SYBASE®

Client-Library/C Programmers Guide

Open Client™

15.5

DOCUMENT ID: DC35570-01-1550-01

LAST REVISED: October 2009

Contents

About This Book

This book contains information on how to write C applications using Open Client™ Client-Library.

Audience

This book is written for application programmers familiar with the C programming language.

How to use this book

This book contains these chapters:

- Chapter 1, “Getting Started with Client-Library” explains how to structure a basic Client-Library program and includes a simple, complete Client-Library application.
- Chapter 2, “Understanding Structures, Constants, and Conventions” contains information about Client-Library structures, constants, and parameter conventions.
- Chapter 3, “Using Open Client and Server Datatypes” contains a summary of datatypes that can be used in a Client-Library application.
- Chapter 4, “Handling Errors and Messages” explains how to handle Client-Library and server errors in your application.
- Chapter 5, “Choosing Command Types” explains when and how to use the different command types in your application.
- Chapter 6, “Writing Results-Handling Code” explains Client-Library’s results processing model.
- Chapter 7, “Using Client-Library Cursors” explains how to declare and manipulate Client-Library cursors.
- Chapter 8, “Using Dynamic SQL Commands” explains how to use dynamic SQL queries in your applications.
- Chapter 9, “Using Directory Services” contains information on how to use Client-Library directory services.
- Appendix A, “Logical Sequence of Calls” contains diagrams of the legal call sequences in Client-Library applications.

Related documents

You can see these books for more information:

- The *Open Server Release Bulletin for Microsoft Windows* contains important last-minute information about Open Server™.
- The *Software Developer's Kit Release Bulletin for Microsoft Windows* contains important last-minute information about Open Client and SDK.
- The *jConnect for JDBC Release Bulletin* versions 6.05 and 7.0 contains important last-minute information about jConnect™.
- The *Open Client and Open Server Configuration Guide for Microsoft Windows* contains information about configuring your system to run Open Client and Open Server.
- The *Open Client Client-Library/C Reference Manual* contains reference information for Open Client Client-Library.
- The *Open Server Server-Library/C Reference Manual* contains reference information for Open Server Server-Library.
- The *Open Client and Open Server Common Libraries Reference Manual* contains reference information for CS-Library, which is a collection of utility routines that are useful in both Client-Library and Server-Library applications.
- The *Open Client and Open Server Programmers Supplement for Microsoft Windows* contains platform-specific information for programmers using Open Client and Open Server. This document includes information about:
  - Compiling and linking an application
  - The sample programs that are included with Open Client and Open Server
  - Routines that have platform-specific behaviors
- The *jConnect for JDBC Installation Guide* version 6.05 contains installation instructions for jConnect for JDBC™.
- The *jConnect for JDBC Programmers Reference* describes the jConnect for JDBC product and explains how to access data stored in relational database management systems.
- The *Adaptive Server Enterprise ADO.NET Data Provider Users Guide* provides information on how to access data in Adaptive Server® using any language supported by .NET, such as C#, Visual Basic .NET, C++ with managed extension, and J#.

- The *Adaptive Server Enterprise ODBC Driver by Sybase Users Guide* for Windows and Linux, provides information on how to access data from Adaptive Server on Microsoft Windows, Linux, and Apple Mac OS X platforms, using the Open Database Connectivity (ODBC) Driver.
- The *Adaptive Server Enterprise OLE DB Provider by Sybase Users Guide for Microsoft Windows* provides information on how to access data from Adaptive Server on Microsoft Windows platforms, using the Adaptive Server OLE DB Provider.

Other sources of information

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

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

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

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

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

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

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

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

2. Click Partner Certification Report.
3. In the Partner Certification Report filter select a product, platform, and timeframe and then click Go.
4. Click a Partner Certification Report title to display the report.

❖ Finding the latest information on component certifications

1. Point your Web browser to Availability and Certification Reports at http://certification.sybase.com/.
2. Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
3. Select Search to display the availability and certification report for the selection.

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

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

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ Finding the latest information on EBFs and software maintenance

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.
4. Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

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

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

Conventions

Table 1: Syntax conventions

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

Accessibility features

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

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

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

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

If you need help

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

CHAPTER 1

Getting Started with Client-Library

This chapter includes the fundamental concepts required to develop Client-Library/C applications.

Client-Library overview

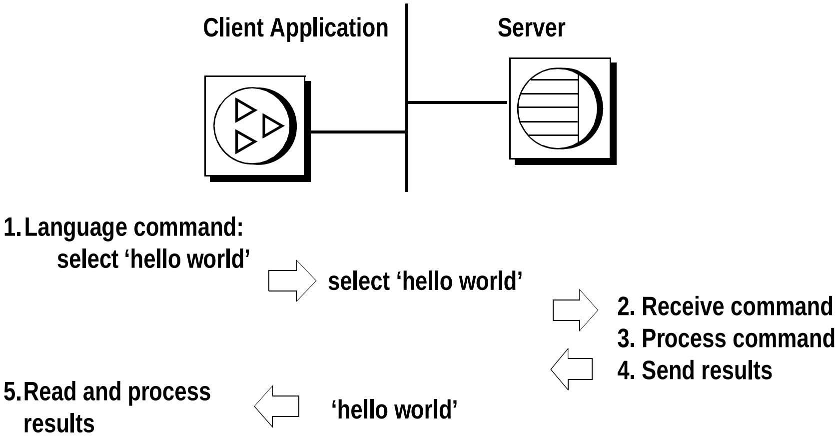
Client-Library is a collection of routines for sending commands to and retrieving results from Sybase servers.

For an overview of Sybase's client/server architecture and products, see Chapter 1, "Introducing Client-Library," in the *Open Client Client-Library/C Reference Manual*.

Types of Client-Library applications

Client-Library applications vary mainly in the types of commands that they send. Once connected to a server, all client applications use the "send commands, process results" paradigm illustrated in Figure 1-1:

Figure 1-1: The commands/results paradigm



Adaptive Server Enterprise client applications

The following examples illustrate what kinds of tasks an Adaptive Server Enterprise client application might carry out:

- SQL interpreter – the client application prompts the user for queries, sends these queries to the server as language commands, retrieves the results from the Adaptive Server Enterprise, and displays the results. The Sybase isql utility is such an application; it calls the following Client-Library routines:
    - ct_command(CS_LANG_CMD) to define a language command and its text
    - ct_send to send it to the server
    - ct_results to read the results
    - ct_res_info and ct_describe to find out column formats
    - ct_bind and ct_fetch to retrieve rows

  See “Language commands” on page 76. See also the example application shown in “A simple sample program” on page 4.

- Data-entry – an application that always runs the same queries. The application uses Adaptive Server Enterprise stored procedures to implement application logic for performing inserts, updates, and menu population. The client program invokes the stored procedures by sending RPC commands. Such an application calls:
  - ct_command(CS_RPC_CMD) to define an RPC command
  - ct_param or ct_setparam to define parameter values with which to call the procedure
  - ct_send to send the command to the server
  - ct_results, ct_bind, ct_fetch, and so forth, to read the results

  See “RPC commands” on page 78.
- Interactive query-by-example – an application that prompts for queries that can contain markers, indicated by a question mark (?), for values to be supplied at runtime. The application uses dynamic SQL commands to:
  - Prepare the statement, by sending a ct_dynamic(CS_PREPARE) command and handling the results
  - Query for parameter formats, by sending a ct_dynamic(CS_DESCRIBE_INPUT) command and handling the results
  - After prompting for input values, execute the statement by sending a ct_dynamic(CS_EXECUTE) command and handling the results

  See Chapter 8, “Using Dynamic SQL Commands.”

Open Server client or gateway applications

Open Server Server-Library is a collection of routines that allows you to create custom server applications. Server-Library routines are documented in the *Open Server Server-Library/C Reference Manual*.

The following examples illustrate the tasks that an Open Server client application might carry out:

- Client for custom Open Server application – a client application sends RPC commands to invoke custom server routines that have been "registered" as callable server procedures in the Open Server application program. See the *Open Server Server-Library/C Reference Manual* for information on registered procedures. See "RPC commands" on page 78 for a description of how client applications send RPC commands.
- Notification client – Open Server provides a feature called "registered procedure notification" that allows client applications to watch for invocations of selected registered procedures. For example, a client application that caches copies of important data might watch for a notification on a registered procedure that updates the data. The notification indicates when the cached copy must be refreshed. See the "Registered Procedures" topics page in the *Open Client Client-Library/C Reference Manual*.
- Gateway application – a server application acts as an intermediary between its own clients and other servers. The gateway accepts client commands, forwards them to a remote server, reads the results, and forwards the results to its own client. If the remote server is a Sybase server, the gateway makes Client-Library calls to communicate with the remote server.

A simple sample program

This section walks you through an sample program that connects to a server, sends a query, processes the results, then exits. Most Client-Library applications exhibit a program structure similar to this.

Building programs

The *Open Client and Open Server Programmers Supplement for Microsoft Windows* and *Open Client and Open Server Programmers Supplement for UNIX* describe how to build a Client-Library application on your platform and includes information about required compile/link options, library file names, and runtime requirements.

Steps in the example

The following steps show a simple Client-Library application:

1. Set up the Client-Library programming environment:
   a. Use cs_ctx_alloc to allocate a context structure.
   b. Use cs_config to set any CS-Library properties for the context.
   c. Use ct_init to initialize Client-Library.
   d. Use ct_config to set Client-Library properties for the context.
2. Define error handling. Most applications use callback routines to handle errors:
   a. Use cs_config(CS_MESSAGE_CB) to install a CS-Library error callback.
   b. Use ct_callback to install a client message callback.
   c. Use ct_callback to install a server message callback.

   Warning! Applications that do not define error handling do not receive notification of errors that occur in the program, on the network, or on the server. Code your applications to handle errors and server messages. Applications that do not perform error handling are difficult to debug and maintain.

3. Connect to a server:
   a. Use ct_con_alloc to allocate a connection structure.
   b. Use ct_con_props to set any properties in the connection structure
   c. Use ct_connect to open a connection to a server.
   d. Use ct_options to set any server options for this connection.
4. Send a language command to the server:
   a. Use ct_cmd_alloc to allocate a command structure.
   b. Use ct_command to initiate a language command.
   c. Use ct_send to send the command.
5. Process the results of the command:
   a. Use ct_results to set up results for processing (called in a loop).
   b. Use ct_res_info to get information about a result set.

   c Use ct_describe to get information about a result item.

   d Use ct_bind to bind a result item to program data space.

   e Use ct_fetch to fetch result rows (called in a loop).

6 Finish:

   a Use ct_cmd_drop to deallocate the command structure.

   b Use ct_close to close the connection with the server.

   c Use ct_exit to exit Client-Library.

   d Use cs_ctx_drop to deallocate the context structure.

Source listing

The following sample program, called *firstapp.c*, demonstrates the steps outlined in the previous section. Commentary for each step follows the example (beginning with "Step 1: Set up the Client-Library programming environment" on page 18).

The source code for this application is included with the Client-Library sample programs. See the Client-Library chapter in the *Open Client and Open Server Programmers Supplement for Microsoft Windows* or *Open Client and Open Server Programmers Supplement for UNIX* for information on making and running the sample programs.

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


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctpublic.h>
#include "example.h"

#define MAXCOLUMNS   2
#define MAXSTRING    40
#define ERR_CH stderr
#define OUT_CH stdout

/*
** Define a macro that exits if a function return code indicates
```

```
** failure.
*/
#define EXIT_ON_FAIL(context, ret, str) \
  if (ret != CS_SUCCEED) \
  { \
    fprintf(ERR_CH, "Fatal error: %s\n", str); \
    if (context != (CS_CONTEXT *) NULL) \
    { \
      (CS_VOID) ct_exit(context, CS_FORCE_EXIT); \
      (CS_VOID) cs_ctx_drop(context); \
    } \
    exit(-1); \
  }


/*
** Callback routines for library errors and server messages.
*/
CS_RETCODE CS_PUBLIC csmsg_callback PROTOTYPE((
        CS_CONTEXT     *context,
        CS_CLIENTMSG   *clientmsg ));
CS_RETCODE CS_PUBLIC clientmsg_callback PROTOTYPE((
        CS_CONTEXT     *context,
        CS_CONNECTION  *connection,
        CS_CLIENTMSG   *clientmsg ));
CS_RETCODE CS_PUBLIC servermsg_callback PROTOTYPE((
        CS_CONTEXT     *context,
        CS_CONNECTION  *connection,
        CS_CLIENTMSG   *servermsg ));


/*
** Main entry point for the program.
*/
int
main(int argc, char *argv[])
{
    CS_CONTEXT     *context;     /* Context structure     */
    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;
CS_RETCODE      results_ret;
CS_INT          result_type;
CS_CHAR         name[MAXSTRING];
CS_CHAR         city[MAXSTRING];

EX_SCREEN_INIT();

/*
** Step 1: Initialize the application.
*/
```

For more commentary, see “Step 1: Set up the Client-Library programming environment” on page 18.

```
/*
** First allocate a context structure.
*/
context = (CS_CONTEXT *) NULL;
ret = cs_ctx_alloc(EX_CTLIB_VERSION, &context);
EXIT_ON_FAIL(context, ret, "cs_ctx_alloc failed");

/*
** Initialize Client-Library.
*/
ret = ct_init(context, EX_CTLIB_VERSION);
EXIT_ON_FAIL(context, ret, "ct_init failed");


/*
** Step 2: Set up the error handling. Install callback handlers
** for: - CS-Library errors - Client-Library errors - Server
** messages
*/
```

For more commentary, see “Step 2: Define error handling” on page 20.

```
/*
** Install a callback function to handle CS-Library errors
*/
ret = cs_config(context, CS_SET, CS_MESSAGE_CB,
                (CS_VOID *)csmsg_callback,
                CS_UNUSED, NULL);
EXIT_ON_FAIL(context, ret,
            "cs_config(CS_MESSAGE_CB) failed");

/*
```

```
** Install a callback function to handle Client-Library errors
**
** The client message callback receives error or informational
** messages discovered by Client-Library.
*/
ret = ct_callback(context, NULL, CS_SET, CS_CLIENTMSG_CB,
                  (CS_VOID *) clientmsg_callback);
EXIT_ON_FAIL(context,ret,
            "ct_callback for client messages failed");

/*
** The server message callback receives server messages sent by
** the server. These are error or informational messages.
*/
ret = ct_callback(context, NULL, CS_SET, CS_SERVERMSG_CB,
                  (CS_VOID *) servermsg_callback);
EXIT_ON_FAIL(context, ret,
            "ct_callback for server messages failed");


/*
** Step 3: Connect to the server. We must: - Allocate a connection
** structure. - Set user name and password. - Create the
** connection.
*/
```

For more commentary, see “Step 3: Connect to a server” on page 22.

```
/*
** First, allocate a connection structure.
*/
ret = ct_con_alloc(context, &connection);
EXIT_ON_FAIL(context, ret, "ct_con_alloc() failed");

/*
** These two calls set the user credentials (username and
** password) for opening the connection.
*/
ret = ct_con_props(connection, CS_SET, CS_USERNAME,
           Ex_username, CS_NULLTERM, NULL);
EXIT_ON_FAIL(context, ret, "Could not set user name");
ret = ct_con_props(connection, CS_SET, CS_PASSWORD,
           Ex_password, CS_NULLTERM, NULL);
EXIT_ON_FAIL(context, ret, "Could not set password");

/*
** Create the connection.
*/
```

```
if(EX_SERVER==NULL)
    ret = ct_connect(connection, (CS_CHAR *) NULL, 0);
else
    ret = ct_connect(connection, (CS_CHAR *)EX_SERVER, strlen(EX_SERVER));
EXIT_ON_FAIL(context, ret, "Could not connect!");

/*
** Step 4:  Send a command to the server, as follows: - Allocate a
** CS_COMMAND structure - Build a command to be sent with
** ct_command. - Send the command with ct_send.
*/
```

For more commentary, see “Step 4: Send commands to the server” on page 24.

```
/*
** Allocate a command structure.
*/
ret = ct_cmd_alloc(connection, &cmd);
EXIT_ON_FAIL(context, ret, "ct_cmd_alloc() failed");


/*
** Initiate a language command. This call associates a query with
** the command structure.
*/
ret = ct_command(cmd, CS_LANG_CMD,
         "select au_lname, city from pubs2..authors \
         where state = 'CA'",
         CS_NULLTERM, CS_UNUSED);
EXIT_ON_FAIL(context, ret, "ct_command() failed");

/*
** Send the command.
*/
ret = ct_send(cmd);
EXIT_ON_FAIL(context, ret, "ct_send() failed");

/*
** Step 5: Process the results of the command.
*/
```

For more commentary, see “Step 5: Process the results of the command” on page 25.

```
while ((results_ret = ct_results(cmd, &result_type))
       == CS_SUCCEED)
{

    /*
```

```
** ct_results sets result_type to indicate when data is
** available and to indicate command status codes.
*/
switch ((int)result_type)
{
case CS_ROW_RESULT:

   /*
    ** This result_type value indicates that the rows
    ** returned by the query have arrived. We bind and
    ** fetch the rows.
    **
    ** We're expecting exactly two character columns:
    ** Column 1 is au_lname, 2 is au_city.
    **
    ** For each column, fill in the relevant fields in
    ** the column's 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;
    ret = ct_bind(cmd, 1, &columns[0],
                 name, &datalength[0],
                 &indicator[0]);
    EXIT_ON_FAIL(context, ret,
      "ct_bind() for au_lname failed");

    /*
    ** Same thing for the 'city' column.
    */
    columns[1].datatype = CS_CHAR_TYPE;
    columns[1].format = CS_FMT_NULLTERM;
    columns[1].maxlength = MAXSTRING;
    columns[1].count = 1;
    columns[1].locale = NULL;

    ret = ct_bind(cmd, 2, &columns[1], city,
                 &datalength[1],
                 &indicator[1]);
    EXIT_ON_FAIL(context, ret,
      "ct_bind() for city failed");

    /*
    ** 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(ERR_CH,
                            "Error on row %ld.\n",
                            (long)(count+1));
                }
                /*
                ** We have a row, let's print it.
                */
                fprintf(OUT_CH, "%s: %s\n", name, city);
            }

            /*
            ** We're finished processing rows, so check
            ** ct_fetch's final return value to see if an
            ** error occurred. The final return code should be
            ** CS_END_DATA.
            */
            if ( ret == CS_END_DATA )
            {
                fprintf(OUT_CH,
                        "\nAll done processing rows.\n");
            }
            else /* Failure occurred. */
            {
                EXIT_ON_FAIL(context, CS_FAIL,
                            "ct_fetch failed");
            }

            /*
            ** All done with this result set.
            */
            break;
      case CS_CMD_SUCCEED:

            /*
            ** We executed a command that never returns rows.
```

```
                */
                fprintf(OUT_CH, "No rows returned.\n");
                break;

            case CS_CMD_FAIL:

                /*
                ** The server encountered an error while
                ** processing our command. These errors will be
                ** displayed by the server-message callback that
                ** we installed earlier.
                */
                break;

            case CS_CMD_DONE:

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

            default:

                /*
                ** We got something unexpected.
                */
                EXIT_ON_FAIL(context, CS_FAIL,
                   "ct_results returned unexpected result type");
                break;
            }
        }

        /*
        ** We've finished processing results. Check the return value
        ** of ct_results() to see if everything went okay.
        */
        switch( (int) results_ret)
        {
        case CS_END_RESULTS:

                /*
                ** Everything went fine.
                */
                break;

            case CS_FAIL:
```

```
            /*
            ** Something terrible happened.
            */
            EXIT_ON_FAIL(context, CS_FAIL,
                        "ct_results() returned CS_FAIL.");
            break;

        default:

            /*
            ** We got an unexpected return value.
            */
            EXIT_ON_FAIL(context, CS_FAIL,
               "ct_results returned unexpected return code");
            break;
    }

    /*
    ** Step 6:  Clean up and exit.
    */
```

For more commentary, see “Step 6: Finish” on page 27.

```
    /*
    ** Drop the command structure.
    */
    ret = ct_cmd_drop(cmd);
    EXIT_ON_FAIL(context, ret, "ct_cmd_drop failed");

    /*
    ** Close the connection and drop its control structure.
    */
    ret = ct_close(connection, CS_UNUSED);
    EXIT_ON_FAIL(context, ret, "ct_close failed");
    ret = ct_con_drop(connection);
    EXIT_ON_FAIL(context, ret, "ct_con_drop failed");

    /*
    ** ct_exit tells Client-Library that we are done.
    */
    ret = ct_exit(context, CS_UNUSED);
    EXIT_ON_FAIL(context, ret, "ct_exit failed");

    /*
    ** Drop the context structure.
    */
    ret = cs_ctx_drop(context);
    EXIT_ON_FAIL(context, ret, "cs_ctx_drop failed");
```

```
    /*
    ** Normal exit to the operating system.
    */
    exit(0);
}

/*
** Handler for server messages. Client-Library will call this
** routine when it receives a message from the server.
*/
CS_RETCODE CS_PUBLIC
servermsg_callback(CS_CONTEXT *cp, CS_CONNECTION *chp, CS_SERVERMSG *msgp)
{

    /*
    ** Print the message info.
    */
    fprintf(ERR_CH,
        "Server message:\n\t");
    fprintf(ERR_CH,
        "number(%ld) severity(%ld) state(%ld) line(%ld)\n",
        (long) msgp->msgnumber, (long) msgp->severity,
        (long) msgp->state, (long) msgp->line);

    /*
    ** Print the server name if one was supplied.
    */
    if (msgp->svrnlen > 0)
        fprintf(ERR_CH, "\tServer name: %s\n", msgp->svrname);

    /*
    ** Print the procedure name if one was supplied.
    */
    if (msgp->proclen > 0)
        fprintf(ERR_CH, "\tProcedure name: %s\n", msgp->proc);

    /*
    ** Print the null terminated message.
    */
    fprintf(ERR_CH, "\t%s\n", msgp->text);

    /*
    ** Server message callbacks must return CS_SUCCEED.
    */
    return(CS_SUCCEED);
}

/*
```

```
**  Client-Library error handler. This function will be invoked
**  when a Client-Library has detected an error. Before Client-
**  Library routines return CS_FAIL, this handler will be called
**  with additional error information.
*/
CS_RETCODE CS_PUBLIC
clientmsg_callback(CS_CONTEXT *context, CS_CONNECTION *conn, CS_CLIENTMSG
*emsgp)
{

     /*
     ** Error number:
     ** Print the error's severity, number, origin, and
     ** layer. These four numbers uniquely identify the error.
     */
     fprintf(ERR_CH,
             "Client Library error:\n\t");
     fprintf(ERR_CH,
        "severity(%ld) number(%ld) origin(%ld) layer(%ld)\n",
        (long) CS_SEVERITY(emsgp->severity),
        (long) CS_NUMBER(emsgp->msgnumber),
        (long) CS_ORIGIN(emsgp->msgnumber),
        (long) CS_LAYER(emsgp->msgnumber));

     /*
     ** Error text:
     ** Print the error text.
     */
     fprintf(ERR_CH, "\t%s\n", emsgp->msgstring);


     /*
     ** Operating system error information: Some errors, such as
     ** network errors, may have an operating system error associated
     ** with them. If there was an operating system error, this code
     ** prints the error message text.
     */
     if (emsgp->osstringlen > 0)
     {
          fprintf(ERR_CH,
                  "Operating system error number(%ld):\n",
                  (long) emsgp->osnumber);
          fprintf(ERR_CH, "\t%s\n", emsgp->osstring);
     }

     /*
     ** If we return CS_FAIL, Client-Library marks the connection as
```

```
     ** dead. This means that it cannot be used anymore. If we return
     ** CS_SUCCEED, the connection remains alive if it was not already
     ** dead.
     */
     return (CS_SUCCEED);
}


/*
**  CS-Library error handler. This function will be invoked
**  when CS-Library has detected an error.
*/
CS_RETCODE CS_PUBLIC
csmsg_callback(CS_CONTEXT *context, CS_CLIENTMSG *emsgp)
{
 /*
  ** Print the error number and message.
  */
  fprintf(ERR_CH,
          "CS-Library error:\n");
  fprintf(ERR_CH,
           "\tseverity(%ld) layer(%ld) origin(%ld) number(%ld)",
           (long) CS_SEVERITY(emsgp->msgnumber),
           (long) CS_LAYER(emsgp->msgnumber),
           (long) CS_ORIGIN(emsgp->msgnumber),
           (long) CS_NUMBER(emsgp->msgnumber));
  fprintf(ERR_CH, "\t%s\n", emsgp->msgstring);

