SYBASE®

Programmer’s Reference for C

Mainframe Connect Client Option

12.6

IBM CICS, IMS, and MVS

DOCUMENT ID: DC35396-01-1260-01

LAST REVISED: March 2005

Copyright © 1991-2005 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

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

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase, the Sybase logo, ADA Workbench, Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Anywhere, Adaptive Server Enterprise, Adaptive Server Enterprise Monitor, Adaptive Server Enterprise Replication, Adaptive Server Everywhere, Adaptive Warehouse, Afaria, Answers Anywhere, Anywhere Studio, Application Manager, AppModeler, APT Workbench, APT-Build, APT-Edit, APT-Execute, APT-Translator, APT-Library, AvantGo Mobile Delivery, AvantGo Mobile Inspection, AvantGo Mobile Marketing Channel, AvantGo Mobile Pharma, AvantGo Mobile Sales, AvantGo Pylon, AvantGo Pylon Application Server, AvantGo Pylon Conduit, AvantGo Pylon PIM Server, AvantGo Pylon Pro, Backup Server, BizTracker, ClearConnect, Client-Library, Client Services, Convoy/DM, Copernicus, Data Pipeline, Data Workbench, DataArchitect, Database Analyzer, DataExpress, DataServer, DataWindow, DataWindow .NET, DB-Library, dbQueue, Developers Workbench, Direct Connect Anywhere, DirectConnect, Distribution Director, e-ADK, E-Anywhere, e-Biz Impact, e-Biz Integrator, E-Whatever, EC Gateway, ECMAP, ECRTP, eFulfillment Accelerator, Embedded SQL, EMS, Enterprise Application Studio, Enterprise Client/Server, Enterprise Connect, Enterprise Data Studio, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, eProcurement Accelerator, EWA, Financial Fusion, Financial Fusion Server, Gateway Manager, GlobalFIX, iAnywhere, iAnywhere Solutions, ImpactNow, Industry Warehouse Studio, InfoMaker, Information Anywhere, Information Everywhere, InformationConnect, InternetBuilder, iScript, Jaguar CTS, jConnect for JDBC, M2M Anywhere, Mail Anywhere Studio, MainframeConnect, Maintenance Express, Manage Anywhere Studio, M-Business Channel, M-Business Network, M-Business Server, MDI Access Server, MDI Database Gateway, media.splash, MetaWorks, mFolio, Mirror Activator, MySupport, Net-Gateway, Net-Library, New Era of Networks, ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, Open Biz, Open Client, Open ClientConnect, Open Client/Server, Open Client/Server Interfaces, Open Gateway, Open Server, Open ServerConnect, Open Solutions, Optima++, PB-Gen, PC APT Execute, PC DB-Net, PC Net Library, PocketBuilder, Pocket PowerBuilder, Power++, power.stop, PowerAMC, PowerBuilder, PowerBuilder Foundation Class Library, PowerDesigner, PowerDimensions, PowerDynamo, PowerScript, PowerSite, PowerSocket, Powersoft, PowerStage, PowerStudio, PowerTips, Powersoft Portfolio, Powersoft Professional, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, QAnywhere, Rapport, RemoteWare, RepConnector, Replication Agent, Replication Driver, Replication Server, Replication Server Manager, Replication Toolkit, Report-Execute, Report Workbench, Resource Manager, RFID Anywhere, RW-DisplayLib, RW-Library, S-Designor, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILS, smart.partners, smart.parts, smart.script, SQL Advantage, SQL Anywhere, SQL Anywhere Studio, SQL Code Checker, SQL Debug, SQL Edit, SQL Edit/TPU, SQL Everywhere, SQL Modeler, SQL Remote, SQL Server, SQL Server Manager, SQL SMART, SQL Toolset, SQL Server/CFT, SQL Server/DBM, SQL Server SNMP SubAgent, SQL Station, SQLJ, STEP, SupportNow, S.W.I.F.T. Message Format Libraries, Sybase Central, Sybase Client/Server Interfaces, Sybase Financial Server, Sybase Gateways, Sybase IQ, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase User Workbench, SybaseWare, Syber Financial, SyberAssist, SybFlex, SyBooks, System 10, System 11, System XI (logo), SystemTools, Tabular Data Stream, TradeForce, Transact-SQL, Translation Toolkit, UltraLite, UltraLite.NET, UNIBOM, Unilib, Uninull, Unisep, Unistring, URK Runtime Kit for UniCode, VisualWriter, VQL, WarehouseArchitect, Warehouse Control Center, Warehouse Studio, Warehouse WORKS, Watcom, Watcom SQL, Watcom SQL Server, Web Deployment Kit, Web.PB, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, XA-Server, XcelleNet, and XP Server are trademarks of Sybase, Inc. 11/04

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names used herein may be trademarks or registered trademarks of their respective companies.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

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

Contents

About This Book .... **vii**

CHAPTER 1 **Open ClientConnect Processing** **1**

What is Open ClientConnect? 1
Understanding three-tier and two-tier environments 2
Open ClientConnect communications 3
Three-tier (gateway-enabled) environment 4
Two-tier (gateway-less) environment 7
Communication flow 8
Open ClientConnect security 9
How to choose a network driver 9
Compatibility 14
Open ClientConnect Client-Library functions 15
Using Client-Library functions 16
Basic control structures 16
Steps in a simple program 18
A simple language program 19
Setting up the Client-Library programming environment 19
Connecting to a server 19
Sending a command to the server 20
Processing the results of the command 20
Finishing up 21

CHAPTER 2 **Topics** **23**

Buffers 23
CS_CLIENTMSG structure 26
Customization 27
DATAFMT structure 28
Datatypes 32
Open ClientConnect datatypes 32
Error and message handling 35
Return codes 36
Messages 36

The CS_EXTRA_INF property 37
Nulls 38
Properties 39
About the properties 43
Remote procedure calls (RPCs) 48
Results 51
CS_SERVERMSG structure 52
SQLCA structure 54
SQLCODE structure 55
Handles 56

CHAPTER 3 **Functions 59**
ct_bind 61
ct_cancel 69
ct_close 72
ct_cmd_alloc 75
ct_cmd_drop 78
ct_cmd_props 81
ct_command 83
ct_con_alloc 88
ct_con_drop 93
ct_config 96
ct_connect 99
ct_con_props 104
ct_describe 109
ct_diag 116
ct_exit 131
ct_fetch 134
ct_get_format 140
ct_init 141
ct_param 145
ct_remote_pwd 152
ct_res_info 154
ct_results 160
ct_send 165
cs_config 171
cs_convert 174
cs_ctx_alloc 179
cs_ctx_drop 182

APPENDIX A **Sample Language Application 187**
Sample program – SYCTSAA6 188

APPENDIX B **Sample RPC Application** **223**
Sample program – SYCTSAR6 224

APPENDIX C **Sybase Product Documentation** **261**
Publications by content 261
Publications by audience 264

Index **267**

About This Book

The Mainframe Connect Client Option *Programmer's Reference for C* contains reference information for the C version of Open ClientConnect™ Client-Library.

Note The Open ClientConnect Client-Library™ is a subset of the generic Sybase® Client-Library.

This chapter includes the following topics:

- Audience
- Product name changes
- How to use this book
- Related documents
- Other sources of information
- Sybase certifications on the Web
- Sybase EBFs and software maintenance
- Conventions
- If you need help

Audience

The Mainframe Connect Client Option *Programmer's Reference for C* is a reference manual for mainframe programmers who write client applications in the C programming language using Open ClientConnect Client-Library.

This manual assumes that the programmer is familiar with the C programming language and knows how to write programs under either CICS or IMS TM. This book does not contain instructions for writing C programs. Rather, it describes the functions that can be called within your C programs to enable them to communicate with remote servers.

Product name changes

The following table describes new names for products in the 12.6 release of the Mainframe Connect Integrated Product Set.

