Dynamic SQL is implemented using temporary stored procedures and therefore, inherits the restrictions of stored procedures.
- The long variable-length binary datatypes, as well as the 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 can contain only a single statement: a **select** statement.
- Stored procedures do not support text and image parameters.
- Event notifications are not supported.
- Message commands are not supported.
- The POSIX locale method of localization is not supported.

What Client/Server Features are Supported?

To ascertain some of the client and server features supported by a particular connection, an application can call **ct_capability**. **ct_capability**'s *value* parameter returns information about whether the capability is enabled or not.

An application can find out, 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** manual page.

SQLCA Structure

A SQLCA structure can be 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 on how Client-Library maps message numbers to *sqlcode*, see “SQLCODE Structure”on page 2-183.

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.
- 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 can be 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 be mapped to a SQLCODE of 0 as well.

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 command:

```
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 be mapped to a SQLCODE of 0 as well.

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

For a list of Client-Library messages, see the **Client-Library Messages** topics page.

SQLSTATE Structure

A SQLSTATE structure can be 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 bytes.

Client-Library always sets SQLSTATE and the *sqlstate* field of the CS_CLIENTMSG and CS_SERVERMSG structure identically.

Structures

Client-Library structures fall into two categories: "hidden" structures, whose internals are not documented, and "exposed" structures, whose internals are documented.

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 de-allocate 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.

The following table lists the routines and macros that allocate, manipulate, and de-allocate hidden structures:

Structure:	**Routines:**	**For More Information, See:**
CS_BLKDESC	**blk_alloc, blk_drop**	The *Open Client and Open Server Common Libraries Reference Manual.*

Table 2-34: Routines that manipulate hidden structures

Structure:	Routines:	For More Information, See:
CS_CAP_TYPE	**CS_CLR_CAPMASK**, **CS_SET_CAPMASK**, **CS_TST_CAPMASK**	"Setting and Retrieving Multiple Capabilities" on page 2-34
CS_COMMAND	**ct_cmd_alloc**, **ct_cmd_props**, **ct_cmd_drop**	"Basic Control Structures" on page 1-7
CS_CONNECTION	**ct_con_alloc**, **ct_con_props**, **ct_con_drop**	"Basic Control Structures" on page 1-7
CS_CONTEXT	**cs_ctx_alloc**, **ct_config**, **cs_config**, **cs_ctx_drop**	"Basic Control Structures" on page 1-7
CS_LOCALE	**cs_loc_alloc**, **cs_locale**, **cs_loc_drop**	"International Support" on page 2-84 of this manual. The *Open Client and Open Server Common Libraries Reference Manual*.
CS_LOGINFO	**ct_getloginfo**, **ct_setloginfo**	The manual pages for **ct_getloginfo** and **ct_setloginfo** in Chapter 3 of this manual.

Table 2-34: Routines that manipulate hidden structures (continued)

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 on topics pages.

Text and Image

text and *image* are SQL Server datatypes designed to hold large text or image values. The *text* datatype will hold up to 2, 147,483,647 bytes of printable characters. The *image* datatype will hold 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 pointer to the text or image value is stored in the table. This pointer is called a “text pointer.”

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 on the CS_IODESC structure, see the **CS_IODESC** topics page.

Retrieving a Text or Image Column

An application can retrieve text or image columns in two ways:

- It can select the columns, bind the columns, and fetch rows. In other words, an application can retrieve and process text and image columns in the same way it retrieves and processes any other type of column.
- It can select the columns, use **ct_fetch** to loop through result rows, and use **ct_get_data** to retrieve data in the text and image columns. An application will typically use 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. It can still, however, use **ct_get_data** to retrieve the text in the fourth column.

To retrieve a text or image value using **ct_get_data**, an application follows these steps:

1. Execute a command that generates a result set that contains text or image columns.

   An application can use 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. Process 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 can call **ct_get_data** in a loop to retrieve the text or image data for the column.
   - The application can call **ct_data_info** to get an I/O descriptor that can be used to update the column at a later time.

   Most applications will 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 can:

   - Call **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.
   - Call **ct_data_info** 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.

- Call **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

An application can only update a value in a text or image column if it has a current I/O descriptor for the column value that it needs to update.

To retrieve the current I/O descriptor for a column value, an application must:

1. Call **ct_fetch** to fetch the row of interest.
2. Call **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. Call **ct_data_info** to retrieve the I/O descriptor.

Once it has the current I/O descriptor for a column value, the application can perform the update:

1. Call **ct_command** to initiate a send-data command.
2. Modify the I/O descriptor, if necessary. Most applications will change only the values of the *locale*, *total_txtlen*, or *log_on_update* fields.
3. Call **ct_data_info** 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. Call **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. Call **ct_send** to send the command.
6. Call **ct_results** to process the results of the command. An update of a text or image value 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 will 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
            endwhile
            ct_send(cmd2) to send the send-data command
            ct_results(cmd2) to process the send-data results
        endfor
    endwhile
endwhile
```

Figure 2-2: Updating text or image columns

Populating a Table Containing Text or Image Columns

An application’s method of populating a table containing text or image columns will depend on the size of the data values to be inserted.

Smaller Text and Image Values

Most applications can embed text or image values of less than about 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")
```

➤ ***Note***

Be aware that MS Windows applications are limited by the amount of memory available in the data segment.

Larger Text and Image Values

Because it results in improved performance, the following method is recommended when populating a SQL 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. **select** the row. You must specifically **select** the text or image columns. This step is necessary in order to provide Client-Library with current I/O descriptor information.
4. Call **ct_results** and **ct_fetch** to process the results of the **select**. Although the actual data returned by this **select** can be thrown away, the application must retrieve the I/O descriptor for each text or image column in the row.

   For information on retrieving an I/O descriptor, see "Updating a Text or Image Column" on page 2-190.
5. Update the columns as described in "Updating a Text or Image Column" on page 2-190.

Types

Open Client Client-Library supports a wide range of datatypes. These datatypes are shared with Open Client CS-Library and Open Server Server-Library. In most cases, they correspond directly to SQL Server datatypes.

The “Datatype Summary” chart, below, lists Open Client / Server type constants, their corresponding typedefs, and their corresponding SQL Server or Secure SQL Server datatypes, if any.

A list of Open Client routines that are useful in manipulating datatypes follows the summary chart, together with more detailed information on each datatype.

For additional information on datatypes, see Chapter 3, “Structures, Datatypes, Constants, and Conventions” in the *Client-Library Programmer’s Guide*.

Datatype Summary

The following table lists Open Client / Server type constants, their corresponding typedefs, and their corresponding SQL Server or Secure SQL Server datatypes, if any:

	Open Client/Server Type Constant	**Description**	**Corresponding Open Client/Server Typedef**	**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	*boolean*
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

Table 2-35: Datatype summary

	Open Client/Server Type Constant	Description	Corresponding Open Client/Server Typedef	Corresponding Server Datatype
Datetime types	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 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_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*
Security types	CS_BOUNDARY_TYPE	Secure SQL Server boundary type	CS_CHAR	*sensitivity_ boundary*
	CS_SENSITIVITY_TYPE	Secure SQL Server sensitivity type	CS_CHAR	*sensitivity*
Text and image types	CS_TEXT_TYPE	Text type	CS_TEXT	*text*
	CS_IMAGE_TYPE	Image type	CS_IMAGE	*image*

Table 2-35: Datatype summary (continued)

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

Binary Types

Open Client has three binary types, CS_BINARY, CS_LONGBINARY, and CS_VARBINARY.

CS_BINARY corresponds to the SQL 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 SQL Server datatypes. Specifically, CS_VARBINARY does not correspond to the SQL Server datatype varbinary.

CS_LONGBINARY does not correspond to any SQL Server type, but some Open Server applications may support CS_LONGBINARY. An application can use the CS_DATA_LBIN capability to determine whether an Open Server connection supports CS_LONGBINARY. If it does, then **ct_describe** can return 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 SQL 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 three character types, CS_CHAR, CS_LONGCHAR, and CS_VARCHAR:

CS_CHAR corresponds to the SQL 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 SQL Server datatypes. Specifically, CS_VARCHAR does not correspond to the SQL Server datatype varchar.

CS_LONGCHAR does not correspond to any SQL Server type, but some Open Server applications may support CS_LONGCHAR. An application can use the CS_DATA_LCHAR capability to determine whether an Open Server connection supports CS_LONGCHAR. If it does, then **ct_describe** can return 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 SQL 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 a null-terminated string.

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.

Datetime Types

Open Client supports two datetime types, CS_DATETIME and CS_DATETIME4. These datatypes are intended to hold 8-byte and 4-byte datetime values, respectively.

An Open Client application can use the CS-Library Routine **cs_dt_crack** to extract date parts (year, month, day, etc.) from a datetime structure.

CS_DATETIME corresponds to the SQL 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 milliseconds):

```
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 SQL 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:

```
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.

Numeric Types

Open Client supports a wide range of numeric types.

Integer types include **CS_TINYINT**, a 1-byte integer; **CS_SMALLINT**, a 2-byte integer, and **CS_INT**, a 4-byte integer:

```
typedef unsigned char    CS_TINYINT;

typedef short            CS_SMALLINT;

typedef long             CS_INT;
```

CS_REAL corresponds to the SQL Server datatype *real*. It is implemented as a C-language *float* type:

```
typedef float    CS_REAL;
```

CS_FLOAT corresponds to the SQL Server datatype *float*. It is implemented as a C-language *double* type:

```
typedef double   CS_FLOAT;
```

CS_NUMERIC and **CS_DECIMAL** correspond to the SQL Server datatypes *numeric* and *decimal*. These types provide platform-independent support for numbers with precision and scale.

The SQL 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 precision of the numeric value. 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.

scale is the scale of the numeric value. 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_PREC 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 SQL Server *money* datatype. The range of legal CS_MONEY values is between + / - $922,337,203,685,477.5807:

```
typedef struct _cs_money
{
    CS_INT      mnyhigh;
    CS_UINT     mnylow;
} CS_MONEY;
```

CS_MONEY4 corresponds to the SQL 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;
```

Security Types

Open Client supports Secure SQL Server's *sensitivity_boundary* and *sensitivity* types by defining the type constants CS_BOUNDARY_TYPE and CS_SENSITIVITY_TYPE.

These type constants differ from other Open Client type constants in that they do not correspond to similarly-named typedefs. Instead, they correspond to CS_CHAR.

This means that although Open Client routines accept and return CS_BOUNDARY_TYPE and CS_SENSITIVITY_TYPE to describe a column or variable's datatype, any corresponding program variable must be of type CS_CHAR.

For example, if an application calls **ct_bind** with the *datatype* field of the CS_DATAFMT structure set to CS_SENSITIVITY_TYPE, the program variable to which the data is being bound must be of type CS_CHAR.

Text and Image Types

Open Client supports a text datatype, CS_TEXT, and an image datatype, CS_IMAGE.

CS_TEXT corresponds to the 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_IMAGE corresponds to the 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 can create a user-defined datatype.

An Client-Library application creates a user-defined type by declaring it:

```
typedef char    CODE_NAME;
```

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

```
#define CODE_NAME_TYPE   CS_USERTYPE + 2;
```

Once a user-defined type has been created, an application can:

- Call **cs_set_convert** to install custom conversion routines to convert between standard Open Client types and the user-defined type
- Call **cs_setnull** to define a null substitution value for the user-defined type.

After conversion routines are installed, an application can bind 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_bind** and **cs_convert**.

➤ ***Note***

Do not confuse Open Client user-defined types with SQL Server user-defined types. Open Client user-defined types are C-language types, declared within an application. SQL Server user-defined types are database column datatypes, created using the system stored procedure ***sp_addtype***.

Routines

3 Routines

This chapter contains a manual page for each Client-Library routine.