  /*
  ** Print any operating system error information.
  */
  if(emsgp->osstringlen > 0)
  {
    fprintf(ERR_CH, "Operating System Error: %s\n",
      emsgp->osstring);
  }
  return (CS_SUCCEED);
}
```

Step 1: Set up the Client-Library programming environment

A Client-Library programming environment is defined by:

- A CS_CONTEXT structure, which defines a programming context
- A Client-Library version level, which is indicated by an application's call to ct_init

Header files

All Client-Library/C applications require the header file *ctpublic.h*, which contains typedefs and declarations required by Client-Library routines.

Allocating a context structure

A Client-Library application calls the CS-Library routine cs_ctx_alloc to allocate a context structure. A Client-Library application must allocate a context structure before initializing Client-Library.

Note CS-Library routines start with the prefix "cs." Client-Library routines start with the prefix "ct". All Client-Library programs include at least two calls to CS-Library, because they must allocate and drop a context structure.

Setting CS-Library context properties

After allocating a context structure, a Client-Library application can call cs_config to set CS-Library properties for the context structure.

Context properties define aspects of an application's behavior at the context level. *firstapp.c* calls cs_config to set the CS_MESSAGE_CB property. This property defines a CS-Library message callback routine. An application needs to set this property if it will be handling CS-Library errors using the callback method. See Chapter 4, "Handling Errors and Messages."

You may need to code your application to set other CS-Library context properties as well. Besides CS_MESSAGE_CB, applications most commonly set the following properties with cs_config:

- CS_LOC_PROP – describes localization information for the context. An application must set this property if a context requires localization information that differs from the localization information that is available in the operating system environment. For example, if an application that is running in a German environment requires a French context, it can call cs_config to set the CS_LOC_PROP property.
- CS_EXTERNAL_CONFIG – specifies whether ct_init will read default application property settings from the OCS runtime configuration file. See "External configuration" on page 20.
- CS_APP_NAME – specifies a name for the application. If external configuration is enabled (CS_EXTERNAL_CONFIG is CS_TRUE), then the application name specifies a section of the configuration file from which to read settings. CS_APP_NAME is also inherited by allocated CS_CONNECTION structures.

See cs_config in the *Open Client and Open Server Common Libraries Reference Manual*.

Initializing Client-Library

To initialize Client-Library, an application calls ct_init, which sets up internal control structures and defines the version of Client-Library behavior that the application requires. ct_init must be the first Client-Library call in an application.

Most applications call ct_init only once; however, it is not an error for an application to call ct_init multiple times. Client-Library permits multiple ct_init calls because some applications cannot guarantee which of several modules will execute first. These types of applications need to call ct_init in each module.

ct_init takes as its parameter a symbol describing the version of Client-Library behavior that the application expects.

If Client-Library cannot provide this behavior, ct_init returns CS_FAIL.

Setting Client-Library context properties

firstapp.c calls ct_config to set the CS_MAX_CONNECT context property. This property specifies the maximum number of connections for a context.

Client-Library context properties serve one of two purposes:

- They define aspects of a context's behavior.

  CS_MAX_CONNECT is an example of this category.

- They define default properties for connections created from the context.

  The CS_NETIO property is an example of this category. If a context CS_NETIO property is set to CS_SYNC_IO, to indicate synchronous connections, then any connection structure allocated within the context will be synchronous. ct_con_props can be called to change the value of CS_NETIO for a specific connection after it has been allocated.

For a complete list of Client-Library context properties, see the "Properties" topics page in the *Open Client Client-Library/C Reference Manual*.

Applications that are not multithreaded can call ct_config to change a context's properties at any time during the program's execution. Multithreaded applications must set context properties in single-threaded, start-up code or limit all access to a context and its child connections to a single thread. See the "Multithreaded Programming" topics page in the *Open Client Client-Library/C Reference Manual*.

When an application calls ct_config to change a context property, property values for existing connections do not change, but connections allocated after the ct_config call will pick up the new property values.

External configuration

As an alternative to setting properties with hard-coded ct_config calls, Client-Library allows external configuration of property values for applications that have been configured to use this feature. See the topics page "Using the Runtime Configuration File" in the *Open Client Client-Library/C Reference Manual*.

Step 2: Define error handling

Errors can be handled inline or with callback functions. The sample program uses callback functions. See "Two methods for handling messages" on page 64 for information on the inline method.

ct_callback installs Client-Library callback routines. which are application routines that Client-Library calls automatically when a triggering event of the appropriate type occurs.

There are several types of callbacks, but the sample 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.

The client message callback is called automatically whenever Client-Library generates an error or informational message. For example, if the application passes an invalid parameter value, or calls routines out of sequence, then Client-Library generates an error and calls the client message callback with a description of the error.

The server message callback is called whenever the server sends an informational or error message during results processing. For example, if the application sends a language command that contains a syntax error or refers to a nonexistent table, then the server sends a message that describes the error.

The sample program also calls cs_config to install a CS-Library error handler. CS-Library calls the application's CS-Library error handler when an error occurs in a CS-Library call.

Other types of callbacks include:

- Completion callbacks, used by asynchronous connections to handle asynchronous operation completions
- Notification callbacks, used to handle registered procedure notifications received from an Open Server
- Signal callbacks, used by UNIX applications to handle non-Client-Library signals

See the ct_callback reference page and the "Callbacks" topics page in the *Open Client Client-Library/C Reference Manual*.

Note A CS-Library message callback is not installed in the same way as Client-Library message callbacks: An application installs a CS-Library message callback by calling cs_config rather than ct_callback. Once installed, both types of callbacks function similarly.

Step 3: Connect to a server

Connecting to a server is a three-step process. An application:

- Allocates a connection structure
- Sets properties for the connection, if necessary
- Logs in to a server

Allocating a connection structure

An application calls ct_con_alloc to allocate a connection structure.

Setting connection structure properties

An application calls ct_con_props to set, retrieve, or clear connection structure properties.

Connection properties define various aspects of a connection's behavior. For example:

- The CS_USERNAME property defines the user name that a connection will use when logging in to a server.
- The CS_APPNAME property specifies the application name that appears in the Adaptive Server Enterprise *sysprocess* table after the connection is opened.
- The CS_PACKETSIZE property defines the Tabular Data Stream™ (TDS) packet size, which determines the size of network packets that the application will send and receive over this connection. By default, Open Client Server (OCS) allows the server to choose a packetsize between 512 and 65535 bytes. Servers supporting Server Specified Packetsize like Adaptive Server Enterprise may choose a packetsize freely. The packetsize may also be smaller or larger than the packetsize specified with CS_PACKETSIZE.

When a connection structure is allocated, it picks up some default property values from its parent context. For example, if the CS_APPNAME property is set at the context level, all connection structures allocated from that context inherit the application name. Other properties that do not exist at the context level, such as CS_PACKETSIZE, default to standard Client-Library values.

For a complete list of connection properties, see the ct_con_props reference page in the *Open Client Client-Library/C Reference Manual*.

Required connection properties

At a minimum, an application must set the connection properties that specify the connection's user name (CS_USERNAME) and allow the server to authenticate the user's identity. Servers can confirm a user's identity in two ways:

- By requiring a valid password
- By using network-based user authentication

If the server requires a password, then the application must set the CS_PASSWORD property to the value of the user's server password.

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

Logging in to a server

An application calls ct_connect to connect to a server. In the process of establishing a connection, ct_connect sets up communication with the network, logs in to the server, and communicates any connection-specific property information to the server.

For example, if the server supports network-based user authentication and the client application requests it, then Client-Library and the server query the network's security system to see if the user (whose name is specified by CS_USERNAME) is logged in to the network. Applications must request network-based user authentication by setting the CS_SEC_NETWORKAUTH connection property.

Step 4: Send commands to the server

In Client-Library, a *command* is a request for action sent from the client application to the server. Each command belongs to a command type and may have input data associated with it. Client-Library bundles this information into a symbolic format and sends it over the network to the server, where it is executed.

firstapp.c sends a language command to the server. This command instructs the server to parse and execute the query that was defined as ct_command's *text* (third) parameter. For information on other command types, see Chapter 5, "Choosing Command Types."

An application defines and sends commands to a server by using a CS_COMMAND structure. To define and send a command, the application:

- Allocates a CS_COMMAND structure
- If necessary, sets properties for the command structure
- Initiates the command
- Defines any parameters required for the command
- Sends the command

Allocating a command structure

An application calls ct_cmd_alloc to allocate a command structure. Several command structures can be allocated from the same connection.

Setting command structure properties

An application calls ct_cmd_props to set, retrieve, or clear command structure properties.

Command-structure properties determine aspects of Client-Library behavior at the command-structure level. For example, the CS_HIDDEN_KEYS property determines whether or not Client-Library exposes any hidden keys that are returned as part of a result set.

firstapp.c sets no command-structure properties; instead, it uses the default command-level behavior. Command structures inherit default property values from their parent connection.

For a complete list of command-structure properties, see the ct_cmd_props reference page in the *Open Client Client-Library/C Reference Manual*.

Executing a command

An application calls ct_command, ct_cursor, or ct_dynamic to initiate a command. ct_send sends any type of command to the server.

firstapp.c calls ct_command to initiate a language command. ct_send sends the command text to the server, which parses, compiles, and executes it.

See Chapter 5, "Choosing Command Types."

Step 5: Process the results of the command

Applications call ct_results repeatedly to handle the results returned by the server. Almost all Client-Library programs process results by executing a loop controlled by ct_results return status. Inside the loop, a switch takes place on the current type of result. Different types of results require different types of processing.

The results-processing model used in the example is based on this pseudocode:

```
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 command done ....
             case command failed ....
             case other result type....
             ... raise an error ...
         end switch
 end while
```

```
check ct_results’ final return code
```

Note Sybase strongly recommends that you use this type of program structure, even in the case of a simple language command. In more complex programs, you cannot predict the number and type of result sets that an application will receive in response to a command. Code that calls ct_results in a loop is also easier to maintain, enhance, or reuse, since the results-handling logic is centralized.

---

ct_results sets up results for processing and sets the return parameter *result_type* to indicate the type of result data that is available for processing.

If the select statement sent by *firstapp.c* executes successfully on the server, the sample program receives result types of CS_ROW_RESULT and CS_CMD_DONE, in that order. If the statement does not execute successfully on the server, the program receives a result type of CS_CMD_FAIL.

Because this program is so simple, most result types are not included as cases in the *result_type* switch. However, the code does raise an error for unexpected values of *result_type*. Code this check into your program’s results loop—the error raised may help you trap coding bugs early in the development cycle.

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 and ct_describe to get a description of each column. However, in *firstapp.c*, these calls are not necessary because the example was coded with knowledge of how many columns were selected and their format.

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 sample program prints it.

ct_fetch is called until there are no more rows, then the sample program checks ct_fetch’s final return code to find out whether the loop terminated normally or because of failure.

For information on the other result types that an application can receive, see Chapter 6, “Writing Results-Handling Code.”

Step 6: Finish

Before exiting, a Client-Library application must:

1. Deallocate all command structures for each connection.
2. Close and deallocate all open connections.
3. Exit Client-Library.
4. Deallocate all context structures.

As noted in "Exiting Client-Library" on page 27, step 2 can be included with step 3.

Deallocating command structures

An application calls ct_cmd_drop to deallocate a command structure. It is an error to deallocate a command structure that has pending results or an open cursor.

Closing and deallocating connections

An application calls ct_close to close a connection and ct_con_drop to deallocate a closed connection. It is an error to deallocate a connection that has not been closed.

Exiting Client-Library

An application calls ct_exit to exit Client-Library for a specific context. ct_exit closes and deallocates any open connections and cleans up internal Client-Library data space. ct_exit must be the last Client-Library call for a context.

Because ct_exit closes and deallocates all open connections, it is not strictly necessary for an application to close and deallocate connections by calling ct_close and ct_con_drop; instead, the application can just call ct_exit.

Deallocating a context structure

The CS-Library routine cs_ctx_drop deallocates a context structure.

CHAPTER 2

Understanding Structures, Constants, and Conventions

This chapter contains information about Client-Library structures, constants, and conventions.

Hidden structures

Hidden structures are structures whose internals are not documented. For example, a Client-Library application needs to call CS-Library or Client-Library routines to allocate, inspect, modify, and deallocate hidden structures. The application cannot access the structure contents directly. Hidden structures include:

- CS_CONTEXT, which defines a Client-Library programming context.
- CS_CONNECTION, which defines an individual client/server connection.
- CS_COMMAND, which is used to send commands and process results.

- 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.
- CS_DS_OBJECT, which contains information about a directory entry.
- CS_BLKDESC, a control structure used by applications that call Bulk-Library routines. For information on Bulk-Library, see the *Open Client and Open Server Common Libraries Reference Manual*.
- CS_LOCALE, which is used to store localization information.

CS_CONTEXT

Before an application can initialize Client-Library, it must allocate a CS_CONTEXT, or context, structure.

A CS_CONTEXT structure stores configuration information that describes a particular *context*, or operating environment, for a set of server connections. CS_CONTEXT is shared by CS-Library, Client-Library, and Server-Library. A CS_CONTEXT structure is allocated and dropped using the CS-Library routines cs_ctx_alloc and cs_ctx_drop.

Although an application can use more than one context, a simple application typically requires only one.

Note An Open Client application that is running under CICS on an IBM host is restricted to one context per application.

Some context information is stored in the form of *properties*. Properties have values that an application can change to customize a context. Properties include CS_MAX_CONNECT, which defines the maximum number of connections allowed within the context, and CS_NETIO, which determines whether or a context's connections default to synchronous or asynchronous behavior.

Connection and command structures also have properties. When a connection is allocated, it picks up default property values from its parent context. When a command structure is allocated, it picks up default property values from its parent connection.

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

CS_CONNECTION

A CS_CONNECTION structure stores information about a particular client/server connection, including the user name and password for the connection, the packet size the connection will use, and whether the connection is synchronous or asynchronous.

As with a context, some connection information is stored in the form of properties. When a connection is created, it picks up some default property values from its parent context. Other properties (those that do not exist at the context level, such as CS_PACKETSIZE), default to standard Client-Library values.

Multiple connections to one or more servers can exist simultaneously within a single context.

CS_COMMAND

A CS_COMMAND, or command, structure is used to send commands to a server and to process the results of those commands.

A command structure is associated with a specific parent connection. Multiple command structures can exist simultaneously for a single connection.

Control structure hierarchy

CS_CONTEXT, CS_CONNECTION, and CS_COMMAND are the basic control structures to set up the Client-Library environment, connect to a server, send commands, and process results. All three of these structures are hidden.

Connection and command rules

The following rules apply to connection and command structures:

- Within a connection, the results of a command must be completely processed before another command can be sent.

  The exception to this rule is a ct_cursor (CS_CURSOR_OPEN) command, which generates a cursor result set. After ct_results returns CS_CURSOR_RESULT to indicate that cursor results are available:

    - The command structure that sent the cursor open command can be used to send a cursor update or cursor delete command related to the newly opened cursor.
    - Any other command structure within the connection can be used to send a command not related to the newly opened cursor.
- A separate command structure must be used for each Client-Library cursor. A Client-Library cursor is one that is declared through ct_cursor. See Chapter 7, "Using Client-Library Cursors."

CS_LOGINFO

A CS_LOGINFO, or login information, structure, is used internally to contain connection structure information, such as user name and password, that is used when logging in to a server.

Connection properties that reside in this structure are known as *login properties*.

The Client-Library routines ct_getloginfo and ct_setloginfo use a CS_LOGINFO structure. An application can use these routines to copy login properties from an open connection to a new connection structure.

CS_DS_OBJECT

A CS_DS_OBJECT, or directory object, structure, contains information about a directory entry. Client-Library and Server-Library use a directory to store the network address information required to create connections. Storage for the directory can be provided by the Sybase interfaces file or a network-based directory, such as the Windows Registry.

An application receives pointers to one or more CS_DS_OBJECT structures as the result of a directory search by the Client-Library routine ct_ds_lookup.

See Chapter 9, "Using Directory Services."

CS_BLKDESC

Bulk-library routines use a CS_BLKDESC, or bulk descriptor structure. The bulk descriptor is the control structure for bulk copy operations.

An application calls blk_alloc to allocate a CS_BLKDESC structure.

After completing a bulk copy operation, an application frees a CS_BLKDESC by calling blk_drop.

Bulk-Library routines are documented in the *Open Client and Open Server Common Libraries Reference Manual*.

CS_LOCALE

A CS_LOCALE, or locale structure, can be used to specify localization information at the context, connection, command structure, or data element levels.

A CS_LOCALE structure specifies:

- A language, character set, and collating sequence
- How to represent dates, times, numeric, and monetary values in character format

An application can call the CS-Library routines cs_loc_alloc, cs_locale, and cs_loc_drop to allocate, set values for, and drop a CS_LOCALE structure.

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

Exposed structures

Exposed structures are structures whose internals are documented. A Client-Library application must allocate any exposed structures it intends to use. Type definitions for the exposed structures are included in the header file *ctpublic.h*. In addition, Chapter 2, “Topics,” in the *Open Client Client-Library/C Reference Manual* contains a topics page for each exposed structure.

Exposed structures include:

- CS_BROWSEDESC – the browse descriptor structure
- CS_CLIENTMSG – the Client-Library message structure
- CS_DATAFMT – the data format structure
- CS_DATEREC – the datetime descriptor structure
- CS_IODESC – the I/O descriptor structure
- CS_PROP_SSL_LOCALID – the decryption structure
- CS_SERVERMSG – the server message structure
- SQLCA – the SQL communications area structure
- SQLCODE – the SQL code structure
- SQLSTATE – the SQL state structure

CS_BROWSEDESC

ct_br_column uses a CS_BROWSEDESC structure to return information about a browse mode column. Browse mode columns are returned by a Transact-SQL select ... for browse statement.

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

For a description of the fields in a CS_BROWSEDESC structure, see the "CS_BROWSEDESC Structure" topics page in the *Open Client Client-Library/C Reference Manual*.

CS_CLIENTMSG

Client-Library uses a CS_CLIENTMSG structure to describe a Client-Library error or informational message.

For a discussion of Client-Library message handling, see Chapter 4, "Handling Errors and Messages."

For a description of the fields in a CS_CLIENTMSG structure, see the "CS_CLIENTMSG Structure" topics page in the *Open Client Client-Library/C Reference Manual*.

CS_DATAFMT

Client-Library routines use the CS_DATAFMT structure to describe data values and program variables.

Some routines require a CS_DATAFMT structure as an input parameter. For example, ct_bind requires a data format structure describing the destination variable for a bind, and ct_param requires a data format structure describing the parameter being passed.

Other routines fill in CS_DATAFMT fields with a description of output data, which an application can then access directly. For example, ct_describe initializes a CS_DATAFMT structure with a description of a result data item.

Client-Library routines that use the CS_DATAFMT structure include ct_bind, ct_describe, and ct_param. CS-Library routines that use CS_DATAFMT include cs_convert and cs_set_convert.

For a description of the fields in a CS_DATAFMT structure, see the "CS_DATAFMT Structure" topics page in the *Open Client Client-Library/C Reference Manual*.

When a CS_DATAFMT structure is an input parameter to a routine, the routine ignores the contents of any fields in the structure that it does not use. For example, ct_bind ignores the contents of the *name*, *namelen*, *status*, and *usertype* fields.

The reference page for each routine that uses CS_DATAFMT contains a table listing the fields that are used and the values they can have.

CS_DATEREC

The CS_DATEREC structure is used with the CS-Library routine cs_dt_crack to interpret date and time data returned from the server. Date and time data is represented on the server by the date, time, datetime, datetime4, bigdatetime, and bigtime datatypes. These datatypes are packed structures. cs_dt_crack unpacks the date and time components into the CS_DATEREC fields.

For a description of the server datetime datatype and the equivalent Client-Library types, see "Datetime types" on page 56. For a description of the CS_DATEREC structure, see the cs_dt_crack reference page in the *Open Client and Open Server Common Libraries Reference Manual*.

CS_IODESC

Client-Library uses a CS_IODESC structure to describe text or image data.

For a discussion of how the CS_IODESC is used to process text and image values, see the “text and image Data Handling” topics page in the *Open Client Client-Library/C Reference Manual*.

For a description of the fields in a CS_IODESC structure, see the “CS_IODESC Structure” topics page in the *Open Client Client-Library/C Reference Manual*.

CS_PROP_SSL_LOCALID

Client-Library uses a CS_PROP_SSL_LOCALID structure to specify the path to the Local ID (certificates) file. The CS_PROP_SSL_LOCALID structure contains a file name and a password used to decrypt the information in the file.

For information about CS_PROP_SSL_LOCALID, see the *Open Client Client-Library/C Reference Manual*.

CS_SERVERMSG

Client-Library uses a CS_SERVERMSG structure to describe a server error or informational message.

For a discussion of Client-Library message handling, see Chapter 4, “Handling Errors and Messages.”

For a description of the fields in a CS_SERVERMSG structure, see the “CS_SERVERMSG Structure” topics page in the *Open Client Client-Library/C Reference Manual*.

SQLCA, SQLCODE, and SQLSTATE

When an application is handling error and informational messages inline, the Client-Library routine ct_diag can return message information in a SQLCA, SQLCODE, or SQLSTATE structure.

For a discussion of Client-Library message handling, see Chapter 4, “Handling Errors and Messages.”

For a description of the SQLCA, SQLCODE, and SQLSTATE structures, see the “SQLCA Structure,” “SQLCODE Structure,” and “SQLSTATE Structure” topics pages in the *Open Client Client-Library/C Reference Manual*.

SQLDA

Applications can use a SQLDA structure with the Client-Library routine ct_dynsqlda to pass parameters for server commands and handle the results from server commands.

For a description of the SQLDA structure and its use in applications, see the ct_dynsqlda reference page in the *Open Client Client-Library/C Reference Manual*.

Constants

Client-Library makes use of a wide variety of constants, including type constants, format constants, and other symbolic constants.

Constants related to a routine (for example, symbolic constants used as return values) are listed on the reference page for the routine in the *Open Client Client-Library/C Reference Manual*.

Type constants

Open Client and Open Server use type constants to describe the datatypes of program variables. For example, when calling ct_bind to describe a bind variable of type CS_DATETIME, an application sets the datatype field of the CS_DATAFMT structure to CS_DATETIME_TYPE.

Client-Library routines that use type constants include ct_bind, ct_describe, and ct_param. In addition, the CS-Library routine cs_convert uses type constants.

The type constant for a datatype is the name of the datatype with “_TYPE” appended. For example, the type constant for the datatype CS_CHAR is CS_CHAR_TYPE.

With the exception of CS_CHAR, all datatypes correspond to a single type constant.

CS_CHAR corresponds to three: CS_CHAR_TYPE, CS_BOUNDARY_TYPE, and CS_SENSITIVITY_TYPE. This means that variables described as CS_BOUNDARY_TYPE or CS_SENSITIVITY_TYPE must be declared as CS_CHAR.

Table 3-3 on page 52 lists Open Client type constants.

Format constants

Open Client and Open Server use format constants to describe how to format character and binary data. In particular, the format field of the CS_DATAFMT structure is a bitmask of format constants indicating how to format character, text, and binary data.

Table 2-1 lists Open Client format constants:

Table 2-1: Format constants

Format constant	Valid types	Resulting format
CS_FMT_NULLTERM	Character and text	The data is null-terminated.
CS_FMT_PADBLANK	Character and text	The data is padded with blanks to the full length of the variable.
CS_FMT_PADNULL	Character, text, binary, and image	The data is padded with nulls to the full length of the variable.
CS_FMT_UNUSED	All	No formatting takes place.

Other symbolic constants

Open Client makes use of a wide variety of other symbolic constants. Many Client-Library routines use symbolic constants as input and output parameter values.

Table 2-2 lists some of the symbolic constants used in Open Client:

Table 2-2: Other symbolic constants

Symbolic constant	Meaning
CS_FAIL	A return code indicating failure
CS_FALSE	A Boolean false value.
CS_MAX_NAME	The maximum column name length allowed by Adaptive Server Enterprise.
CS_NULLTERM	CS_NULLTERM passed as a buffer's length indicates that the value contained in the buffer is null-terminated.
CS_SUCCEED	A return code indicating successful execution of a library call.
CS_TRUE	A Boolean true value.

Note The underlying values of symbolic constants may change from version to version. For this reason, Client-Library application programmers should always code using the symbolic constants themselves and not their underlying values.

Conventions

This section contains information about Client-Library's parameter conventions.

Topics include NULL and unused parameters, string parameters, and the standard Client-Library parameters *action*, *buffer*, *buflen*, and *outlen*.

NULL and unused parameters

This section contains information about 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 (using cs_ctx_alloc, ct_con_alloc, or ct_cmd_alloc) before using them as parameters.

If an application passes an invalid control structure address to a Client-Library routine, the routine returns CS_FAIL, and Client-Library does not call the application's client message callback routine. Client-Library requires the address of a valid control structure to retrieve the address of the application's callback routine.

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 obtain 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 in which the parameters were originally listed in the stored procedure's create procedure statement. This is not necessarily the same order as specified in the Remote Procedure Call (RPC) command that invoked the stored procedure. In determining what number to pass as *item*, do not count nonreturn 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* – 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 places the requested information.

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

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

- *buflen* – 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* – 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.

  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.

Table 2-3 summarizes the interaction between *action*, *buffer*, *buflen,* and *outlen*:

Table 2-3: Interaction between action, buffer, buflen, and outlen parameters

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.

action	buffer	buflen	outlen	What happens
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, noncharacter 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.
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, noncharacter 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, noncharacter 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.

action	buffer	buflen	outlen	What happens
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.

CHAPTER 3 Using Open Client and Server Datatypes

This chapter summarizes the datatypes that are shared by Open Client and Open Server.

Types and type constants

Client-Library supports a wide range of datatypes, which are shared with CS-Library and Server-Library. In most cases, they correspond directly to Adaptive Server Enterprise datatypes.

Where are datatypes declared?

The header file *cstypes.h* contains type definitions (typedefs) for all of the Open Client and Open Server datatypes. The *cstypes.h* file is included in Client-Library applications using *ctpublic.h*—there is no need to include it explicitly.

An application declaring program variables uses these type definitions in its declaration section. For example:

```
CS_CHAR          buffer[40];
CS_INT           result_type, count;
CS_MONEY         profit;
```

Why use Open Client and Open Server datatypes?

There are two reasons why you should use Open Client and Open Server datatypes in your application rather than the native C datatypes: heterogeneous architecture, and portability of application code.

In a client/server application, data may be shared among machines with different architectures.

Open Client and Open Server datatypes provide a platform-independent representation for data that is transported between machines with different architectures. For example, if a client program is compiled and run on a machine that stores the bytes of integer values in a different order from the machine where the server is running, the bytes are swapped when CS_INT values are transported over a connection. For this reason, always use the correct CS_TYPEDEF to declare any variable that holds data to be sent to the server or read from the results of a server command.

Open Client and Open Server datatypes also permit application source code to be ported between platforms. For example, a CS_INT is always mapped to a system datatype that matches a 4-byte integer. Always use the correct CS_TYPEDEF to declare variables that are used in calls to Client-Library or CS-Library routines.

unichar datatype

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

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

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

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

Datatypes and capabilities

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

Following are the Open Client ct_capability() parameters:

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

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

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

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

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

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

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

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

```
...
strcpy (datafmt.name, “@charparam”);
datafmt.namelen =CS_NULLTERM;
datafmt.datatype = CS_UNICHAR_TYPE;
datafmt.maxlength = CS_MAX_CHAR;
```

```
datafmt.status = CS_RETURN;
datafmt.locale = NULL;
...
```

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

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

Table 3-1 lists the CS_DATAFMT bitmask fields.

Table 3-1: CS_DATAFMT structure

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

isql and *bcp* utilities

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

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

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

Limitations

The server to which the Open Client and Open Server is connecting must support 2-byte Unicode datatypes, and use UTF-8 as the default character set.

If the server does not support 2-byte Unicode datatypes, the server returns an error message: "`Type not found. Unichar/univarchar is not supported.`"

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

CS_UNICHAR does not provide C programming operations on UTF-16 encoded Unicode data such as Unicode character strings.

unitext datatype

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

Datatypes and capabilities

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

The Open Client ct_capability() parameters:

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

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

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

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

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

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

Table 3-2: CS_DATAFMT structure

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

isql and *bcp* utilities

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

Limitations

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

If the server does not support 2-byte Unicode datatypes, the server returns an error message. However, the client can convert other datatypes to or from CS_UNITEXT.

CS_UNITEXT does not provide C programming operations on UTF-16 encoded Unicode data such as Unicode character strings.

xml datatype

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

Datatypes and capabilities

Following are the Open Client ct_capability() parameters:

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

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

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

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

isql and *bcp* utilities

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

Limitations

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

Note the following syntax rules of XML:

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

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

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

What are type constants?

Type constants are symbolic values that identify the datatype of a program variable. Many CS-Library, Client-Library, and Server-Library routines take the address of a program variable as a CS_VOID * parameter. Type constants are required to identify the datatype when passing CS_VOID * parameters. Typically, a type constant is passed to a routine as the *datatype* field of a CS_DATAFMT structure. (See "CS_DATAFMT" on page 35.)

Datatype summary

Table 3-3 lists Open Client and Open Server type constants, their corresponding type definitions, and their corresponding Adaptive Server Enterprise datatypes.

Adaptive Server Enterprise datatypes are identified by Transact-SQL keywords. See the Adaptive Server Enterprise documentation for descriptions of the Adaptive Server Enterprise datatypes.

Table 3-3: Datatype summary

Type category	Open Client and Open Server type constant	Description	Corresponding C datatype	Corresponding server datatype
Binary types	CS_BINARY_TYPE	Binary type	CS_BINARY	binary, varbinary
	CS_LONGBINARY_TYPE	Long binary type	CS_LONGBINARY	None
	CS_VARBINARY_TYPE	Variable-length binary type	CS_VARBINARY	None
Bit types	CS_BIT_TYPE	Bit type	CS_BIT	bit
Character types	CS_CHAR_TYPE	Character type	CS_CHAR	char, varchar
	CS_LONGCHAR_TYPE	Long character type	CS_LONGCHAR	None
	CS_VARCHAR_TYPE	Variable-length character type	CS_VARCHAR	None
	CS_UNICHAR_TYPE	Fixed-length or variable-length character type	CS_UNICHAR	unichar univarchar
	CS_XML_TYPE	Variable-length character type	CS_XML	xml

Type category	Open Client and Open Server type constant	Description	Corresponding C datatype	Corresponding server datatype
Datetime type	CS_DATE_TYPE	4-byte date type	CS_DATE	date
	CS_TIME_TYPE	4-byte time type	CS_TIME	time
	CS_DATETIME_TYPE	8-byte datetime type	CS_DATETIME	datetime
	CS_DATETIME4_TYPE	4-byte datetime type	CS_DATETIME4	smalldatetime
	CS_BIGDATETIME_TYPE	8-byte binary type	CS_BIGDATETIME	bigdatetime
	CS_BIGTIME_TYPE	8-byte binary type	CS_BIGTIME	bigtime
Numeric types	CS_TINYINT_TYPE	1-byte unsigned integer type	CS_TINYINT	tinyint
	CS_SMALLINT_TYPE	2-byte integer type	CS_SMALLINT	smallint
	CS_INT_TYPE	4-byte integer type	CS_INT	int
	CS_BIGINT_TYPE	8-byte integer type	CS_BIGINT	bigint
	CS_USMALLINT_TYPE	2-byte unsigned integer type	CS_USMALLINT	usmallint
	CS_UINT_TYPE	4-byte unsigned integer type	CS_UINT	uint
	CS_UBIGINT_TYPE	8-byte unsigned integer type	CS_UBIGINT	ubigint
	CS_DECIMAL_TYPE	Decimal type	CS_DECIMAL	decimal
	CS_NUMERIC_TYPE	Numeric type	CS_NUMERIC	numeric
	CS_FLOAT_TYPE	8-byte float type	CS_FLOAT	float
	CS_REAL_TYPE	4-byte float type	CS_REAL	real
Money types	CS_MONEY_TYPE	8-byte money type	CS_MONEY	money
	CS_MONEY4_TYPE	4-byte money type	CS_MONEY4	smallmoney
Text and image types	CS_TEXT_TYPE	Text type	CS_TEXT	text
	CS_IMAGE_TYPE	Image type	CS_IMAGE	image
	CS_UNITEXT_TYPE	Variable-length character type	CS_UNITEXT	unitext

Binary types

Open Client includes three binary types, CS_BINARY, CS_LONGBINARY, and CS_VARBINARY:

- CS_BINARY corresponds to the Adaptive Server Enterprise 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_LONGBINARY does not correspond to any Adaptive Server Enterprise type, but some Open Server applications may support CS_LONGBINARY. An application can call ct_capability and check 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_VARBINARY does not correspond to any Adaptive Server Enterprise type, and Open Client routines do not return CS_VARBINARY_TYPE. CS_VARBINARY is a structure that holds a byte array and its length:

```
typedef struct _cs_varybin
 {
 CS_SMALLINT      len;
 CS_BYTE          array[CS_MAX_CHAR];
 } CS_VARBINARY;
```

  CS_VARBINARY is provided so that programmers can write non-C programming language veneers to be written for Open Client. Typical client applications do not use CS_VARBINARY.

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 nonzero values are converted to 1.

Character types

Open Client has four character types, CS_CHAR, CS_LONGCHAR, CS_VARCHAR, and CS_XML:

- CS_CHAR corresponds to the Adaptive Server Enterprise types char and varchar. In other words, 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_LONGCHAR does not correspond to any Adaptive Server Enterprise datatype, but some Open Server applications may support CS_LONGCHAR. An application can call ct_capability and check 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_VARCHAR does not correspond to any Adaptive Server Enterprise type. For this reason, Open Client routines do not return CS_VARCHAR_TYPE. CS_VARCHAR is a structure provided to enable non-C programming language veneers to be written for Open Client. It holds a string and its length:

  ```
  typedef struct_cs_varchar
   {
   CS_SMALLINT      len;
   CS_CHAR          str[CS_MAX_CHAR];
   } CS_VARCHAR;
  ```

  Typical client applications do not use CS_VARCHAR.
- Corresponding directly to the xml datatype, CS_XML is an addition to CS_TEXT and CS_IMAGE datatypes for representing XML data. CS_XML represents XML data in an unparsed format and can be used anywhere CS_TEXT and CS_IMAGE is used, for example, in cs_convert, ct_bind, or ct_param.

  CS_XML only fetches data if the server supports XML datatypes. CS_DATA_XML (request) and CS_DATA_NOXML (response), are added to ct_capability to determine the server's capability to support XML datatypes.

  CS_XML is always activated, and its datatype constant is CS_XML_TYPE. The xml datatype is mapped to TDS_XML.

Datetime types

Open Client supports six datetime types: CS_DATE, CS_TIME, CS_DATETIME, CS_DATETIME4, CS_BIGDATETIME, and CS_BIGTIME. These datatypes are intended to hold 8-byte and 4-byte datetime values.

The CS_BIGDATETIME and CS_BIGTIME datatypes provide microsecond-level precision for time data. These datatypes are intended to hold 8-byte binary values.These datatypes function similarly to the respective CS_DATETIME and CS_TIME datatypes: The CS_BIGDATETIME datatype can be used anywhere that the CS_DATETIME datatype can be used, and the CS_BIGTIME datatype can be used anywhere that the CS_TIME datatype can be used. All Open Client and Open Server routines that can be applied to the CS_DATETIME and CS_TIME datatypes can also be applied to the CS_BIGDATETIME and CS_BIGTIME datatypes.

- CS_DATE corresponds to the Adaptive Server Enterprise date datatype with a range of legal values from January 1, 0001 to December 31, 9999.
- CS_TIME corresponds to the Adaptive Server Enterprise time datatype, with a range of legal values from 12:00:00.000 to 11:59:59:999 with a precision of 1/300th of a second (3.33 ms).
- CS_DATETIME corresponds to the Adaptive Server Enterprise datetime datatype, with a range of legal values from January 1, 1753 to December 31, 9999, with a precision of 1/300th of a second (3.33 ms).
- CS_DATETIME4 corresponds to the Adaptive Server Enterprise smalldatetime datatype, with a range of legal values from January 1, 1900 to June 6, 2079, with a precision of 1 minute.
- CS_BIGDATETIME corresponds to the Adaptive Server Enterprise bigdatetime datatype and contains the number of microseconds that have passed since January 1, 0000 00:00:00.000000. The range of legal CS_BIGDATETIME values is from January 1, 0001 00:00:00.000000 to December 31, 9999 23:59:59.999999.

  Note January 1, 0000 00:00:00.000000 is the base starting value from which microseconds are counted. Any value earlier than January 1, 0001 00:00:00.000000 is invalid.

- CS_BIGTIME corresponds to the Adaptive Server Enterprise bigtime datatype and indicates the number of microseconds that have passed since the beginning of the day. The range of legal CS_BIGTIME values is from 00:00:00.000000 to 23:59:59.999999.

- CS_BIGDATETIME and CS_BIGTIME data is presented to the client in the native-byte order (endianness) of the underlying client platform. Any necessary byte-swapping is performed at the server before the data is sent to the client, or after the data is received from the client.

An application can call the CS-Library routine cs_convert to initialize a datetime type from a character string. cs_convert recognizes all of the date and time formats valid for Transact-SQL datetime character strings. See the "Datatypes" topic in the *Adaptive Server Enterprise Reference Manual*.

cs_convert can also convert a CS_DATETIME or CS_DATETIME4 value into a character string.

Other routines that are useful when working with datetime values include:

- cs_cmp, which compares two data values.
- cs_dt_crack, which maps a datetime value to a CS_DATEREC structure. A CS_DATEREC contains distinct fields for the different parts of a datetime value.
- cs_dt_info, which retrieves language-specific datetime information such as day names. This routine also configures the format for converting datetime data values to character strings.

cs_convert, cs_cmp, cs_dt_crack, and cs_dt_info use locale information that is specified indirectly, using the CS_CONTEXT, or directly, using a CS_LOCALE structure. (See "CS_LOCALE" on page 33.) An application can change the locale information for a CS_CONTEXT by calling cs_config to set the CS_LOC_PROP property for the context.

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, CS_INT, a 4-byte integer, CS_BIGINT, an 8-byte integer, CS_USMALLINT, an unsigned 2-byte integer, CS_UINT, an unsigned 4-byte integer and CS_UBIGINT, an unsigned 8-byte integer.
- CS_REAL corresponds to the Adaptive Server Enterprise datatype real and is implemented as a C-language float type.
- CS_FLOAT corresponds to the Adaptive Server Enterprise datatype float and is implemented as a C-language double type.

- CS_NUMERIC and CS_DECIMAL correspond to the Adaptive Server Enterprise datatypes numeric and decimal. These datatypes provide platform-independent support for numbers with precision and scale.

  The Adaptive Server Enterprise datatypes numeric and decimal are equivalent, and CS_DECIMAL is defined as CS_NUMERIC.

Money types

Open Client supports two money datatypes, CS_MONEY and CS_MONEY4. These datatypes are intended to hold 8-byte and 4-byte money values, respectively:

- CS_MONEY corresponds to the Adaptive Server Enterprise money datatype, with legal values between -$922,337,203,685,477.5807 and +$922,337,203,685,477.5807.
- CS_MONEY4 corresponds to the Adaptive Server Enterprise smallmoney datatype, with legal values between -$214,748.3648 and +$214,748.3647.

An application can call the CS-Library routine cs_convert to initialize a money type from a character string. The cs_convert routine recognizes all of the money formats valid for Transact-SQL money character strings. See “Datatypes” in the *Adaptive Server Enterprise Reference Manual*.

The cs_convert routine can also convert a CS_MONEY or CS_MONEY4 value into a character string.

Money values cannot be manipulated with standard C operators because they are stored in structures. To perform arithmetic operations on money values, an application can either:

- Call the CS-Library routine cs_calc to perform the arithmetic operation, or
- Call cs_convert to convert the money type to a datatype with a standard C equivalent (such as CS_FLOAT).

The cs_cmp routine can be called to compare money values.

Text and image types

Open Client supports a *text* datatype, CS_TEXT, a *unitext* datatype, CS_UNITEXT, 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_UNITEXT corresponds to the server datatype unitext. As with text, unitext describes a variable length column containing up to 2,147,483,647 bytes of printable data. The difference is that unitext character data is stored with Unicode UTF-16 encoding rather than the default character set on the server.
- 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.

Small *text*, *unitext* and *image* data values require no special handling. Result values can be bound to program variables and subsequently fetched, and input data values can be entered into a database using the Transact-SQL insert and update commands. However, when *text*, *unitext* and *image* values are large, it is usually more practical for an application to use routines that allow the text, unitext or image data to be handled one chunk at a time.

These routines are:

- ct_data_info, which sets or retrieves a CS_IODESC structure. A CS_IODESC structure describes the text, unitext or image data that is to be read from or written to the server.
- ct_get_data, which reads a chunk of data from the result stream.
- ct_send_data, which writes a chunk of data to the command stream.

See the “text and image Data Handling” topics page in the *Open Client Client-Library/C Reference Manual*.

Null substitution values

When a row containing NULL values is fetched from a server, Client-Library substitutes specified “null substitution values” for the null columns when copying the row data to program variables.

Table 3-4 lists Client-Library’s default null substitution values:

Table 3-4: Default null substitution values

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

To change null substitution values, an application can call the CS-Library routine cs_setnull.

Open Client user-defined datatypes

If an application that needs to use a datatype that is not included in the standard Open Client datatypes, you can create a user-defined datatype. For example, you might create a user-defined datatype that represents encrypted character data. To create a user-defined datatype:

1. Create the new datatype name. For example:

   ```
   typedef char ENCRYPTED_CHAR;
   ```

2. Define a type constant that represents the datatype. For example:

   ```
   #define ENCRYPTED_TYPE      CS_USERTYPE + 2;
   ```

   Because the Open Client routines ct_bind and cs_set_convert use symbolic type constants to identify datatypes, you must define a type constant for each user-defined type. User-defined type constants must be greater than or equal to CS_USERTYPE.

3. Call cs_set_convert to install custom conversion routines to convert between standard Open Client datatypes and the user-defined datatype. For the ENCRYPTED_CHAR user-defined datatype in the example above, you might define and install custom conversion routines that encrypt and decrypt character data. You might, for example, install an encryption routine for conversions from CS_CHAR_TYPE to ENCRYPTED_TYPE, and install a decryption routine for conversions from ENCRYPTED_TYPE to CS_CHAR_TYPE.

4. Call cs_setnull to define a null substitution value for the user-defined datatype.

After conversion routines are installed, an application can bind server results to a user-defined datatype:

```
mydatafmt.datatype = ENCRYPTED_CHAR;
 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 datatypes with Adaptive Server Enterprise user-defined datatypes. Open Client user-defined datatypes are C-language types, declared within an application. Adaptive Server Enterprise user-defined datatypes are database column datatypes, created with the system stored procedure sp_addtype.

CHAPTER 4 Handling Errors and Messages

This chapter describes how to program your applications to handle Client-Library and server error and informational messages.

About messages

Client-Library generates messages in response to a wide range of error and informational conditions. These messages are called “Client-Library messages” or “client messages.”

Servers also generate messages in response to error and informational conditions. These messages are called “server messages.”

How to identify messages

Do not confuse Client-Library messages with Client-Library return codes, or server messages with message results.

Client-Library messages and Client-Library return codes

Client-Library messages are generated in response to Client-Library errors and other conditions of interest. Each Client-Library message includes a number, text, and severity level.

Return codes are symbolic values that indicate success, failure, or other conditions of interest. All Client-Library routines use return codes.

Generally speaking, when a Client-Library routine returns CS_FAIL, Client-Library generates a message, but Client-Library can also generate messages at other times.

Applications need to handle messages in addition to checking return codes.

Server messages and message results

Do not confuse server messages and message results.

Server messages are generated by a server in response to server errors or other exceptional conditions. Each server message includes a number, text, and severity level.

Message results are a type of result that can be sent in response to normal command execution—see “Processing Message Results” on page 6-12.

Server messages and message results are not related.

Two methods for handling messages

An application can handle Client-Library and server messages using one of two methods:

- Callbacks – the application installs its own routines to handle Client-Library and server messages. When a message is generated, Client-Library calls the appropriate callback and passes details about the message using the callback’s input parameters.
- Inline message handling – in mainline code, the application periodically calls ct_diag to retrieve messages.

Callbacks have these advantages:

- They are relatively automatic. Once installed, callbacks are triggered whenever a message occurs.
- They centralize message-handling code.
- They provide a way for an application to gracefully handle unexpected errors. An application that handles errors using the inline method may not successfully trap unanticipated errors.

Inline error handling, on the other hand, has the advantage of operating under an application's direct control, which allows an application to check for messages at particular times. For example, an application might call ct_con_props a dozen times to customize a connection but check for errors only after the last call.

Most applications use callbacks to handle messages, but an application that is running on a platform-and-language combination that does not support callbacks must use the inline method.

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

Combining the methods

An application can use different methods on different connections and can switch back and forth between the two methods, but these techniques are not useful in typical applications.

When moving from the inline to the callback method, installing either type of message callback for a connection turns off inline error handling. Client-Library discards any saved messages.

When moving from the callback to the inline method, calling ct_diag to initialize inline message handling deinstalls a connection's message callbacks. If this occurs, the connection's first call to ct_diag retrieves a warning message.

Handling messages with callback routines

Most applications use callbacks to handle Client-Library and server messages. The application defines and installs callback routines to handle Client-Library and server messages. When a message is generated, Client-Library calls the appropriate callback and passes details about the message using the callback's input parameters.

To use the callback method, an application must define and install:

- A client-message callback to handle Client-Library messages
- A server-message callback to handle server messages

An application calls ct_callback to install a message callback. Once installed, the callbacks are automatically triggered when a Client-Library or server message occurs.

Client-Library stores callback locations 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. Instead, the routine that caused the error returns CS_FAIL.

Defining a client-message callback

A client-message callback is a C function that 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. *connection* can be NULL.
- *message* is a pointer to a CS_CLIENTMSG structure containing Client-Library message information. For information about the CS_CLIENTMSG structure, see the "CS_CLIENTMSG Structure" topics page in the *Open Client Client-Library/C Reference Manual*.

  message can have a new value each time the client-message callback is called.

Like other callbacks, a client-message callback is limited as to which Client-Library routines it can call. A client-message callback can call only the following 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)

A client-message callback must return one of the following return codes:

- CS_SUCCEED, to instruct Client-Library to continue any processing that is occurring on this connection. In the case of timeout errors, CS_SUCCEED causes Client-Library to wait for one additional timeout period. At the end of this period, Client-Library calls the client-message callback again.
- 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. To continue using the connection, the application must close the connection and then reopen it.

Defining a server-message callback

A server-message callback is a C function that 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. *connection* can be NULL.
- *message* is a pointer to a CS_SERVERMSG structure containing server message information. See the “CS_SERVERMSG Structure” topics page in the *Open Client Client-Library/C Reference Manual* for CS_SERVERMSG field descriptions.

  message can have a new value each time the server-message callback is called.

Like other callbacks, a server-message callback is limited as to which Client-Library routines it can call. A server-message callback can call only the following 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)
- ct_res_info, ct_bind, ct_describe, ct_fetch, and ct_get_data, to process extended error data only

A server-message callback must return CS_SUCCEED.

Installing callbacks

An application calls ct_callback to install a client or server-message callback.

If an application installs callbacks at the context level, all connection structures allocated within the context inherit the callbacks.

To “deinstall an existing callback routine, call ct_callback with *action* as CS_SET and *func* as NULL.

To replace an existing callback routine with a new one, call ct_callback with *action* as CS_SET install the new routine. ct_callback replaces the existing callback with the new callback.

To obtain a pointer to an existing callback, call ct_callback with *action* as CS_SET and *func* as the address of a CS_VOID * variable. ct_callback places the address of the callback in the variable.

Handling messages inline

A Client-Library application calls ct_diag to handle Client-Library and server messages inline.

An application can use inline error handling at the connection level only. That is, inline error handling cannot be enabled for a context. If an application has more than one connection, it must make separate ct_diag calls for each connection.

An application calls ct_diag to:

- Initialize inline error handling.

- Clear messages.
- Get messages.
- Limit the number of saved messages.
- Find out how many messages are currently saved.
- Retrieve the CS_COMMAND structure on which extended error data (if any) is available. See "Extended error data" on page 71.

Client-Library does not start saving messages for a connection until inline error handling has been initialized for the connection.

An application can retrieve client-message information into a CS_CLIENTMSG structure or a SQLCA, SQLCODE, or SQLSTATE structure. An application can retrieve server-message information with a CS_SERVERMSG structure or a SQLCA, SQLCODE, or SQLSTATE structure. For information about these structures, see the *Open Client Client-Library/C Reference Manual*.

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.

The CS_EXTRA_INF property

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

The CS_EXTRA_INF property determines whether or not Client-Library returns certain kinds of informational messages, such as the number of rows affected by a command. Normally, an application can call ct_res_info to obtain this information. With CS_EXTRA_INF set to CS_TRUE, the information is returned as a Client-Library message.

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 CS_DIAG_TIMEOUT_FAIL property

When inline error handling is in effect, the CS_DIAG_TIMEOUT_FAIL property determines whether Client-Library fails or retries on Client-Library timeout errors.

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 instruct 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 contains CS_MAX_MSG bytes of the message and the second contains a null terminator.

If an application is using callback routines to handle messages, Client-Library calls the callback routine once for each message chunk.

If an application use ct_diag to handle messages, it must call ct_diag once for each message chunk.

Note The SQLCA, SQLCODE, and SQLSTATE structures do not support sequenced messages. An application cannot use these structures to retrieve sequenced messages. Messages that are too long for these structures are truncated.

Operating system messages are reported in the *osstring* field of the CS_CLIENTMSG structure. Client-Library does not sequence operating system messages.

See the "Error and Message Handling" topics page in the *Open Client Client-Library/C Reference Manual*.

Extended error data

Some server messages have extended error data associated with them, which is additional information about the error. For Adaptive Server Enterprise messages, the additional information usually describes 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 any datatype.

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

Uses of extended error data

Applications that allow end users to enter or edit data often need to report errors to their users at the column level. However, the standard server message mechanism 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 au_id and title_id values that match those in an existing row causes a "duplicate key" message to be sent to the application.

On receiving this message, the application must identify the problem column or columns to the end user so that the user can readily correct them. This information is also available in the text of the duplicate key message, but an application must parse the text to extract the column names.

For information about how to identify and process extended error data, see the "Error and Message Handling" topics page in the *Open Client Client-Library/C Reference Manual*.

Server transaction states

Server transaction state information is useful when an application needs to determine the outcome of a transaction. Table 4-1 lists the symbolic values that represent transaction states.

Table 4-1: Transaction states

Symbolic value	Meaning
CS_TRAN_IN_PROGRESS	A transaction is in progress.
CS_TRAN_COMPLETED	The most recent transaction completed successfully.
CS_TRAN_STMT_FAIL	The most recently executed statement in the current transaction failed.
CS_TRAN_FAIL	The most recent transaction failed.
CS_TRAN_UNDEFINED	A transaction state is not defined.

For information about how to retrieve server transaction states in mainline code and from within a server callback routine, see the "Error and Message Handling" topics page in the *Open Client Client-Library/C Reference Manual*.

CHAPTER 5 Choosing Command Types

Client-Library provides several command types. This chapter introduces each command type, explains how they are used, and discusses their advantages and disadvantages.

Command overview

In a Client-Library application, a command is a stream of TDS protocol symbols and data sent from a client to the server. The command describes some operation that the server is to perform and provides parameter data for the operation. In response to an application's API calls, Client-Library encodes commands in the TDS protocol.