Old product names	New product name
• Open ClientConnect for CICS • Open ClientCONNECT for CICS	Mainframe Connect Client Option for CICS
• Open Client Connect for IMS and MVS • Open ClientCONNECT for IMS and MVS	Mainframe Connect Client Option for IMS and MVS
• Open ServerConnect for CICS • Open ServerCONNECT for CICS	Mainframe Connect Server Option for CICS
• Open ServerConnect for IMS and MVS • Open ServerCONNECT for IMS and MVS	Mainframe Connect Server Option for IMS and MVS
• MainframeConnect™ for DB2 UDB • MainframeCONNECT for DB2/MVS-CICS	Mainframe Connect DB2 UDB Option for CICS
• DirectConnect™ for OS/390 • DirectCONNECT for DB2/MVS	DirectConnect for z/OS

The old product names are used throughout this book, except for on the title page.

Note This book also uses the terms MVS and OS/390 where the newer term z/OS would otherwise be used.

How to use this book

Table 1 shows where to find the information you need in this book.

Table 1: Chapter descriptions

Chapter	Contents
Chapter 1, "Open ClientConnect Processing"	An overview of Open ClientConnect including discussion of different kinds of client requests and explanations of how Open ClientConnect programs process them. **Note** Everyone who writes programs using Open ClientConnect should read this chapter.

Chapter	Contents
Chapter 2, “Topics”	Descriptions of Gateway-Library™ concepts, and information on how to accomplish specific programming tasks. This chapter discusses tasks, resources, and other topics that the application programmer needs to understand to write Gateway-Library applications. It includes a detailed discussion of the Gateway-Library cursor, dynamic SQL and Japanese language support and a list of supported datatypes and models for structures used to store data.
Chapter 3, “Functions”	Reference pages for each Gateway-Library function. Each function description contains sections on functionality, syntax, explanatory comments and related functions, as well as an example.
Appendix A, “Sample Language Application”	A sample C application program that processes client language requests under CICS.
Appendix B, “Sample RPC Application”	A sample C application program that processes client RPC requests under CICS.
Appendix C, “Sybase Product Documentation”	A list and description of all relevant Sybase product documentation.

Related documents

For a complete list and description of all related documents, see Appendix C, “Sybase Product Documentation.”

Other sources of information

Use the Sybase Getting Started CD, the Sybase Technical Library CD, and the Technical Library 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 Technical Library CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader (downloadable at no charge from the Adobe Web site, using a link provided on the CD).
- The Technical Library CD contains product manuals and is included with your software. The DynaText reader (included on the Technical Library CD) allows you to access technical information about your product in an easy-to-use format.

  Refer to the *Technical Library Installation Guide* in your documentation package for instructions on installing and starting the Technical Library.
- The Technical Library Product Manuals Web site is an HTML version of the Technical Library 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 Technical Library 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. Select Products from the navigation bar on the left.
3. Select a product name from the product list and click Go.
4. Select the Certification Report filter, specify a time frame, and click Go.
5. Click a Certification Report title to display the report.

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

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

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

Sybase EBFs and software maintenance

❖ Finding the latest information on EBFs and software maintenance

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. Enter user name and password information, if prompted (for existing Web accounts) or create a new account (a free service).
3. Select a product.
4. Specify a time frame and click Go.
5. Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions

This section describes the syntax and style conventions used in this book.

Note Throughout this book, all references to Adaptive Server™ Enterprise also apply to its predecessor, SQL Server®. Also, Adaptive Server Enterprise (ASE) and Adaptive Server (AS) are used interchangeably.

Open ClientConnect uses eight-character function names, while other versions of Client-Library use longer names. This book uses the long version of Client-Library names with one exception: the eight-character version is used in syntax statements. For example, CTBCMDPROPS has eleven letters. In the syntax statement, it is written CTBCMDPR, using eight characters. You can use either version in your code.

Table 2 explains syntax conventions used in this book.

Table 2: Syntax conventions

Symbol	Explanation
( )	Parentheses indicate that parentheses are included as part of the command.
{ }	Braces indicate that you must choose at least one of the enclosed options. Do not type the braces when you type the option.
[ ]	Brackets indicate that you can choose one or more of the enclosed options, or none. Do not type the brackets when you type the options.
|	The vertical bar indicates that you can select only one of the options shown. Do not type the bar in your command.
,	The comma indicates that you can choose one or more of the options shown. Separate each choice by using a comma as part of the command.

Table 3 explains style conventions used in this book.

Table 3: Style conventions

This type of information	Looks like this
Gateway-Library function names	TDINIT, TDRESULT
Client-Library function names	CTBINIT, CTBRESULTS
Other executables (DB-Library routines, SQL commands) in text	the dbrpcparam routine, a select statement
Directory names, path names, and file names	*/usr/bin directory, interfaces* file
Variables	*n* bytes
Adaptive Server datatypes	datetime, float
Sample code	01 BUFFER PIC S9(9) COMP SYNC. 01 BUFFER PIC X(n).
User input	01 BUFFER PIC X(n)

This type of information	Looks like this
Client-Library and Gateway-Library function argument names	*BUFFER, RETCODE*
Client-Library function arguments that are input (I) or output (O)	*COMMAND* – (I) *RETCODE* – (O)
Names of objects stored on the mainframe	SYCTSAA5
Symbolic values used with function arguments, properties, and structure fields	CS-UNUSED, FMT-NAME, CS-SV-FATAL
Client-Library property names	CS-PASSWORD, CS-USERNAME
Client-Library and Gateway-Library datatypes	CS-CHAR, TDSCHAR

All other names and terms appear in this typeface.

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 Open ClientConnect Processing

This chapter includes the following topics:

- What is Open ClientConnect?
- Understanding three-tier and two-tier environments
- Open ClientConnect communications
- Open ClientConnect security
- How to choose a network driver
- Compatibility
- Open ClientConnect Client-Library functions
- Using Client-Library functions
- Steps in a simple program
- A simple language program

What is Open ClientConnect?

Open ClientConnect is a programming environment that provides Open Client Client-Library routines for use in building mainframe client applications.

Open ClientConnect runs on an IBM System/390 or plug-compatible mainframe computer. It uses LU 6.2 or TCP/IP communications protocols and is available for CICS, IMS TM, and native MVS host transaction processors.

Note Some information in this guide is specific to version 4.0 of Open ClientConnect. This information will apply to Open ClientConnect for CICS only. All other information will apply to both Open ClientConnect for CICS and Open ClientConnect for IMS and MVS.

Open ClientConnect applications can communicate with two kinds of servers:

- Adaptive Server Enterprise and Open Server™ on PCs and several mid-range UNIX platforms
- Open ServerConnect applications running in a separate region on the mainframe

Open ClientConnect applications can send requests to Adaptive Server Enterprise, Open ServerConnect applications, and MainframeConnect for DB2 UDB (or OmniSQL Access Module™ for DB2).

Adaptive Server Enterprise

Open ClientConnect applications can send requests to Adaptive Server Enterprise indirectly through either of the following:

- The three-tier (gateway-enabled) environment using Mainframe ClientConnect (MCC).
- The two-tier (gateway-less) environment using TCP. See "Two-tier (gateway-less) environment" on page 7 for more information on two-tier environments.

Open ServerConnect

Open ClientConnect applications can send requests directly to Open ServerConnect running in a different CICS region. If using TCP, Open ClientConnect may send requests to Open ServerConnect running in the same CICS region.

Note Due to an IBM SNA restriction, connections from Open ClientConnect to Open ServerConnect require that they reside in different regions when connecting through LU 6.2. For TCP/IP, they can reside in the same region.

Understanding three-tier and two-tier environments

Open ClientConnect supports both three-tier (gateway-enabled) and two-tier (gateway-less) environments. When installing and using Open ClientConnect, follow the instructions in this book that are specific to your environment.

Three-tier (gateway-enabled) environments

If you use SNA as your protocol, you must use a three-tier environment for routing.

Note The DirectConnect product no longer comes with an MCC for TCP. A three-tier (gateway-enabled) environment using TCP as your protocol is no longer an option.