List of Routines

ct_bind
Bind server results to program variables.

ct_br_column
Retrieve information about a column generated by a browse-mode **select**.

ct_br_table
Return information about browse mode tables.

ct_callback
Install or retrieve a Client-Library callback routine.

ct_cancel
Cancel a command or the results of a command.

ct_capability
Set or retrieve a client / server capability.

ct_close
Close a server connection.

ct_cmd_alloc
Allocate a CS_COMMAND structure.

ct_cmd_drop
De-allocate a CS_COMMAND structure.

ct_cmd_props
Set or retrieve command structure properties.

ct_command
Initiate a language, package, RPC, message, or send-data command.

ct_compute_info
Retrieve compute result information.

ct_con_alloc
Allocate a CS_CONNECTION structure.

ct_con_drop
De-allocate a CS_CONNECTION structure.

ct_con_props
Set or retrieve connection structure properties.

ct_config
Set or retrieve context properties.

ct_connect
Connect to a server.

ct_cursor
Initiate a Client-Library cursor command.

ct_data_info
Define or retrieve a data I/O descriptor structure.

ct_debug
Manage debug library operations.

ct_describe
Return a description of result data.

ct_diag
Manage in-line error handling.

ct_dynamic
Initiate a prepared dynamic SQL statement command.

ct_dyndesc
Perform operations on a dynamic SQL descriptor area.

ct_exit
Exit Client-Library.

ct_fetch
Fetch result data.

ct_get_data
Read a chunk of data from the server.

ct_getformat
Return the server user-defined format string associated with a result column.

ct_getloginfo
Transfer TDS login response information from a CS_CONNECTION structure to a newly-allocated CS_LOGINFO structure.

ct_init
Initialize Client-Library for an application context.

ct_keydata
Specify or extract the contents of a key column.

ct_labels
Define a security label or clear security labels for a connection.

ct_options
Set, retrieve, or clear the values of server query-processing options.

ct_param
Define a command parameter.

ct_poll
Poll connections for asynchronous operation completions and registered procedure notifications.

ct_recvpassthru
Receive a TDS (Tabular Data Stream) packet from a server.

ct_remote_pwd
Define or clear passwords to be used for server-to-server connections.

ct_res_info
Retrieve current result set or command information.

ct_results
Set up result data to be processed.

ct_send
Send a command to the server.

ct_send_data
Send a chunk of text or image data to the server.

ct_sendpassthru
Send a TDS (Tabular Data Stream) packet to a server.

ct_setloginfo
Transfer TDS login response information from a CS_LOGINFO structure to a CS_CONNECTION structure.

ct_wakeup
Call a connection’s completion callback.

ct_bind

Function

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 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 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 – A pointer to the CS_DATAFMT structure that describes the destination variable(s).

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:

Field name:	When is the field used?	Set the field 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. All type constants listed on the **Types** topics page are valid. Open Client user-defined types are also valid, provided that user-supplied conversion routines have been installed via **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. For a list of the data conversions provided by Client-Library, see the manual page for **cs_willconvert**. If *datatype* is CS_BOUNDARY_TYPE or CS_SENSITIVITY_TYPE, the **buffer* program variable must be of type CS_CHAR.
format	When binding results to character- or binary-type destination variables; otherwise CS_FMT_UNUSED.	A bit-mask 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.

*Table 3-1: Fields in the CS_DATAFMT structure (***ct_bind***)*

Field name:	When is the field used?	Set the field 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	Only when binding to numeric or decimal destinations.	The scale to be used for 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	Only when binding to numeric or decimal destinations.	The precision to be used for 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.
count	When binding all types of results.	*count* is the number of result rows to be copied to program variables per **ct_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 will ever 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.
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.

*Table 3-1: Fields in the CS_DATAFMT structure (***ct_bind***) (continued)*

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.

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:

Value of indicator variable:	Indicates:
-1	The fetched data was NULL. In this case, no data is copied to **buffer*.
0	The fetch was successful.
integer value	The actual length of the server data, if the fetch resulted in truncation.

Table 3-2: Values for indicator *(***ct_bind***)*

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-3: Return values (***ct_bind***)*

Common reasons for a **ct_bind** failure include:

- An illegal datatype specified via *datafmt→datatype*.
- A bad *datafmt→locale* pointer. Initialize *datafmt→locale* to NULL if it is not used.
- Requested conversion is not available.

Comments

- **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 on how to process these types of results, see the Results topics page, 2-164.

- 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 on **ct_get_data**, see the **ct_get_data** manual page, 3-148, and the **Text and Image** topics page, 2-188.

- **ct_bind** binds only the current result type. **ct_results** indicates the current result type via 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 can call **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 re-bind 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.
- 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. This saves an application the trouble of re-binding interspersed regular row results and compute row results that are generated by the same command.

  For example, a language command containing a **select** statement with **compute** and **order by** clauses can generate multiple buffers full of regular row results intermixed with compute row results. Because they are generated by the same command, each buffer of regular row results and each buffer of compute row results will contain identical columns. An application need only bind the first buffer of regular row results and the first buffer of compute results. These bindings will remain in effect until both result sets are completely processed.
- 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 on Open Client user-defined types, see "Open Client User-Defined Datatypes" on page 2-200.

Clearing Bindings

- To clear the binding for a result item, call **ct_bind** with *buffer*, *datafmt*, *copied*, and *indicator* as NULL.
- To clear all bindings, call **ct_bind** with item as CS_UNUSED and *buffer*, *datafmt*, *copied*, and *indicator* as NULL.
- It is not an error to clear a non-existent binding.

Array Binding

- Array binding is the process of binding a result column to an array of program variables. At fetch time, multiple rows' worth of a column are copied to an array of variables with a single **ct_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.

Example

```
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 */
....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 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 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 *exutils.c* example program.For further examples of using **ct_bind**, see the *compute.c*, *ex_alib.c*, *getsend.c*, and *i18n.c* example programs.

See Also

ct_describe, ct_fetch, ct_res_info, ct_results, Types

ct_br_column

Function

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 on the CS_BROWSEDESC structure, see the **CS_BROWSEDESC** topics page.

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed. **ct_br_column** returns CS_FAIL if the current result set was not generated by a **select...for browse**.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-4: Return values (***ct_br_column***)*

Comments

- **ct_br_column** fills **browsedesc* with information about the column specified by *colnum*.
- A column can be updated through browse mode only if it meets three conditions:
  - It belongs to a browsable table.
  - It is the result of a **select...for browse.**
  - It is not the result of a SQL expression, such as **max(colname)**.
- It 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 the **Browse Mode** topics page for more information on browse mode.

See Also

Browse Mode, ct_br_table

ct_br_table

Function

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 2, and so forth.

type – The type of information to return. The following table lists the symbolic values that are legal for *type*:

Value of *type*:	ct_br_table returns:	****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.

Table 3-5: Values for type *(***ct_br_table***)*

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

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed. **ct_br_table** returns CS_FAIL if the current result set was not generated by a **select...for browse**.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-6: Return values (***ct_br_table***)*

Comments

- **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.
- For more information on browse mode, see the **Browse Mod**e topics page.

See Also

Browse Mode, ct_br_column

ct_callback

Function

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:*	**ct_callback:**
CS_SET	Installs a callback.
CS_GET	Retrieves the currently-installed callback of this type.

Table 3-7: Values for action *(***ct_callback***)*

type – The type of callback routine of interest. The following table lists the symbolic values that are legal for *type*:

Value of *type:*	**ct_callback installs:**
CS_CLIENTMSG_CB	A client message callback.
CS_COMPLETION_CB	A completion callback.
CS_ENCRYPT_CB	An encryption callback.
CS_MESSAGE_CB	A message callback.
CS_CHALLENGE_CB	A negotiation callback.
CS_SERVERMSG_CB	A server message callback.
CS_NOTIF_CB	A registered procedure notification callback.
CS_SIGNAL_CB + *signal_number*	A signal callback. 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.

Table 3-8: Values for type *(***ct_callback***)*

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.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-9: Return values (***ct_callback***)*

Comments

- 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 re-use 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 de-install 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 will automatically replace 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.
- For information on how to declare specific types of callback routines, see the **Callbacks** topics page.

Example

```
/*
** 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* example program. For further examples of using **ct_callback**, see the *ex_alib.c* and *exutils.c* example programs.

See Also

Callbacks, ct_capability, ct_config, ct_con_props, ct_connect

ct_cancel

Function

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*:

Value of *type*:	**Behavior of ct_cancel:**	**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.

Table 3-10: Values for type *(***ct_cancel***)*

Returns

Returns:	**To Indicate:**
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. For more information, see the **Asynchronous Programming** topics page.
CS_CANCELED	The cancel operation was canceled. Only a CS_CANCEL_CURRENT type of cancel can be canceled.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.
CS_TRYING	A cancel operation is already pending for this connection.

*Table 3-11: Return values (***ct_cancel***)*

Comments

- 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 discarding a buffer's worth of results. 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 immediately discard the canceled command's results (if any).

- 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 in order 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 main-line 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 main-line 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**.

- 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.

Example

```
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 example program. For further examples of using **ct_cancel**, see the *ex_alib.c, ex_amain.c,* and *getsend.c* example programs.

See Also

ct_fetch, ct_results

ct_capability

Function

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*:	ct_capability:
CS_SET	Sets a capability.
CS_GET	Retrieves a capability.

Table 3-12: Values for action *(***ct_capability***)*

type – The type category of the capability. The following table lists the symbolic values that are legal for *type*:

Value of *type*:	What it means:
CS_CAP_REQUEST	Request capabilities. These capabilities describe the types of requests that a connection can support. Request capabilities are retrieve-only.

Table 3-13: Values for type *(***ct_capability***)*

Value of *type*:	What it means:
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.

Table 3-13: Values for type (**ct_capability**) *(continued)*

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 will only need to set or retrieve a small number of capabilities.

CS_CAP_REQUEST Capabilities

CS_CAP_REQUEST Capability:	Meaning:	Capability relates to:
CS_CON_INBAND	In-band (non-expedited) attentions.	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_DATA_BIN	Binary datatype.	Datatypes
CS_DATA_VBIN	Variable-length binary type.	Datatypes

Table 3-14: Request capabilities

CS_CAP_REQUEST Capability:	Meaning:	Capability relates to:
CS_DATA_LBIN	Long binary datatype.	Datatypes
CS_DATA_BIT	Bit datatype.	Datatypes
CS_DATA_BITN	Nullable bit values.	Datatypes
CS_DATA_BOUNDARY	Secure Server boundary datatypes.	Datatypes
CS_DATA_CHAR	Character datatype.	Datatypes
CS_DATA_VCHAR	Variable-length character datatype.	Datatypes
CS_DATA_LCHAR	Long character 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_DATA_INTN	NULL integers.	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_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

Table 3-14: Request capabilities (continued)

CS_CAP_REQUEST Capability:	Meaning:	Capability relates to:
CS_PROTO_DYNPROC	Client-Library prepends SQL to a Dynamic SQL prepare statement.	Dynamic SQL
CS_REQ_BCP	Bulk copy requests.	Commands
CS_REQ_CURSOR	Cursor 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_URGNOTIF	Send notifications with the “urgent” bit set in the TDS packet header.	Registered procedures
CS_REQ_RPC	Remote procedure requests.	Commands

Table 3-14: Request capabilities (continued)

CS_CAP_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_NOVBIN	No variable-length binary type.	Datatypes
CS_DATA_NOLBIN	No long binary datatype.	Datatypes
CS_DATA_NOBIT	No bit datatype.	Datatypes
CS_DATA_NOBOUNDARY	No Secure Server boundary datatypes.	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

Table 3-15: Response capabilities

CS_CAP_RESPONSE Capability:	Meaning:	Capability relates to:
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_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_RES_NOEED	No extended error results.	Results
CS_RES_NOMSG	No message results.	Results
CS_RES_NOPARAM	Don’t use PARAM / PARAMFMT TDS streams for RPC results.	Results
CS_RES_NOSTRIPBLANKS	The server shouldn’t 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

Table 3-15: Response capabilities (continued)

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, value must point to a CS_CAP_TYPE structure.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-16: Return values (***ct_capability***)*

Comments

- 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. Note that 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 will support.
  - 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. 4.0 TDS does, however, 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 will overwrite 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 in order 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 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.

See Also

Capabilities, ct_con_props, ct_connect, ct_options, Properties

ct_close

Function

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, if any, to use for the close. The following table lists the symbolic values that are legal for *option*:

Value of *option*:	What it means:
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.

Table 3-17: Values for option *(***ct_close***)*

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. For more information, see the **Asynchronous Programming** topics page. 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. For more information, see the **Asynchronous Programming** topics page. Note that **ct_close** does not return CS_BUSY when called with *option* as CS_FORCE_CLOSE.

*Table 3-18: Return values (***ct_close***)*

The most common reason for a **ct_close**(CS_UNUSED) failure is pending results on the connection.

Comments

- To de-allocate 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, if any.
- The behavior of **ct_close** depends on the value of *option*, which determines the type of close. Each section below contains information on a type of close.

Default Close Behavior

- If the connection has any pending results, **ct_close** returns CS_FAIL. If the connection has an open cursor, the server closes the cursor 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.

Example

```
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* example program. For another example of using **ct_close**, see the *ex_amain.c* example program.

See Also

ct_callback, ct_con_drop, ct_connect, ct_con_props

ct_cmd_alloc

Function

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.

In case of error, **ct_cmd_alloc** sets **cmd_pointer* to NULL.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-19: Return values (**ct_cmd_alloc**)*

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

Comments

- 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.)