Types of commands

Table 5-1 summarizes the Client-Library command types.

Table 5-1: Summary of command types

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

Executing commands

All commands are executed with these steps:

1. Initiate the command – This step identifies the command type and what it executes.
2. Define parameter values – Some commands require parameter data as input.
3. Send the command – ct_send writes the command symbols and data to the network. The server then reads, interprets, and executes the command.
4. Process the results of the command – ct_results, called in a loop, reads the results of the command. See “Structure of the basic loop” on page 90.

Initiating a command

An application can send several types of commands to a server:

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

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

Defining parameters for a command

The following types of commands can take parameters:

- A language command, when the command text contains variables
- An RPC command, when the stored procedure takes parameters
- A cursor-declare command, when the body of the cursor contains host language parameters
- A cursor-open command, when the body of the cursor contains host language parameters
- A message command
- A dynamic SQL execute command

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

Processing results

Each time a command is sent, the application must process or cancel the results. A typical application calls ct_results until it returns a value other than CS_SUCCEED. See “Structure of the basic loop” on page 90.

Resending a command

For most command types, Client-Library allows an application to resend the command immediately after the results of previous execution have been processed. The application resends commands as follows:

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

The application must have specified the addresses of the parameter source variables with ct_setparam when defining the command.

- The application calls ct_send to resend the command.

An application can resend all types of commands except:

- Send-data commands initiated by ct_command(CS_SEND_DATA_CMD)
- Send-bulk commands initiated by ct_command(CS_SEND_BULK_CMD)

Language commands

A language command sends the text of a query to the server. The server responds by parsing and executing the command.

Language commands for Adaptive Server Enterprise must be written in Transact-SQL. Other servers, such as Replication Server®, use a different language.

Building language commands

Your application initiates a language command by calling ct_command with *type* as CS_LANG_CMD and **buffer* as the language text. For example, the call below initiates a language command to select rows from the authors table in the pubs2 database:

```
ret = ct_command(cmd, CS_LANG_CMD,
 "select au_lname, city from pubs2..authors \
 where state = 'CA'",
 CS_NULLTERM, CS_UNUSED);
```

Language commands can take parameters. For Adaptive Server Enterprise client applications, parameter placement is indicated by undeclared variables in the command text. For example, a language command such as the one below takes a parameter whose value is substituted for “@state_name”:

```
select au_lname, city from pubs2..authors \
       where state = @state_name
```

Parameters are useful when your code executes the same language command more than once.

Results-handling for language commands

Code your application to handle the results of a language command with a standard results loop, as discussed in "Structure of the basic loop" on page 90.

Language commands can return the result types listed in Table 5-2, for the given reasons:

Table 5-2: Result types from the execution of a language command

Result type	Meaning/when received
CS_ROW_RESULT	Regular rows, sent in response to a select statement executed by the language batch or by a called stored procedure.
CS_COMPUTE_RESULT	Compute rows, sent in response to a selectstatement that contains a compute clause. The select statement can be executed by the language batch or by a called stored procedure.
CS_PARAM_RESULT	Output parameter values, sent in response to an exec statement that passes parameter values. (Parameters must be qualified with output in the exec statement.) Output parameter values are received after the results of all statements executed by the procedure.
CS_STATUS_RESULT	A stored procedure's return status, sent in response to an exec statement. The return status is received after the results from all statements executed by the procedure.
CS_COMPUTEFMT_ RESULT, CS_ROWFMT_RESULT	Format results, seen only if the CS_EXPOSE_FMTS connection property is CS_TRUE (the default is CS_FALSE).
CS_CURSOR_ RESULT	Cursor result rows are retrievable using ct_fetch or ct_scroll_fetch.
CS_CMD_DONE	Placeholder to indicate that the results of one logical command have been processed. Seen after the following events: • The results from each statement executed in the language batch have been processed. • The results of each select statement executed by a called stored procedure have been completely processed.
CS_CMD_SUCCEED	Indicates the success of an insert, update, or exec statement that was executed directly by the language batch.
CS_CMD_FAIL	Indicates that the command or a statement within the language batch failed to execute.

When to use language commands

Language commands are useful to applications that execute ad hoc queries. For example, the Sybase isql command interpreter allows an end user to enter queries, sends the queries to the server as a language command, and displays the results.

Language commands are also useful in client-side middleware applications that pass SQL queries to a Sybase server through Client-Library.

When not to use language commands

For better performance, you can code applications that always execute the same query to invoke stored procedures instead. Instead of coding the query in the C application code, you can create a stored procedure to execute the query and use an RPC command to invoke the stored procedure. This method can be faster because the server does not need to parse and interpret the query each time it executes.

Stored procedures can be considerably faster when a single invocation of the procedure replaces several client commands.

Stored procedures can be executed either by an execute language command or by an RPC command. See “RPCs versus execute language commands” on page 83 for a discussion of the differences between these methods.

RPC commands

An RPC command sends the name of a stored procedure or registered procedure to the server, plus values for the procedure’s parameters, if any. If the procedure exists, the server executes it and returns the results.

RPC commands to Adaptive Server Enterprise invoke stored procedures. RPC commands to an Open Server application invoke either registered procedures or the Open Server’s RPC event handlers.

See the *Transact-SQL Users Guide* for information on creating Adaptive Server Enterprise stored procedures. See the “Registered Procedures” topics page in the *Open Server Server-Library/C Reference Manual* for information on registered procedures.

Building RPC commands

Your application initiates an RPC command by calling ct_command with *type* as CS_RPC_CMD, **buffer* as the procedure name, and *option* as CS_NO_RECOMPILE, CS_RECOMPILE, or CS_UNUSED. For example:

```
ct_command(cmd, CS_RPC_CMD, rpc_name, CS_NULLTERM,
CS_NO_RECOMPILE)
```

The *option* value indicates whether the server should recompile the procedure. When invoking an Adaptive Server Enterprise stored procedure, CS_RECOMPILE is equivalent to specifying the with recompile clause in an equivalent execute statement. See the Adaptive Server Enterprise documentation for an explanation of when recompilation is useful.

Parameter values for an RPC command are passed with calls to ct_param or ct_setparam. These routines are identical, except that ct_param copies a data value, while ct_setparam copies pointers to data values. Both routines require a CS_DATAFMT structure, an indicator variable, and the address of a data value. See the reference pages for ct_param and ct_setparam in the *Open Client Client-Library/C Reference Manual*.

For RPC commands, code your ct_param or ct_setparam calls according to the following rules:

- Pass parameter values in a datatype that matches the declaration of the parameter in the stored procedure.

  Client-Library does not convert outgoing parameter values. If necessary, use cs_convert to convert the parameter value into the matching datatype.

- Pass all parameters by name or all parameters by position.

  To pass a parameter by name, copy its name into the *name* field of ct_param's or ct_setparam's *datafmt* parameter, and set *datafmt.length* to match. Parameters for which you do not call ct_param or ct_setparam are effectively passed as NULL.

  To pass parameters by position, set *datafmt.length* to 0 and call ct_param or ct_setparam in the order in which the parameters appear in the procedure's definition. To pass a parameter as NULL, set the associated *indicator* variable to -1.

  All parameters must be passed using the same method. RPC commands that pass parameters by position usually perform better than those that pass parameters by name.

- Set *datafmt.status* to indicate whether the parameter is a return parameter.

  CS_RETURN indicates a return parameter; use CS_INPUTVALUE for non-return parameters.

Return parameters are similar to the "pass by reference" facility offered by some programming languages. The value of the parameter, with any changes made by the procedure code, is available to the client application after the procedure completes execution. See "Return parameter values" on page 81.

- Use ct_setparam rather than ct_param when the command will be sent multiple times with varying parameter values.

  ct_setparam binds a parameter source variable to the initiated command, allowing the application to change the parameter's value between calls to ct_send.

For an example that illustrates how to define an RPC command with parameters, see the reference page for ct_param in the *Open Client Client-Library/C Reference Manual*.

RPC command results handling

Code your application to handle the results of an RPC command with a standard results loop, as discussed in "Structure of the basic loop" on page 90.

RPC commands can return the result types listed in Table 5-3, for the given reasons:

Table 5-3: Result types from the execution of an RPC command

Result type	Meaning/when received
CS_ROW_RESULT	Regular rows, sent in response to a select statement executed by the procedure.
CS_COMPUTE_RESULT	Compute rows, sent in response to a selectstatement that contains a compute by clause.
CS_PARAM_RESULT	Return (output) parameter values, received after results from all statements in the procedure have been processed.
CS_STATUS_RESULT	The procedure's return status, received after results from all statements in the procedure have been processed.
CS_COMPUTEFMT_ RESULT, CS_ROWFMT_RESULT	Format results, seen only if the CS_EXPOSE_FMTS connection property is CS_TRUE (the default is CS_FALSE).
CS_CMD_DONE	Placeholder that indicates the results of one logical command have been processed. Seen after the following events: • The results from each statement executed in the language batch have been processed • The results of each select statement executed by a called stored procedure have been completely processed

Result type	Meaning/when received
CS_CMD_SUCCEED	Indicates that the procedure was invoked successfully, but does not mean that all the statements in the stored procedure executed successfully. Applications must always check the stored procedure's return status value to determine whether an error occurred (see "Return status values" on page 81).
CS_CMD_FAIL	Indicates that the procedure call failed. Not all errors cause CS_CMD_FAIL to be returned. A statement may fail in the stored procedure, but the server still returns a result type of CS_CMD_SUCCEED. Applications must always check the stored procedure's return status value to determine whether an error occurred (see "Return status values" on page 81).

Return parameter values

The server returns parameter values in the results of an RPC command for each parameter for which both of the following statements are true:

- The parameter is passed as a return parameter in the RPC command.
- The parameter is defined as an output parameter in the definition of the procedure.

If parameter data is returned, all parameter values are returned in a CS_PARAM_RESULT result set.

Return status values

Return status values are returned as a CS_STATUS_RESULT result set (see "Processing return status results" on page 97).

Note SQL statements that return a result type of CS_CMD_FAIL when executed by a language command may return CS_CMD_SUCCEED when executed by a stored procedure. Always check a stored procedure's return status to determine whether the procedure executed successfully.

If a procedure successfully completes execution, the return status is either the value explicitly returned by the procedure or 0 if the procedure lacks an explicit return statement. However, some runtime errors cause a stored procedure to abort before it executes to completion. For example, a select statement in the procedure may refer to a table that no longer exists. For these errors, Adaptive Server Enterprise aborts the execution of the procedure and returns a return status value that indicates the error—see the return reference page in the *Adaptive Server Enterprise Reference Manual* for a list of return status codes and their meaning.

When a runtime error occurs inside a stored procedure, Adaptive Server Enterprise does not return a result type of CS_CMD_FAIL. To determine whether a server-side error has occurred inside the procedure, applications should always check the return status of the stored procedure. Adaptive Server Enterprise also sends server messages that describe runtime errors.

When to use RPC commands

RPC commands offer the following unique benefits:

- Stored procedure parameter values do not require conversion on the server.

  When invoking a stored procedure with an RPC command, parameters are passed in their declared datatypes. The server does not need to convert the parameters from character format to their declared datatypes.

- There is no other way to execute Open Server registered procedures.

  Open Server registered procedures provide a relatively simple way to develop a distributed application with Open Client and Open Server. Registered procedures can be either a function in the Open Server application code, or a special type of procedure that is created by a client application and exists only to trigger client notification events when it is executed. The latter type is created when the client application invokes the sp_regcreate Open Server system registered procedure.

  - See the *Open Server Server-Library/C Reference Manual* for information on defining C functions that can be called as a registered procedure.

  - See the sp_regcreate reference page in the *Open Server Server-Library/C Reference Manual* for details on how Client-Library applications can create a registered procedure on an Open Server.

- See the "Registered Procedures" topics page in the *Open Client Client-Library/C Reference Manual* for information on how Client-Library applications can receive registered procedure notifications.

RPCs versus *execute* language commands

A stored procedure can be executed either by an RPC command or by an execute language statement. Remote procedure calls have a few advantages over execute statements:

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

  A Transact-SQL language command can be used only to execute an Adaptive Server Enterprise 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 in character format, within the text of the language command. 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 character-format 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. (A return parameter must be specified as such when it is originally added to the RPC command stream with ct_param or ct_setparam.)

  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.

Client-Library cursor commands

A cursor is a symbolic name that an application attaches to a select statement. The cursor supports operations to manipulate the select's result set. See "Cursor overview" on page 105 for a list of cursor operations.

A Client-Library cursor is created with a ct_cursor or ct_dynamic cursor-declare command.

Building Client-Library cursor commands

Chapter 7, "Using Client-Library Cursors" explains how to use Client-Library cursor commands in your application. See "Using Client-Library cursors" on page 111 for the typical call sequence.

When to use Client-Library cursors

Use Client-Library cursors when you want to process two or more commands at the same time while using only one server connection.

A Client-Library cursor-open command is the only command type that allows the application to send new commands over the same connection while still retrieving rows. After sending any other type of command, your application must completely process the results of the command before another command can be sent on the same connection. If the application design requires this functionality, then there is no alternative to using Client-Library cursor commands. See "Benefits of Client-Library cursors" on page 109 and "Connection and command rules" on page 31.

Note that cursors can only be declared to execute a single select statement. See "Step 1: Declare the cursor" on page 113.

When not to use Client-Library cursors

Cursors do incur a performance penalty relative to executing a select statement using a language or RPC command. The difference occurs because the cursor requires internal Client-Library cursor-fetch commands to retrieve cursor rows, while a regular-row result set does not. Thus, processing the results of the cursor-open command requires more network round trips. (See “Step 2: Set cursor rows” on page 119.) There is also additional Adaptive Server Enterprise internal overhead associated with cursor processing.

Dynamic SQL commands

Dynamic SQL is the process of generating, preparing, and executing SQL statements at runtime using commands initiated by Client-Library’s ct_dynamic routine.

Building Dynamic SQL commands

Chapter 8, “Using Dynamic SQL Commands” explains how to use Client-Library cursor commands in your application. See “Program structure for the prepare-and-execute method” on page 132 for the typical call sequence.

When to use dynamic SQL commands

Dynamic SQL prepared statement commands are the only command type that allows the application to query the server for the inputs required to execute the command and for the format of the command’s results:

- A ct_dynamic describe-input command causes the server to send the number and format of parameters that are required to execute the statement. See “Step 2: Get a description of command inputs” on page 134 for details.
- A ct_dynamic describe-output command causes the server to send the number and formats of result columns that the statement returns. See “Step 3: Get a description of command outputs” on page 136 for details.

When not to use dynamic SQL

In general, dynamic SQL should not be used in applications where the design does not require the specific advantages listed under "Benefits of dynamic SQL" on page 128. Dynamic SQL commands incur more overhead than language commands. Also, since they are implemented internally as temporary stored procedures, they can cause resource-contention issues in the Adaptive Server Enterprise tempdb database.

See "Limitations of dynamic SQL" on page 128 and "Alternatives to dynamic SQL" on page 130.

Message commands

Message commands can be used with custom Open Server applications. Adaptive Server Enterprise does not support message commands. From the client-application programmer's perspective, a message command is equivalent to an RPC command that is called by number rather than by name.

Your application initiates a message command by calling ct_command with *type* as CS_MESSAGE_CMD and **buffer* as the address of a CS_INT variable that contains the identifier for the message command. For example:

```
CS_INT     msg_id;
if (ct_command(cmd, CS_MSG_CMD, (CS_VOID *)&msg_id,
CS_UNUSED, CS_UNUSED)
!= CS_SUCCEED)
{
fprintf(stderr, "ftclient: ct_command(MSG_CMD)
failed.\n");
return CS_FAIL;
}
```

Message identifiers must be known to both the client application and the Open Server application. Typically, the message command identifiers that a server responds to are defined in a shared header file. Sybase reserves message identifiers in the range CS_USER_MSGID to CS_USER_MAX_MSGID (inclusive) for customer use.

Message commands can take parameters. These are supplied with ct_param or ct_setparam. Whether parameters are passed by name or by position depends on how the Open Server application is coded.

Code your application to handle the results of a message command with a standard results loop, as discussed in "Structure of the basic loop" on page 90. Among other result types, message commands can return message results (result type of CS_MSG_RESULT). See "Processing message results" on page 100.

When to use message commands

Message commands provide an alternative to RPC commands in the design of the client interface for a custom Open Server application. A message command uses an integer identifier rather than a string RPC name and lacks the fixed-parameter list of an Open Server registered procedure.

In the Open Server code, message commands are handled by the message event handler. See the *Open Server Server-Library/C Reference Manual*.

When not to use message commands

Adaptive Server Enterprise does not support message commands.

Package commands

Package commands are supported only on connections to an Open Server on CICS. Package commands are otherwise similar to RPC commands.

Send-data commands

Send-data commands, initiated with ct_command(CS_SEND_DATA), are used to upload text or image column values in chunks.

See the "text and image Data Handling" topics page in the *Open Client Client-Library/C Reference Manual* for details on how to use send-data commands in your application.

When to use send-data commands

For Adaptive Server Enterprise client applications, send-data commands are the only way to upload large *text* or *image* column values a chunk at a time. If your application uploads text or image values that are to large to fit in a contiguous memory buffer, then send-data commands are the only practical method to perform the update.

For *text* or *image* column values that are small enough to fit into a contiguous memory buffer, the application may achieve better performance by embedding the values in insert language commands. See the “text and image Data Handling” topics page in the *Open Client Client-Library/C Reference Manual* for details on this method.

When not to use send-data commands

Generally, send-data commands should be avoided when designing the client interface for a custom Open Server application. Open Server application processing for send-data commands is quite complicated. If the server must allow uploads of large values in chunks, you can design the interface so that values are uploaded with multiple invocations of a message, RPC, or language command. For example, with message commands, one message command identifier might indicate the beginning of an upload operation, and another might indicate a command that contains (as a parameter) a chunk of the data value.

CHAPTER 6 Writing Results-Handling Code

This chapter explains Client-Library's results-processing model. It covers the following topics:

Types of results

After an application sends a command to a server, it must process any results generated by the command. Types of results include:

- Regular row results – rows returned when the server processes a select statement.
- Cursor row results – rows returned when the server processes a ct_cursor Client-Library cursor-open command.
- Parameter results – fetchable data that can represent:
  - Output values for an Adaptive Server Enterprise stored procedure's return parameters
  - Output values for an Open Server registered procedure's return parameters

- A new timestamp value for an updated *text*/*image* column (seen only when processing the results of a ct_command send-data command)
- A new timestamp value for a row that was updated with a language command containing a browse-mode update statement

- Stored procedure return status results – the return value from an Adaptive Server Enterprise stored procedure or Open Server registered procedure.
- Compute row results –intermediate rows returned when the server processes a select statement with a compute by clause.
- Message results – a message ID returned by an Open Server application's message command handler while processing the results of a message command.
- Describe results – informational results that describe the format of a prepared dynamic SQL statement's input parameters or result columns.
- Format results – informational results used by Open Server gateway applications to retrieve regular row and compute row formats before the actual data arrives.

A single command can generate more than one type of result. For example, a language command that executes a stored procedure can generate multiple regular row and compute row result sets, a parameter result set, and a return status result set. For this reason, it is important that you code applications to handle all types of results that a server can generate.

The simplest way for an application to handle all result types is to process results in a loop as described in the following section.

Structure of the basic loop

Most synchronous Client-Library programs process results using a loop controlled by ct_results. Inside the loop, a switch takes place on the type of result that is currently available for processing, as indicated by the value of ct_results' parameter *result_type*. Different types of results require different types of processing.

result_type is also used to indicate the outcome of a server command that returns no results, for example, an insert or delete command.

Most synchronous applications use a program structure similar to the following one to process results:

```
while ct_results returns CS_SUCCEED
    (optional) ct_res_info to get current
        command number
    switch on result_type
        /*
        ** Values of result_type that indicate
        ** fetchable results:
        */
        case CS_COMPUTE_RESULT...
        case CS_CURSOR_RESULT...
        case CS_PARAM_RESULT...
        case CS_ROW_RESULT...
        case CS_STATUS_RESULT...
        /*
        ** Values of result_type that indicate
        ** non-fetchable results:
        */
        case CS_COMPUTEFMT_RESULT...
        case CS_MSG_RESULT...
        case CS_ROWFMT_RESULT...
        case CS_DESCRIBE_RESULT...
        /*
        ** Other values of result_type:
        */
        case CS_CMD_DONE...
            (optional) ct_res_info to get the
                number of rows affected by
                the current command
        case CS_CMD_FAIL...
        case CS_CMD_SUCCEED...
    end switch
end while
switch on ct_results’ final return code
    case CS_END_RESULTS...
    case CS_CANCELED...
    case CS_FAIL...
end switch
```

Processing 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.

An application typically calls the following routines to process a regular row result set:

- ct_res_info, which returns information about the current result set. Most often, an application uses ct_res_info to get the number of columns in the current result set. However, ct_res_info also returns other types of information—for example, the number of rows affected by the current command.
- ct_describe, which returns information about a particular result item in the current result set. An application generally needs to call ct_describe once for each result item before binding each result item to a program variable.
- ct_bind, which binds a result item to a program variable. Binding creates an association between a result item and a data space.
- ct_fetch, which copies result data into bound variables.

Binding is the process of associating a result item with program data space. *Fetching* is the process of retrieving a data instance of a result item. If binding has been specified for a result item, then fetching causes a data instance of the item to be copied into the program data space.

Most synchronous applications use a program structure similar to the following one to process a regular row result set:

```
case CS_ROW_RESULT
    ct_res_info(CS_NUMDATA) 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
```

Processing cursor 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 a language command containing a Transact-SQL open statement. The open statement opens an Adaptive Server Enterprise language cursor, which returns regular rows each time the application executes a Transact-SQL fetch statement. See “Language cursors versus Client-Library cursors” on page 106.

A cursor row result set contains zero or more rows of tabular data.

In general, when an application sends a command to a server, it cannot send another command on the same connection until ct_results indicates that the results of the first command have been completely processed (by returning CS_END_RESULTS, CS_CANCELED, or CS_FAIL).

An exception to this rule occurs when ct_results indicates cursor results. In this case, an application can call ct_cursor and ct_send to send cursor-update, cursor-delete, or cursor-close commands while processing the cursor result set. Using a a different CS_COMMAND structure, the application can also send new commands over the same connection to the server. See “Benefits of Client-Library cursors” on page 109.

In addition to ct_res_info, ct_describe, ct_bind, and ct_fetch, an application can call ct_keydata, ct_cursor, ct_param, ct_send, ct_results, and ct_cancel while processing a cursor result set.

Most synchronous applications use a program structure similar to the following one to process a cursor result set:

```
case CS_CURSOR_RESULT
ct_res_info(CS_NUMDATA) 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
           and cursor has not been closed
         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 a nested cursor update, delete, or
close command: */
          ct_cursor to initiate the cursor command
          /* For updates/deletes whose “where” clause
contains variables */
          ct_param or ct_setparam for each parameter
          /* End for updates/deletes whose ... */
          ct_send to send the command
          while ct_results returns CS_SUCCEED
              (...process results...)
          end while
          /* End to send a nested cursor command */

   end while
    switch on ct_fetch’s final return code
          case CS_END_DATA...
          case CS_CANCELED...
          case CS_FAIL...
      end switch
     if cursor was closed
          break out of outer ct_results loop
      end if

  end case
```

Calls to ct_results are nested within a ct_fetch loop and a larger ct_results loop (not shown).

For nested cursor-update or cursor-delete commands, after the inner ct_results indicates that the results from the nested command have been completely processed (by returning CS_END_RESULTS, CS_FAIL, or CS_CANCELED), any subsequent calls to ct_results will operate on results generated by the original cursor command.

For nested cursor-close commands, there are no results remaining after the cursor is closed. In this case, the application breaks out of the outer ct_results loop after the results of the nested cursor-close command have been processed.

To cancel the cursor rows returned by the cursor-open command, an application can call ct_cancel with *type* as CS_CANCEL_CURRENT. However, it is more efficient to close the cursor with a nested cursor-close command. A CS_CANCEL_CURRENT ct_cancel call retrieves the unwanted rows and discards them. (It is equivalent to clearing all binds, then calling ct_fetch until ct_fetch returns CS_END_DATA.)

Note In your cursor application, do not use any other type of cancel besides CS_CANCEL_CURRENT on a connection that has an open cursor—CS_CANCEL_ALL or CS_CANCEL_ATTN can put a connection's cursors into an undefined state. Instead of canceling, the application can simply close the cursor.

Processing scrollable cursor results

The program structure for processing scrollable cursor results is similar to that for regular cursors. The key difference is that ct_scroll_fetch returns CS_SCROLL_CURSOR ENDS when you use the CS_FALSE option. This is indicated as follows:

```
      end while
       switch on ct_scroll_fetch's final return code
            case CS_SCROLL_CURSOR_ENDS...
        end switch
       if cursor was closed
            break out of outer ct_results loop
        end if

    end case
```

Note ct_scroll_fetch never returns CS_END_DATA as a valid return.

Note A warning message is generated if certain sequences of operations cause the cursor to move beyond the resultset boundary. Examples of this are the sequential use of CS_PREV, CS_NEXT, CS_ABSOLUTE or CS_RELATIVE calls, with offsets of such magnitude (relative to the current cursor position), that the cursor moves beyond the resultset boundary. The warning messages does not indicate that an error has occurred. See the *Open Client Client-Library/C Reference Manual*.

Processing parameter results

A parameter result set contains a single row of parameters.