Two-tier (gateway-less)

If you have standardized to TCP for connectivity from CICS to Adaptive Server Enterprise, you must use the two-tier environment for routing. The two-tier environment allows Open ClientConnect to directly login to an Adaptive Server Enterprise, which eliminates the need for an MCC gateway.

An Open ClientConnect network configuration using two-tier (gateway-less) processing consists of the following:

- A host-based client, which is an Open ClientConnect program running under CICS. The client program selects a server and sends requests to that server.
- A server, which can be any server that Open ClientConnect applications can access, including servers on the LAN—Sybase Open Servers and Adaptive Server Enterprises—as well as Open ServerConnect running in a separate CICS region. If TCP is the protocol, Open ClientConnect can access servers running in the same CICS region.

Open ClientConnect communications

This section describes Open ClientConnect communications in three-tier and two-tier environments. It also explains the communication flow for both environments.

Note Although it is not shown in any of the following figures, Open ClientConnect for CICS also works with Open ServerConnect for IMS and MVS.

Three-tier (gateway-enabled) environment

The following diagrams show a basic Open ClientConnect configuration for CICS in three-tier (gateway-enabled) SNA and TCP/IP environments:

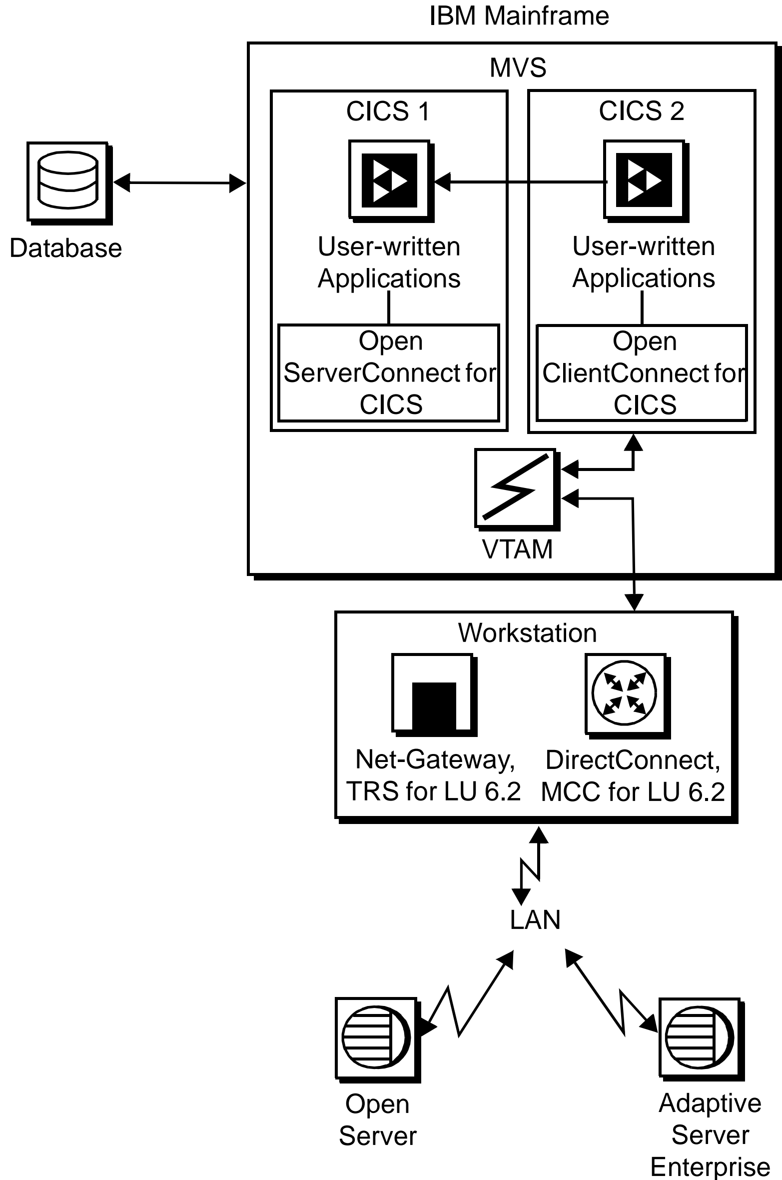
- Figure 1-1 on page 5
- Figure 1-2 on page 6

Note For three-tier, gateway-enabled environments, DirectConnect 11.1 (or Net-Gateway version 3.0.1 or higher) is a required companion product for full-feature compatibility with Open ServerConnect version 4.0 and Open ClientConnect version 3.2.

Three-tier SNA environment

Figure 1-1 depicts Open ClientConnect communications in a three-tier (gateway-enabled) SNA environment.

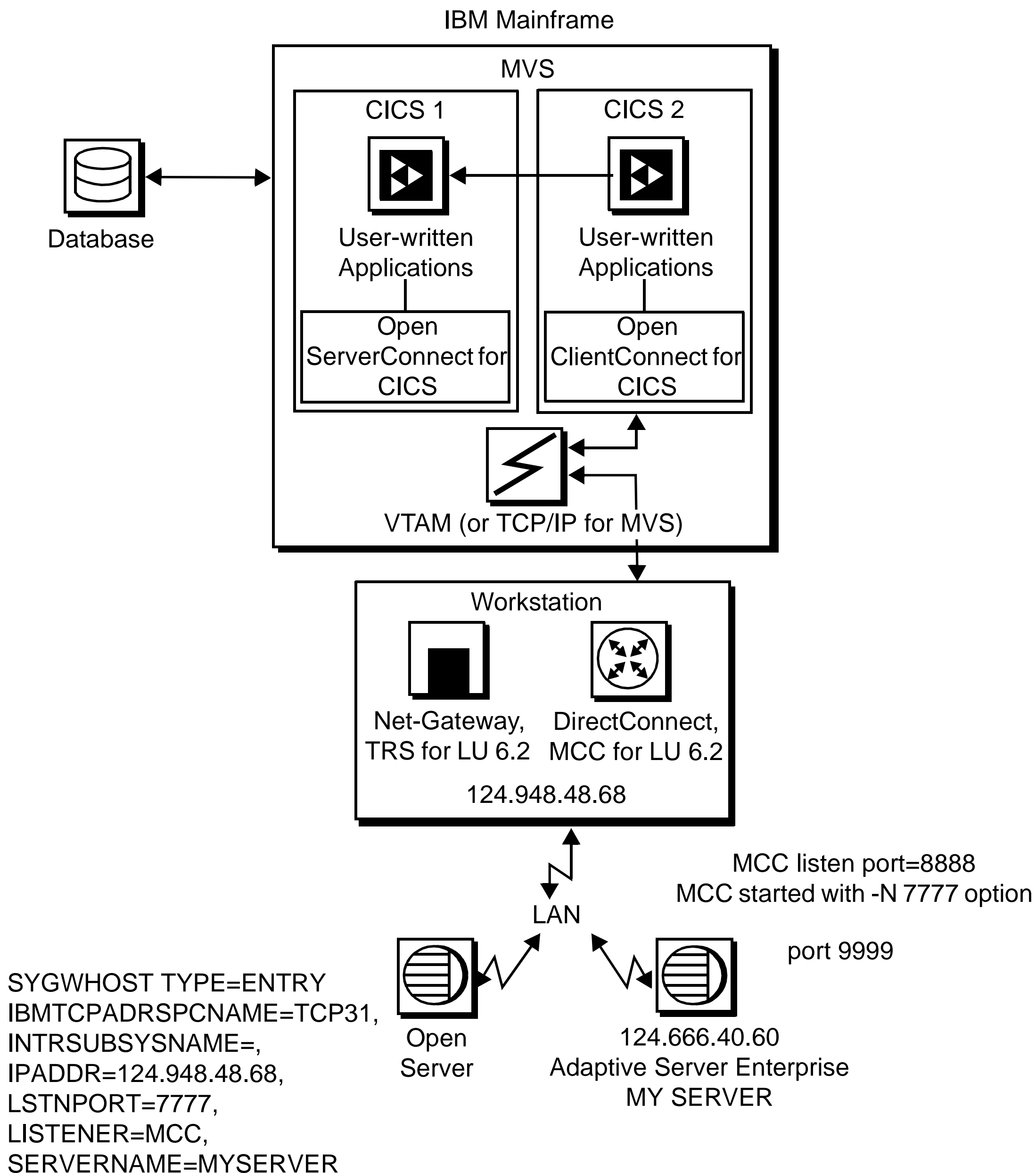
Figure 1-1: Open ClientConnect in a three-tier SNA environment



Three-tier TCP environment

Figure 1-2 depicts Open ClientConnect communications in a three-tier (gateway-enabled) TCP environment.