Example

```
/* 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* example program. For further examples of using **ct_cmd_alloc**, see the *compute.c*, *csr_disp.c*, *ex_alib.c*, *exutils.c*, *i18n.c*, and *rpc.c* example programs.

See Also

ct_command, ct_cmd_drop, ct_cmd_props, ct_con_alloc, ct_cursor, ct_dynamic

ct_cmd_drop

Function

De-allocate a CS_COMMAND structure.

Syntax

```
CS_RETCODE ct_cmd_drop(cmd)

CS_COMMAND      *cmd;
```

Parameters

cmd – A pointer to a CS_COMMAND structure.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-20: Return values (***ct_cmd_drop***)*

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.

Comments

- 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 de-allocated, it cannot be re-used. To allocate a new CS_COMMAND structure, an application can call **ct_cmd_alloc**.

- Before de-allocating a command structure, an application should cancel any active commands, process or cancel any pending results, and close and de-allocate any open cursors on the command structure.

Example

```
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* example program. For further examples of using **ct_cmd_drop**, see the *csr_disp.c*, *ex_alib.c*, *exutils.c*, and *i18n.c* example programs.

See Also

ct_command, ct_cmd_alloc

ct_cmd_props

Function

Set or retrieve command structure properties.

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:*	ct_cmd_props:
CS_SET	Sets the value of the property.
CS_GET	Retrieves the value of the property.
CS_CLEAR	Clears the value of the property by resetting it to its Client-Library default value.

Table 3-21: Values for action (**ct_cmd_props**)

property – The symbolic name of the property whose value is being set or retrieved. The **Properties** topics page lists 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_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 null-terminated, pass *buflen* as CS_NULLTERM.

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

outlen – A pointer to an integer variable.

outlen is not used if a property value is being set 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, an application can use the value of **outlen* to determine how many bytes are needed to hold the information.

Summary of Parameters

For information on *action*, *buffer*, *buflen*, and *outlen*, see ‘‘action, buffer, buflen, and outlen’’ on page 2-125.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-22: Return values (***ct_cmd_props***)*

Comments

- 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 the **Properties** topics page for more information on properties.
- An application can use **ct_cmd_props** to set or retrieve the following properties:

Property name:	**What it is:**	***buffer* is:**	**Client-Library can set or retrieve at what 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_CUR SOR_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.	Set to a CS_INT-sized bit-mask.	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, 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, command.	Retrieve only.

Table 3-23: Client-Library properties

Property name:	What it is:	***buffer* is:**	Client-Library can set or retrieve at what level?	Notes
CS_USERDATA	User-allocated data.	User-allocated data.	Connection, command. To set CS_USERDATA at the context level, call **cs_config**.	None.

Table 3-23: Client-Library properties (continued)

Example

```
/*
** 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;

}
```

This code excerpt is from the *ex_alib.c* example program. For another example of using **ct_cmd_props**, see the *rpc.c* example programs.

See Also

ct_config, ct_cmd_alloc, ct_con_props, ct_res_info

ct_command

Function

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. The table in the **Summary of Parameters** section lists the symbolic values that are legal for *type*.

buffer – A pointer to data space.

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, if any.

Currently, 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 that are legal for *option*:

***type* is:**	**Value of *option:***	**Meaning:**
CS_RPC_CMD	CS_RECOMPILE	Recompile the stored procedure before executing it.
	CS_NORECOMPILE	Do not recompile the stored procedure before executing it.
	CS_UNUSED	Equivalent to CS_NORECOMPILE.
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.

Table 3-24: Values for option (**ct_command**)

Summary of Parameters

Value of *type:*	**ct_command initiates:**	***buffer* is:**	***buflen* is:**
CS_LANG_CMD	A language command.	A pointer to the text of the language command.	The length of the **buffer* data or CS_NULLTERM.
CS_MSG_CMD	A message command.	A pointer to a CS_SMALLINT representing the message id.	CS_UNUSED
CS_PACKAGE_CMD	A package command.	A pointer to 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 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 the database table name.	The length of the **buffer* data or CS_NULLTERM.

*Table 3-25: Summary of parameters (***ct_command***)*

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-26: Return values (***ct_command***)*

Comments

- Initiating a command is the first step in sending it to a server.
- Sending a command to a server is a four step process. To send a command to a server, an application must:
  - Initiate the command by calling **ct_command**. This routine sets up internal structures that are used in building a command stream to send to the server.

  - Pass parameters for the command (if required) by calling **ct_param** 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.

  - Send the command to the server by calling **ct_send**.
  - Verify the success of the command by calling **ct_results**.

    This last step does not imply that an application need only call **ct_results** once. An application needs to continue calling **ct_results** until it no longer returns CS_SUCCEED. See the *Open Client Client-Library/C Programmer's Guide* for a discussion of processing results.

- An application can call **ct_cancel** with *type* as CS_CANCEL_ALL to clear a command that has been initiated but not yet sent.
- Within a single connection, the following rules apply to the use of **ct_command**:
  - After calling **ct_command** to initiate a command, an application must either send the initiated command or clear it before calling **ct_command** a second time.
  - After sending a command initiated via **ct_command**, an application must completely process or cancel all results generated by the command before calling **ct_command** to initiate another command.
  - An application cannot call **ct_command** to initiate a command on a command structure that is managing a cursor.
- Each section below contains information on one of the types of commands that **ct_command** can initiate.

Language Commands

- Language commands contain a character string that represents one or more commands 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 authors", CS_NULLTERM,
        CS_UNUSED);
```

- The character string must represent one or more **entire** server commands. Unlike DB-Library's **dbcmd** routine, **ct_command** does not append text to an internal buffer. If an application calls **ct_command** twice in succession, the second **ct_command** call will fail.

- The character string can represent more than one server command. For example, the following language command contains three Transact-SQL commands:

```
ct_command(cmd, CS_LANG_CMD, “use pubs2 \
        select * from titles \
        select * from authors “, CS_NULLTERM,
        CS_UNUSED);
```

- A language command can be in any language, so long as the server to which it is directed can understand it. SQL Server understands Transact-SQL, but an Open Server application constructed with SYBASE 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** once for each variable that the language string contains.
- Transact-SQL command variables must begin with an ‘@’ symbol.
- A language cursor generates a regular row result set when an application calls **ct_command** to fetch against the cursor. The fetch command generates regular row results containing a number of rows equal to the current “cursor rows” setting for the cursor.

Message Commands

- Message commands and results provide a way for clients and servers to communicate specialized information to one another.
- A message has an “id”, which an application provides via **ct_command**’s *buffer* parameter.
- 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 can call **ct_param** once for each parameter that the message requires.

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 can call **ct_param** once for each parameter that the package requires.

RPC (remote procedure call) 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 can call **ct_param** once for each parameter the stored procedure requires.
- After sending an RPC command with **ct_send**, an application can process 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. For more information, see the **Remote Procedure Calls** topics page, 2-160.

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.
- For more information on writing text or image values, see the **Text and Image** topics page.

Send-Bulk-Data Commands

- Internally, SYBASE uses send-bulk-data commands as part of its implementation of Client-Library's bulk copy routines.

Example

```
/*
** 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 */
...CODE DELETED.....
}
```

This code excerpt is from the *exutils.c* example program. For further examples of using **ct_command**, see the *compute.c*, *ex_alib.c*, *getsend.c*, *i18n.c*, and *rpc.c* example programs.

See Also

ct_cmd_alloc, ct_cursor, ct_dynamic, ct_param, ct_send

ct_compute_info

Function

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 that are legal for *type*, see the chart in the **Summary of Parameters** section.

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 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.

ct_compute_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.

Summary of Parameters

Value of *type*:	Value of *colnum*:	ct_compute_info returns:	**buffer* is set to:	**outlen* is set to:
CS_BYLIST_LEN	CS_UNUSED	The number of elements in the bylist array.	An integer value.	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.	A symbolic value, one of: CS_OP_SUM CS_OP_AVG CS_OP_COUNT CS_OP_MIN CS_OP_MAX	sizeof(CS_INT)

*Table 3-27: Summary of parameters (***ct_compute_info***)*

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-28: Return values (***ct_compute_info***)*

Comments

- 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 will contain 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 select-list id of the column from which the compute column derives.

The Compute ID for this 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:

***buffer set to:**	**To indicate:**
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.

Table 3-29: Aggregate operator types

Examples

Assume 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*.

For another example of using **ct_compute_info**, see the *compute.c* example program.

See Also

ct_bind, ct_describe, ct_res_info, ct_results

ct_con_alloc

Function

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.

In case of error, **ct_con_alloc** sets **con_pointer* to NULL.

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

*Table 3-30: Return values (***ct_con_alloc***)*

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

Comments

- 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:
  - Calls **ct_con_alloc** to allocate a CS_CONNECTION structure.

- Calls **ct_con_props** to set the values of connection-specific properties, if desired.
- 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.

• In order to send commands to a server, one or more command structures must be allocated for a connection. **ct_cmd_alloc** allocates a command structure.

Example

```
/*
** 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 */
    ...CODE DELETED.....

    /* Open the connection */
    ...CODE DELETED.....
}
```

For further examples of using **ct_con_alloc**, see the *blktxt.c*, *ex_amain.c*, and *exutils.c* example programs.

See Also

cs_ctx_alloc, ct_cmd_alloc, ct_close, ct_connect, ct_con_props

ct_con_drop

Function

De-allocate 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.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-31: Return values (***ct_con_drop***)*

The most common reason for a **ct_con_drop** failure is that the connection is still open.

Comments

- When a CS_CONNECTION structure is de-allocated, all CS_COMMAND structures associated with it are de-allocated.
- A CS_CONNECTION structure contains information about a particular client / server connection.
- Once a CS_CONNECTION has been de-allocated, it cannot be reused. To allocate a new CS_CONNECTION, an application can call **ct_con_alloc**.
- An application cannot de-allocate 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.

Example

```
/* 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* example program. For further examples of using **ct_con_drop**, see the *blktxt.c* and *ex_amain.c* example programs.

See Also

ct_con_alloc, ct_close, ct_connect, ct_con_props

ct_con_props

Function

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:*	**ct_con_props:**
CS_SET	Sets the value of the property.
CS_GET	Retrieves the value of the property.
CS_CLEAR	Clears the value of the property by resetting it to its Client-Library default value.

Table 3-32: Values for action (**ct_con_props**)

property – The symbolic name of the property whose value is being set or retrieved. The **Properties** topics page lists 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_con_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 null-terminated, pass *buflen* as CS_NULLTERM.

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

outlen – A pointer to an integer variable.

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

Summary of Parameters

For information on *action*, *buffer*, *buflen*, and *outlen*, see "action, buffer, buflen, and outlen" on page 2-125.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-33: Return values (***ct_con_props***)*

Comments

- Connection properties define aspects of Client-Library behavior at the connection level.
- 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. See the "Notes" column in *Table 3-34: Client-Library connection properties*, on page 3-70.
- See the **Properties** topics page for more information on properties.
- An application can use **ct_con_props** to set or retrieve the following properties:

Property name:	**What it is:**	***buffer is:**	**Client-Library can set or retrieve at what level?**	**Notes**
CS_ANSI_BINDS	Whether or not 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.	Connection.	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 or not a connection is enabled to perform bulk copy "in" operations.	CS_TRUE or CS_FALSE.	Connection.	Login property. Cannot be set after connection is established.
CS_CHARSETCNV	Whether or not character set conversion is taking place.	CS_TRUE or CS_FALSE.	Connection.	Retrieve only, after connection is established.

Table 3-34: Client-Library connection properties

Property name:	What it is:	*buffer is:	Client-Library can set or retrieve at what level?	Notes
CS_COMMBLOCK	A pointer to a communication sessions block. This property is specific to IBM-370 systems and is ignored by all other platforms.	A pointer value.	Connection.	Cannot be set after connection is established.
CS_CON_STATUS	The connection's status.	A CS_INT-sized bit-mask.	Connection.	Retrieve only.
CS_DIAG_TIMEOUT	When in-line 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 or not 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_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 or not to expose results of type CS_ROWFMT_RESULT and CS_COMPUTEFMT_RE SULT.	CS_TRUE or CS_FALSE.	Context, connection.	Cannot be set after connection is established.

Table 3-34: Client-Library connection properties (continued)

Property name:	What it is:	*buffer is:	Client-Library can set or retrieve at what level?	Notes
CS_EXTRA_INF	Whether or not to return the extra information that's required when processing Client-Library messages in-line using a SQLCA, SQLCODE, SQLSTATE.	CS_TRUE or CS_FALSE.	Context, connection.	
CS_HIDDEN_KEYS	Whether or not to expose hidden keys.	CS_TRUE or CS_FALSE.	Context, connection, command.	
CS_HOSTNAME	The host machine name.	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 or not the connection is open.	CS_TRUE or CS_FALSE.	Connection.	Retrieve only.
CS_NETIO	Whether network I/O is synchronous or asynchronous.	CS_SYNC_IO, CS_ASYNC_IO.	Context, connection.	Asynchronous connections are either fully or deferred asynchronous, to match their parent context.
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.