Several types of data can be returned to an application in the form of a parameter result set, including:

- Return parameter values

  An Adaptive Server Enterprise stored procedure or an Open Server registered procedure can return output parameter data. The CS_PARAM_RESULT result set contains new values for the procedure's parameters, as set by the procedure code. See "RPC commands" on page 78 for a description of how applications execute stored procedures or registered procedures.

- Browse mode timestamp values

  Browse mode is a scheme that interactive applications can use to perform ad hoc row updates of retrieved rows. Tables involved in browse mode require a timestamp column to control simultaneous access to the data. After a client application executes a browse-mode update statement, Adaptive Server Enterprise returns a parameter result set that contains the new timestamp value for the updated row. See the "Browse Mode" topics page in the *Open Client Client-Library/C Reference Manual* for more details.

- A text or image column timestamp

  After a client application updates a text or image column with a send-data command, Adaptive Server Enterprise returns the new text timestamp for the column as a parameter result set. See the "text and image Data Handling" topics page in the *Open Client Client-Library/C Reference Manual* for more details.

- Message result parameters

  A message result set consists of a message identifier (see "Processing message results" on page 100). The message result set can be followed immediately by a parameter result set containing parameter values that accompany the message result.

An application calls ct_res_info, ct_describe, ct_bind, and ct_fetch to process a parameter result set.

Most synchronous applications use a program structure similar to the following one to process a parameter result set:

```
case CS_PARAM_RESULT
```

```
    ct_res_info(CS_NUMDATA) 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
```

Processing return status results

A return status result set is generated by the execution of a stored procedure. All stored procedures return a status number. See the description of the return command in the *Adaptive Server Enterprise Reference Manual*.

A return status result set consists of a single row containing a return status.

An application calls ct_bind and ct_fetch to process a return status.

Most synchronous applications use a program structure similar to the following one to process a return status result set:

```
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
```

Processing compute results

A compute result set is generated by the execution of a Transact-SQL select statement that contains a compute clause. A compute clause generates a compute result set every time the value of its bylist changes. A compute result set consists of a single row containing a number of columns equal to the number of row aggregates in the compute clause.

For example, consider the query:

```
select type, price from titles
 where price > $12 and type like "%cook"
 order by type, price compute sum(price) by type
```

The query returns regular rows (with columns type and price). Intermixed with the regular rows, the query returns compute result sets each time the value of type changes in the regular row results. Each compute result set contains a single row with one column for the sum(price) expression.

See the *Adaptive Server Enterprise Reference Manual* for more examples of queries with a compute clause.

In addition to ct_res_info, ct_describe, ct_bind, and ct_fetch, an application can call ct_compute_info while processing compute row results. ct_compute_info provides a variety of compute row information. The information available from ct_compute_info includes:

- The compute ID for a compute row

  A query can have more than one compute clause. ct_compute_info(CS_COMP_ID) retrieves the number of the compute clause that generated a compute result set. A compute row ID of 1 corresponds to the first compute clause in the query.

- The compute bylist

  The compute bylist is the list of columns that follows the by keyword in the compute clause. In the application, the bylist is represented by an array of CS_SMALLINT values, each of which represents the position of a column in the select list. For example:

```
select dept, name, year, sales from employee
           order by dept, name, year
          compute count(name) by dept, name
```

If you execute this query, then the bylist values are 1 and 2, corresponding to the positions of dept and name in the select list.

ct_compute_info(CS_BYLIST_LEN) returns the length of the bylist, and ct_compute_info(CS_BYLIST) populates an application-allocated array with the bylist column numbers.

- Compute row select-list column IDs

  Select-list column IDs are available for each column in a compute row. The select-list column ID is the select-list position of the column from which the compute-row column was derived. For example, this query returns compute rows containing one column for the sum(price) expression:

  ```
  select type, price from titles
   where price > $12 and type like "%cook"
   order by type, price compute sum(price) by type
  ```

  The corresponding select-list column ID is 2, which is the position of the price column in the select list.

  ct_compute_info retrieves compute column IDs when called with *type* as CS_COMP_COLID and *colnum* as the compute column number.

- Compute column operators

  ct_compute_info, when called with *type* as CS_COMP_OP and *colnum* as the compute column number, retrieves a symbolic constant that indicates the operator with which the column value was computed. See the ct_compute_info reference page in the *Open Client Client-Library/C Reference Manual* for a list of these operators.

Most synchronous applications use a program structure similar to the following one to process a compute result set:

```
case CS_COMPUTE_RESULT
    (optional)ct_compute_info to get bylist length,
    bylist, or compute row id
    ct_res_info(CS_NUMDATA) 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
```

Processing message results

All types of servers can return message results.

A message result set contains no fetchable results. Instead, a message has an ID, which an application can retrieve by calling ct_res_info(CS_MSGTYPE).

Message IDs in the range 1–32,767 are reserved for Adaptive Server Enterprise and Sybase internal use.

Application-defined message IDs must be in the range CS_USER_MSGID to CS_USER_MAX_MSGID.

If parameter values are associated with a message, they are returned as a separate parameter result set following the message result set. See “Processing parameter results” on page 96.

Note A message result set is not the same thing as a server message. Server messages are generated in response to error conditions or to indicate server conditions of interest. They are generally handled within an application’s server-message callback. See Chapter 4, “Handling Errors and Messages.”

An application calls ct_res_info to retrieve a message ID.

Most synchronous applications use a program structure similar to the following one to process a message result set:

```
case CS_MSG_RESULT
    ct_res_info to get the message ID
    code to handle the message ID
end case
```

Processing describe results

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

See “Step 2: Get a description of command inputs” on page 134 and “Step 3: Get a description of command outputs” on page 136.

An application can retrieve this information by calling ct_describe, ct_dyndesc, or ct_dynsqlda. See “Processing parameter descriptions” on page 134 and “Processing column descriptions” on page 136.

Most applications use a program structure similar to the following one to process a describe result set:

```
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
```

Processing format results

Normally, format information for regular row and compute row result sets is only available while the application is processing the result set. At that time, the application can call ct_res_info to retrieve the number of items in the result set, ct_describe to get a description of each item, and ct_compute_info to get compute information.

This mechanism works well for most applications. Some applications, however, need to be able to get format information for a result set before they process the result set. An example of this type of application is a gateway application that repackages Adaptive Server Enterprise results before sending them on to a non-Sybase client.

Client-Library makes advance format information available to an application in the form of *format results*. There are two types of format results: regular row format results and compute row format results.

Format result sets contain no fetchable results. Instead, an application can call ct_res_info, ct_describe, and ct_compute_info to retrieve format information after ct_results indicates format results.

To receive format results, an application must set the Client-Library CS_EXPOSE_FMTS property to CS_TRUE.

An application can call ct_describe and ct_compute_info to retrieve format information.

A gateway application might use a program structure similar to the following one to process format results:

```
case CS_ROWFMT_RESULT
    ct_res_info(CS_NUMDATA) 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
```

Values of *result_type* that indicate command status

In addition to indicating the type of result set that is available for processing, ct_results sets *result_type* to the values below to indicate the status of command processing:

- CS_CMD_DONE – indicates that the results of a *logical command* have been completely processed. See "Logical commands" on page 103 for an explanation of this term.
- CS_CMD_SUCCEED – indicates the success of a command that returns no data, such as a Transact-SQL insert or delete command.
- CS_CMD_FAIL – indicates that, due to error, the server failed to execute a server command. For example, the text of a language command might contain a syntax error or refer to a nonexistent object. In most cases, the server returns a server message that describes the error.

  Because a Client-Library command can execute multiple server commands, an application must either:

  - Continue to call ct_results to process results generated by any other server commands contained in the original Client-Library command, or
  - Call ct_cancel(CS_CANCEL_ALL) to cancel the Client-Library command and discard its results.

Logical commands

ct_results sets *result_type* to CS_CMD_DONE to indicate that the results of a logical command have been completely processed. A *logical command* is any command defined using ct_command, ct_dynamic, or ct_cursor, with the following exceptions:

- Each Transact-SQL select statement that returns data inside a stored procedure is a logical command. Other Transact-SQL statements inside stored procedures do not count as logical commands (including select statements that assign values to local variables).
- Each Transact-SQL statement executed by a dynamic SQL command is a distinct logical command.
- Each Transact-SQL statement in the text of language command is a logical command.

Logical commands and Client-Library commands are not equivalent. A Client-Library command can execute multiple logical commands on the server, for example, a stored procedure can execute multiple select statements that return data, and each such statement represents one logical command. A logical command can generate one or more result sets; for example, a select statement can return multiple regular-row and compute results sets.

ct_results final return code

After handling all the results of the command, your code should check the final return code from ct_results to see if errors are indicated.

Final return code values can be the following:

- CS_END_RESULT – indicates a normal loop exit.
- CS_CANCELED – indicates that results were canceled: ct_cancel(CS_CANCEL_ALL) or ct_cancel(CS_CANCEL_ATTN) was called while processing results.
- CS_FAIL – indicates a serious client-side or network error, such as a communication failure or a memory shortage.

CHAPTER 7 Using Client-Library Cursors

This chapter explains Client-Library cursors. It covers the following topics:

Cursor overview

A cursor is a symbolic name that an application attaches to a select statement. The statement can be executed and its result set manipulated by performing operations on the cursor.

Cursors support the following operations:

- Declare – create a new cursor by giving it a name and defining its query.
- Set cursor rows – specify the number of rows from the result table to be returned with each fetch operation.
- Open – execute the cursor's query and prepare it for fetch operations.
- Fetch – retrieve rows from the cursor, which must be open. Each fetch operation retrieves a single row from the query's result table. This applies to both scrollable and non-scrollable cursors. Under certain conditions (as defined by the "set cursor rows" operation), more rows can be returned per fetch call.
- Update – modify the values in a fetched row. The update affects the tables from which the row was selected.
- Delete – remove a fetched row from an underlying table.

- Close – ready the cursor to be either reopened or deallocated.
- Deallocate – free the cursor's resources.

In an Adaptive Server Enterprise client application, cursors can either be created and manipulated with language commands or with ct_cursor commands. Cursors created using Transact-SQL language commands are called *language cursors*. Cursors created with ct_cursor commands are called *Client-Library cursors*. Table 7-1 on page 107 compares the two types of cursors.

Language cursors versus Client-Library cursors

Table 7-1 compares Transact-SQL (language) cursor commands with Client-Library cursor commands:

Table 7-1: Transact-SQL cursor commands versus Client-Library cursor commands

Operation	Language command	Client-Library cursor command
Declare	declare cursor	ct_cursor(CS_CURSOR_DECLARE) or ct_dynamic(CS_CURSOR_DECLARE)
Set cursor rows	set cursor rows	ct_cursor(CS_CURSOR_ROWS)
Open	open	ct_cursor(CS_CURSOR_OPEN)
Fetch	fetch	ct_fetch or ct_scroll_fetch, after ct_results has returned with a *result_type* of CS_CURSOR_RESULT.
Update	update ... where current of *cursor_name*	ct_cursor(CS_CURSOR_UPDATE) By default, affects the last fetched row, but can be redirected to any previously fetched row.
Delete	delete ... where current of *cursor_name*	ct_cursor(CS_CURSOR_DELETE) By default, affects the last fetched row, but can be redirected to any previously fetched row.
Close	close	ct_cursor(CS_CURSOR_CLOSE)
Deallocate	deallocate cursor	ct_cursor(CS_CURSOR_DEALLOC) or ct_cursor(CS_CURSOR_CLOSE) The cursor is closed and deallocated with one command if the CS_DEALLOC bit is set in ct_cursor's *option* parameter.

Language cursors

On Adaptive Server Enterprise, a language cursor is declared with the declare cursor statement, opened with an open statement, and fetched from using fetch statements. See the *Adaptive Server Enterprise Reference Manual* for descriptions of these commands. A Client-Library program can send all of these statements as normal language commands.

Once a language cursor has been declared and opened, each fetch language command returns a set of regular rows (ct_results *result_type* is CS_ROW_RESULT) and can be handled just like the results of a select command (see "Processing regular row results" on page 91). As with any other language command, the results of each command must be processed with ct_results (and ct_fetch, if necessary) before another command can be sent on the connection.

When declared within a language command sent by a client connection, a language cursor has scope limited to that connection. In other words, only language commands sent over the same connection can reference the cursor.

Language cursors provide the following advantage over Client-Library cursors:

- On Adaptive Server Enterprise, you can declare a cursor and open inside a Transact-SQL stored procedure. Such a cursor is called a *server cursor*. Complex tasks that are implemented using a stored procedure and server cursors should perform better than an equivalent implementation that uses Client-Library cursors. The performance difference is mainly due to the fact that the Client-Library cursor requires many network round trips to fetch the cursor rows (and to execute any nested update commands), while the server cursor does not.
- Language cursors can be used with an existing client application that handles ad hoc language commands. For example, a user of the Sybase isql client application can use language cursors, even though isql contains no special code to support cursors.

The *Adaptive Server Enterprise Reference Manual* contains more detailed information on language cursors.

Client-Library cursors

A Client-Library cursor requires application programmers to code ct_cursor calls that declare and open the cursor. A Client-Library cursor-open command returns a single fetchable result set of type CS_CURSOR_RESULT.

A Client-Library cursor's scope is limited to a single command structure. In fact, once a cursor is declared with a command structure, that command structure becomes a dedicated "handle" for further operations on the cursor.

Client-Library cursors provide the following advantages over language cursors:

- Fetching from a Client-Library cursor is more simple.

  Each fetch from a Client-Library cursor involves a single ct_fetch or ct_scroll_fetch call; after each ct_fetch or ct_scroll_fetch call that returns rows, the application can send new commands over the connection.

  Each fetch from a language cursor is a separate Client-Library command that involves calls to ct_command, ct_send, ct_results, ct_fetch, and so forth. The results of the fetch language command must be completely processed before the application can send new commands over the same connection.

- A Client-Library cursor can be used to modify any previously fetched row. A language cursor can only be used to delete or update the most recently fetched row.

- A Client-Library cursor can be declared to execute a stored procedure (as long as the stored procedure only executes a single select statement—for more details, see “Step 1: Declare the cursor” on page 113). A language cursor must be declared with a select statement.

When to use Client-Library cursors

Client-Library cursors offer some unique benefits, but they also may incur a performance penalty relative to other command types.

Benefits of Client-Library cursors

Client-Library cursors provide the following unique benefits to an application:

- They allow the application to execute simultaneous commands on the same connection.
- They allow an application to update a table while fetching from it using only a single connection.

A ct_cursor cursor-open command is the only command type that allows simultaneous command processing on a single connection. After sending any other type of command, the application must completely process the results of the command before sending another command. When processing the results of a cursor-open command, the client application execute two categories of new commands:

- Nested cursor commands on the same command structure
- Unrelated commands executed using a different command structure

Nested cursor commands

A *nested cursor command* is a cursor-close, cursor-delete, or cursor-update command that is sent while fetching the rows returned by a cursor-open command; the processing of these commands is "nested" within the processing of the cursor-open command that returned the cursor rows. Before sending a nested cursor command, the application must call ct_fetch to retrieve at least one cursor row.

See "Nested cursor-update or cursor-delete commands" on page 122 and "Nested cursor-close commands" on page 124.

Client-Library's browse mode feature also allows an application to update a table while fetching from it. However, browse mode requires two connections to the server. For a description of this feature, see the "Browse Mode" topics page in the *Open Client Client-Library/C Reference Manual*.

Commands executed using a different command structure

While fetching the rows returned by a cursor-open command, any command can be executed using a separate command structure. For example, the application might issue a select or an update command based on the cursor data. In this case, the application must completely process the results on the separate command structure before fetching the next cursor row or sending a nested cursor command. The application could also open a new cursor. In this case, the new cursor must be opened and its command handle must be ready to return cursor rows before the application can perform another operation on the original cursor.

As an example, consider an application that selects rows from an example table employee that contains the following data:

emp_fname	emp_lname	emp_id	mgr_id
Bob	Burnett	3349	4572
Alice	Williams	4572	5237
Thomas	Cooper	7028	3198
Samuel	Jones	6193	4572
Jennifer	Uribe	0969	4572
Joachin	Palmer	3198	4572

emp_fname	emp_lname	emp_id	mgr_id
Jerry	Howe	5939	5237
George	Latimer	5237	NULL
...	...	...	...

Here, emp_id is the employee ID number and mgr_id specifies the employee ID number of each employee's manager. One of the application requirements is that for each fetched employee row, the application must issue another query to find out which employees work for the last-fetched employee.

If the application uses a Client-Library cursor to select rows from the employee table, it could send the second query by using a separate CS_COMMAND structure. If the application was not using cursors, it would have to issue the second query by using a second connection to the server, or wait until it had processed all the results from the original query to send a new command over the same connection.

Performance issues when using Client-Library cursors

In general, a Client-Library cursor performs worse than an equivalent select statement that is executed using a language or RPC command. An application that does not require the special benefits listed above achieves higher performance using language commands or RPC commands.

However, cursors may improve performance when the application would otherwise require several connections or some sort of row-buffering mechanism to accomplish the same task.

Using Client-Library cursors

A typical application uses the steps below to declare and open a Client-Library cursor.

1 Send a cursor-declare command.

For cursors declared with a select statement:

- ct_cursor(CS_CURSOR_DECLARE)
- ct_param or ct_setparam to define host variable formats

- ct_send (if not batching commands)
- ct_results, in a loop (if not batching commands)

For cursors declared with an execute statement:

- ct_cursor(CS_CURSOR_DECLARE)
- ct_send (if not batching commands)
- ct_results, in a loop (if not batching commands)

For cursors declared with a prepared dynamic SQL statement:

- ct_dynamic(CS_CURSOR_DECLARE)
- (Optional) ct_cursor(CS_CURSOR_OPTION)
- ct_send
- ct_results, in a loop
- If a cursor is declared with a ct_cursor command, the commands in steps 1, 2, and 3 can be batched: they can be sent to the server with a single call to ct_send.

2 (Optional) Send a cursor-rows command.

- ct_cursor(CS_CURSOR_ROWS)
- ct_send (if not batching commands)
- ct_results, in a loop (if not batching commands)

3 Send a cursor-open command.

- ct_cursor(CS_CURSOR_OPEN)
- ct_param or ct_setparam to pass parameter values
- ct_send
- ct_results, called in a standard results loop.

A successful open command returns a CS_CURSOR_RESULT result set. If batching commands, several calls to ct_results are required (to retrieve the status results from the batched commands) before the cursor rows are available.

4 Process cursor rows.

- ct_bind to bind to cursor rows
- ct_fetch or ct_scroll_fetch (called in a loop to retrieve each row)

- New commands can be sent inside the ct_fetch or ct_scroll_fetch loop, after at least one row has been fetched. See “Step 4: Process cursor rows” on page 122.

5 Close the cursor.

- ct_cursor(CS_CURSOR_CLOSE)
- ct_send
- ct_results

An application can close and deallocate the cursor with one command by setting the CS_DEALLOC bit in the ct_cursor *option* parameter when defining the cursor-close command. In that case, the step 6 is unnecessary.

6 Deallocate the cursor.

- ct_cursor(CS_CURSOR_DEALLOC)
- ct_send
- ct_results

Each step in the process above sends one Client-Library cursor command to the server. After sending each command, the application must handle the results with ct_results. Code your application to handle the results of a cursor command with a standard results loop, as discussed in “Structure of the basic loop” on page 90.

Step 1: Declare the cursor

There are three types of cursor-declare commands. Each one executes the cursor’s select statement differently:

- The cursor executes a select statement directly.

  The application calls ct_cursor and passes the select statement as the ct_cursor *text* argument.

- The cursor executes a stored procedure.

  The select statement is executed by a stored procedure that has been created ahead of time, either by the application itself or by the application administrator. To declare the cursor, call ct_cursor and pass, as the *text* argument, an execute statement that invokes the procedure. Cursors can be declared only on a stored procedure that contains a single select statement.

- The cursor executes a prepared dynamic SQL statement.

  The application calls ct_dynamic(CS_PREPARE) to create a prepared statement that executes the select statement. Then the application calls ct_dynamic(CS_CURSOR_DECLARE) and passes the statement identifier as the ct_dynamic *id* argument.

Declaring a cursor to directly execute a select statement

To create a cursor or scrollable cursor that directly executes a select statement, call ct_cursor with *type* as CS_CURSOR_DECLARE and *text* as a select statement.

A simple cursor declaration

The following code declares a Client-Library cursor. Return code checking is omitted for simplicity:

```
CS_CHAR body[1024];
strcpy(body, “select * from titles for read only”);
 ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          “a cursor”, CS_NULLTERM,
          body, CS_NULLTERM, CS_UNUSED);
```

The following code declares a Client-Library scrollable cursor. Return code checking is omitted for simplicity:

```
CS_CHAR body[1024];
strcpy(body, “select * from titles”);
 ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          “s cursor”, CS_NULLTERM,
          body, CS_NULLTERM, CS_SCROLL_CURSOR);
```

Declaring a cursor that takes parameters

The select statement can also contain host language variables of the form *@variable_name* to indicate where parameters will be substituted in the statement when the cursor is opened. Adaptive Server Enterprise allows variables to substitute for values in the cursor’s where clause. For example, the following statement could be used to declare a cursor that takes a variable int value:

```
SELECT title_id, title, price FROM titles
 WHERE total_sales > @sales_val
```

In this case, you must specify the parameter format by calling ct_param or ct_setparam with a NULL *data* pointer after declaring the cursor. Each time the cursor is opened, the application supplies parameter values by calling ct_param or ct_setparam again. This case is demonstrated by the example below:

```
 CS_CHAR     body[1024];
 CS_DATAFMT  intfmt;
```

```
 CS_INT      sales_val;
  strcpy(body, “select title_id, title, price from
       titles where total_sales > @sales_val
       for read only”);
 ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          “a cursor”, CS_NULLTERM,
          body, CS_NULLTERM, CS_UNUSED);
 ... error checking deleted ...

(CS_VOID)memset(&intfmt, 0,  sizeof(intfmt));
/*
 ** Define the format of @sales_val.
 */
 intfmt.datatype = CS_INT_TYPE;
 intfmt.maxlength = CS_SIZEOF(CS_INT);
 intfmt.status = CS_INPUTVALUE;
ret = ct_param(cmd, &intfmt, (CS_VOID *)NULL,
              CS_UNUSED, 0);
 ... error checking deleted ...
ret = ct_cursor(cmd, CS_CURSOR_OPEN, NULL,
               CS_UNUSED, NULL, CS_UNUSED,
               CS_UNUSED);
 ... error checking deleted ...
/*
 ** Supply a value for @sales_val. intfmt fields
 ** were set above.
 */
sales_val = 1;
 ret = ct_param(cmd, &intfmt,
             (CS_VOID *)&sales_val, CS_UNUSED, 0);
 ... error checking deleted ...
/*
 ** Send the batched cursor declare and open
 ** commands.
 */
 ret = ct_send(cmd);
 ... error checking deleted ...
```

Specifying which columns can be updated

For applications that connect to Adaptive Server Enterprise, use the for read only or for update of clauses in the select statement to specify which columns, if any, will be updated. In the ct_cursor(CS_CURSOR_DECLARE) call, pass the ct_cursor *option* parameter as CS_UNUSED to indicate that the server should decide which columns can be updated. For example, a cursor declared with this following statement allows updates of the price column:

```
SELECT title_id, title, price FROM titles
 FOR UPDATE OF price
```

Other servers, such as custom Open Servers, may not recognize or use the for read only or for update of clauses in the select statement. These servers require the client application to indicate which columns are to be updated with separate calls to ct_param or ct_setparam. For details, see the reference page for ct_cursor in the *Open Client Client-Library/C Reference Manual*.

Declaring a cursor to execute a stored procedure

You can declare cursors to execute a stored procedure that in turn executes a single select statement. You create this style of cursor by calling ct_cursor with *type* as CS_CURSOR_DECLARE and *text* as an execute statement that invokes the procedure.

For example, the select statement in the example above could be invoked by a stored procedure:

```
CREATE PROCEDURE titlecursorproc
  @sales_val INT
 AS
  SELECT title_id, price, title FROM titles
  WHERE ( total_sales > @sales_val )
  FOR READ ONLY
```

For Client-Library cursors that execute an Adaptive Server Enterprise stored procedure, you do not use host language variables and do not define any variable formats with ct_param—the server determines parameter formats from the declaration of the stored procedure. The steps required to declare and open the cursor are otherwise similar to those illustrated under “Declaring a cursor that takes parameters” on page 114. The example below shows how to declare and open a Client-Library cursor on the *titlecursorproc* stored procedure:

```
CS_CHAR     body[1024];
 CS_DATAFMT  intfmt;
 CS_INT      sales_val;
strcpy(body, “EXECUTE titlecursorproc”);
 ret = ct_cursor(cmd, CS_CURSOR_DECLARE,
          “a cursor”, CS_NULLTERM,
          body, CS_NULLTERM, CS_UNUSED);
 ... error checking deleted ...
ret = ct_cursor(cmd, CS_CURSOR_OPEN, NULL,
               CS_UNUSED, NULL, CS_UNUSED,
               CS_UNUSED);
 ... error checking deleted ...
/*
 ** Supply a value for the @sales_val parameter for
```

```
 ** titlecursorproc.
 */
 (CS_VOID)memset(&intfmt, 0,  sizeof(intfmt));
 intfmt.datatype = CS_INT_TYPE;
 intfmt.maxlength = CS_SIZEOF(CS_INT);
 intfmt.status = CS_INPUTVALUE;
sales_val = 1;
 ret = ct_param(cmd, &intfmt,
              (CS_VOID *)&sales_val, CS_UNUSED, 0);
 ... error checking deleted ...
/*
 ** Send the batched cursor declare and open
 ** commands.
 */
 ret = ct_send(cmd);
 ... error checking deleted ...
... results processing deleted ...
```

Declaring a cursor to execute a prepared dynamic SQL statement

You can declare cursors on a prepared dynamic SQL statement that executes a single select statement. For example, you could prepare a statement to execute the select statement below:

```
SELECT title_id, title, price FROM titles
 WHERE total_sales > ? FOR READ ONLY
```

The "?" character (the dynamic parameter marker) is a placeholder for a parameter value that will be provided when the cursor is opened. Dynamic SQL statements are created by sending a ct_dynamic(CS_PREPARE) command to the server and handling the results. See "Step 1: Prepare the statement" on page 134 for details.