Figure 1-2: Open ClientConnect in a three-tier TCP environment

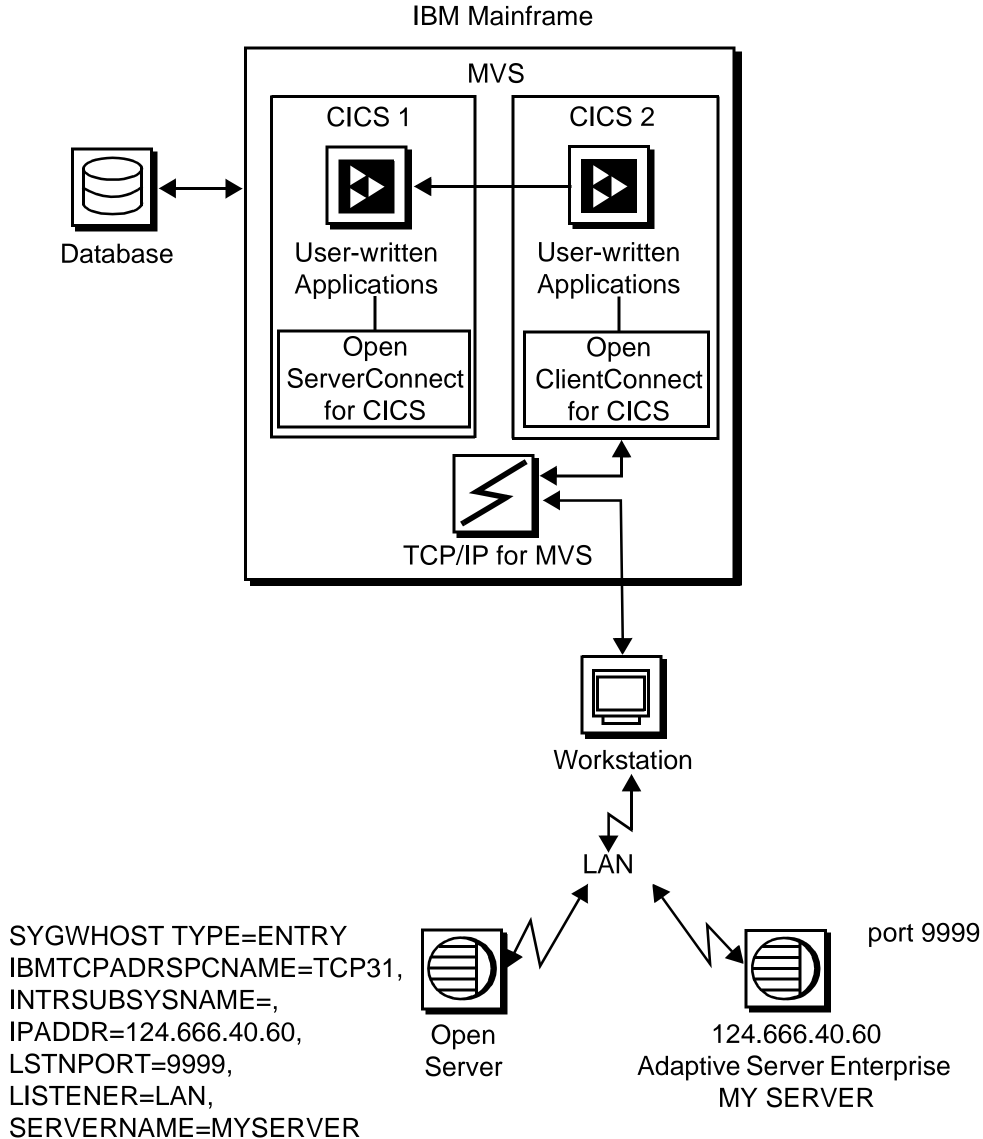


Two-tier (gateway-less) environment

Two-tier TCP environment

Figure 1-3 depicts Open ClientConnect communication in a two-tier (gateway-less) TCP environment.

Figure 1-3: Open ClientConnect in a two-tier TCP environment



Communication flow

This section describes what happens at the mainframe, at the DirectConnect installation, and at the server in Open ClientConnect processing.

At the mainframe

An Open ClientConnect application calls a pre-written procedure, such as a stored procedure or an Open ServerConnect application. All calls from Open ClientConnect to remote nodes are processed using the LU 6.2 (three-tier only) or TCP/IP (two-tier only) communications protocol. For requests to an Open Server, the client can access any data available to the Open Server application. If the request is to Open ServerConnect, the client can access any data storage system accessible through CICS.

The called procedure or transaction executes and returns results to the calling Open ClientConnect application, which can use the results for local processing. If the client has permission, the client transaction can update data at remote sites by inserting, modifying, and deleting entries in database tables or other data storage systems. For additional information on any of the following topics, refer to the Mainframe Connect Client Option *Installation and Administration Guide*.

isql utility

Open ClientConnect includes isql, a utility that allows users to send SQL language commands interactively. Users specify the server, whether or not to enable tracing, and type SQL commands in a 3270 panel.

Connection Router Table (SNA Only)

For SNA environments, Open ClientConnect includes a Connection Router Table that allows you to define servers and connections, and to set traffic priorities.

Server-Host Mapping Table (TCP/IP only)

For TCP/IP, Open ClientConnect includes a Server-Host Mapping Table that allows you to define servers for both three-tier and two-tier environments.

Side Information (SNA only)

Open ClientConnect for CICS uses the APPC Side Information File to define servers.

At the server

Typically, a server accepts requests from a client and returns results. In an Open ClientConnect environment, the server can be an Adaptive Server Enterprise, an Open Server on PCs and several mid-range UNIX platforms, or an Open ServerConnect running on the mainframe.

From the server standpoint, a request from an IBM host is no different than a request from a Sybase client. Open ClientConnect participates in ASCII-to-EBCDIC translations and datatype conversions.

Open ClientConnect security

Security for Open ClientConnect processing can be configured to require permission to:

- Log into the target server or desired CICS region.
- Use specific commands, stored procedures or transactions, and data objects at the target server.

For more security-related information regarding:

- Adaptive Server Enterprise, refer to the chapter called "Security Administration," in the Adaptive Server Enterprise *System Administration Guide*.
- DirectConnect, refer to the Mainframe Connect DirectConnect for z/OS Option *User's Guide for Transaction Router Services*.
- Mainframes, refer to documentation provided with CICS and MVS, or the appropriate mainframe security system.

How to choose a network driver

Open ClientConnect supports concurrent use of multiple network drivers, providing additional flexibility and easy installation for sites configured to run mixtures of SNA and TCP/IP.

Note Dynamic network driver support is a new feature in Open ClientConnect version 4.0.

The network drivers can be invoked from the same Open Client/Open Server common code base. The appropriate network driver is loaded dynamically at the time the program executes.

You must use the SYGWDRIV macro to define the network drivers to be used with Open ClientConnect and Open ServerConnect. For each operating environment (CICS and MVS), the default SYGWXCPH member provided on the tape contains the SYGWDRIV macro definitions for *all* supported network drivers pertinent to the technology. The person installing Open ClientConnect should edit the appropriate SYCTCUST member to comment-out the drivers that your site does not intend to use.

This section provides an overview of network communication, lists and describes general criteria for choosing a driver, as well as how to choose between a CPI-C/LU 6.2 and a TCP/IP driver.

Overview of network communication definitions

You need to choose which dynamic network drivers to use at your site. Your choice depends on the protocols installed at your site and the types of processing you want to achieve.