Table 3-34: Client-Library connection properties (continued)

Property name:	What it is:	*buffer is:	Client-Library can set or retrieve at what level?	Notes
CS_PARENT_ HANDLE	The address of the connection structure's parent context.	Set to an address.	Connection, command.	Retrieve only.
CS_PASSWORD	The password used to log into the server.	A character string.	Connection.	Login property.
CS_SEC_ APPDEFINED	Whether or not 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 or not 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_ ENCRYPTION	Whether or not the connection will use encrypted password security handshaking.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SEC_ NEGOTIATE	Whether or not the connection will use trusted-user security handshaking.	CS_TRUE or CS_FALSE.	Connection.	Cannot be set after connection is established.
CS_SERVERNAME	The name of the server to which this connection is 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.	
CS_TRANSACTION_ NAME	A transaction name.	A string value.	Connection.	

Table 3-34: Client-Library connection properties (continued)

Property name:	What it is:	*buffer is:	Client-Library can set or retrieve at what level?	Notes
CS_USERDATA	User-allocated data.	User-allocated data.	Connection, command.	
CS_USERNAME	The name used to log into the server.	A character string.	Connection.	Login property. Cannot be set after connection is established.

Table 3-34: Client-Library connection properties (continued)

Example

```
/*
** 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 username, 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 username 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.....
}
```

This code excerpt is from the *blktxt.c* example program. For further examples of using **ct_con_props**, see the *ex_alib.c*, *ex_amain.c*, *exutils.c*, and *rpc.c* example programs.

See Also

ct_capability, ct_cmd_props, ct_connect, ct_config, ct_init, Properties

ct_config

Function

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:*	ct_config:
CS_SET	Sets the value of the property.
CS_GET	Retrieves the value of the property.
CS_CLEAR	Clears the value of the property by resetting it to its Client-Library default value.

Table 3-35: Values for action *(***ct_config***)*

property – The symbolic name of the property whose value is being set or retrieved. The **Properties** topics page lists 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.

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_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.

Summary of Parameters

For information on *action*, *buffer*, *buflen*, and *outlen*, see "action, buffer, buflen, and outlen" on page 2-125.

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

*Table 3-36: Return values (***ct_config***)*

Comments

- Context properties define aspects of Client-Library behavior at the context level.
- 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 actually 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 via **cs_config** affect only CS-Library.

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

srv_props sets and retrieves the values of Server-Library-specific context properties. Properties set via **srv_props** affect only Server-Library.

- See the **Properties** topics page for more information on properties.
- An application can use **ct_config** to set or retrieve the following properties:

Property name:	What it is:	**buffer* is:	Client-Library can set or retrieve at what level?	Notes
CS_ANSI_BINDS	Whether or not to use ANSI-style binds.	CS_TRUE or CS_FALSE.	Context, connection.	
CS_DISABLE_POLL	Whether or not 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_EXPOSE_FMTS	Whether or not 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_EXTRA_INF	Whether or not to return the extra information that's required when processing Client-Library messages in-line using a SQLCA, SQLCODE, or SQLSTATE.	CS_TRUE or CS_FALSE.	Context, connection	

Table 3-37: Client-Library context properties

Property name:	What it is:	**buffer* is:	Client-Library can set or retrieve at what level?	Notes
CS_HIDDEN_KEYS	Whether or not 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.	
CS_LOGIN_TIMEOU T	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.
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 or CS_FALSE.	Context.	
CS_NOINTERRUPT	Whether or not the application can be interrupted.	CS_TRUE or CS_FALSE.	Context.	
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.	

Table 3-37: Client-Library context properties (continued)

Property name:	What it is:	**buffer* is:	Client-Library can set or retrieve at what level?	Notes
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. Currently, the only possible value is CS_VERSION_100	Context.	Retrieve only.

Table 3-37: Client-Library context properties (continued)

Example

```
/* 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* example program.

See Also

cs_config, ct_cmd_props, ct_capability, ct_con_props, ct_connect, ct_init, Properties

ct_connect

Function

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 this structure with login parameters.

server_name – A pointer to the name of the server to connect to. **server_name* is the name given to the server in the interfaces file on the application's host machine. **ct_connect** looks up **server_name* in the interfaces file to determine how to connect to this server.

If *server_name* is NULL, **ct_connect** looks up the interfaces entry that corresponds to the value of the DSQUERY environment variable or logical name. If DSQUERY has not been explicitly set, it has a value of "SYBASE". For more information on the interfaces file, see the *Open Client/Server Supplement*.

➤ ***Note***

An interfaces file may not be used on some platforms. For information on whether your platform uses an interfaces file, see the *Open Client/Server Supplement* for your platform.

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.

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.
CS_PENDING	Asynchronous network I/O is in effect. For more information, see the **Asynchronous Programming** topics page.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-38: Return values (**ct_connect**)*

Common reason for a **ct_connect** failure include:

- Unable to allocate sufficient memory.
- The maximum number of connections is already established. **ct_config** is used to set the maximum number of connections allowed per context.
- Unable to open socket.
- Server name not found in interfaces file.
- Unknown host machine name.
- SQL Server is unavailable or does not exist.
- Login incorrect.
- Could not open interfaces file.

When **ct_connect** returns CS_FAIL, it generates a Client-Library error number that indicates the error.

Comments

- 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 will be synchronous or asynchronous.

  For more information on asynchronous connections, see the **Asynchronous Programming** topics page, 2-3.

- 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 on platform-specific property values, see the *Open Client/Server Supplement*.

- 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 there is no server listening on the specified port, the machine that the server is supposed to be on will signal the client, via a network error, that the connection can't 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 signal 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.

- To close a connection, an application calls **ct_close**.

Multiple QUERY Entries in an Interfaces File

- It is possible to set up an interfaces file so that if **ct_connect** fails to establish a connection with a server, it attempts to establish a connection with an alternate server.

  An application can use the **ct_connect** call:

  ```
  ct_connect(connection, "MARS", CS_NULLTERM)
  ```

  to connect to the server MARS. An interfaces file containing an entry for MARS might look like this:

```
#
MARS
        query tcp hp-ether violet 1025
        master tcp hp-ether violet 1025
        console tcp hp-ether violet 1026
#
VENUS
        query tcp hp-ether plum 1050
        master tcp hp-ether plum 1050
        console tcp hp-ether plum 1051
#
NEPTUNE
        query tcp hp-ether mauve 1060
        master tcp hp-ether mauve 1060
        console tcp hp-ether mauve 1061
```

The application is directed to port number 1025 on the machine *violet*. If MARS is not available, the **ct_connect** call fails. If the interfaces file has multiple *query* entries in it for MARS, however, then when the first connection attempt fails, **ct_connect** will automatically attempt to connect to the next server listed. Such an interfaces file might look like this:

```
#
MARS
        query tcp hp-ether violet 1025
        query tcp hp-ether plum 1050
        query tcp hp-ether mauve 1060
        master tcp hp-ether violet 1025
        console tcp hp-ether violet 1026
#
VENUS
        query tcp hp-ether plum 1050
        master tcp hp-ether plum 1050
        console tcp hp-ether plum 1051
#
NEPTUNE
        query tcp hp-ether mauve 1060
        master tcp hp-ether mauve 1060
        console tcp hp-ether mauve 1061
```

Note that the second *query* entry under MARS is identical to the *query* entry under VENUS, and that the third *query* entry is identical to the *query* entry under NEPTUNE. If this interfaces file is used, then if the application fails to connect with MARS it will automatically attempt to connect with VENUS. If it fails to connect with VENUS, it will automatically attempt to connect with NEPTUNE.

There is no limit on the number of alternate servers that may be listed under a server's interfaces file entry, but each alternate server must be listed in the same interfaces file.

Two numbers may be added after the server's name in the interfaces file:

```
#
MARS retries seconds
        query tcp hp-ether violet 1025
        query tcp hp-ether plum 1050
        query tcp hp-ether mauve 1060
        master tcp hp-ether violet 1025
        console tcp hp-ether violet 1026
```

retries represents the number of additional times to loop through the list of query entries if no connection is achieved during the first pass. *seconds* represents the amount of time, in seconds, that **ct_connect** will wait at the top of the loop before going through the list again. These numbers are optional. If they are not included, **ct_connect** will try to connect to each query entry only once.

Looping through the list and pausing between loops is useful in case any of the candidate servers is in the process of booting.

Multiple query lines can be particularly useful when alternate servers contain mirrored copies of the primary server's databases.

Example

```
/* 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* example program. For further examples of using **ct_connect**, see the *blktxt.c* and *ex_amain.c* example programs.

See Also

ct_close, ct_con_alloc, ct_con_drop, ct_con_props, ct_remote_pwd

ct_cursor

Function

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. The chart in the **Summary of Parameters** section lists the symbolic values that are legal for *type*.

name – A pointer to the name associated with the cursor command, if any. The chart in the **Summary of Parameters** section 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, if any. The chart in the **Summary of Parameters** section 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, if any. The chart in the **Summary of Parameters** section indicates which commands take an option and what that option can be.

Summary of Parameters

Value of *type*:	ct_cursor initiates:	*name* is:	**text* is:	*option* is:
CS_CURSOR_CLOSE	A cursor close command.	NULL	NULL	CS_DEALLOC to close and de-allocate the cursor. CS_UNUSED to close the cursor without de-allocating it.
CS_CURSOR_DEALLOC	A de-allocate cursor command.	NULL	NULL	CS_UNUSED
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_FOR_UPDATE to indicate that the cursor is for update. CS_READ_ONLY to indicate that the cursor is read-only. CS_UNUSED. The meaning of CS_UNUSED is server-defined.
CS_CURSOR_DELETE	A cursor delete command.	A pointer to the name of the table to delete from.	NULL	CS_UNUSED
CS_CURSOR_OPEN	A cursor open command.	NULL	NULL	CS_UNUSED
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. The meaning of CS_UNUSED is server-defined.
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.

*Table 3-39: Summary of parameters (**ct_cursor**)*

Value of *type*:	**ct_cursor** initiates:	***name*** **is:**	****text*** **is:**	***option*** **is:**
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

*Table 3-39: Summary of parameters (***ct_cursor***) (continued)*

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-40: Return values (***ct_cursor***)*

Comments

- 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 de-allocate a cursor as well as commands to update and delete rows in an underlying table.
- Sending a command to a server is a four step process. To send a command to a server, an application must:
  - 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.
  - Pass parameters for the command (if required) by calling **ct_param** once for each parameter that the command requires.

    Not all commands require parameters. See the "Client-Library Cursor Declare" section for an explanation of when to call **ct_param**.
  - Send the command to the server by calling **ct_send**.
  - Verify the success of the command by calling **ct_results**.

This last step does not imply that an application need only call **ct_results** once. If the value of **ct_results**' *result_type* parameter indicates that there are fetchable results, the application will most likely process the results using a loop controlled by **ct_results**. See the *Open Client Client-Library/C Programmer's Guide* for a discussion of processing results.

Batching Client-Library Cursor Commands

- An application can "batch" together commands to achieve reduced network traffic and improved application performance.

  To batch together commands to declare, set rows for, and open a Client-Library cursor, the application:

  - Calls **ct_cursor** to declare the cursor.
  - Calls **ct_param** (if necessary) to define the format(s) of host variable(s).
  - Calls **ct_cursor** (optional) to set rows for the cursor.
  - Calls **ct_cursor** to open the cursor.
  - Calls **ct_param** (if necessary) to supply value(s) for the host variable(s).
  - Calls **ct_send** to send the command batch to the server.

  The sequence of calls is:

  ```
  ct_cursor(CS_CURSOR_DECLARE)
  ct_param
  ct_cursor(CS_CURSOR_ROWS)
  ct_cursor(CS_CURSOR_OPEN)
  ct_param
  ct_send
  ```

  Commands must be batched in the logical order: declare, set cursor rows, open.

Client-Library Cursor Close

- A **ct_cursor**(CS_CURSOR_CLOSE) command throws away the cursor result set that was generated when the cursor was opened.
- An application can re-open a closed cursor.

Client-Library Cursor De-allocate

- A **ct_cursor**(CS_CURSOR_DEALLOC) command de-allocates a Client-Library cursor. If a cursor has been de-allocated, it cannot be re-opened.
- An application cannot de-allocate an open cursor.
- To initiate a command to both close and de-allocate a Client-Library cursor, call **ct_cursor** with *type* as CS_CURSOR_CLOSE and *option* as CS_DEALLOC.

Client-Library Cursor Declare

- Declaring a Client-Library cursor is equivalent to associating the cursor name with a SQL statement. This SQL statement is called the **body** of the cursor.