After preparing the statement, the application can call ct_dynamic with *type* as CS_CURSOR_DECLARE and *id* as the statement identifier.

Use the for read only or for update of clauses in the select statement to specify which columns, if any, to be updated. If the statement does not have one of these clauses, the application can call ct_cursor(CS_CURSOR_OPTION) immediately after calling ct_dynamic to initiate the cursor-declare command.

You cannot batch the ct_dynamic cursor-declare command c with ct_cursor cursor-rows or ct_cursor cursor-open commands.

The following example fragment shows how to declare and open a cursor with a prepared statement:

```
  /*
  ** Prepare the statement.
  */
  strcpy(body, “SELECT title_id, title, price FROM titles
                WHERE price > ? FOR READ ONLY”);
  strcpy(stmt_id, "dyn_a");
  retcode = ct_dynamic(cmd, CS_PREPARE, stmt_id, CS_NULLTERM,
             body, CS_NULLTERM);
  if (retcode != CS_SUCCEED)
  {
    ex_error("DoCursor: ct_dynamic(prepare) failed");
    return retcode;
  }
if ((retcode = ct_send(cmd)) != CS_SUCCEED)
  {
    ex_error("DoCursor: ct_send() failed");
    return retcode;
  }

  ... ct_results() loop goes here. No fetchable results are
      returned ...
 /*
  ** Declare the cursor
  */
  retcode = ct_dynamic(cmd, CS_CURSOR_DECLARE,
                       stmt_id, CS_NULLTERM,
                       "cursor_a", CS_NULLTERM);
  if (retcode != CS_SUCCEED)
  {
    ex_error("DoCursor: ct_dynamic(cursor declare) failed");
    return retcode;
  }
if ((retcode = ct_send(cmd)) != CS_SUCCEED)
  {
    ex_error("DoCursor: ct_send() failed");
    return retcode;
  }

  ... ct_results() loop goes here. No fetchable results are
      returned by the cursor-declare command ...
```

Step 2: Set cursor rows

After a Client-Library cursor is declared, an application can call ct_cursor to specify a cursor-rows setting for the cursor. The value of the cursor-rows setting defines the number of rows that the server returns to Client-Library per internal fetch request, not the number of rows returned to an application per ct_fetch call. An internal fetch request is made when more rows are needed from the server to satisfy ct_fetch requests.

By default, the cursor-rows setting is 1. If the application does not send a cursor-rows command that precedes the cursor-open command, the cursor rows setting is 1. For cursors declared with ct_cursor commands, the cursor-rows command can be batched with the cursor-open command.

The cursor-rows settings determines how many rows Client-Library receives from the server in response to each internal Client-Library fetch request. For example, if cursor-rows is set to 5 and the application does not use array binding, Client-Library makes an internal fetch request when an application calls ct_fetch the first time, the sixth time, and so on.

To facilitate a multi-row return from ct_scroll_fetch, you must use a cursor-row setting greater than 1. For maximum efficiency, you must also use array binding. Your array bind count should be equal to the CS_CURSOR_ROWS value.

Note Array binding is required for ct_scroll_fetch if the CS_CURSOR_ROWS setting is greater than 1. Array binding can be used with both ct_fetch and ct_scroll_fetch. If CS_CURSOR_ROWS is set to the default value of 1, normal program variables may be used with either API call.

If you specify a cursor-rows setting greater than 1, Client-Library buffers also handles additional internal row fetches transparently. When an application calls ct_fetch to fetch a cursor row, Client-Library may read the row directly from the network, send an internal fetch request to the server to get more rows, or retrieve the row from an internal row buffer. Two situations require Client-Library to buffer cursor rows internally:

- When the application sends a nested cursor-update or cursor-delete command.
- When the application sends a command on a different command structure than the cursor's.

In these situations, Client-Library must read and buffer any unread rows to clear the connection for writing.

In general, a higher cursor-rows setting can benefit application performance when processing a read-only cursor. A higher cursor-rows setting decreases the number of network round trips required to fetch rows. However, if cursor-rows is set too high and Client-Library must buffer rows, the buffering overhead can outweigh the gains achieved by decreasing the number of round trips.

To minimize the likelihood that Client-Library will need to buffer rows, use array binding with an array size that matches the cursor-rows setting. See the reference page for ct_bind in the *Open Client Client-Library/C Reference Manual*.

Step 3: Open the cursor

You initiate a cursor-open command by calling ct_cursor(CS_CURSOR_OPEN). If the cursor requires input parameters, define them by calling ct_param or ct_setparam once for each parameter value. Parameter values are required if any of the following conditions are true:

- The body of the cursor is a SQL text string 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.

Applications that restore cursor-open commands should call ct_setparam rather than ct_param to specify parameter values for the cursor-open command. When ct_setparam is used, the application can change the parameter values for the restored cursor-open command. (See “Reopening a cursor” on page 121.)

Cursor command batching

Cursors declared with ct_cursor can be batched. The first time a cursor is opened, an application can send the cursor-declare, cursor-rows, and cursor-open commands with a single call to ct_send and process the results with a single results loop.

When a cursor is reopened, the application can batch a cursor-rows command with the cursor-open command.

Batching the commands reduces the number of network round trips required to open the cursor.

Reopening a cursor

After the results of a cursor-open command have been processed, the previous cursor-open command can be restored with a single ct_cursor call (with the syntax described below). The restore operation readies the command structure to send the previous cursor-open command. The following command information is restored:

- Any cursor-rows commands that were batched with the cursor-open command.
- Parameter values for the cursor-open command that were passed with ct_param.
- Bindings to parameter source variables that were established with ct_setparam. ct_send reads the current values when the restored command is sent.

Cursor-declare commands that were batched with the cursor-open command are not restored.

An application restores a cursor-open command by calling ct_cursor with *type* as CS_CURSOR_OPEN and *option* as CS_RESTORE_OPEN. Most applications use the program structure below to restore and send a cursor-open command.

```
 /*
 ** Assign new variables in the program variables
 ** bound with ct_setparam.
 */
 ... assignment statement for each parameter
      source variable ...
ct_cursor(CS_CURSOR_OPEN, ..., CS_RESTORE_OPEN)
 ct_send
 ... handle cursor results ...
```

You can also reopen a cursor by initiating a new cursor open command (preceded by a cursor-rows command if necessary). However, applications that restore the previous command can eliminate several Client-Library calls.

Step 4: Process cursor rows

Cursor results should be processed by calling ct_results in a standard loop structure (see "Structure of the basic loop" on page 90). Cursor rows are available when ct_results returns with *result_type* equal to CS_CURSOR_RESULT. Cursor rows are handled like any other fetchable result set. (See "Processing cursor results" on page 93.)

The difference from other result types is that the application can issue new commands while fetching cursor rows. These commands can be either of two types:

- Nested cursor commands – cursor-close, cursor-delete, or cursor-update commands executed using the command structure that controls the cursor, or
- All other commands – any command executed using a separate command structure.

Nested cursor-update or cursor-delete commands

While processing a cursor result set, an application can update or delete any previously fetched row in the cursor result set. The modification is propagated back to the base tables from which the cursor result set derives.

A cursor update command is initiated by calling ct_cursor with *type* as CS_CURSOR_UPDATE, *name* as the name of the base table, and *text* as a SQL update clause. For example, the following call builds a command to update a row in the *authors* table of the pubs2 database:

```
ret_code = ct_cursor(cmd, CS_CURSOR_UPDATE,
     "authors", CS_NULLTERM, "update authors \
     set au_lname = 'Barr'", CS_NULLTERM,
     CS_UNUSED);
ct_send(cmd);
ct_results(cmd, &res_type);
```

The cursor update can update columns from one table only. Separate commands can be sent to update columns from more than one table.

A cursor-delete command is initiated by calling ct_cursor with *type* as CS_CURSOR_DELETE, *name* as the name of the base table from which to delete the row, and *text* as NULL.

After sending a cursor-update or cursor-delete command, the application must completely process the update or delete operation before calling ct_fetch again.

Key columns

An application should avoid updating columns that are part of the cursor result set's primary key. ct_describe sets the CS_KEY bit in the *datafmt.status* field to indicate that a column is a primary key for the result set.

Redirected updates or deletes

By default, a cursor-update or a cursor-delete affects the last-fetched row. However, you can redirect the update or delete to affect any previously fetched row. Redirected updates or deletes are most commonly used by applications that perform array binding to process the cursor rows.

Cursor updates or deletes are redirected by calling ct_keydata before sending the command.

For an application that redirects updates, you must ensure that the command structure's CS_HIDDEN_KEYS property is CS_TRUE before opening the cursor. (Use ct_cmd_props to set the property for the command structure before opening the cursor, or use ct_con_props to set it at the connection level before allocating command structures.) CS_HIDDEN_KEYS determines whether the cursor's hidden-key columns are exposed to the application.

A *hidden-key column* is returned with a cursor's result set but was not specified in the cursor's select list. ct_describe sets the CS_HIDDEN bit in the *datafmt.status* field to indicate that a column was not part of the cursor's select list.

Hidden-key columns provide additional information that the server requires to find the destination rows for cursor updates and deletes. Normally, Client-Library handles these additional columns internally and does not expose them to the application. However, applications that perform redirected updates or deletes must handle the hidden-key columns explicitly.

To redirect a cursor update or delete, an application must call ct_keydata and specify values for every column in the row that is a version key or a primary key (including hidden columns). These terms are explained below:

- A *primary-key* column is part of the primary key for the cursor result set. ct_describe sets the CS_KEY bit in the *datafmt.status* field to indicate that a column is a primary key for the result set.
- A *version-key* column is a real table column (not an expression in the select list) that is not part of the primary key for the cursor result set. ct_describe sets the CS_VERSION_KEY bit in the *datafmt.status* field to indicate that a column is a version key for the result set.

A hidden-key column can be either a primary-key column or a version-key column.

Applications that redirect cursor updates must be coded according to the rules below:

- Make sure the CS_HIDDEN_KEYS property is CS_TRUE for the command structure before the cursor is opened.
- When processing the cursor rows, call ct_describe to obtain CS_DATAFMT information for all cursor columns, including hidden columns. Save the information for use with later updates.
- In interactive applications, use the CS_HIDDEN bit in the CS_DATAFMT *status* field to determine whether a column should be displayed.
- When retrieving rows, save column values for all rows that can be updated. These values are required as input to ct_keydata.
- To update a previously fetched row, call ct_keydata for every column in the row whose matching CS_DATAFMT *status* field has either the CS_KEY or CS_VERSION_KEY bit set.
- Avoid updating key columns. Check the CS_KEY bit in the CS_DATAFMT status field to determine whether a column is a key column.

Nested cursor-close commands

An application can close a cursor before fetching all its rows by sending a cursor-close command and handling the results. See “Step 5: Close the cursor” on page 125 for more details.

Closing a cursor is preferred over calling ct_cancel to discard unwanted cursor rows for the following reasons:

- Calling ct_cancel(CS_CANCEL_ALL) or ct_cancel(CS_CANCEL_ATTN) can cause a connection’s cursors to go into an undefined state.
- Calling ct_cancel(CS_CANCEL_CURRENT) can waste network bandwidth. This call causes Client-Library to fetch the remaining rows over the network and discard them.

Sending commands on a different command structure

An application can send commands, which are unrelated to the original cursor, on a separate command structure while fetching the rows from the original cursor.

For example, the application might issue a select or an update based on the cursor data. In this case, the application must completely process the results of the new command before fetching the next cursor row. The application could also open a new cursor. In this case, the new cursor must be opened and its command handle must be ready to return cursor rows before the application can perform another operation on the original cursor.

Step 5: Close the cursor

An application initiates a cursor-close command by calling ct_cursor with *type* as CS_CURSOR_CLOSE. If the application will not use the cursor again, it can close and deallocate the cursor with one command by passing ct_cursor's *option* parameter as CS_DEALLOC. Otherwise, *option* should be CS_UNUSED.

Step 6: Deallocate the cursor

An application initiates a cursor-deallocate command by calling ct_cursor with type as CS_CURSOR_DEALLOC. If an application does not explicitly deallocate a cursor, it is deallocated when the application disconnects.

Client-Library cursor properties

Once a Client-Library cursor is declared, it is associated with only one command structure. Applications can obtain information about the cursor associated with a command structure by calling ct_cmd_props to retrieve the following properties:

- CS_CUR_ID – contains the cursor's server identification number. A cursor's identification number can be retrieved after calling ct_cmd_props(CS_CUR_STATUS) to confirm that a cursor exists in a particular command space.
- CS_CUR_NAME – contains the cursor's name. An application can use the CS_CUR_NAME property to retrieve a cursor's name any time after its ct_cursor(CS_CURSOR_DECLARE) call returns CS_SUCCEED.

- CS_CUR_ROWCOUNT – contains the cursor-rows setting. This setting is the number of rows returned to Client-Library per internal fetch request. A cursor's row count can be retrieved after calling ct_cmd_props(CS_CUR_STATUS) to confirm that a cursor exists in a particular command space.
- CS_CUR_STATUS – indicates the cursor status. An application can use the CS_CUR_STATUS property to determine:
  - Whether a cursor exists within a command space
  - Whether the cursor is open
  - Whether the cursor can be used for updates
  - Whether the cursor is read-only, has sensitivity, and has scrollability.

  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_HAVE_CUROPEN – indicates whether the command structure has a cursor-open command that can be restored. See "Reopening a cursor" on page 121.

All of these properties are retrieve-only command structure properties whose values can be retrieved by calling ct_cmd_props. See the reference page for ct_cmd_props in the *Open Client Client-Library/C Reference Manual*.

CHAPTER 8 Using Dynamic SQL Commands

This chapter explains Dynamic SQL, including:

Dynamic SQL overview

Dynamic SQL is the process of generating, preparing, and executing SQL statements at run time using commands initiated by Client-Library's ct_dynamic routine.

Dynamic SQL is primarily useful for precompiler support, but it can also be used by interactive applications.

Client-Library and Adaptive Server Enterprise allow two methods of dynamic SQL command execution:

- Execute-immediate – the client application sends the server a ct_dynamic command that executes a literal statement. This is essentially the same process as sending a language command, but with more restrictions. (See "Language commands" on page 76.)
- Prepare-and-execute – the client application sends the server a sequence of server commands that prepares a statement and executes it one or more times. The application can send additional commands to query the server for the formats of the statement's input parameters and the result set that it returns.

With the prepare-and-execute method, the client application sends a ct_dynamic(CS_PREPARE) command to the server to create a *prepared statement*. A prepared statement is similar to an Adaptive Server Enterprise stored procedure. When either is created, the server checks the SQL statement syntax, builds an optimized query plan, and stores the query plan in preparation for later execution. The key differences are as follows:

- The prepared statement is dropped automatically when the client program disconnects, while the stored procedure is not.
- The prepared statement is referenced by an identifier that is visible only to the connection that created the statement, while a stored procedure name is visible to any client connection. However, the procedure's permissions may restrict which users can execute it.
- The client program can easily determine the input (parameter) and output (result) column formats for a prepared statement without executing it.

Benefits of dynamic SQL

Using dynamic SQL commands, an application can prepare a "generic" SQL statement once and execute it multiple times. Statements can also contain markers for parameter values to be supplied at execution time, so that the statement can be executed with varying inputs.

Limitations of dynamic SQL

Dynamic SQL has some significant limitations.

Performance of dynamic SQL commands

A dynamic SQL implementation of an application generally performs worse than an implementation where permanent Adaptive Server Enterprise stored procedures are created and the client program invokes them with RPC commands.

When you create Adaptive Server Enterprise stored procedures for an application, SQL statement compilation and optimization are performed once when the procedure is created. On the other hand, a dynamic SQL application incurs compilation and optimization overhead every time the client program runs. A dynamic SQL implementation also incurs database space overhead because each instance of the client program must create separate compiled versions of the application's prepared statements. In contrast, when you design an application to use stored procedures and RPC commands, all instances of the client program can share the same stored procedures.

Adaptive Server Enterprise restrictions and database requirements

Adaptive Server Enterprise 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 deallocated. A prepared statement can be deallocated 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 Adaptive Server Enterprise and using dynamic SQL is subject to the restrictions of Adaptive Server Enterprise stored procedures. Some of the implications of this are:

- Temporary tables are destroyed when the prepared statement is deallocated.
- Parameters of text and image datatypes are not supported.
- The maximum number of parameters supported is 255.
- If the dynamic SQL statement itself executes a stored procedure (with a Transact-SQL execute statement), output parameter values and the return status are unavailable to the client application.
- The datatype of the parameters represented by placeholders must be known at parsing time. The following statements are not valid:

```
? <op> ?, (? is null)
```

```
CONVERT(<type>, ?)
```

See the *Transact-SQL Users Guide* for a complete discussion of stored procedures.

Alternatives to dynamic SQL

Developers who learn Sybase after learning another DBMS system should not confuse Sybase's dynamic SQL implementation with that of other vendors. With Adaptive Server Enterprise, most command types are "dynamic." The closest analogy that Adaptive Server Enterprise offers to "static SQL commands" are stored procedures. However, any client application can invoke a stored procedure, as long as the procedure's permissions allow the client program's user to execute it. Other DBMS systems may limit the scope of a precompiled static SQL command to the precompiled application.

For Adaptive Server Enterprise applications, many tasks that require you to use dynamic SQL with another DBMS can be implemented with Client-Library command types other than dynamic SQL. For example:

- For an application that must execute SQL statements whose text is not known prior to runtime, you can code the client program to define language commands by calling ct_command. This method is appropriate for commands that are only executed once or a small number of times.
- For an application that must execute commands whose text is known before runtime and where performance is important, you can create an Adaptive Server Enterprise stored procedure and code the client program to invoke the procedure with RPC commands (defined with ct_command).
- For an application that must interactively define and open cursors, you can code the client program to define the cursor-declare commands with ct_cursor.

Using the execute-immediate method

The execute-immediate method executes a single SQL statement by sending a single command to the server.

When to use the execute-immediate method

A dynamic SQL statement can be executed immediately only if it meets the following criteria:

- It does not return fetchable data (it is not a select statement).

- It does not contain placeholders for parameters (indicated by a question mark (?) in the text of the statement).

  Dynamic parameter markers act as placeholders that allow users to specify actual data to be substituted into a SQL statement at run time.

Generally, you should use the execute-immediate method when the application executes a statement only once. Using the execute-immediate method, an application can execute a statement more than once, but this method incurs the overhead associated with repeated statement preparations.

Coding an execute-immediate command

To execute a dynamic SQL statement using the execute-immediate method, code your application to:

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, *buffer* as the address of the string containing the SQL statement, and *id* as NULL.
3. Call ct_send to send the command to the server.
4. Call ct_results in a standard loop, as described in "Structure of the basic loop" on page 90. The value of the **result_type* parameter indicates whether the command succeeded (CS_CMD_SUCCEED) or failed (CS_CMD_FAIL).

Using the prepare-and-execute method

For the prepare-and-execute method, the server performs the compilation and execute operations separately in response to distinct commands.

When to use prepare-and-execute method

An application must use this method if the dynamic SQL statement meets any of the following criteria:

- It returns data.
- It contains placeholders for values to be supplied at execution time, represented by a question mark (?) character in the text of the statement.

An application should use this method if it will execute the statement multiple times because it incurs the overhead associated with statement preparation only when it first prepares the statement. Each subsequent execution of the statement does not incur the cost of recompiling the statement.

The prepare-and-execute method offers the following advantages over the execute-immediate method:

- select statements can be executed.
- Performance is better when statements are executed more than once.
- The statement can take parameters whose values can change each time the statement executes.

Program structure for the prepare-and-execute method

Most applications will use the steps below to prepare and execute a dynamic SQL statement:

1. Prepare the dynamic SQL statement.
   - ct_dynamic(CS_PREPARE)
   - ct_send
   - ct_results, in a loop

   The prepare command returns no fetchable results.
2. (Optional) Get a description of the parameters required to execute the prepared statement.
   - ct_dynamic(CS_DESCRIBE_INPUT)
   - ct_send
   - ct_results, in a loop

   ct_results returns with a *result_type* of CS_DESCRIBE_RESULT to indicate that the parameter descriptions are available.
3. (Optional) Get a description of the result columns returned by the prepared statement.

- ct_dynamic(CS_DESCRIBE_OUTPUT)
- ct_send
- ct_results, in a loop

ct_results returns with a *result_type* of CS_DESCRIBE_RESULT to indicate that the description is available.

4 Execute the prepared statement or declare and open a cursor on the prepared statement.

To execute the prepared statement (without a cursor):

- ct_dynamic(CS_EXECUTE).
- If necessary, define parameter values with ct_param, ct_setparam, ct_dyndesc, or ct_dynsqlda.
- ct_send.
- ct_results, in a loop. Fetchable results may require processing.

For a description of how to execute a prepared statement with a cursor, see “Using Client-Library cursors” on page 111.

5 Deallocate the prepared statement.

If a cursor is declared on the statement, first close and deallocate the cursor:

- ct_cursor(CS_CURSOR_CLOSE, CS_DEALLOC) or, if the cursor is not open, ct_cursor(CS_CURSOR_DEALLOC)
- ct_send
- ct_results, in a loop
- Initiate and send a command to deallocate the prepared statement:
- ct_dynamic(CS_DEALLOC)
- ct_send
- ct_results, in a loop

The deallocate command returns no fetchable results.

Each step in the process above sends one dynamic SQL command to the server. After sending each command, the application must handle the results with ct_results. Code your application to handle the results of a dynamic SQL command with a standard results loop, as discussed in “Structure of the basic loop” on page 90.

Step 1: Prepare the statement

To initiate a command that prepares a dynamic SQL statement, an application calls ct_dynamic with *type* as CS_PREPARE, *id* as a character string statement identifier, and *buffer* as the statement to prepare. For example:

```
char     *query = “select type, title, price \
         from titles \
         where title_id = ?”
ct_dynamic(cmd, CS_PREPARE, “myid”, CS_NULLTERM,
     query, CS_NULLTERM);
```

Statement identifiers must be unique among other dynamic SQL statements prepared on the same connection.

ct_send sends the prepare command to the server, and a standard ct_results loop handles the results.

Step 2: Get a description of command inputs

After a statement is prepared, the application can send a describe-input command to the server to obtain a description of any parameters that are required to execute the statement. This description includes the number of input values, as well as their datatypes, 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.

Initiating a describe-input command

To initiate a describe-input command, the application calls ct_dynamic with *type* as CS_DESCRIBE_INPUT and *id* as the statement identifier. ct_send sends the command to the server, and a standard ct_results loop handles the results.

Processing parameter descriptions

ct_results returns with *result_type* of CS_DESCRIBE_RESULT to indicate that the input parameter formats are available. Applications can retrieve the parameter formats in one of two ways:

- With ct_res_info and ct_describe

The application calls ct_res_info to determine the number of parameters; then, for each parameter, it calls ct_describe to initialize a CS_DATAFMT structure with a description of the parameter.

Typically, an application using this method keeps the CS_DATAFMT structures in an array or list for use with later calls to ct_param or ct_setparam.

- With ct_dyndesc or ct_dynsqlda

  Both these routines allow the application to retrieve formats into a structure that can later be used to pass parameters for the command that executes the statement. Both of these routines:

  - Retrieve a description of the input parameters required to execute a prepared dynamic SQL statement
  - Define input parameter values for the execution of a prepared statement
  - Retrieve a description of the data results that will be returned when a prepared statement is executed
  - Retrieve data values in the result set returned by the execution of a prepared statement

  The differences between the routines are:

  - ct_dynsqlda – retrieves formats into a SQLDA structure. The application must allocate the memory for this structure before retrieving formats into it. ct_dynsqlda requires only a single call to perform each operation.
  - ct_dyndesc – retrieves formats into an internal Client-Library data structure that is hidden from the application. ct_dyndesc requires several calls to perform a single operation.

  ct_dyndesc and ct_dynsqlda both call ct_res_info and ct_describe internally. When used to pass parameter values, ct_dyndesc and ct_dynsqlda both call ct_param internally.

Step 3: Get a description of command outputs

The application can send a describe-output command to get the format of the result columns that will be returned when the prepared statement executes. For example, an interactive application might use a describe-output command to determine the number and format of result columns to prepare data structures that are used when displaying the query results. A describe-output command allows the application to determine the results format without executing the prepared statement.

Note A single dynamic SQL batch may contain multiple SQL statements. The description of the prepared statement output, however, only describes the first resultset. You will receive full descriptions of each resultset only when the dynamic SQL statement is executed.

Initiating a describe-output command

To initiate a describe-output command, the application calls ct_dynamic with *type* as CS_DESCRIBE_OUTPUT and *id* as the statement identifier. ct_send sends the command to the server, and a standard ct_results loop handles the results.