Use the following topic overview to understand issues involved in selecting your drivers:

- System Application Architecture (SAA)
- Common Programming Interface (CPI)
- APPC/MVS
- Systems Network Architecture (SNA)
- LU 6.2
- Advanced Program-to-Program Communications (APPC)
- Common threads between APPC MVS, CICS, and IMS TM

System Application Architecture (SAA)

SAA is an architecture composed of a set of selected software interfaces, conventions, and protocols designed to provide a framework for developing distributed applications. The key benefits of SAA are portability, consistency, and connectivity. The components of SAA are specifications for the key application interface points such as:

- Common user access
- Common communication support
- Common Programming Interface (CPI)

  CPI is explained in the following section.

Common Programming Interface (CPI)

The SAA Common Programming Interface specifies the languages and services used to develop applications across SAA environments. The elements of the CPI specification are divided into two parts:

- Processing logic
  - High level language—COBOL, C, Fortran, RPG
  - Procedure language—REXX
  - Application generator—Cross Systems Product/Application Development (CSP/AD)
- Services
  - Communication interface or CPI-C—API for writing APPC applications
  - Database interface—Structured Query Language (SQL)
  - Dialog interface—Interactive System Productivity Facility (ISPF)

APPC/MVS

APPC/MVS is an SNA application that extends APPC support to the MVS operating system. The primary role of APPC/MVS is to provide full LU 6.2 capability to MVS applications to allow communication with other applications in a distributed SNA network.

APPC/MVS provides programming support by providing an API based on the CPI-C interface. This interface is implemented in a lower-level API that is MVS-specific. The CPI-C calls all begin with CM; for example, CMALLC (Allocate). The MVS calls all begin with ATB; for example, Send_data (ATBSEND). The CPI-C calls are portable to non-MVS platforms while the ATB calls are not portable to non-MVS platforms.

Systems Network Architecture (SNA)

SNA is an IBM Network Architecture composed of a set of software interfaces, protocols, and operational sequences for transmitting information through and controlling the configuration and operation of networks.

LU 6.2

LU 6.2 refers to the SNA Logical Unit Type 6.2, which supports general communication between programs in a distributed environment. LU 6.2 is characterized by peer-to-peer communications support, comprehensive end-to-end error processing, optimized data transmission flow and a generic API.

The LU 6.2 system is layered functionally. It can be represented by a set of finite-state machines. Each of these machines has a finite number of states and a set of rules that govern the transition from one state to another. These finite state machines govern the behavior of LU 6.2 devices by guaranteeing that a given input always produces the same output.

Advanced Program-to-Program Communications (APPC)

APPC is peer level data communication support based on the SNA LU 6.2 protocols.

Common threads between APPC MVS, CICS, and IMS TM

All inbound transactions require a scheduler. Under MVS, the ASCH address space performs this function by scheduling inbound transactions in initiators under its control. The relationship between ASCH and its initiators is very similar to that of JES (Job Entry System), which schedules jobs in initiators under its control.

The Control region is the scheduler running under CICS. The Message Region running under the Control region corresponds to the initiators used by ASCH.

CICS differs from MVS and IMS TM because it does not schedule transactions in a separate address space. It schedules them as a task within its own address space.

Outbound transactions use a file called the Side Information File to map a name to an SNA logical unit. MVS and IMS TM both use this file.

General criteria for choosing a driver

The choice of a network driver depends on several factors:

- Network type—SNA or TCP/IP
- Network environment—two-tier (TCP/IP only) or three-tier (SNA only)
- Operating environment—CICS or MVS

This section explains why you might want to choose a particular driver in each environment.

Network type and environment

Because non-MVS platforms do not support LU 6.2 in their operating systems, if your network type is SNA, then you *must* use a three-tier network environment with a gateway. In a three-tier (gateway-enabled) environment, you must use either an LU 6.2 or CPI-C driver.

If your network type is TCP/IP, your network environment must be two-tier (gateway-less). The choice between IBM TCP/IP and Interlink depends on which product is installed in the network environment, although Open ClientConnect supports both concurrently.

Operating environment

This section explains the drivers used in CICS and MVS environments.

CICS environment

The following drivers are supported in the CICS environment:

- LU 6.2
- CPI-C
- IBM TCP/IP
- Interlink TCP/IP

The LU 6.2 driver supports only incoming transactions sent to the CICS message queue. It does not support Open Client outbound requests from the mainframe. With the LU 6.2 driver, CICS builds an entire result set in the message queue and sends that entire result set to the MSG.

The CPI-C driver supports both Open Client and Open Server requests. It does not use the CICS message queue to send or receive requests. Therefore, result sets sent by Open Server using the CPI-C driver can be interrupted. However, with the LU 6.2 driver, if a client does a CTRL-C to cancel the result set, the gateway must read the entire result set and throw it away.

MVS environment

The following drivers are supported in the MVS environment:

- CPI-C
- IBM TCP/IP
- Interlink TCP/IP

Choosing between a CPI-C/LU 6.2 driver and a TCP/IP driver

When choosing between a CPI-C/LU 6.2 driver and a TCP/IP driver, consider the following factors:

- Network type - SNA or TCP/IP
- Reliability
- Performance
- Network operating environment - Two-tier or three-tier

Network type

If your current network is SNA-only or TCP/IP-only, choose the driver that supports your network protocol.

Reliability

SNA networks have been running on IBM mainframes much longer than TCP/IP based-networks, and the SNA operational procedures are well established. However, if your LAN-side staff is not very familiar with SNA on a particular vendor platform, the SNA setup can be difficult.

TCP/IP is simpler to set up and maintain from the LAN, although it can be a challenge to get it running under MVS for the first time. In addition, TCP/IP on mainframes is a relatively new technology compared to SNA, and as such, SNA is probably more robust and reliable.

Performance

TCP/IP performance appears to equal and in some cases exceed SNA-based performance. When going from the LAN to a mainframe, SNA requires a gateway, while TCP/IP does not.

Network operating environment

Small, two-tier (gateway-less) Client Server networks are easier to set up and maintain than three-tier (gateway-enabled) networks, because three-tier networks have a gateway between the mainframe and the LAN. However, three-tier networks scale better, as well as provide a single point of entry for security and tracing facilities that can be easily enabled.

Compatibility

For full functionality with the current version, use the mainframe access components listed in Table 1-1, as available at your site.

Table 1-1: Open ClientConnect version compatibility

Component	Version level
Open ServerConnect	3.1 or higher
Open ClientConnect	3.2 or higher
MainframeConnect for DB2 UDB or OmniSQL Access Module for DB2	11.1 or higher 10.1 with latest Emergency Bug Fix (EBF) tapes
DirectConnect Transaction Router Service or Net-Gateway	11.1 3.0.1 for full functionality
Japanese Conversion Module	3.1

Open ClientConnect Client-Library functions

Open ClientConnect includes a programming interface of functions that are used in writing mainframe client applications. Some of these functions prepare and send requests to a server; others retrieve and process the results. Additional functions set application properties, handle error conditions, and provide a variety of information about an application's interaction with a server.

Open ClientConnect's Client-Library functions are similar in name and function to Open Client Client-Library's routines. However, not all Sybase Open Client routines are supported. For example, Open ClientConnect does not currently support cursors or compute rows.

Most mainframe Client-Library function names begin with "ct_"; for example, the function that exits from Client-Library is named ct_exit. This corresponds to the Client-Library/C routine ct_exit.

Open ClientConnect includes utility functions which allocate, drop, and assign properties to context handles, and one function used in datatype conversion. These functions begin with the prefix: "cs_" instead of "ct_." This naming convention derives from related Sybase products, but it is irrelevant for Open ClientConnect. Use cs_xxxxx functions just as if they were ct_xxxxx functions.

Structures, types, and values used by Client-Library functions are defined in header files.

Client-Library functions are described in detail in Chapter 3, "Functions."