  The SQL statement associated with a cursor can be a command to execute a stored procedure. For example:

  ```
  ct_cursor (cmd, CS_CURSOR_DECLARE, "mycursor",
      CS_NULLTERM, "execute my_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 SQL statement associated with a cursor can contain host variables. If it does, an application must define the variables' formats for the server by calling **ct_param** at cursor declare time, once for each variable.

➤ ***Note***

Defining a variable's format is not the same thing as supplying a value for the variable. An application supplies values for host variables at cursor open time.

  Skip this step if the SQL statement is a stored procedure or a dynamic SQL statement, but supply values for a stored procedure's input parameter(s) or a dynamic SQL statement's placeholder(s) at cursor open time.
- To declare a cursor as 'read-only', an application specifies *option* as CS_READ_ONLY. This means that the cursor cannot be used to change values in the underlying server tables.

- To declare a cursor 'for update', an application specifies *option* as CS_FOR_UPDATE. This means that the cursor can be used to change values in the underlying server tables.

  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.

  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,
      CS_UNUSED);

  format.status = CS_UPDATECOL;

  ct_param(cmd, &format, "au_lname", CS_NULLTERM,
      CS_UNUSED);

  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);
  ```

Client-Library Cursor Delete

- A **ct_cursor**(CS_CURSOR_DELETE) command deletes the current cursor row from the cursor result set. The delete is propagated back to the underlying server tables.

Client-Library Cursor Open

- 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**.

- Some cursors require input parameter values at cursor open time. An application can pass input parameter values for a cursor open command by calling **ct_param** after calling **ct_cursor**. A cursor open command requires parameters if any of the following are 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)).

Dynamic SQL Cursor Option

- A **ct_cursor**(CS_CURSOR_OPTION) command sets Client-Library cursor options ('read-only' or 'for update') for dynamic SQL prepared statements.

  Applications that declare a cursor on a dynamically prepared SQL statement (**ct_dynamic**(CS_CURSOR_DECLARE)) must follow the cursor declaration with a call to **ct_cursor**(CS_CURSOR_OPTION). Unlike a Client-Library cursor declare command, the dynamic SQL cursor declare command does not provide a way to specify cursor options.

  A dynamic SQL application declares a cursor on a prepared SQL statement to make use of Client-Library cursor functionality. Once an application calls **ct_dynamic**(CS_CURSOR_DECLARE), it makes **ct_cursor** calls from that point onwards, beginning with **ct_cursor**(CS_CURSOR_OPTION).

  An application that uses **ct_cursor** to declare a cursor does not need to use the cursor option command because the cursor declare command (**ct_cursor**(CS_CURSOR_DECLARE)) provides a way to specify cursor options (via the *option* parameter). However, it is not illegal to follow a cursor declare command with a cursor option command; the newly-specified option simply replaces the option that was originally specified.
- To declare a cursor as 'read-only', an application specifies *option* as CS_READ_ONLY. This means that the cursor cannot be used to change values in the underlying server tables.

- To declare a cursor 'for update', an application specifies *option* as CS_FOR_UPDATE. This means that the cursor can be used to change values in the underlying server tables.

  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.
- An application can only specify cursor options *before* opening a cursor.

Client-Library Cursor Rows

- 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 only set cursor rows *before* opening a cursor.
- The cursor rows setting defaults to one row.

Client-Library Cursor Update

- 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.
- When updating a SQL 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*...).
- An application can update only a single table.

Example

```
/* 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;
    }
```

This code excerpt is from the *csr_disp.c* example program.

See Also

Cursors, ct_cmd_alloc, ct_keydata, ct_param, ct_results, ct_send

ct_data_info

Function

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 of *action:*	**ct_data_info:**
CS_SET	Defines an I/O descriptor.
CS_GET	Retrieves an I/O descriptor.

Table 3-41: Values for action (**ct_data_info**)

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 in a **select** statement's select-list is column number 1, the second 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 on this structure, see "CS_IODESC Structure" on page 2-54.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-42: Return values (***ct_data_info***)*

Comments

- **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 *log_on_update* field in the CS_IODESC to indicate 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, if any.
- 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.
- It is illegal to call **ct_data_info** to retrieve the I/O descriptor for a column before calling **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.

- For more information on the I/O descriptor structure, see “CS_IODESC Structure” on page 2-54.
- For more information on text and image, see “Text and Image” on page 2-188.

Example

```
/*
** 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 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* example program.

See Also

ct_get_data, ct_send_data, Text and Image

ct_debug

Function

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, use the chart in the *flag* parameter section 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 in order 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 the chart in the *flag* parameter section to determine whether or not to supply *connection*.

operation – The operation to perform. The table in the **Summary of Parameters** section lists the symbolic values that are legal for *operation*.

flag – A bit mask representing debug subsystems. The following table lists the symbolic values that can make up *flag*:

Value of *flag*:	**Requires:**	**When the flag is enabled, Client-Library:**
CS_DBG_ALL	*context* and *connection*	Takes all possible debug actions.
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*	Captures information exchanged with a server in protocol-specific (for example, TDS) format. This information is not human readable.
CS_DBG_PROTOCOL_ STATES	*connection*	Prints information relating to Client-Library protocol-level state transitions.

Table 3-43: Values for flag *(**ct_debug**)*

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*.

Summary of Parameters

***operation* is:**	***flag* is:**	***filename* is:**	**ct_debug:**
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-form debug information.

*Table 3-44: Summary of parameters (***ct_debug***)*

Returns

Returns:	**To Indicate:**
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 the **Asynchronous Programming** topics page.

*Table 3-45: Return values (***ct_debug***)*

Comments

- **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. The chart for the *flag* parameter 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* and protocol-form debug information to *connect.dat* 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.

Example

```
...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* example program. For further examples of using **ct_debug**, see the *ex_alib.c*, and *ex_amain.c* example programs.

See Also

Error and Message Handling, Client-Library Messages, ct_callback, ct_con_alloc, ct_diag

ct_describe

Function

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** statement's select-list is column number 1, the second 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:

Field name:	**For which types of result items?**	**ct_describe sets the field to:**
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 on the **Types** topics page are valid, with the exceptions of 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 of the data for the column or parameter.

*Table 3-46: Fields in the CS_DATAFMT structure (**ct_describe**)*

Field name:	For which types of result items?	ct_describe sets the field to:
scale	Regular columns, column formats, return parameters, compute columns, or compute column formats of type numeric or decimal.	The scale of the result data item.
precision	Regular columns, column formats, return parameters, compute columns, or compute column formats of type numeric or decimal.	The precision of the result data item.
status	Regular columns and column formats.	A bitmask of the following symbols, or-ed together: 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 the **Properties** topics page. 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.
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 SQL Server user-defined datatype of the column or parameter, if any. *usertype* is set in addition to (not instead of) *datatype*.
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.

*Table 3-46: Fields in the CS_DATAFMT structure (**ct_describe**) (continued)*

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed. **ct_describe** returns CS_FAIL if *item* does not represent a valid result data item.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-47: Return values (***ct_describe***)*

Comments

- 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 the CS_DATAFMT topics page for a description of the CS_DATAFMT structure.
- See the **Results** topics page for a description of result types.

Example

```
/* 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* example program. For further examples of using **ct_describe**, see the *compute.c*, *ex_alib.c*, *getsend.c*, and *i18n.c* example programs.

See Also

ct_bind, ct_fetch, ct_res_info, ct_results, Results

ct_diag

Function

Manage in-line 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. The table in the **Summary of Parameters** section lists the symbolic values that are legal 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. The following table lists the symbolic values that are legal for *type*:

Value of *type*:	To indicate:
SQLCA_TYPE	A SQLCA structure.
SQLCODE_TYPE	A SQLCODE structure, which is a long integer.
SQLSTATE_TYPE	A SQLSTATE structure, which is an array of bytes.
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.

Table 3-48: Values for type *(**ct_diag**)*

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.

Summary of Parameters

Value of *operation:*	**ct_diag:**	***type* is:**	***index* is:**	***buffer* is:**
CS_INIT	Initializes in-line 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.	One of the legal *type* values: 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 legal 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.

*Table 3-49: Summary of parameters (***ct_diag***)*

Value of *operation:*	ct_diag:	*type* is:	*index* is:	*buffer* is:
CS_GET	Retrieves a specific message.	Any legal *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 legal 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.

*Table 3-49: Summary of parameters (***ct_diag***) (continued)*

Returns

Returns:	To Indicate:
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, when there are only 2 messages queued.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-50: Return values (***ct_diag***)*

Common reasons for a **ct_diag** failure include:

- Invalid *connection*.
- Inability to allocate memory.
- Invalid parameter combination.

Comments

- 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 in-line, using **ct_diag**.

  It is possible for an application to switch back and forth between the two methods. For information on how to do this, see the **Errors and Messages** topics page.
- **ct_diag** manages in-line 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 in-line 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:

  ```
  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 In-Line Error Handling

- To initialize in-line error handling, an application calls **ct_diag** with *operation* as CS_INIT.
- Generally, if a connection will use in-line error handling, an application should call **ct_diag** to initialize in-line 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, SQLCODE, or CS_ALLMSG_TYPE.
- If *type* is not CS_ALLMSG_TYPE:
  - **ct_diag** assumes that *buffer* points to a structure of type *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 operations 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.
- See the **SQLCA, SQLCODE, CS_CLIENTMSG** and **CS_SERVERMSG** topics pages for information on these structures.

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 on extended error data, see "Extended Error Data" on page 2-79.

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) call with *index* i returns the i'th message chunk, not the i'th message.
  - 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.
- For more information on sequenced messages, see "Sequencing Long Messages" on page 2-77.

See Also

Error and Message Handling, Client-Library Messages, ct_callback, ct_options

ct_dynamic

Function

Initiate a prepared dynamic SQL statement 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. The table in the **Summary of Parameters** sections lists the symbolic values that are legal 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.

Summary of Parameters

Value of *type:*	**ct_dynamic:**	***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	De-allocates a prepared SQL statement.	The prepared statement identifier.	NULL
CS_DESCRIBE_INPUT	Retrieves input parameter information. An application can access this information via **ct_describe** or via **ct_dyndesc**.	The prepared statement identifier.	NULL
CS_DESCRIBE_OUTPUT	Retrieves column list information. An application can access this information via **ct_describe** or via **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	Execute 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.

*Table 3-51: Summary of parameters (***ct_dynamic***)*

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-52: Return values (***ct_dynamic***)*

Comments

- Initiating a command is the first step in sending it to a server.
- Sending a command to a server is a four step process. To send a command to a server, an application must:
  - 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.
  - Pass parameters for the command, if required. Most applications will pass parameters by calling **ct_param** once for each parameter that the command requires, but it is also possible to pass parameters for a command by using **ct_dyndesc**.
  - Send the command to the server by calling **ct_send**.
  - Verify the success of the command by calling **ct_results**.

    This last step does not imply that an application need only call **ct_results** once. If the value of **ct_results**' *result_type* parameter indicates that there are fetchable results, the application will most likely process the results using a loop controlled by **ct_results**. See the *Open Client Client-Library/C Programmer's Guide* for a discussion of processing results.

About Prepared Statements

- A prepared SQL statement is a SQL statement which 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.
- If a prepared statement is a Transact-SQL command containing host variables, each variable must begin with a colon (:).
- If a prepared statement requires parameters, they are passed using **ct_param** or **ct_dyndesc** at execute time.
- Once a statement is successfully prepared, it can be executed repeatedly until it is de-allocated.
- For more information on dynamic SQL, see the **Dynamic SQL** topics page.

Preparing a Statement

- To initiate a command to prepare a statement, an application calls **ct_dynamic** with *type* as CS_PREPARE.

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.
- An application must declare a cursor on a prepared statement prior to executing the prepared statement.

Setting Options

- After declaring a cursor on a prepared statement, an application can call **ct_cursor**(CS_CURSOR_OPTION) to set options ('readonly' and 'for update') for the cursor.

Getting a Description of Prepared Statement Input

- To initiate a command to get a description of prepared statement input parameters, an application calls **ct_dynamic** with *type* as CS_DESCRIBE_INPUT.
- An application typically retrieves a description of prepared statement input parameters before passing input values to a prepared statement.
- For information on how to access the information returned as the result of a CS_DESCRIBE_INPUT command, see "Getting a Description of Prepared Statement Input," on the **Dynamic SQL** topics page.

Getting a Description of Prepared Statement Output

- To initiate a command to get a description of prepared statement output columns, an application calls **ct_dynamic** with *type* as CS_DESCRIBE_OUTPUT.
- For information on how to access the information returned as the result of a CS_DESCRIBE_OUTPUT command, see "Getting a Description of Prepared Statement Output," on the **Dynamic SQL** topics page.

Executing a Prepared Statement

- To initiate a command to execute a prepared statement, an application calls **ct_dynamic** with *type* as CS_EXECUTE.

Executing a Literal Statement

- To initiate a command to execute a literal SQL statement, an application calls **ct_dynamic** with *type* as CS_EXEC_IMMEDIATE.

De-allocating a Prepared Statement

- To initiate a command to de-allocate a prepared statement, an application calls **ct_dynamic** with *type* as CS_DEALLOC.

See Also

Dynamic SQL, ct_dyndesc, ct_param, ct_send

ct_dyndesc

Function

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 connection.