Processing column descriptions

ct_results returns with *result_type* of CS_DESCRIBE_RESULT to indicate that the result column formats are available. Applications can retrieve the column formats in one of two ways:

- With ct_res_info and ct_describe

  The application calls ct_res_info to get the number of columns, then, for each parameter, calls ct_describe to initialize a CS_DATAFMT structure with a description of the column.

  Typically, an application using this method maintains an array or list of CS_DATAFMT structures for use with later calls to ct_bind.

- With ct_dyndesc or ct_dynsqlda

  Both these routines allow the application to retrieve formats into a structure that can later be used to retrieve row data when the prepared statement executes.

- ct_dynsqlda retrieves formats into a SQLDA structure. The application must allocate the memory for this structure before retrieving formats into it.
- ct_dyndesc retrieves formats into an internal Client-Library data structure that is hidden from the application.

ct_dyndesc and ct_dynsqlda both call ct_res_info and ct_describe internally. When used to retrieve row data, ct_dyndesc and ct_dynsqlda both call ct_bind internally.

Step 4: Execute the prepared statement

To initiate a command to execute the prepared statement, the application calls ct_dynamic with *type* as CS_EXECUTE and *id* as the statement identifier. The application must define any parameters required to execute the prepared statement. Parameter values can be defined in one of several ways:

- By calling ct_param once for each parameter. ct_param and ct_setparam offer the best performance. ct_param does not allow the application to change parameter values before resending the command.
- By calling ct_setparam once for each parameter. ct_setparam takes pointers to parameter source values. This method is the only one that allows parameter values to be changed before resending the command.
- By calling ct_dyndesc several times to allocate a dynamic descriptor area, populate it with data values, and apply it to the command. ct_dyndesc(CS_USE_DESC) calls ct_param internally.
- By calling ct_dynsqlda to apply the contents of a user-allocated SQLDA structure to the command. Note that ct_dynsqlda(CS_SQLDA_PARAM) calls ct_param internally.

The application can determine the number and format of a prepared statement's parameters by sending a describe-input command and handling the results before executing the prepared statement. See "Step 2: Get a description of command inputs" on page 134.

ct_send sends the command to the server, and a standard ct_results loop handles the results. Code your application to handle the results with a standard results loop, as discussed in "Structure of the basic loop" on page 90.

Step 5: Deallocate the prepared statement

Deallocating a prepared statement frees any resources associated with it. Explicit deallocation is optional; if the application does not explicitly deallocate prepared statements, the server deallocates them when the client program disconnects.

If a cursor is declared on the prepared statement, the application must first deallocate the cursor before deallocating the statement. See "Step 6: Deallocate the cursor" on page 125 for details.

To initiate a command to deallocate the prepared statement, the application calls ct_dynamic with *type* as CS_DEALLOC and *id* as the statement identifier. ct_send sends the command to the server, and a standard ct_results loop handles the results.

Dynamic SQL versus stored procedures

For improved performance compared to dynamic SQL, application designers can use Adaptive Server Enterprise stored procedures as an alternative where the application requirements allow it.

There are similarities between dynamic SQL and stored procedures:

- Creating a stored procedure is analogous to preparing a dynamic SQL statement.
- A stored procedure's input parameters serve the same purpose as dynamic parameter markers.
- Executing a stored procedure is equivalent to executing a prepared statement.

Stored procedures and dynamic SQL prepared statements offer identical functionality, with the following exceptions:

- Dynamic SQL allows retrieval of a prepared statement's parameter formats, while stored procedures do not. See "Step 2: Get a description of command inputs" on page 134.

- The format for stored procedure results cannot easily be determined programmatically without executing the procedure. Dynamic SQL allows retrieval of a prepared statement's result column formats without executing the statement. See "Step 3: Get a description of command outputs" on page 136.
- User-created stored procedures are permanent database objects, while prepared statements are automatically deallocated when the user disconnects from the server.

A dynamic SQL statement can be replaced by a stored procedure that returns the same results. For example, the following dynamic SQL statement queries the *pubs2..titles* table for books of a certain type in a certain price range:

```
select * from pubs2..titles
     where type = ?
     and price between ? and ?
```

Here, the dynamic SQL statement has dynamic parameter markers (?) for a *type* value and two *price* values.

You can create an equivalent stored procedure as follows:

```
create proc titles_type_pricerange
   @type char(12),
   @price1 money,
   @price2 money
 as
   select * from titles
     where
        type = @type
        and price between @price1 and @price2
```

When executed with the same input parameter values, the prepared statement and the stored procedure return the same rows. In addition, the stored procedure returns a return status result.

CHAPTER 9 Using Directory Services

This chapter describes how Client-Library applications can use a directory service.

Directory service overview

A **directory** stores information as *directory entries* and associates a logical name with each entry. Each directory entry contains information about some network entity, such as a user, a server, or a printer.

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

By default, Client-Library uses the Sybase interfaces file as the directory source. Sybase also provides directory drivers for several network-based directory services such as DCE's Cell Directory Service (CDS) and the Windows Registry service. For information about the directory drivers that are available on your platform, see the *Open Client and Server Configuration Guide for Microsoft Windows* and *Open Client and Server Configuration Guide for UNIX*.

How do applications use a directory service?

Information about Sybase servers is stored in the directory. When an application calls ct_connect to open a connection to a server, it passes the name of the server's directory entry as the ct_connect *server_name* parameter. ct_connect looks up the entry and retrieves the server's network address and any other information needed to establish the connection.

Applications can also search for available servers using Client-Library routines.

Searching the directory

Before an application can search a directory, it must have set up the Client-Library programming environment and allocated a CS_CONNECTION structure. See Chapter 1, "Getting Started with Client-Library" if you do not already know how to initialize Client-Library and allocate a connection structure.

Example code

The *usedir.c* sample program demonstrates how Client-Library applications perform a directory search. All of the code fragments in this chapter are taken from *usedir.c*.

Program structure

To perform directory search, code your application to follow the steps below:

1 Begin the search.

- ct_con_props to set directory service properties
- ct_callback to install a pointer to the application's directory callback in the connection structure

Execute application code to initialize a list or array that will collect directory objects

- ct_ds_lookup to begin the search

Note that instead of calling ct_callback here, the application could have installed the callback in the connection's parent context structure before allocating the connection. Then it would become the default directory callback for all connections allocated from the context.

2 Collect search results in the directory callback.

- (Optional) ct_ds_objinfo to inspect the object
- (Optional) ct_ds_dropobj to drop unwanted objects

Execute application code to collect directory objects with an application defined list or array.

During the directory search, ct_ds_lookup invokes the directory callback once for each entry that is found in the search.

3 Inspect the directory objects. For each directory object:

- ct_ds_objinfo to get the object's fully qualified name
- ct_ds_objinfo to get the number of attributes
- ct_ds_objinfo to get each attribute's metadata and values

4 Clean up.

For each object, ct_ds_dropobj to deallocate the directory object

Step 1: Starting the search

An application starts a directory search by initializing the application data structures that will hold the results, installing a directory callback, and calling ct_ds_lookup.

Initialize data structures

The example code in this chapter collects directory objects in a data structure called SERVER_INFO_LIST, which can be implemented as an array or list of CS_DS_OBJECT pointers.

The code calls the following example routines to collect directory object structures:

- sil_init_list – allocate and initialize an empty SERVER_INFO_LIST.

- sil_add_object – add a directory object to the end of a SERVER_INFO_LIST.
- sil_extract_object – given a 1-based index number, retrieve a directory object from the SERVER_INFO_LIST.
- sil_list_len – get the number of objects stored in a SERVER_INFO_LIST.
- sil_drop_list – deallocate a SERVER_INFO_LIST and all its constituents. Calls ct_ds_dropobj to deallocate each directory object in the list.

These routines simply manage a list of CS_DS_OBJECT pointers. Their implementation is not shown here, but complete code can be found in the *usedir.c* sample file in the Client-Library sample programs.

Setting directory service properties

Applications call ct_con_props to set directory service properties for a connection. Applications most commonly set the following properties to control a directory search:

- CS_DS_DITBASE – specifies the node in the directory where the search begins. DIT-base values must follow the syntax rules of the directory service. See the “Directory Services” topics page in the *Open Client Client-Library/C Reference Manual* for example DIT-base values.
- CS_DS_SEARCH – constrains the depth that the search descends beneath the DIT base. The possible values of CS_DS_SEARCH are as follows:

Value	Meaning
CS_SEARCH_ONE_LEVEL (default)	Search includes only the leaf entries that are immediate descendants of the node specified by CS_DS_DITBASE.
CS_SEARCH_SUBTREE	Search the entire subtree whose root is specified by CS_DS_DITBASE.

Note The DCE directory driver does not allow CS_DS_SEARCH to be set to a value other than the default, CS_SEARCH_ONE_LEVEL.

All directory service properties have a symbolic name that begins with “CS_DS”. See the “Properties” topics page in the *Open Client Client-Library/C Reference Manual* for a complete list of Client-Library properties.

Installing the directory callback

An application installs a directory callback by calling ct_callback with the *action* parameter as CS_SET, the *type* parameter as CS_DS_LOOKUP_CB, and *func* as the address of the applications directory callback routine.

A directory callback can be installed at the context level or the connection level. Connections that are allocated from a context inherit the context's directory callback. These steps install the callback at the connection level.

Coding of the callback routine is discussed under "Step 2: Collecting search results in the directory callback" on page 148.

Calling ct_ds_lookup

Applications begin a search by calling ct_ds_lookup with *action* as CS_SET.

ct_ds_lookup takes a CS_DS_LOOKUP_INFO structure as its *lookup_info* parameter that describes the search request. *lookup_info–>objclass* must point at a CS_OID structure that indicates the directory object class CS_OID_OBJSERVER. The other CS_DS_LOOKUP_INFO fields are currently unused and should be all passed as NULL.

ct_ds_lookup also takes a pointer to user-allocated data as its *userdata* parameter. When ct_ds_lookup invokes the application's directory callback, the callback receives the same pointer value as an input parameter.

Example code to start a directory search

The following fragment declares an application routine, get_servers, that searches for server directory class objects:

```
/*
** get_servers() -- Query the directory for servers and
**    get a list of directory objects that contain details
**    for each.
**
** Parameters
**   conn -- Pointer to allocated connection structure.
**   pserver_list -- Address of a pointer to a SERVER_INFO_LIST.
**       Upon successful return, the list will be initialized
**       and contain an object for each server found in the
**       search.
**
```

```
**       NOTE: The caller must clean up the list with sil_drop_list()
**       when done with it.
**
** Returns
**   CS_SUCCEED or CS_FAIL.
*
CS_RETCODE get_servers (conn, pserver_list)
CS_CONNECTION *conn;
SERVER_INFO_LIST **pserver_list;
{
  CS_RETCODE          ret;
  CS_INT              reqid;
  CS_VOID             *oldcallback;
  CS_OID              oid;
  CS_DS_LOOKUP_INFO   lookup_info;

  /*
  ** Steps for synchronous-mode directory searches:
  **
  ** 1. If necessary, initialize application specific data structures
  **    (Our application collects directory objects in *pserver_list).
  ** 2. Save the old directory callback and install our own.
  ** 3. Set the base node in the directory to search beneath
  **    (CS_DS_DITBASE property).
  ** 4. Call ct_ds_lookup to begin the search, passing any application
  **    specific data structures as the userdata argument.
  ** 5. Client-Library invokes our callback once for each found object
  **    (or once to report that no objects were found). The callback
  **    (directory_cb) receives pointers to found servers and appends
  **    each to the list of servers.
  ** 6. Check the return status of ct_ds_lookup.
  ** 7. Restore callbacks and properties that we changed.
  */

  /*
  ** Step 1. Initialize the data structure (*pserver_list).
  */
  ret = sil_init_list(pserver_list);
  if (ret != CS_SUCCEED || (*pserver_list) == NULL)
  {
    ex_error("get_servers: Could not initialize list.");
    return CS_FAIL;
  }

  /*
  ** Step 2. Save the old directory callback and install our own callback,
  **   directory_cb(), to receive the found objects.
  */
```

```
ret = ct_callback(NULL, conn, CS_GET,
                  CS_DS_LOOKUP_CB, &oldcallback);
if (ret == CS_SUCCEED)
{
  ret = ct_callback(NULL, conn, CS_SET,
                    CS_DS_LOOKUP_CB, (CS_VOID *)directory_cb);
}
if (ret != CS_SUCCEED)
{
  ex_error("get_servers: Could not install directory callback.");
  return CS_FAIL;
}

/*
** Step 3. Set the base node in the directory to search beneath
**    (the CS_DS_DITBASE connection property).
*/

ret = provider_setup(conn);
if (ret != CS_SUCCEED)
{
  ex_error("get_servers: Provider-specific setup failed.");
  return CS_FAIL;
}

/*
** Step 4. Call ct_ds_lookup to begin the search, passing the server list
**    pointer as userdata.
** Step 5. Client-Library invokes our callback once for each found object
**    (or once to report that no objects were found). Our callback,
**    directory_cb, will receive a pointer to each found server object
**    and appends it to the list.
** Step 6. Check the return status of ct_ds_lookup.
*/

/*

** Set the CS_DS_LOOKUP_INFO structure fields.
*/
lookup_info.path = NULL;
lookup_info.pathlen = 0;
lookup_info.attrfilter = NULL;
lookup_info.attrselect = NULL;

strcpy(oid.oid_buffer, CS_OID_OBJSERVER);
oid.oid_length = STRLEN(oid.oid_buffer);
lookup_info.objclass = &oid;
```

```
  /*
  ** Begin the search.
  */
  ret = ct_ds_lookup(conn, CS_SET, &reqid,
                   &lookup_info, (CS_VOID *)pserver_list);
  if (ret != CS_SUCCEED)
  {
    ex_error("get_servers: Could not run search.");
    return CS_FAIL;
  }

  /*
  ** Step 7.  Restore callbacks and properties that we changed.
  */
  ret = ct_callback(NULL, conn, CS_SET,
                  CS_DS_LOOKUP_CB, oldcallback);
  if (ret != CS_SUCCEED)
  {
    ex_error("get_servers: Could not restore directory callback.");
    return CS_FAIL;
  }

return CS_SUCCEED;

} /* get_servers() *
```

Step 2: Collecting search results in the directory callback

During the directory search, ct_ds_lookup invokes the directory callback once for each entry that is found in the search.

Defining the directory callback

A directory callback has the following prototype:

```
CS_RETCODE CS_PUBLIC
directory_cb (connection, reqid, status,
             numentries, ds_object, userdata)
CS_CONNECTION   *connection;
CS_INT          reqid;
CS_RETCODE      status;
```

```
CS_INT          numentries;
CS_DS_OBJECT    *ds_object;
CS_VOID         *userdata;
```

where:

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

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

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

  userdata provides a way for the callback to communicate with mainline code.

The callback can return CS_CONTINUE or CS_SUCCEED.

- A return of CS_SUCCEED truncates the search results: Client-Library discards any remaining directory objects and stops invoking the callback.
- A return of CS_CONTINUE causes Client-Library to invoke the callback with the next directory object in the search results.

Directory callback example

The following example fragment defines a directory callback. This callback:

- Confirms that the directory object pointer is valid.
- Adds the directory object to the application's list of servers by calling the sil_add_object example routine. When the mainline code calls ct_ds_lookup, it passes the address of an initialized SERVER_INFO_LIST as the ct_ds_lookup *userdata* parameter. The callback receives the same address as its own *userdata* parameter.
- If the list of servers is full, the callback returns CS_SUCCEED to truncate the search results. Otherwise, the callback returns CS_CONTINUE.

```
/*
** directory_cb() -- Directory callback to install in Client-Library.
**   When we call ct_ds_lookup(), Client-Library calls this function
**   once for each object that is found in the search.
**
**   This particular callback collects the objects in
**   the SERVER_INFO_LIST that is received as userdata.
**
** Parameters
**   conn -- The connection handle passed to ct_ds_lookup() to
**      begin the search.
**   reqid -- The request id for the operation (assigned by Client-Library).
**   status -- CS_SUCCEED when search succeeded (ds_object is valid).
**      CS_FAIL if the search failed (ds_object is not valid).
**   numentries -- The count of objects to be returned for the
**      search. Includes the current object. Can be 0 if search
**      failed.
**   ds_object -- Pointer to a CS_DS_OBJECT structure. Will
**      be NULL if the search failed.
**   userdata -- The address of user-allocated data that was
**      passed to ct_ds_lookup().
**
**      This particular callback requires userdata to be the
**      address of a valid, initialized SERVER_INFO_LIST pointer.
**      (SERVER_INFO_LIST is an application data structure defined
**      by this sample).
**
** Returns
**   CS_CONTINUE unless the SERVER_INFO_LIST pointed at by userdata fills
**     up, then CS_SUCCEED to truncate the search results.
*/

S_RETCODE          CS_PUBLIC
```

```
directory_cb(conn, reqid, status, numentries, ds_object, userdata)
CS_CONNECTION      *conn;
CS_INT              reqid;
CS_RETCODE          status;
CS_INT              numentries;
CS_DS_OBJECT       *ds_object;
CS_VOID            *userdata;
{
    CS_RETCODE          ret;
    SERVER_INFO_LIST   *server_list;

    if (status != CS_SUCCEED || numentries <= 0)
    {
        return CS_SUCCEED;
    }
    /*
    ** Append the object to the list of servers.
    */
    server_list = *((SERVER_INFO_LIST **)userdata);
    ret = sil_add_object(server_list, ds_object);
    if (ret != CS_SUCCEED)
    {

        /*
        ** Return CS_SUCCEED to discard the rest of the objects that were
        ** found in the search.
        */
        ex_error(
           "directory_cb: Too many servers! Truncating search results.");
        return CS_SUCCEED;
    }
    /*
    ** Return CS_CONTINUE so Client-Library will call us again if more
    ** entries are found.
    */
    return CS_CONTINUE;

} /* directory_cb() */
```

Step 3: Inspecting directory objects

Applications inspect the contents of a directory object with several calls to ct_ds_objinfo. To an application, a directory object consists of the following visible pieces:

- The object class that the object belongs to
- The object's fully qualified name
- A numbered set of attributes

An object's directory object class determines the object's attributes and the expected syntax (that is, datatype) for each attributes' values.

Although object attributes appear as a numbered set, an application should be coded to work independently of the order in which attributes are returned. A directory object class does not define an ordering of attributes, and most directory services do not guarantee that attributes will be ordered consistently for different directory objects in the same object class.

Most applications use a program structure similar to the one below to inspect a directory object:

```
ct_ds_objinfo to get the directory object class (optional)
 ct_ds_objinfo to get the fully qualified name
... application code to process fully qualified name ...
for each desired attribute type
  ct_ds_objinfo to get number of attributes
  i = 0
  while i is less than number of attributes
    i = i + 1
    ct_ds_objinfo to retrieve the metadata for attribute i
    compare returned attribute type to desired attribute type
    if attribute types match
       /* i is the number of the desired attribute */
       break while
    end if
  end while
  allocate sufficient space for attribute i's values
  ct_ds_objinfo to retrieve attribute i's values
  ... application code to process attribute values ...
end for
```

Attribute data structures

An attribute's metadata is represented by a CS_ATTRIBUTE structure:

```
typedef struct _cs_attribute
 {
     CS_OID          attr_type;
     CS_INT          attr_syntax;
     CS_INT          attr_numvals;
 } CS_ATTRIBUTE;
```

where:

- *attr_type* is a CS_OID structure that uniquely describes the type of the attribute. This field tells the application which of an object's attributes it has received.
- *attr_syntax* is a syntax specifier that tells how the attribute value is expressed. Attribute values are passed within a CS_ATTRVALUE union, and the syntax specifier tells which member of the union to use.
- *attr_numvals* tells how many values the attribute contains. This information can be used to size an array of CS_ATTRVALUE unions to hold the attribute values.

An attribute's value(s) are represented by a CS_ATTRVALUE union:

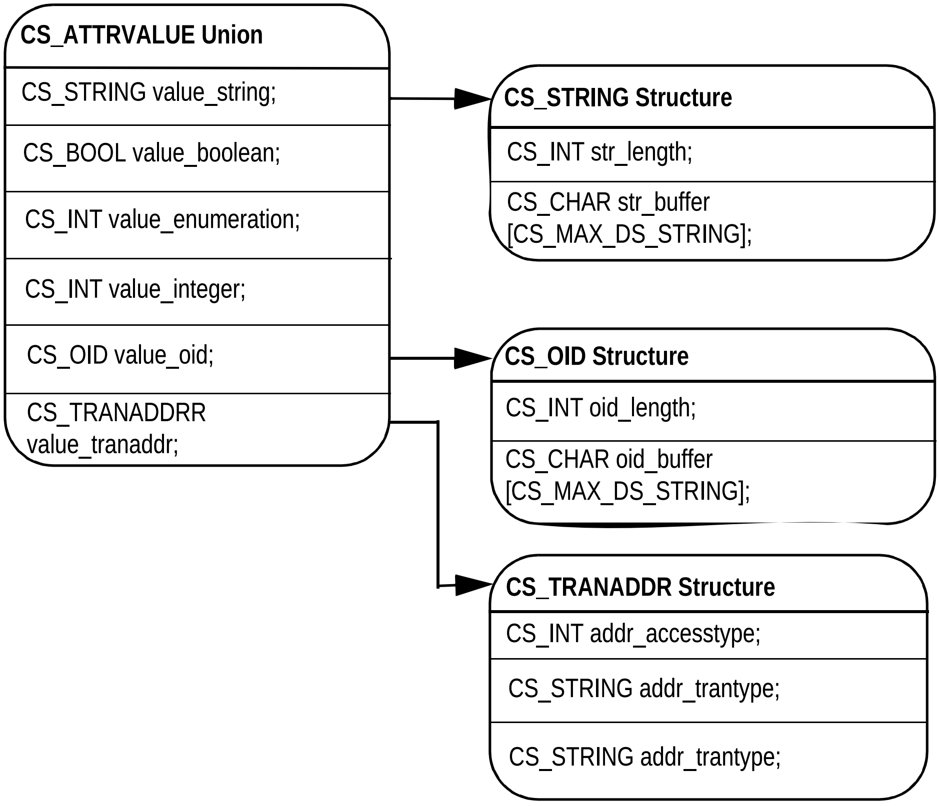
```
typedef struct _cs_ds_lookup_info
 {
     CS_OID          *objclass;
     CS_CHAR         *path;
     CS_INT          pathlen;
     CS_DS_OBJECT    *attrfilter;
     CS_DS_OBJECT    *attrselect;
 } CS_DS_LOOKUP_INFO;
```

Applications check the *syntax* field of the CS_ATTRIBUTE structure to determine which member of a CS_ATTRVALUE union contains the actual value. The following table shows the correspondence:

CS_ATTRIBUTE syntax specifier	CS_ATTRVALUE union member
CS_ATTR_SYNTAX_STRING	*value_string*
CS_ATTR_SYNTAX_BOOLEAN	*value_boolean*
CS_ATTR_SYNTAX_INTEGER	*value_integer*
CS_ATTR_SYNTAX_TRANADDR	*value_tranaddr*
CS_ATTR_SYNTAX_OID	*value_oid*

Figure 9-1 shows an exploded view of the CS_ATTRVALUE union and its member structures:

Figure 9-1: An exploded CS_ATTRVALUE union



Example code to inspect a directory object

The following fragment declares an example routine, show_server_info, that prints the contents of a directory object as text.

The code uses a static array, *AttributesToDisplay*, that lists the attribute types (as OID strings) for the attributes whose values should be retrieved, in the order that they should be printed.