Using Client-Library functions

An application programmer writes a client program, adding calls to Client-Library functions to set up control structures, connect to servers, send commands, process results, and clean up. A Client-Library program is compiled, linked, and run in the same way as any other C program under CICS or IMS TM.

Note An application program can act as both client and server. Such a program, called a *mixed-mode* program, contains both Client-Library calls to send requests and Gateway-Library calls to accept and process requests. For more information about an example of a mixed-mode program, see the Mainframe Connect Server Option *Programmer's Reference for COBOL.*

Basic control structures

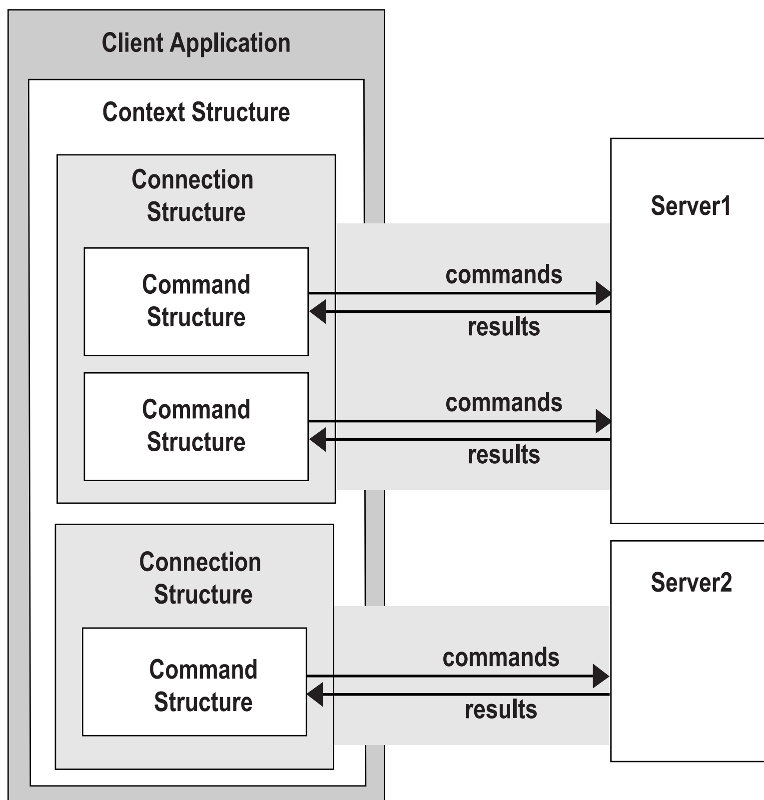
In order to send commands to a server, an application must allocate three types of structures:

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

An application allocates these structures by calling the functions cs_ctx_alloc, ct_con_alloc, and ct_cmd_alloc.

The relationship between these control structures is illustrated in Figure 1-4 on page 17.

Figure 1-4: Client-Library's control structures



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

Note An Open ClientConnect application is restricted to one context per application. This differs from applications written in other versions of Client-Library, which support multiple context structures.

Steps in a simple program

A simple program involves the following steps:

1 Set up the programming environment:

cs_ctx_alloc — Allocate a context structure.
ct_init — Initialize the programming interface.

2 Establish a connection with a server or CICS or IMS region.

ct_con_alloc — Allocate a connection structure.
ct_con_props — Set or retrieve connection structure properties.
ct_connect — Connect to a server.

3 Send a command to the server or to a CICS or IMS region.
(For three-tier processing, send a command to Mainframe ClientConnect which forwards the request to the target server, and routes the results back to the client program.)

ct_cmd_alloc — Allocate a command structure.
ct_command — Initiate a language request or RPC.
ct_send — Send a request to the server.

4 Process the results of the command:

ct_results — Set up result data to be processed.
ct_res_info — Return result set information.
ct_bind — Bind a returned column or parameter to a program variable.
ct_fetch — Fetch result data.

5 Finish up:

ct_cmd_drop — Deallocate a command structure.
ct_close — Close a server connection.
ct_con_drop — Deallocate a connection structure.
ct_exit — Exit the programming interface.
cs_ctx_drop — Deallocate a context structure.

A simple language program

The following walk-through demonstrates the basic framework of an Open ClientConnect application. The program follows the steps outlined in the previous section, sending a language request to an Adaptive Server Enterprise and processing the results. In this case, the language command is a Transact-SQL™ select command.

Several working examples are provided on the product tape.

Note The *CTPUBLIC include* file is required in all source files that contain calls to Open ClientConnect.

Setting up the Client-Library programming environment

cs_ctx_alloc allocates a context structure. A context structure is used to store configuration parameters that describe a particular "context," or operating environment, for a set of connections.

Application properties that can be defined at the context level include the version of Client-Library being used, the login time-out value, and the maximum number of connections allowed within the context.

ct_init initializes your environment. It must be the first call in an application after cs_ctx_alloc.

Connecting to a server

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

ct_con_props sets and retrieves the property values of a connection. Connection properties include:

- User name and password, which are used in logging into a server.
- Application name, which appears in Adaptive Server Enterprise's sysprocesses table.
- Packet size, which determines the size of network packets that an application sends and receives.

- Dynamic network driver (LU 6.2, IBM TCP/IP, Interlink TCP/IP, CPI-C), which defines the type of the network used between Open ClientConnect and the server.

Open ClientConnect includes a Connection Router where servers and server connections are defined.

For a complete list of connection properties, see “Remote procedure calls (RPCs)” on page 48 of this book.

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

Sending a command to the server

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

ct_command initiates the process of sending a command. In this example, it initiates a language command.

ct_send sends the command to the server.

Processing the results of the command

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

For row results, typically the number of columns in the result set is determined and then used to control a loop in which result items are bound to program variables. An application can call ct_res_info to get the number of result columns. After the result items are bound using ct_bind, the application calls ct_fetch to fetch data rows until end-of-data.

The results-processing model used in the example is as follows:

- Retrieve results: ct_results
- Determine the type of results: result_type
- Process results
  - If results are result rows: ct_res_info, ct_describe, ct_bind, ct_fetch

- If results are return parameters: ct_param, ct_describe, ct_bind, ct_fetch
- If results are status: ct_bind, ct_fetch

ct_results sets up results for processing. The ct_results return parameter RESULT_TYPE indicates the type of result data that is available for processing.

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

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

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

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

Finishing up

Use the following functions to close a connection and deallocate the structures.

- ct_cmd_drop deallocates a command structure.
- ct_close closes a server connection.
- ct_con_drop deallocates a connection structure.
- ct_exit cleans up the remaining resources being used by command or connection handles.
- cs_ctx_drop deallocates a context structure.

CHAPTER 2 Topics

The following topics are included in this chapter:

- Buffers
- CS_CLIENTMSG structure
- Customization
- DATAFMT structure
- Datatypes
- Error and message handling
- Nulls
- Remote procedure calls (RPCs)
- Results
- CS_SERVERMSG structure
- SQLCA structure
- SQLCODE structure
- Handles

Buffers

Description

A number of arguments used in Client-Library functions affect the contents of buffers: *action*, *buffer*, *buf_len*, *outlen*.

These arguments are described individually below. For a summary of argument values and their interaction, see Table 2-1.

Arguments

- *action* describes what is done to the data. For most functions, *action* can take the following symbolic values:

CS_SET	Assigns a value

CS_GET	Retrieves a value
CS_CLEAR	Clears the buffer, or, resets to the default property value

- *buffer* is a program variable, called a "buffer."

When *action* is CS_SET	*buffer* is the data being read by the function.
When *action* is CS_GET	*buffer* is the information retrieved by the function.
When *action* is CS_CLEAR	*buffer* remains unchanged.

- *buf_len* is the length, in bytes, of the *buffer* argument.

When the value in the buffer is a fixed-length or symbolic value:	Assign *buf_len* the actual length of the value, or CS_UNUSED.
When the value in the buffer is of variable length:	Assign *buf_len* the maximum number of bytes that the buffer can contain.