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 that are legal for *operation*:

Value of *operation:*	**ct_dyndesc:**
CS_ALLOC	Allocates a descriptor.
CS_DEALLOC	De-allocates a descriptor.
CS_GETATTR	Retrieves a parameter or result item's attributes.
CS_GETCNT	Retrieves the number of parameters or columns.

Table 3-53: Values for operation (**ct_dyndesc**)

Value of *operation:*	**ct_dyndesc:**
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.

Table 3-53: Values for operation (**ct_dyndesc**)

index – When used, an integer variable.

Depending on the value of *operation*, *index* can be either the zero-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.

The following table lists the possible values of **indicator*:

Value of *operation*:	Value of **indicator:*	To Indicate:
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.

Table 3-54: Values for indicator (**ct_dyndesc**)

Returns

Returns:	To Indicate:
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_CANCELED	The dynamic SQL operation was canceled.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-55: Return values (***ct_dyndesc***)*

Comments

- 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 de-allocated by another command structure within the same context.
- For more information about dynamic SQL, see the **Dynamic SQL** topics page.

Allocating a Descriptor

- To allocate a descriptor, an application calls **ct_dyndesc** with *operation* as CS_ALLOC.

- The following table lists parameter values for 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

Table 3-56: Parameter values for CS_ALLOC operations

De-allocating a Descriptor

- To de-allocate a descriptor, an application calls **ct_dyndesc** with *operation* as CS_DEALLOC.
- The following table lists parameter values for CS_DEALLOC operations:

descriptor, desclen:	*index:*	*datafmt:*	*buffer, buflen:*	*copied:*	*indicator:*
The name of the descriptor to de-allocate, the length of the name or CS_NULLTERM.	CS_UNUSED	NULL	NULL, CS_UNUSED	NULL	NULL

Table 3-57: Parameter values for CS_DEALLOC operations

Retrieving a Parameter or Result Item’s Attributes

- To retrieve a parameter’s or a result data item’s attributes, an application calls **ct_dyndesc** with *operation* as CS_GETATTR.

- The following table lists parameter values for 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.	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.

Table 3-58: Parameter values for CS_GETATTR operations

- An application needs to set the **datafmt* fields for a CS_GETATTR operation exactly as they would be set for a **ct_bind** call. The following table lists the fields that are used:

Field name:	**Set the field to:**
datatype	The datatype of the *buffer* variable.
format	A bit-mask of format symbols.
maxlength	The length of the *buffer* data space.
scale	The scale of a numeric or decimal buffer; ignored for all other datatypes.
precision	The precision of a numeric or decimal buffer; 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.

Table 3-59: CS_DATAFMT fields to set for CS_GETATTR operations

- **ct_dyndesc**(CS_GETATTR) sets the **datafmt* fields exactly as **ct_describe** would set them. The following table lists the fields in **datafmt* that **ct_dyndesc** sets:

Field name:	**ct_dyndesc sets the 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 on the **types** topics page 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 scale of the result data item.
precision	The precision of the result data item.
status	A bitmask of the following symbols, or-ed together: 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 the **Properties** topics page. 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.
count	*count* represents the number of rows copied to program variables per **ct_fetch** call.**ct_dyndesc** sets *count* to 1 to provide a default value in case an application uses **ct_dyndesc**’s return CS_DATAFMT as **ct_bind**’s input CS_DATAFMT.

Table 3-60: CS_DATAFMT fields set during CS_GETATTR operations

Field name:	**ct_dyndesc sets the field to:**
usertype	The SQL Server user-defined datatype of the column or parameter, if any. *usertype* is set in addition to (not instead of) *datatype*.
locale	A pointer to a CS_LOCALE structure that contains locale information for the data. This pointer can be NULL.

Table 3-60: CS_DATAFMT fields set during CS_GETATTR operations (continued)

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:

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

Table 3-61: Parameter values for CS_GETCNT operations

Setting a Parameter's Attributes

- To set a parameter's attributes, an application calls **ct_dyndesc** with *operation* as CS_SETATTR.

- The following table lists parameter values for 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.	**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.

Table 3-62: Parameter values for CS_SETATTR operations

- An application needs to set the **datafmt* fields for a CS_SETATTR operation exactly as they would be set for a **ct_param** call. The following table lists the fields that are used:

Field name:	**Set the 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.

Table 3-63: CS_DATAFMT fields for CS_SETATTR operations

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.
- The following table lists parameter values for CS_SETCNT 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 new descriptor count.	NULL	NULL, CS_UNUSED	NULL	NULL

Table 3-64: Parameter values for CS_SETCNT operations

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.
- The following table lists parameter values for CS_USE_DESC operations:

descriptor, desclen:	*index:*	*datafmt:*	*buffer, buflen:*	*copied:*	*indicator:*
The name of the descriptor to allocate, the length of the name or CS_NULLTERM.	CS_UNUSED	NULL	NULL, CS_UNUSED	NULL	NULL

Table 3-65: Parameter Values for CS_USE_DESC operations

- 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
```

See Also

ct_cursor, ct_dynamic, ct_fetch

ct_exit

Function

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 table lists the symbolic values that are legal for *option*:

Value of *option*:	Behavior of ct_exit:
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.

Table 3-66: Values for option (**ct_exit**)

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

*Table 3-67: Return values (***ct_exit***)*

Comments

- **ct_exit** terminates Client-Library for a specific context. It closes all open connections, de-allocates internal data space and cleans up any platform-specific initialization.
- **ct_exit** must be the last Client-Library routine called within an Client-Library context.
- If an application finds it needs to call Client-Library routines after it has called **ct_exit**, it can re-initialize 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 connections' 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 de-allocate the context before calling **ct_exit**.

Example

```
/*
** 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* example program. For another example of using **ct_exit**, see the *ex_amain.c* example program.

See Also

ct_close, ct_init

ct_fetch

Function

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 in order to ensure compatibility with future versions of Client-Library.

offset –This parameter is currently unused and must be passed as CS_UNUSED in order to ensure compatibility with future versions of Client-Library.

option – This parameter is currently unused and must be passed as CS_UNUSED in order 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.

Returns

Returns:	To Indicate:
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.
CS_ROW_FAIL	A recoverable error occurred while fetching a row. Recoverable errors include memory allocation failures and conversion errors that occur while copying row values to program variables. An application can continue calling **ct_fetch** to keep retrieving rows, or can call **ct_cancel** to cancel the remaining results. **ct_fetch** places the number of rows fetched in **rows_read*. This number includes the row on which the error occurred. The application must continue to call **ct_fetch**, as the result data is not yet completely fetched.
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 **cs_close**(CS_FORCE_CLOSE) to force the connection closed.
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*.

*Table 3-68: Return values (***ct_fetch***)*

Returns:	To Indicate:
CS_PENDING	Asynchronous network I/O is in effect. For more information, see the **Asynchronous Programming** topics page.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-68: Return values (**ct_fetch**) (continued)*

A common reason for a **ct_fetch** failure is that a program variable specified via **ct_bind** is not large enough for a fetched data item.

Comments

- "Result data" is an umbrella term for all the types of data that a server can return to an application. These types of data include:
  - Regular rows.
  - Cursor rows.
  - Return parameters. Types of data that are returned as parameters include message parameters, stored procedure return parameters, extended error data, and registered procedure notification parameters.
  - Stored procedure status numbers.
  - 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 calling **ct_results**, an application can:
  - Process the result set by binding the result items and fetching the data. (A typical application will call **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. However, an application can also use **ct_fetch** and **ct_dyndesc** to process a result set.)
  - Discard the result set, using **ct_cancel**.
- 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 via 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.

Example

```
/* 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* example program. For further examples of using **ct_fetch**, see the *compute.c, ex_alib.c,getsend.c,* and *i18n.c* example programs.

See Also

ct_bind, ct_describe, ct_get_data, ct_results, Cursors, Results

ct_get_data

Function

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 be incremented only; 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** statement's select-list is column number 1, the second 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 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, *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.*

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page for more information.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-69: Return values (**ct_get_data**)*

Comments

- 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 on 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 2-188.
- **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 multi-byte 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 columns number 1 through 4 and binds columns number 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.

➤ ***Note***

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.

- For more information on how to use **ct_get_data**, see"Text and Image" on page 2-188.

Example

```
/*
** 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 illustratethis, 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* example program.

See Also

ct_bind, ct_data_info, ct_fetch, ct_send_data, Text and Image

ct_getformat

Function

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** statement's select-list is column number 1, the second 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).

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-70: Return values (**ct_getformat**)*

Comments

- 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

Function

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.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-71: Return values (***ct_getloginfo***)*

Comments

- 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 pass-through.

- In order 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 which may not be re-entrant.

TDS Pass-Through

- 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 pass-through, 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 only be made in an Open Server SRV_CONNECT event handler, are:

1. **srv_getloginfo** to allocate a CS_LOGINFO structure and fill it with TDS information from a client login request.
2. **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.
3. **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.
4. **srv_setloginfo** to send the remote server's response, retrieved in step 3, to the client.

Copying Login Properties

For information on using **ct_getloginfo** to copy login properties from an open connection to a newly-allocated connection structure, see the **Properties** topics page.

See Also

ct_recvpassthru, ct_sendpassthru, ct_setloginfo

ct_init

Function

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. The following table lists the symbolic values that are legal for *version*:

Value of *version:*	To Indicate:	Features Supported:
CS_VERSION_100	10.0 behavior.	Cursors, registered procedures, remote procedure calls. This is the initial version of Client-Library.

Table 3-72: Values for version *(***ct_init***)*

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_MEM_ERROR	The routine failed due to a memory allocation error.

*Table 3-73: Return values (***ct_init***)*

Returns:	To Indicate:
CS_FAIL	The routine failed for other reasons. **ct_init** returns CS_FAIL if Client-Library cannot provide *version*-level behavior.

*Table 3-73: Return values (**ct_init**) (continued)*

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.

Comments

- **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 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.