For each row in *AttributesToDisplay*, the example retrieves the values for the attribute type (if any) and prints them.

```
/*
** AttributesToDisplay is a read-only static array used by
** the show_server_info() function. It contains the Object
** Identifier (OID) strings for the server attributes to
** display, in the order that they are to be displayed.
*/
typedef struct
{
   CS_CHAR             type_string[CS_MAX_DS_STRING];
   CS_CHAR             english_name[CS_MAX_DS_STRING];
} AttrForDisplay;
#define N_ATTRIBUTES 7
CS_STATIC AttrForDisplay AttributesToDisplay[N_ATTRIBUTES + 1] =
{
   {CS_OID_ATTRSERVNAME, "Server name"},
   {CS_OID_ATTRSERVICE, "Service type"},
   {CS_OID_ATTRVERSION, "Server entry version"},
   {CS_OID_ATTRSTATUS, "Server status"},
   {CS_OID_ATTRADDRESS, "Network addresses"},
   {CS_OID_ATTRRETRYCOUNT, "Connection retry count"},
   {CS_OID_ATTRLOOPDELAY, "Connection retry loop delay"},
   {"", ""}
};
/*
** show_server_info()
**   Selectively display the attributes of a server directory
**   object.
**
** Parameters
**   ds_object -- Pointer to the CS_DS_OBJECT that describes the
**     server's directory entry.
**   outfile -- Open FILE handle to write the output to.
**
** Dependencies
**   Reads the contents of the AttributesToDisplay global array.
**
** Returns
**   CS_SUCCEED or CS_FAIL.
*/
CS_RETCODE
show_server_info(ds_object, outfile)
CS_DS_OBJECT       *ds_object;
```

```
FILE               *outfile;
{
   CS_RETCODE          ret;
   CS_CHAR             scratch_str[512];
   CS_INT              outlen;
   CS_INT              cur_attr;
   CS_ATTRIBUTE        attr_metadata;
   CS_ATTRVALUE       *p_attrvals;

/*
** Distinguished name of the object.
*/
   ret = ct_ds_objinfo(ds_object, CS_GET, CS_DS_DIST_NAME, CS_UNUSED,
                      (CS_VOID *)scratch_str, CS_SIZEOF(scratch_str),
                       &outlen);
   if (ret != CS_SUCCEED)
   {
      ex_error("show_server_info: get distinguished name failed.");
      return CS_FAIL;
   }

   fprintf(outfile, "Name in directory: %s\n", scratch_str);

   for (cur_attr = 0; cur_attr < N_ATTRIBUTES; cur_attr++)
   {
   /*
   ** Look for the attribute. attr_get_by_type() fails if the object
   ** instance does not contain a value for the attribute. If this
   ** happens, we just go on to the next attribute.
   */
      ret = attr_get_by_type(ds_object,
                             AttributesToDisplay[cur_attr].type_string,
                             &attr_metadata, &p_attrvals);
      if (ret == CS_SUCCEED)
      {
         fprintf(outfile, "%s:\n",
         AttributesToDisplay[cur_attr].english_name);
        /*
        ** Display the attribute values.
         */
         ret = attr_display_values(&attr_metadata, p_attrvals, outfile);
         if (ret != CS_SUCCEED)
         {
            ex_error(
            "show_server_info: display attribute values failed.");
            free(p_attrvals);
            return CS_FAIL;
```

```
         } /* if */
         free(p_attrvals);
      } /* if */
   } /* for */

   return CS_SUCCEED;
} /* show_server_info() */
```

Retrieving an attributes value

The example fragment below contains the code for the attr_get_by_type example utility routine. attr_get_by_type takes an OID string that specifies the desired attribute type, searches for the desired attribute in the directory object's attribute set, and returns the attribute's metadata and values if they are found.

```
/*
** get_attr_by_type()
**   Get metadata and attribute values for a given attribute type.
**
** Parameters
**   ds_object -- Pointer to a valid CS_DS_OBJECT hidden structure.
**   attr_type_str -- Null-terminated string containing the OID for the
**            desired attribute type.
**   attr_metadata -- Pointer to a CS_ATTRIBUTE structure to
**            fill in.
**   p_attrvals -- Address of a CS_ATTRVALUE union pointer.
**            If successful, this routine allocates an array
**            of size attr_metadata->numvalues, retrieves values into
**            it, and returns the array address in *p_attr_values.
**            NOTE: The caller must free this array when it is no longer
**            needed.
**
** Returns
**   CS_FAIL if no attribute of the specified type was found.
**   CS_SUCCEED for success.
*/

CS_RETCODE
attr_get_by_type(ds_object, attr_type_str, attr_metadata, p_attrvals)
CS_DS_OBJECT       *ds_object;
CS_CHAR            *attr_type_str;
CS_ATTRIBUTE       *attr_metadata;
CS_ATTRVALUE      **p_attrvals;
{
   CS_RETCODE          ret;
   CS_INT              num_attrs;
   CS_INT              cur_attr;
```

```
CS_INT              outlen;
CS_INT              buflen;
CS_BOOL             found = CS_FALSE;

/*
** Check input pointers. If not NULL, make them fail safe.
*/
if (attr_metadata == NULL || p_attrvals == NULL)
{
   return CS_FAIL;
}
attr_metadata->attr_numvals = 0;
*p_attrvals = NULL;
/*
** Get number of attributes.
*/
ret = ct_ds_objinfo(ds_object, CS_GET, CS_DS_NUMATTR, CS_UNUSED,
                   (CS_VOID *)#_attrs, CS_SIZEOF(num_attrs),
                    NULL);
if (ret != CS_SUCCEED)
{
   ex_error("attr_get_by_type: get number of attributes failed.");
   return CS_FAIL;
}

/*
** Look for the matching attribute, get the values if found.
*/
for (cur_attr = 1;
     cur_attr <= num_attrs && found != CS_TRUE;
     cur_attr++)
{
   /*
   ** Get the attribute's metadata.
   */
   ret = ct_ds_objinfo(ds_object, CS_GET, CS_DS_ATTRIBUTE, cur_attr,
                      (CS_VOID *)attr_metadata,
                       CS_SIZEOF(CS_ATTRIBUTE), NULL);
   if (ret != CS_SUCCEED)
   {
      ex_error("attr_get_by_type: get attribute failed.");
      return CS_FAIL;
   }

   /*
   ** Check for a match.
   */
   if (match_OID(&(attr_metadata->attr_type), attr_type_str))
```

```
      {
         found = CS_TRUE;
         /*
         ** Get the values -- we first allocate an array of
         ** CS_ATTRVALUE unions.
         */
         *p_attrvals = (CS_ATTRVALUE *) malloc(sizeof(CS_ATTRVALUE)
                                    * (attr_metadata->attr_numvals));
         if (p_attrvals == NULL)
         {
            ex_error("attr_get_by_type: out of memory!");
            return CS_FAIL;
         }
         buflen = CS_SIZEOF(CS_ATTRVALUE) * (attr_metadata->attr_numvals);
         ret = ct_ds_objinfo(ds_object, CS_GET, CS_DS_ATTRVALS, cur_attr,
                              (CS_VOID *)(*p_attrvals), buflen, &outlen);
         if (ret != CS_SUCCEED)
         {
            ex_error("attr_get_by_type: get attribute values failed.");
            free(*p_attrvals);
            *p_attrvals = NULL;
            attr_metadata->attr_numvals = 0;
            return CS_FAIL;
         }
      }
   }

   /*
   ** Got the attribute.
   */
   if (found == CS_TRUE)
   {
      return CS_SUCCEED;
   }
/*
** Not found.
*/
   attr_metadata->attr_numvals = 0;
   return CS_FAIL;
} /* attr_get_by_type() */


/*
** match_OID()
**              Compare a pre-defined OID string to the contents of a
**              CS_OID structure.
**
```

```
** Parameters
**            oid -- Pointer to a CS_OID structure. OID->oid_length should be
**            the length of the string, not including any null-terminator.
**            oid_string -- Null-terminated OID string to compare.
**
** Returns
**            Non-zero if contents of oid->oid_buffer matches contents
**            of oid_string.
*/
int match_OID(oid, oid_string)

CS_OID *oid;
CS_CHAR *oid_string;
{
   return ((strncmp(oid_string, oid->oid_buffer, oid->oid_length) == 0)
   && ((oid->oid_length == strlen(oid_string))));
} /* match_OID() */
```

Processing attribute values

The code fragment below declares an example routine, attr_display_values, which prints the values of an attribute as text. attr_display_values calls two other utility routines to perform its work:

- attr_val_as_string – formats an attribute value as text and puts the result in a character array.
- attr_enum_english_name – converts an integer or enumerated attribute value into a printable character string

.

```
/*
** attr_display_values()
**   Writes an attribute's values to the specified text
**   file.
**
** Parameters
**   attr_metadata -- address of the CS_ATTRIBUTE structure that
**      contains metadata for the attribute.
**   attr_vals -- address of an array of CS_ATTRVALUE structures.
**       This function assumes length is attr_metadata->attr_numvals
**       and value syntax is attr_metadata->attr_syntax.
**   outfile -- Open FILE handle to write to.
**
** Returns
**   CS_SUCCEED or CS_FAIL.
```

```
*/
CS_RETCODE
attr_display_values(attr_metadata, attr_vals, outfile)
CS_ATTRIBUTE       *attr_metadata;
CS_ATTRVALUE       *attr_vals;
FILE               *outfile;
{
   CS_INT              i;
   CS_CHAR             outbuf[CS_MAX_DS_STRING * 3];
   CS_RETCODE          ret;

   /*
   ** Print each value.
   */
   for (i = 0; i < attr_metadata->attr_numvals; i++)
   {
      ret = attr_val_as_string(attr_metadata, attr_vals + i,
            outbuf, CS_MAX_DS_STRING * 3, NULL);
      if (ret != CS_SUCCEED)
      {
         ex_error("attr_display_values: attr_val_as_string() failed.");
         return CS_FAIL;
      }
      fprintf(outfile, "\t%s\n", outbuf);
   }

   return CS_SUCCEED;

} /* attr_display_values() */
/*
** attr_val_as_string() -- Convert the contents of a CS_ATTRVALUE union to
**   a printable string.
**
** Parameters
**   attr_metadata -- The CS_ATTRIBUTE structure containing metadata
**       for the attribute value.
**   val -- Pointer to the CS_ATTRVALUE union.
**   buffer -- Address of the buffer to receive the converted value.
**   buflen -- Length of *buffer in bytes.
**   outlen -- If supplied, will be set to the number of bytes written
**      to *buffer.
**
** Returns
**   CS_SUCCEED or CS_FAIL.
*/

CS_RETCODE
```

```
attr_val_as_string(attr_metadata, val, buffer, buflen, outlen)
CS_ATTRIBUTE       *attr_metadata;
CS_ATTRVALUE       *val;
CS_CHAR            *buffer;
CS_INT              buflen;
CS_INT             *outlen;
{
   CS_CHAR             outbuf[CS_MAX_DS_STRING * 4];
   CS_CHAR             scratch[CS_MAX_DS_STRING];
   CS_RETCODE          ret;

   if (buflen == 0 || buffer == NULL)
   {
      return CS_FAIL;
   }
   if (outlen != NULL)
   {
      *outlen = 0;
   }
   switch ((int)attr_metadata->attr_syntax)
   {
      case CS_ATTR_SYNTAX_STRING:
         sprintf(outbuf, "%.*s",
            (int)(val->value_string.str_length),
            val->value_string.str_buffer);
         break;
      case CS_ATTR_SYNTAX_BOOLEAN:
         sprintf(outbuf, "%s",
            val->value_boolean == CS_TRUE ? "True" : "False");
         break;
      case CS_ATTR_SYNTAX_INTEGER:
      case CS_ATTR_SYNTAX_ENUMERATION:
      /*
      ** Some enumerated or integer attribute values should be converted
      ** into an english-language equivalent. attr_enum_english_name()
      ** contains all the logic to convert #define's into human
      ** language.
      */
         ret = attr_enum_english_name((CS_INT)(val->value_enumeration),
               &(attr_metadata->attr_type),
               scratch, CS_MAX_DS_STRING, NULL);
         if (ret != CS_SUCCEED)
         {
            ex_error("attr_val_as_string: attr_enum_english_name() failed.");
            return CS_FAIL;
         }
         sprintf(outbuf, "%s", scratch);
```

```
        break;

    case CS_ATTR_SYNTAX_TRANADDR:
    /*
    ** The access type is an enumerated value. Get an english language
    ** string for it.
    */
        switch ((int)(val->value_tranaddr.addr_accesstype))
        {
            case CS_ACCESS_CLIENT:
                sprintf(scratch, "client");
                break;
            case CS_ACCESS_ADMIN:
                sprintf(scratch, "administrative");
                break;
            case CS_ACCESS_MGMTAGENT:
                sprintf(scratch, "management agent");
                break;
            default:
                sprintf(scratch, "%ld",
                       (long)(val->value_tranaddr.addr_accesstype));
                break;
        }

        sprintf(outbuf,
                "Access type '%s'; Transport type '%s'; Address '%s'",
                 scratch,
                 val->value_tranaddr.addr_trantype.str_buffer,
                 val->value_tranaddr.addr_tranaddress.str_buffer);

        break;

    case CS_ATTR_SYNTAX_OID:
        sprintf(outbuf, "%.*s",
        (int)(val->value_oid.oid_length),
        val->value_oid.oid_buffer);
        break;
    default:
        sprintf(outbuf, "Unknown attribute value syntax");
        break;

} /* switch */
if (strlen(outbuf) + 1 > buflen || buffer == NULL)
{
    return CS_FAIL;
}
else
{
```

```
      sprintf(buffer, "%s", outbuf);
      if (outlen != NULL)
      {
         *outlen = strlen(outbuf) + 1;
      }
   }
   return CS_SUCCEED;
} /* attr_val_as_string() */
/*
** attr_enum_english_name()
**   Based on the attribute type, associate an english phrase with
**   a CS_INT value. Use this function to get meaningful names for
**   CS_ATTR_SYNTAX_ENUMERATION or CS_ATTR_SYNTAX_INTEGER attribute
**   values.
**
**   If the attribute type represents a quantity and not a numeric code,
**   then the value is converted to the string representation of the
**   number. Unknown codes are handled the same way.
**
** Parameters
**   enum_val -- The integer value to convert to a string.
**   attr_type -- Pointer to an OID structure containing the OID string
**      that tells the attribute's type.
**   buffer -- Address of the buffer to receive the converted value.
**   buflen -- Length of *buffer in bytes.
**   outlen -- If supplied, will be set to the number of bytes written
**      to *buffer.
**
** Returns
**   CS_SUCCEED or CS_FAIL
*/

CS_RETCODE
attr_enum_english_name(enum_val, attr_type, buffer, buflen, outlen)
CS_INT              enum_val;
CS_OID             *attr_type;
CS_CHAR            *buffer;
CS_INT              buflen;
CS_INT             *outlen;
{
   CS_CHAR             outbuf[CS_MAX_DS_STRING];

   if (buffer == NULL || buflen <= 0)
   {
      return CS_FAIL;
   }
   if (outlen != NULL)
```

```
{
   *outlen = 0;
}

/*
** Server version number.
*/
if (match_OID(attr_type, CS_OID_ATTRVERSION))
{
   sprintf(outbuf, "%ld", (long)enum_val);
}
/*
** Server's status.
*/
else if (match_OID(attr_type, CS_OID_ATTRSTATUS))
{
   switch ((int)enum_val)
   {
      case CS_STATUS_ACTIVE:
         sprintf(outbuf, "running");
         break;
      case CS_STATUS_STOPPED:
         sprintf(outbuf, "stopped");
         break;
      case CS_STATUS_FAILED:
         sprintf(outbuf, "failed");
         break;
      case CS_STATUS_UNKNOWN:
         sprintf(outbuf, "unknown");
         break;
      default:
         sprintf(outbuf, "%ld", (long)enum_val);
         break;
   }
}

/*
** Anything else is either an enumerated type that we don't know
** about, or it really is just a number. We print the numeric value.
*/
else
{
   sprintf(outbuf, "%ld", (long)enum_val);
}

/*
** Transfer output to the caller's buffer.
*/
```

```
   if (strlen(outbuf) + 1 > buflen || buffer == NULL)
   {
      return CS_FAIL;
   }
   else
   {
      sprintf(buffer, "%s", outbuf);
      if (outlen != NULL)
      {
         *outlen = strlen(outbuf) + 1;
      }
   }
   return CS_SUCCEED;
} /* attr_enum_english_name() */
```

Step 4: Cleaning up

An application can call ct_ds_dropobj to deallocate each directory object that it received through its directory callback.

Alternatively, directory objects are dropped implicitly when the application calls ct_con_drop to drop the parent connection.

APPENDIX A Logical Sequence of Calls

Client-Library uses a *state machine* to enforce a logical order of operations. It stores information about the last call that an application made and limits the calls that can follow to those that are legal. For example, an application must call ct_connect to connect to a server before it can call ct_send to send commands.

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 “Hidden structures” on page 29 for a discussion of the basic control structures.

At the context level, an application sets up its environment by: allocating one or more context structures, setting CS-Library properties for the contexts, initializing Client-Library, and setting Client-Library properties for the contexts. See “Step 1: Set up the Client-Library programming environment” on page 18.

At the connection level, an application connects to a server by: 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. See “Step 3: Connect to a server” on page 22.

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. See “Step 4: Send commands to the server” on page 24.

Command-level sequence of calls

It is at the command level that the logical sequence of calls becomes complex, due to the larger number of routines that are managed at the command level.

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: the initiated-commands state table and the result-types state table.

Commands state table

The commands table defines the *states* of an application. For example, it defines a command-sent state to indicates that the last call an application made was 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 170 for a detailed description of each of the command states. See "Callable routines in each command state" on page 172 for a list of legal calls in each command state.

Initiated-commands state table

The initiated-commands table controls the use of 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 are processed, the state machine is in the Idle state.

From the Idle state, the command state machine permits an application to initiate a new command. 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 183 for a detailed description of each of the initiated command states. See “Callable routines for initiated commands” on page 185 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) that indicate when results are available. The result-types table goes a step further by indicating the type of available results.

This information is important because certain routines make sense only 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. In cases like these, the result-types table enforces the logical sequence of calls.

See “Result types” on page 188 for a detailed description of each of the result type states. See “Callable routines for each result type” on page 190 for a mapping of result type states with Client-Library routines.

Summary

The information that follows is 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 Client-Library returns descriptive error messages at runtime 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.

Table A-1: Command states

Command state	Meaning
Idle	The application: • Has not yet initiated a command, • Has completely processed the results of the last command, • Has fetched all cursor rows but has not closed the Client-Library cursor, or • Has closed a Client-Library cursor that is still associated with unprocessed results.
Command initiated	The application called ct_command, ct_cursor, or ct_dynamic to initiate a command, but it has not yet sent it to the server.
Command sent	The application called ct_send to send a command to the server, but it has not 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 non-cursor 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 yet been called.
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.

Command state	Meaning
Fetching nested command	The application initiated one of the following commands while fetching from a result set that contains cursor results: • Cursor-close (ct_cursor(CS_CURSOR_CLOSE)) • Cursor-update (ct_cursor(CS_CURSOR_UPDATE)) • Cursor-delete (ct_cursor(CS_CURSOR_DELETE))
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.

Command-level routines

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

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

Callable routines in each command state

Table A-2 maps each command state to the Client-Library routines that an application can call while in that state. It also identifies the state of the command after the routine has completed.

Table A-2: Callable routines at each command state

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.
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_dynsqlda	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_dynsqlda	Command initiated.
	ct_param	Command initiated.
	ct_setparam	Command initiated.
	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.

Beginning state	Callable routines	Resulting command state
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_dynsqlda	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.
	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.

Beginning state	Callable routines	Resulting command state
Non-fetchable results available	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_dynsqlda	Non-fetchable results available.
	ct_getformat	Non-fetchable results available.
	ct_res_info	Non-fetchable results available.
ANSI-style cursor end-data	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.
	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.
	ct_compute_info	ANSI-style cursor end-data.
	ct_describe	ANSI-style cursor end-data.
	ct_dyndesc	ANSI-style cursor end-data.
	ct_dynsqlda	ANSI-style cursor end-data.
	ct_fetch	• ANSI-style cursor end-data, if CS_END_DATA. • Idle, if CS_CANCELED. • Undefined, if CS_FAIL.

Beginning state	**Callable routines**	**Resulting command state**
ANSI-style cursor end-data	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_dynsqlda	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.
	ct_getformat	Fetchable results.
	ct_res_info	Fetchable results.

Beginning state	Callable routines	Resulting command state
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_dynsqlda	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.
	ct_dynsqlda	Fetchable nested command.
	ct_param	Fetchable nested command.
	ct_setparam	Fetchable nested command.
	ct_send	• Sent fetchable nested, if CS_SUCCEED. • Idle, if CS_CANCELED. • Undefined, if CS_FAIL.

Beginning state	Callable routines	Resulting command state
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_dynsqlda	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.

Beginning state	Callable routines	Resulting command state
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.
	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.
Fetching results	ct_dynsqlda	• 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.

Beginning state	Callable routines	Resulting command state
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.
	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_dynsqlda	• 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.

Beginning state	Callable routines	Resulting command state
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_dynsqlda	Fetching nested command.
	ct_param	Fetching nested command.
	ct_setparam	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.
	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.

Beginning state	Callable routines	Resulting command state
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_dynsqlda	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.

Beginning state	Callable routines	Resulting command state
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.
	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_dynsqlda	• 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.
	ct_dynsqlda	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.

Beginning state	Callable routines	Resulting command state
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.

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:

Table A-3: Initiated command states

Initiated command state	Meaning
Idle	The application either has not 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)), sent the command, 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 has not 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.
ct_command command initiated	The application initiated a language, message, package, or RPC command using ct_command.
Initiated send-data	The application initiated a send-data or send-bulk-data command using ct_command.
Initiated cursor-declare	The application initiated a cursor-declare command (ct_cursor(CS_CURSOR_DECLARE)) but has not yet sent it to a server using ct_send.
Initiated cursor-rows	The application initiated a cursor-rows command using ct_cursor(CS_CURSOR_ROWS).

Initiated command state	Meaning
Initiated cursor-open	The application initiated a cursor-open command (ct_cursor(CS_CURSOR_OPEN)) but has not yet sent it to a server.
Initiated cursor-close	The application initiated a cursor-close command (ct_cursor(CS_CURSOR_CLOSE)) but has not yet sent it to a server.
Initiated cursor-deallocate	The application initiated a cursor-deallocate command (ct_cursor(CS_CURSOR_DEALLOC)) but has not yet sent it to a server.
Initiated cursor-update	The application initiated a cursor-update command (ct_cursor(CS_CURSOR_UPDATE)) but has not yet sent it to a server.
Initiated cursor-delete row	The application initiated a cursor-delete command (ct_cursor(CS_CURSOR_DELETE)) but has not yet sent it to a server.
Initiated dynamic cursor-declare	The application initiated a cursor-declare command on a prepared dynamic SQL statement (ct_dynamic(CS_CURSOR_DECLARE)) but has not yet sent it to a server.
Initiated dynamic deallocate	The application initiated a command to deallocate a prepared SQL statement (ct_dynamic(CS_DEALLOC)) but has not yet sent it to a server.
Initiated dynamic describe	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 has not yet sent it to a server.
Initiated dynamic execute	The application initiated a command to execute a prepared SQL statement (ct_dynamic(CS_EXECUTE)) but has not yet sent it to a server.
Initiated dynamic execute immediate	The application initiated a command to execute a literal SQL statement (ct_dynamic(CS_EXEC_IMMEDIATE)) but has not yet sent it to a server.
Initiated dynamic prepare	The application initiated a command to prepare a SQL statement (ct_dynamic(CS_PREPARE)) but has not yet sent it to a server.
ct_send_data succeeded	The application successfully called ct_send_data at least once.
Initiated send-bulk command	The application initiated a send-bulk-data command (ct_command(CS_SEND_BULK_CMD)) but has not yet sent it to a server.

Initiated command routines

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

ct_cmd_drop	ct_dynamic	ct_send_data
ct_command	ct_dyndesc	ct_setparam
ct_cursor	ct_dynsqlda	ct_sendpassthru
ct_data_info	ct_param	

Callable routines for initiated commands

Table A-4 maps each initiated command state to the Client-Library routines that an application can call while in that state.

Where "none" is specified, an application can call none of the routines listed under "Initiated command routines" on page 184. From states that map to a "none" value in the Callable Routines column, an application's options are to send (ct_send) or cancel (ct_cancel) the initiated command.

Table A-4: Callable routines for initiated command states

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_command(CS_SEND_DATA_NOCMD) 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)
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)
ct_command command initiated	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated send-data	ct_data_info(CS_SET) ct_send_data

Initiated Command	**Callable Routines**
Initiated cursor-declare	ct_cursor(CS_CURSOR_ROWS) ct_cursor(CS_CURSOR_OPEN) ct_cursor(CS_CURSOR_OPTION) ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated cursor-rows	ct_cursor(CS_CURSOR_OPEN)
Initiated cursor-open	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated cursor-close	None
Initiated cursor-deallocate	None
Initiated cursor-update	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated cursor-delete	None
Initiated dynamic cursor-declare	None
Initiated dynamic deallocate	None
Initiated dynamic describe	None
Initiated dynamic execute	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
Initiated dynamic execute immediate	None
Initiated dynamic prepare	ct_dyndesc(CS_USE_DESC) ct_dynsqlda(CS_SQLDA_PARAM) ct_param ct_setparam
ct_send_data succeeded	ct_send_data
Initiated send-bulk command	ct_send_data

Result types

Client-Library restricts the routines that can be called based on the result type 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

Table A-5 briefly describes the different result types:

Table A-5: Result type definitions

Result type	Meaning
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.
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.

See Chapter 6, "Writing Results-Handling Code" 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_data_info	ct_getformat
ct_br_column	ct_describe	ct_keydata
ct_br_table	ct_dyndesc	ct_res_info
ct_compute_info	ct_dynsqlda	

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 the ct_results **result_type* parameter.

Table A-6 maps each result type to the Client-Library routines that an application can legally call to process that result type.

Table A-6: Callable routines for each 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) ct_dynsqlda(CS_SQLDA_BIND)
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) ct_dynsqlda(CS_SQLDA_BIND)
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) ct_dynsqlda(CS_SQLDA_BIND)
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) ct_dynsqlda(CS_SQLDA_BIND)
Message results	ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_MSGTYPE) ct_res_info(CS_TRANS_STATE)

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) ct_dynsqlda(CS_SQLDA_BIND)
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)
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)
Describe results	ct_describe ct_res_info(CS_CMD_NUMBER) ct_res_info(CS_NUMDATA) ct_res_info(CS_TRANS_STATE) ct_dyndesc(CS_GETATTR) ct_dyndesc(CS_GETCNT) ct_dynsqlda(CS_GET_IN) ct_dynsqlda(CS_GET_OUT)
Extended error data results	ct_bind ct_describe ct_res_info(CS_NUMDATA) ct_res_info(CS_TRANS_STATE)

Result type	Callable routines
Notification results	ct_bind ct_describe ct_res_info(CS_NUMDATA) ct_res_info(CS_TRANS_STATE)

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 that two command structures (A and B) share the same 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 remains 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

For a definition of each command state, see Table A-1 on page 170.

Index

D

E

F

H

I

L

M

N

O

P

R

S

T

U

V