Note If *buf_len* is set to CS_GET and the buffer is too small to hold the returned value, the function returns CS_FAIL. When this happens, you can query the length of the incoming data by setting CS_SET to 0 and executing the function. The actual length of the data is returned to the *outlen* argument. Enlarge the buffer, assign the value in *outlen* to *buf_len*, and execute the function again.

- *outlen* is an integer variable where the function returns the actual length, in bytes, of the data being retrieved during a CS_GET operation. For all other operations, *outlen* is ignored.When the retrieved data is longer than *buf_len* bytes, an application can use the value of *outlen* to determine how many bytes are needed to hold the information. To do this:
  - Set the value of *outlen* to 0 and call the function
  - Set the value of *buf_len* to the length returned in *outlen*
  - Call the function again

Summary of buffer entries

Table 2-1 summarizes the interaction among *action*, *buffer*, *buf_len*, and *outlen*.

Table 2-1: Interactions among action, buffer, buf_len, and outlen

For this action	When buffer	buf_len is	outlen is	Result
CS_CLEAR	N/A	CS_UNUSED	Ignored	The property reverts to its default value.

For this action	When buffer	buf_len is	outlen is	Result
CS_SET	Contains a variable-length character string.	The length of the string	Ignored	The data in *buffer* is read by the function.
CS_SET	Contains a variable-length character string for which the terminating character is the last non-blank character.	The maximum length of the string	Ignored	The data in *buffer* is read by the function.
CS_SET	Contains a fixed-length character string or symbolic value.	CS_UNUSED	Ignored	The data in *buffer* is read by the function.
CS_GET	Is large enough for the return character string.	The length of the buffer	Set by Client-Library	The retrieved value is copied to the buffer. *outlen* returns the length of the retrieved value.
CS_GET	Is not large enough for the return character string.	The length of the buffer	Set by Client-Library	No data is copied to the buffer. *outlen* returns the length of the value being retrieved. The function returns CS_FAIL, indicating that you should assign the value returned here to the length argument and execute the program again.
CS_GET	Is assumed to be large enough for a fixed-length or symbolic value.	CS_UNUSED	Set by Client-Library	The retrieved value is copied to the buffer. *outlen* returns the length of the value being retrieved.

CS_CLIENTMSG structure

Description

A CS_CLIENTMSG (client message) structure contains information about an error or informational message returned by Open ClientConnect. This structure is defined within the application. When the error involves interaction with the operating system, the operating system error information is returned to this structure. ct_diag returns a message string and information about the message into CS_CLIENTMSG.

Server messages are returned to a CS_SERVERMSG structure, described in "CS_SERVERMSG structure" on page 52. A sample CS_CLIENTMSG structure is provided in the *CTPUBLIC include* file.

A CS_CLIENTMSG structure is defined as follows:

```
typedef struct_cs_clientmsg {
CS_INT    severity;
CS_CHAR   msgnumber;
CS_INT    msgstring[CS_MAX_MSG];

   /*
    **The error may have involved interactions
    **of the operating system (OS). If so, the
    **following contains the OS info.
      */

CS_INT    osnumber;
CS_CHAR   osstring[CS_MAX_MSG;
CS_INT    osstringlen;
CS_INT    status;
CS_CLIENTMSG;
```

Description of arguments in *CS_CLIENTMSG* structure

- *Severity* is a symbolic value representing the severity of the message. Table 2-2 on page 26 lists the legal values for *severity*.

Table 2-2: Values for the CS_CLIENTMSG severity field

Severity value	Explanation
CS_SV_INFORM (0)	No error occurred. The message is informational.
CS_SV_API_FAIL (1)	A Client-Library routine generated an error. This error is typically caused by a bad parameter or calling sequence. The server connection is probably salvageable.
CS_SV_RETRY_FAIL (2)	An operation failed, but it can be retried.
CS_SV_RESOURCE_FAIL (3)	A resource error occurred. This error is typically caused by an allocation error, a lack of file descriptors, or timeout error. The server connection is probably not salvageable.
CS_SV_CONFIG_FAIL (4)	A configuration error occurred.
CS_SV_COMM_FAIL (5)	An unrecoverable error in the server communication channel occurred. The server connection is not salvageable.

Severity value	Explanation
CS_SV_INTERNAL_FAIL (6)	An internal Client-Library error occurred.
CS_SV_FATAL (7)	A serious error occurred. All server connections are unusable.

- *msgnumber* is the Client-Library message number. Messages are listed in the Mainframe Connect Client Option and Server Option *Messages and Codes*.
- *msgstring* is the text of the Client-Library message string.
- *msgstringlen* is the length, in bytes, of *msgstring*. If there is no message text, the value of *msgstringlen* is 0.
- *osnumber* is the server error number, if any. A value here indicates that the message involved CICS or IMS TM I/O errors, remote server errors, or Transaction Router Service (TRS) errors.
- *osstring* is the text of the operating system message string, if any.
- *osstringlen* is the length, in bytes, of *osstring*. If there is no message text, the value of *osstringlen* is 0.
- *status* is reserved for future use.

Customization

Description

When installing Open ClientConnect, system programmers customize the product for the customer site, defining language and program characteristics locally. Some of the customized items are used by Gateway-Library programs.

Gateway-Library functions use the following locally-defined items:

- An access code, which is required to retrieve a client's password.

  Two customization options are related to the ability to retrieve client passwords:

  - The access code itself is defined during customization.
  - An access code flag is set to indicate whether the access code is required to retrieve the client password.
- The native language used at the mainframe (The default is US-English).
- Whether DB2 LONG VARCHAR data strings with lengths greater than 255 bytes are truncated or rejected when sent to a client.

Customization instructions are in the Mainframe Connect Client Option for CICS *Installation and Administration Guide*. The customization module is loaded during program initialization.

DATAFMT structure

Description

A DATAFMT (data format) structure is used to describe data values and program variables. For example:

- ct_bind requires a DATAFMT structure describing a destination variable
- ct_describe returns a DATAFMT structure describing a result data item
- ct_param requires a DATAFMT structure describing an input parameter
- cs_convert requires a DATAFMT structure describing source and destination data

Most functions use only a subset of the fields in a DATAFMT structure. For example, ct_bind does not use the name, status, and usertype fields, and ct_describe does not use the format field. For information on which DATAFMT fields a function uses, see Table 2-3 on page 29 in this chapter, or reference the desired function in Chapter 3, "Functions."

A DATAFMT structure is defined as follows:

```
/*********************************************************************/
                    CT_DATAFMT  used for client/server API calls
/*********************************************************************/
 typedef struct _ct_datafmt {
            CS_CHAR name[CS_MAX_NAME]; /* name of the column/parm     */
            CS_INT  namelen;           /* actual length of the name   */
            CS_INT  datatype;          /* hostvar datatype            */
            CS_INT  format;            /* Pad or \0 terminate Bin/Char*/
            CS_INT  maxlength;         /* max length the data might be*/
            CS_INT  scale;             /* NUMERIC: scale value (only) */
            CS_INT  precision;         /* NUMERIC: precision value    */
            CS_INT  status;            /* datum status (key, ret parm)*/
            CS_INT  count;             /* BIND: how many rows to bind */
            CS_INT  usertype;          /* user defined datatype (UDT) */
            CS_LOCALE *locale;         /* localization struct pointer */
                   } CS_DATAFMT;
```

Table 2-3 describes fields in the DATAFMT structure.

Table 2-3: Fields in the DATAFMT structure

Field	Contents	Used by
name	The name of the data item.	ct_describe ct_param
namelen	The length of name.	ct_describe ct_param
datatype	The datatype of the data. See the specific call to find which data this refers to.	cs_convert ct_bind ct_describe ct_param
format	The format of the data, represented by symbolic values.	ct_bind
maxlength	The maximum length of the data.	cs_convert ct_bind ct_describe ct_param
scale	The number of digits in the decimal part of a number. This field is used with packed decimal, numeric and Sybase-decimal.	cs_convert ct_bind
precision	The total number of digits in a number. This field is used with packed decimal, numeric and Sybase-decimal.	cs_convert ct_bind
status	Status values.	ct_describe ct_param
count	The number of items.	ct_bind ct_describe
usertype	The user-defined datatype (UDT) of retrieved data. The UDT is assigned by the server.	ct_describe ct_param
locale	Reserved for future use.	cs_convert ct_bind ct_describe ct_param

- name is the name of the data item. This can be a column, a parameter, or a return status name.
- namelen is the length, in bytes, of name. Assign namelen a value of 0 if the data item is unnamed.