Example

```
/*
** ex_init()
** EX_CTLIB_VERSION is defined in the examples header file
** as CS_VERSION_100.
         */

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);
     if (retcode != CS_SUCCEED)
     {
          ...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;
     }

#ifdef EX_API_DEBUG
     ...CODE DELETED.....
#endif

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

     /* Call ct_config to set context properties */
     ...CODE DELETED.....

     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* example program. For another example of using **ct_init**, see the *ex_amain.c* example program.

See Also

cs_ctx_alloc, ct_exit

ct_keydata

Function

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:*	ct_keydata:
CS_SET	Sets the contents of the key column.
CS_GET	Retrieves the contents of the key column.

Table 3-74: Values for action *(***ct_keydata***)*

colnum – The number of the column of interest. The first column in a result set is column number 1, the second 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.

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed. **ct_keydata** returns CS_FAIL if *colnum* does not represent a key column.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-75: Return values (***ct_keydata***)*

Comments

- An application can use **ct_keydata** to redefine "current" 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 allows 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 must have its CS_HIDDEN_KEYS property set to CS_TRUE.
  - At least one fetch must have occurred on the cursor.
- 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

Cursors, ct_cursor, ct_describe, ct_res_info, ct_results

ct_labels

Function

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:*	ct_labels:
CS_SET	Sets a security label.
CS_CLEAR	Clears all security labels previously specified for this connection.

Table 3-76: Values for action (**ct_labels**)

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 one 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 one byte long.

If *action* is CS_CLEAR, pass *valuelen* as CS_UNUSED.

outlen – This parameter is currently unused and must be passed as NULL.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-77: Return values (**ct_labels**)*

Comments

- An application needs to define security labels if it will be connecting to a server that uses trusted-user security handshakes.
- Secure SQL Server uses trusted-user security handshakes. On Secure SQL Server, security labels are known as “sensitivity labels.”
- 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 via **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 / 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

Function

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:*	**ct_options:**
CS_SET	Sets the option.
CS_GET	Retrieves the option. An application can use **ct_options** to retrieve options from 10.0+ SQL Servers only.
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.

Table 3-78: Values for action (**ct_options**)

option – The server option of interest. The chart in the **Summary of Parameters** section lists the symbolic values that are legal for *option*. For more information on these options, see the **Options** topics page.

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_param** 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.

Summary of Parameters

Value of *option:*	**param* is:	Legal values for **param:*	Defaults to:
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
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_CURCLOSEON XACT	A boolean value.	CS_TRUE, CS_FALSE	CS_FALSE
CS_OPT_CURREAD	A string value representing a read level label.	A string value.	NULL

*Table 3-79: Summary of parameters (***ct_options***)*

Value of *option:*	***param** is:	Legal values for ***param:**	Defaults to:
CS_OPT_CURWRITE	A string value representing a write level label.	A string value.	NULL
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_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_ISOLATION	A symbolic value representing the isolation level.	CS_OPT_LEVEL1, CS_OPT_LEVEL3	CS_OPT_ LEVEL1
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 regular rows to return.	An integer value. 0 means all rows are returned.	0, meaning all rows are returned.

*Table 3-79: Summary of parameters (***ct_options***) (continued)*

Value of *option:*	***param* is:**	**Legal values for *param:***	**Defaults to:**
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

*Table 3-79: Summary of parameters (**ct_options**) (continued)*

Returns

Returns:	**To Indicate:**
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 the **Asynchronous Programming** topics page for more information.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-80: Return values (**ct_options**)*

Comments

- 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.
- An application cannot use **ct_options** to retrieve options from pre-10.0 SQL Servers.
- An application can use **ct_options** to set options in pre-10.0 SQL Servers, but pre-10.0 servers do not support all options listed in the **Summary of Parameters** section. For example, the 4.9 SQL Server does not support the CS_OPT_RESTREES option. For information on which options a server supports, see the manual page for the **set** command in the *SQL Server Reference Manual*.

See Also

ct_capability, ct_con_props, Options

ct_param

Function

Define a command parameter.

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 on how to set these fields for specific uses of **ct_param**, see the charts in the **Comments** section.

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.

Summary of Parameters

Type of command:	ct_param called for what purpose?	*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 input parameter values.	CS_INPUTVALUE	The parameter value and length.
Cursor update	To pass input parameter values.	CS_INPUTVALUE	The parameter value and length.
Dynamic SQL execute	To pass input parameter values.	CS_INPUTVALUE	The parameter value and length.
Language	To pass input parameter values.	CS_INPUTVALUE	The parameter value and length.
Message	To pass input parameter values.	CS_INPUTVALUE	The parameter value and length.
RPC	To pass input or return parameter values.	CS_RETURN to pass a return parameter; CS_INPUTVALUE to pass a non-return parameter.	The parameter value and length.

*Table 3-81: Summary of parameters (***ct_param***)*

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-82: Return values (***ct_param***)*

Comments

- 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 on these uses, see the following sections, ‘‘Identifying Update Columns for a Cursor Declare Command’’, ‘‘Defining Host Variable Formats’’, and ‘‘Passing Input Parameter Values’’.
- In some cases an application may need to pass a parameter that has a value of NULL. 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. In this case, *data* and *datalen* are ignored.
  - Pass *data* as NULL and *datalen* as 0 or CS_UNUSED.
- Client-Library does not perform any conversion on parameters before passing them to the server. If parameter conversion is required, it is the server’s responsibility.

Identifying Update Columns for a Cursor Declare Command

- An application needs to identify update columns for a cursor declare command if some, but not all, of the columns are “for update.” Update columns can be used to change values in underlying database tables.
- 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:

Field name:	Set the field to:
status	CS_UPDATECOL
All other fields	Are ignored.

Table 3-83: CS_DATAFMT fields for identifying update columns

Defining Host Variable Formats

- An application needs to define host variable formats for cursor declare commands, when the body 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:

Field name:	Set the field to:
name	The name of the host variable.
namelen	The length, in bytes, of *name*, or 0 to indicate an unnamed parameter.

Table 3-84: CS_DATAFMT fields for defining host variable formats

Field name:	Set the field to:
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. 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.

Table 3-84: CS_DATAFMT fields for defining host variable formats (continued)

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
  - 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.
- 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*$\rightarrow$*status* can be either CS_RETURN, to indicate that the a return parameter, or CS_INPUTVALUE, to indicate a non-return parameter.
- 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 commands require input parameter values when the stored procedure being executed takes parameters.
- The following table lists the fields in **datafmt* that are used when passing input parameter values:

Field name:	**Set the field 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_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. *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.

Table 3-85: CS_DATAFMT fields for passing input parameter values

Example

```
/*
** 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)
{
    ...CODE DELETED.....
}
if ((retcode = ct_con_props(connection, CS_GET,
    CS_PARENT_HANDLE, &context, CS_UNUSED,
    NULL)) != CS_SUCCEED)
{
    ...CODE DELETED.....
}
retcode = cs_convert(context, &srcfmt,
    (CS_VOID *)moneystring, &destfmt, &moneyvar,
    &destlen);
if (retcode != CS_SUCCEED)
{
    ...CODE DELETED.....
}

/*
** Initiate the RPC command for our stored
** procedure.
*/
if ((retcode = ct_command(cmd, CS_RPC_CMD,
    rpc_name, CS_NULLTERM,CS_NO_RECOMPILE)) !=
    CS_SUCCEED)
{
    ...CODE DELETED.....
}
```

```
/*
** Clear and setup 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),
    CS_UNUSED)) != CS_SUCCEED)
{
    ...CODE 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), CS_UNUSED)) !=
    CS_SUCCEED)
{
    ...CODE 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),
    CS_UNUSED))  != CS_SUCCEED)
{
    ...CODE 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),
    CS_UNUSED))  != CS_SUCCEED)
{
    ...CODE 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)
{
    ...CODE 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)
{
    ...CODE 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),
        CS_UNUSED))  != CS_SUCCEED)
    {
        ...CODE DELETED.....
    }

    return retcode;
}
```

This code excerpt is from the *rpc.c* example program.

See Also

ct_command, ct_cursor, ct_dynamic, ct_send

ct_poll

Function

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 either a server response arrives or a system interrupt occurs.

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:

Value of *compid*:	Indicating:
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_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.

Table 3-86: Values for compid *(***ct_poll***)*

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 can be any of the return codes listed for the routine, with the exception of CS_PENDING.

Summary of Parameters

***context*:**	***connection*:**	***compconn*:**	**ct_poll:**
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.

*Table 3-87: Summary of parameters (**ct_poll**)*

Returns

Returns:	**To Indicate:**
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 were found.
CS_INTERRUPT	A system interrupt has occurred.

*Table 3-88: Return values (**ct_poll**)*

ct_poll returns CS_FAIL if it polls a connection that has died.

Comments

- **ct_poll** will poll either a specific connection or all connections within a specific context.
- If a platform does not provide interrupt-driven I/O, then an application must read from the network in order 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** in order to recognize asynchronous operation completions and registered procedure notifications.

- 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 CS_ASYNC_NOTIFS is CS_FALSE, **ct_poll** will not read from the network. This means that an application must be reading results in order for **ct_poll** to report a registered procedure notification.
- For more information see the **Callbacks** and **Asynchronous Programming** topics pages.

Example

```
/*
** BusyWait()
**
** Prints out dots while waiting for an async
** operation to complete. It demonstrates an
** application’s ability to do other work while an
** async operation is pending.
*/

CS_STATIC CS_RETCODE CS_INTERNAL
BusyWait(connection, where)
CS_CONNECTION   *connection;
char            *where;
{
    CS_COMMAND  *compcmd;
    CS_INT      compid;
    CS_RETCODE  compstat;

    fprintf(stdout, "Waiting [%s]", where);
    fflush(stdout);
    while (completed == CS_FALSE)
    {
        fprintf(stdout, ".");
        fflush(stdout);
        ct_poll(NULL, connection, 100, NULL, &compcmd,
            &compid, &compstat);
    }
    fprintf(stdout, "\n");
    return completed_retcode;
}
```

This code excerpt is from the *ex_amain.c* example program.

See Also

Asynchronous Programming, Callbacks, ct_callback, ct_wakeup, Properties

ct_recvpassthru

Function

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.

Returns

Returns:	To Indicate:
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 pass-through operation was canceled.
CS_PENDING	Asynchronous network I/O is in effect. For more information, see the **Asynchronous Programming** topics page.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-89: Return values (***ct_recvpassthru***)*

Comments

- 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 via **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 pass-through 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

Function

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:*	**ct_remote_pwd:**
CS_SET	Sets the remote password
CS_CLEAR	Clears all remote passwords specified for this connection by setting them to NULL.

Table 3-90: Values for action (**ct_remote_pwd**)

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 for it.

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 *pnamelen* 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.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-91: Return values (***ct_remote_pwd***)*

Comments

- **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 call a stored procedure located on another server. To accomplish this server-to-server communication, the first server to which an application has connected via **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 into. Each password must be defined with a separate call to **ct_remote_pwd**.
- If an application does not specify a remote password for a particular server, the password defaults to the password set for this connection via **ct_con_props**, if any. If no password has been defined, the password defaults to NULL. If an application user generally has the same password on different servers, this default behavior may be acceptable.
- 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

Function

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. The table in the **Summary of Parameters** section lists the symbolic values that are legal 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.

Summary of Parameters

Value of *type:*	ct_res_info returns:	The information is available after ct_results sets its **result_type* parameter to:	**buffer* is set to:
CS_BROWSE_INFO	CS_TRUE if browse-mode information is available; CS_FALSE if browse-mode information is not available.	CS_ROW_RESULT	CS_TRUE or CS_FALSE.
CS_CMD_NUMBER	The number of the command that generated the current result set.	Any value.	An integer value.
CS_MSGTYPE	An integer representing the id of the message that makes up the current result set.	CS_MSG_RESULT	A small integer.
CS_NUM_COMPUTES	The number of compute clauses in the current command.	CS_COMPUTE_RESULT	An integer value.
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	An integer value.
CS_NUMORDER COLS	The number of columns specified in the order-by clause of the current command.	CS_ROW_RESULT	An integer value.
CS_ORDERBY_COLS	The select list id numbers of columns specified in a the **order by** clause of the current command.	CS_ROW_RESULT	An array of integers.
CS_ROW_COUNT	The number of rows affected by the current command.	CS_CMD_DONE, CS_CMD_FAIL CS_CMD_SUCCEED	An integer value.
CS_TRANS_STATE	The current server transaction state.	Any value.	A symbolic value.

*Table 3-92: Summary of parameters (***ct_res_info***)*

Returns

Returns:	To Indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed. **ct_res_info** returns CS_FAIL if the requested information is larger than *buflen* bytes, or if there is no current result set.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-93: Return values (***ct_res_info***)*

Comments

- **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 on result sets, see the **ct_results** manual page and the **Results** topics page.
- 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 on browse mode, see the **Browse Mode** topics page in Chapter 2 of this manual.

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 this, it 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 against 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 against 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 1, 2, or 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** are ordered on presentation.

Retrieving the Column ID's of Order-By Columns

- To get the select list column id's 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 id's 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_INTs containing the integers 2 and 1.

Retrieving the Number of Rows for the Current Command

- To determine the number of rows affected 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.
- 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.
- For more information about server transaction states, see "Server Transaction States" on page 2-81.

Example

```
...CODE DELETED.....

CS_INT   num_cols;

/*
** Determine the number of columns in this result
** set.
*/
retcode = ct_res_info(cmd, CS_NUMDATA, &num_cols,
    CS_UNUSED, NULL);
if (retcode != CS_SUCCEED)
{
    ...CODE DELETED.....
}

...CODE DELETED.....

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;

...CODE DELETED.....
```

This code excerpt is from the *rpc.c* example program. For further examples of using **ct_res_info**, see the *compute.c*, *ex_alib.c*, *exutils.c*, and *i18n.c* example programs.

See Also

ct_cmd_props, ct_con_props, ct_results, Options

ct_results

Function

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*:

	Value of **result_type:*	What it indicates:	Result set contains:
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.

Table 3-94: Values for *result_type (**ct_results**)

	Value of ***result_type:***	**What it indicates:**	**Result set contains:**
Values that indicate fetchable results	CS_COMPUTE_RESULT	Compute row results.	A single row of compute results.
	CS_CURSOR_RESULT	Cursor row results.	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.
Values that indicate information is available.	CS_COMPUTEFMT_ RESULT	Compute format information.	No fetchable results. An application can call **ct_describe**, **ct_res_info**, 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.	No fetchable results. An application can call **ct_describe** or **ct_dyndesc** to retrieve the information.

Table 3-94: Values for *result_type *(***ct_results***) (continued)*

Returns

Returns:	**To Indicate:**
CS_SUCCEED	A result set is available for processing.
CS_END_RESULTS	All results have been completely processed.

*Table 3-95: Return values (***ct_results***)*

Returns:	To Indicate:
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 **cs_close**(CS_FORCE_CLOSE) to force the connection closed.
CS_CANCELED	Results have been canceled.
CS_PENDING	Asynchronous network I/O is in effect. For more information, see the **Asynchronous Programming** topics page.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-95: Return values (**ct_results**) (continued)*

Comments

- An application calls **ct_results** after sending a command to the server via **ct_send**, and before reading the results of that command (if any) via **ct_fetch**.
- "Result data" is an umbrella term for all the types of data that a server can return to an application. These 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 the **Error and Message Handling** topics page 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 a result set that spans multiple buffers, an application must call **ct_results** as long as it continues to return CS_SUCCEED, indicating that results are available. The simplest way to do this is in a loop that terminates when **ct_results** fails to return CS_SUCCEED. After the loop, an application can use a case-type statement to test **ct_results**’ final return code to determine why the loop terminated.
- 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, it is 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.

When are the Results of a Command Completely Processed?

- **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 via **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.

• 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
```

When calling **ct_results** to process the results of this language command, an application would see the following **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.
```

• A **result_type* of CS_CMD_SUCCEED or CS_CMD_FAIL is immediately followed by a **result_type* of CS_CMD_DONE.

• 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 on this exception, see "Connection and Command Rules" in Chapter 3, "Structures, Datatypes, Constants, and Conventions," of the *Open Client Client-Library Programmer's Guide*.

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.

Special Kinds of Result Sets

- **A message result set** contains no actual result data. Rather, a message has an "id". An application can call **ct_res_info** to get a message's 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 SQL 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_describe**, **ct_res_info**, and **ct_compute_info**.

  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 information by calling **ct_describe** and **ct_dyndesc**.

ct_results *and Stored Procedures*

- A run-time 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 run-time 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 run-time 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.

Example

```
/*
** 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 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;
}
```

This code excerpt is from the *compute.c* example program. For further examples of using **ct_results**, see the *csr_disp.c*, *ex_alib.c*, *exutils.c*, *getsend.c*, *i18n.c*, and *rpc.c* example programs.

See Also

ct_bind, ct_command, ct_cursor, ct_describe, ct_dynamic, ct_fetch, ct_send

ct_send

Function

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.

Returns:

Returns:	To Indicate:
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 **cs_close**(CS_FORCE_CLOSE) to force the connection closed.
CS_CANCELED	The routine was canceled.
CS_PENDING	Asynchronous network I / O is in effect. For more information, see the **Asynchronous Programming** topics page.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-96: Return values (**ct_send**)*

Comments

- Sending a command to a server is a four step process. To send a command to a server, an application must:

- Initiate the command by calling **ct_command**, **ct_cursor**, or **ct_dynamic**. These routines set up internal structures that are used in building a command stream to send to the server.
- Pass parameters for the command (if required) by calling **ct_param** 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.
- Send the command to the server by calling **ct_send**.
- Verify the success of the command by calling **ct_results**.

  This last step does not imply that an application need only call **ct_results** once. An application needs to continue calling **ct_results** until it no longer returns CS_SUCCEED. See the *Open Client Client-Library/C Programmer's Guide* 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.

  Once an application has sent a command, it must call **ct_results** before calling **ct_cancel** to cancel the command.

• **ct_send** uses an asynchronous write and not does 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.

Example

```
/*
** 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* example program. For further examples of using **ct_send**, see the *compute.c*, *csr_disp.c*, *ex_alib.c*, *getsend.c*, *i18n.c,* and *rpc.c* example programs.

See Also

ct_command, ct_cursor, ct_dynamic, ct_fetch, ct_param, ct_results

ct_send_data

Function

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*.

Returns

Returns:	To Indicate:
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. For more information, see the **Asynchronous Programming** topics page.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-97: Return values (***ct_send_data***)*

Comments

- An application can use **ct_send_data** to write a text or image value to a database column. 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 2-190.

- In order 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.
- For more information on writing a text or image value, see"Text and Image" on page 2-188.

Example

```
/*
** 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);
/*
** NOTE: The following is not needed...
**
strcpy(textdata->iodesc.name, "getsend_tbl.t");
textdata->iodesc.namelen = CS_NULLTERM;
*/
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* example program.

See Also

ct_data_info, ct_get_data, Text and Image

ct_sendpassthru

Function

Send a TDS (Tabular Data Stream) 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.

Returns

Returns:	To Indicate:
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. For more information, see the **Asynchronous Programming** topics page.
CS_BUSY	An asynchronous operation is already pending for this connection. For more information, see the **Asynchronous Programming** topics page.

*Table 3-98: Return values (***ct_sendpassthru***)*

Comments

- 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_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 via **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 pass-through 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

Function

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.

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-99: Return values (***ct_setloginfo***)*

Comments

- TDS (Tabular Data Stream) is a communications protocol used for the transfer of requests and request results between clients and servers.
- Because **ct_setloginfo** frees the CS_LOGINFO structure after transferring the TDS information, an application cannot re-use the CS_LOGINFO. 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 pass-through.
- In order 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 re-entrant.

TDS Pass-Through

- 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 pass-through, 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 only be made in an Open Server SRV_CONNECT event handler, are:

1. **srv_getloginfo** to allocate a CS_LOGINFO structure and fill it with TDS information from a client login request.
2. **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.
3. **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.
4. **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 the **Properties** topics page.

See Also

ct_getloginfo, ct_recvpassthru, ct_sendpassthru

ct_wakeup

Function

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. The following table lists the symbolic values that are legal for *function*:

Value of *function*:	**To indicate:**
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_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.

Table 3-100: Values for function *(***ct_wakeup***)*

status – The return status of the completed routine. This value is passed as the *status* parameter to the completion callback

Returns

Returns:	To Indicate:
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 the **Asynchronous Programming** topics page.

*Table 3-101: Return values (**ct_wakeup**)*

Comments

- **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 the callbacks topics page for information on completion callbacks.
- See the **Asynchronous Programming** topics page for more information on using **ct_wakeup** in asynchronous Client-Library applications.

Example

```
...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* example program.

See Also

ct_callback, ct_poll

Appendixes

A Glossary

array

A structure composed of multiple identical variables which 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.

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 SQL Server by means of a single Client-Library command or Embedded SQL statement.

browse mode

Browse mode is 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 utility for copying data in and out of databases. Also called **bcp**.

callback

A routine that Open Client or Open Server calls in response to a triggering event, known as a callback event.

callback event

In Open Client and Open Server, a callback event is an occurrence that triggers a callback routine.

capabilities

A client / server connection's capabilities determine the types of client requests and server responses permitted for that connection.

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.

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 different character sets. For example, if SQL 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-Library

Part of Open Client, Client-Library is a collection of routines for use in writing client applications. Client-Library is a new library, designed to accommodate cursors and other advanced features in the SYBASE 10.0 product line.

code set

See **character set.**

collating sequence

See **sort order**.

command

In Client-Library, a command is 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 command structure (CS_COMMAND) is a hidden Client-Library structure that Client-Library applications use to send commands and process results.

connection structure

A connection structure (CS_CONNECTION) is a hidden Client-Library structure that defines a client/server connection within a context.

context structure

A context structure (CS_CONTEXT) is 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

See **character set conversion**.

CS-Library

Included with both the Open Client and Open Server products, CS-Library is a collection of utility routines that are useful to both Client-Library and Server-Library applications.

current row

With respect to cursors, the current row is the row to which a cursor points. A fetch against a cursor retrieves the current row.

cursor

A cursor is a symbolic name that is associated with a SQL statement.

In Embedded SQL, a cursor is a data selector that passes multiple rows of data to the host program, one row at a time.

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 collection of routines for use in writing client applications.

deadlock

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. SQL Server detects deadlocks and resolves them by killing one user's process.

default

Describes the value, option, or behavior that Open Client / Server products use when none is explicitly specified.

default database

The database that a user gets by default when he or she logs in to a database server.

default language

1. The language that Open Client / Server products use when an application does no explicit localization. The default language is determined by the "default" entry in the locales file.

2. The language that SQL Server uses for messages and prompts when a user has not explicitly chosen a language.

descriptor area

The area that a DBMS uses to store information about dynamic parameters in a dynamic SQL statement.

dynamic SQL

Dynamic SQL allows an Embedded SQL or Client-Library application to execute SQL statements containing variables whose values are determined at run-time.

error message

A message that an Open Client / Server product issues when it detects an error condition.

event

An occurrence that prompts an Open Server application to take certain actions. Client commands and certain commands within Open Server application code can trigger events. When an event occurs, Open Server calls either the appropriate event-handling routine in the application code or the appropriate default event handler.

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.

exposed structure

An exposed structure is a structure whose internals are exposed to Open Client / Server programmers. Open Client / Server programmers can declare, manipulate, and de-allocate exposed structures directly. The CS_DATAFMT structure is an example of an exposed structure.

extended transaction

In Embedded SQL, an extended transaction is a transaction composed of multiple Embedded SQL statements.

FIPS

FIPS is an acronym for Federal Information Processing Standards. If FIPS flagging is enabled, SQL Server or the Embedded SQL precompiler issue warnings when a non-standard extension to a SQL statement is encountered.

gateway

A gateway is 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.

hidden structure

A hidden structure is a structure whose internals are hidden from Open Client / Server programmers. Open Client / Server programmers must use Open Client / Server routines to allocate, manipulate, and de-allocate 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 host program is the application program that contains the Embedded SQL code.

host variable

In Embedded SQL, a variable which enables data transfer between SQL Server and the application program. See also **indicator variable**, **input variable**, **output variable**, **result variable**, and **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.

input variable

A variable that is used to pass information to a routine, a stored procedure, or SQL 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. Note that not all platforms use the interfaces file. On these platforms, an alternate mechanism directs clients to server addresses.

isql script file

In Embedded SQL, an isql script file is one of the three files the precompiler can generate. An isql script file contains precompiler-generated stored procedures, which 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.

keyword

A word or phrase that is reserved for exclusive use in Transact-SQL or Embedded SQL. Also called a **reserved word**.

listing file

In Embedded SQL, a listing file is 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

The national environment in which a program is running. The locale determines the language, sort order, and date / time formatting conventions.

locales file

A file that maps locale names to language / character set pairs. Open Client / Server products search the locales file when loading localization information.

locale name

A character string that represents a language / character set pair. Locale names are listed in the **locales file**. SYBASE predefines some locale names, but a system administrator can define additional locale names and add them to the locales file.

locale structure

A locale structure (CS_LOCALE) is 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 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.

localization

Localization is the process of setting up an application to run in a particular national language environment. An application that is localized typically generates messages in a local language and character set and uses local datetime formats.

login name

The name a user uses to log in to a server. A SQL Server login name is valid if SQL 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.

multi-byte character set

A character set that includes characters encoded using more than one byte. EUC JIS and Shift-JIS are examples of multi-byte character sets.

mutex

A mutual exclusion semaphore. This is a logical object that an Open Server application uses to ensure exclusive access to a shared object.

null

Having no explicitly assigned value. NULL is not equivalent to zero, or to blank. A value of NULL is not considered to be greater than, less than, or equivalent to any other value, including another value of NULL.

Open Server

A SYBASE product than provides tools and interfaces for creating custom servers.

Open Server application

A custom server constructed with Open Server.

output variable

In Embedded SQL, a variable which 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.

pass-through mode

A state of being pertaining to gateway applications.

When in pass-through mode, a gateway relays Tabular Data Stream (TDS) packets between a client and a remote data source without unpacking the packets' contents.

property

A property is a named value stored in a structure. Context, connection, thread, and command structures have properties. A structure's properties determine how it behaves.

query

1. A data retrieval request; usually a **select** statement.

2. Any SQL statement that manipulates data.

registered procedure

In Open Server, a collection of C statements stored under a name. Open Server-supplied registered procedures are called **system registered procedures**.

remote procedure call

1. One of two ways in which a client application can execute a SQL 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 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. A **stored procedure** executed on a different server from the server to which the user is connected.

result variable

In Embedded SQL, a variable which receives the results of a **select** or **fetch** statement.

server

In client / server systems, the server is the part of the system that processes client requests and returns results to clients.

Server-Library

A collection of routines for use in writing Open Server applications.

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 is a structure that provides a communication path between SQL Server and the application program. After executing each SQL statement, SQL Server stores return codes in the SQLCA.

2. In a Client-Library application, a SQLCA is a structure that the application can use to retrieve Client-Library and server error and informational messages.

sqlcode

1. In an Embedded SQL application, a SQLCODE is a structure that provides a communication path between SQL Server and the application program. After executing each SQL statement, SQL Server 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 is a structure that the application can use to retrieve Client-Library and server error and informational message codes.

SQL Server

A server in Sybase's client / server architecture. SQL 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.

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 which receives the return status value of a stored procedure, thereby indicating the procedure's success or failure.

stored procedure

In SQL Server, a collection of SQL statements and optional control-of-flow statements stored under a name. SQL Server-supplied stored procedures are called **system procedures**.

System Administrator

The user in charge of server system administration, including creating user accounts, assigning permissions, and creating new databases. On SQL Server, the System Administrator's login name is *sa*.

system descriptor

In Embedded SQL, a system descriptor is an area of memory that holds a description of variables used in Dynamic SQL statements.

system procedures

Stored procedures that SQL 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.

system registered procedures

Internal registered procedures that Open Server supplies for registered procedure notification and status monitoring.

target file

In Embedded SQL, a target file is 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 transfers requests and results between clients and servers.

thread

A path of execution through Open Server application and library code and the path's associated stack space, state information, and event handlers.

Transact-SQL

Transact-SQL is an enhanced version of the database language SQL. Applications can use Transact-SQL to communicate with SYBASE SQL 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

Transaction mode refers to the manner in which SQL Server manages transactions. SQL Server supports two transaction modes: Transact-SQL mode (also called "unchained transactions") and ANSI mode (also called "chained transactions").

user name

See **login name**.

Index

Index