- datatype is the datatype of the data. This is one of the Client-Library datatypes listed in Table 2-7 on page 32.

Note Return status values have a datatype of CS_INT.

- format is the destination format of fixed-length character or binary data.
- format can be one of the following values listed in Table 2-4.

Table 2-4: Values for the DATAFMT field format

Value	Meaning	Data type
CS_FMT_PADBLANK	The data should be padded with blanks to the full length of the destination variable.	For fixed-length character data only.
CS_FMT_PADNULL	The data should be padded with zeroes to the full length of the destination variable.	For binary or fixed-length character data.

- maxlength can represent various lengths, depending on which function is using the DATAFMT structure. Lengths are represented in bytes. Table 2-5 lists the meaning of maxlength for each function that uses it.

Table 2-5: Lengths defined by the DATAFMT field maxlength

Function	Length defined
cs_convert	The length of the source variable and the length of the destination variable.
ct_bind	The length of the variable to which the data is bound.
ct_describe	The maximum possible length of the column or parameter being described.
ct_param	The maximum length of return parameter data.

- scale is the number of decimal places in the value being converted. scale is used with ct_bind, cs_convert, and ct_param when converting to or from decimal datatypes CS_PACKED370, numeric, and Sybase-decimal.

Note Legal values for scale are from 0 to 31. If the actual scale is greater than 31, the call fails. To indicate that destination data should use the same scale as the source data, set scale to CS_SRC_VALUE. scale must be less than or equal to precision.

- precision is the precision of the value being converted.

Note Precision is used only with ct_bind, cs_convert, and ct_param when converting to or from decimal datatypes CS_PACKED370, numeric, and Sybase-decimal. Legal values for precision are from 1 to 31. To indicate that destination data should use the same precision as the source data, set precision to CS_SRC_VALUE. precision must be greater than or equal to scale.

- status is one or more of the following symbolic values (added together) listed in Table 2-6.

Table 2-6: Values for the DATAFMT field status

Value	Meaning	For this function
CS_CANBENULL	The column can contain nulls.	ct_describe
CS_NODATA	No data is associated with the result data item.	ct_describe
CS_INPUTVALUE	The parameter is a non-return RPC parameter (input parameter).	ct_param
CS_RETURN	The parameter is an RPC return parameter.	ct_param

- count is the number of rows to copy to program variables per ct_fetch call.
- usertype is the user-defined datatype, if any, of data returned by the server.

Note usertype is used only for datatypes defined at the server, not for datatypes defined by Open ClientConnect, such as Adaptive Server-defined datatypes or datatypes defined with TDSETUDT in Open ServerConnect.

- locale is reserved for future use. It must be set to zero.

Datatypes

Description

Open ClientConnect supports a wide range of datatypes. These datatypes are shared with Open Client, Open Server and Open ServerConnect, and correspond directly to Adaptive Server datatypes.

Table 2-7 lists the Client-Library datatypes, together with the corresponding type constants, Adaptive Server datatypes, and Open ServerConnect datatypes.

Table 2-7: Summary of Open ClientConnect datatypes

This Client-Library datatype	Describes this type of data	Corresponds to this Adaptive Server datatype	Corresponds to this Open ServerConnect datatype
CS_BINARY	Binary	binary	TDSBINARY
CS_CHAR	Character	char	TDSCHAR
CS_DATETIME	8-byte datetime	datetime	TDSDATETIME
CS_DATETIME4	4-byte datetime	smalldatetime	TDSDATETIME4
CS_FLOAT	8-byte float	float	TDSFLT8
CS_INT	4-byte integer	int	TDSINT4
CS_LONGBINARY	Long variable binary	– –	TDSLONGVARBIN
CS_LONGCHAR	Long variable character	– –	TDSLONGVARCHAR
CS_MONEY	8-byte money	money	TDSMONEY
CS_MONEY4	4-byte money	smallmoney	TDSMONEY4
CS_PACKED370	IBM S/370 packed decimal	decimal	TDS_PACKED_DECIMAL
CS_REAL	4-byte float	real	TDSFLT4
CS_SMALLINT	2-byte integer	smallint	TDSINT2
CS_VARBINARY	Variable-length binary	– –	TDSVARYBIN
CS_VARCHAR	Variable-length character	– –	TDSVARYCHAR
CS_NUMERIC		numeric	TDSNUMERIC
CS_DECIMAL		Sybase decimal	TDS_SYBASE_DECIMAL

Open ClientConnect datatypes

Open ClientConnect datatypes are designed to match the corresponding DB2 datatypes. For example, CS_VARCHAR is the same as the DB2 datatype VARCHAR, which is different from the Adaptive Server varchar type. Open ClientConnect datatypes are described in the following subsections:

- Binary
- Character
- Datetime
- Integer
- Real, float, packed decimal, numeric and Sybase-decimal
- Money

Binary

Open ClientConnect supports the following three binary types:

- CS_BINARY is a binary type.
- CS_VARBINARY is a variable-length binary type. It corresponds to the DB2 datatype VARBINARY, the DB-Library datatype DBVARYBIN, and the Gateway-Library datatype TDSVARYBIN, and includes a length specification (the initial two bytes, referred to in print as "LL") along with the data.
- CS_LONGBINARY is a long variable binary type. It does not include the two-byte "LL" length specification prefix. The default maximum length for this datatype is 32K.

Note CS_VARBINARY does not correspond to the Adaptive Server datatype varbinary. Open ClientConnect converts Adaptive Server varbinary data to CS_VARBINARY.

Character

Open ClientConnect supports the following three character types:

- CS_CHAR is a set-length character type.
- CS_VARCHAR is a variable-length character type. It corresponds to the DB2 datatype VARCHAR, the DB-Library datatype DBVARYCHAR, and the Gateway-Library datatype TDSVARYCHAR, and includes a length specification (the initial two bytes, referred to in print as "LL") along with the data.

- CS_LONGCHAR is a long variable-length character type. It does not include the two-byte "LL" length specification prefix.

Note CS_VARCHAR does not correspond to the Adaptive Server datatype varchar. Open ClientConnect converts Adaptive Server varchar data to CS_VARCHAR.

Datetime

Open ClientConnect supports the following two datetime types that hold 8-byte and 4-byte datetime values, respectively:

- CS_DATETIME corresponds to the Adaptive Server datatype datetime. The range of legal CS_DATETIME values is from January 1, 1753, to December 31, 9999, with a precision of 1/300th of a second (3.33 milliseconds).
- CS_DATETIME4 corresponds to the Adaptive Server datatype smalldatetime. The range of legal CS_DATETIME4 values is from January 1, 1900 to June 6, 2079, with a precision of 1 minute.

Integer

Open ClientConnect supports the following two integer types:

- CS_SMALLINT, a 2-byte integer.
- CS_INT, a 4-byte integer.

Real, float, packed decimal, numeric and Sybase-decimal

Open ClientConnect supports the following five decimal types:

- CS_REAL, a 4-byte decimal value.
- CS_FLOAT, an 8-byte decimal value.

- CS_PACKED370, which is used to handle IBM S/370 packed decimal data. It can be converted to the Adaptive Server money, char, numeric, or Sybase-decimal datatype with cs_convert.

Note CS_PACKED370 values can be negative. The maximum number of decimal places for a packed decimal object is 31.

- CS_NUMERIC, which is used to handle Adaptive Server numeric data. It can be converted to character or packed decimal. For example:

```
typedef struct _cs_numeric
 {
     CS_BYTE  precision;
     CS_BYTE  scale;
     CS_BYTE  array CS_MAX_NAMELEN;
 } CS_NUMERIC;
```

- CS_DECIMAL, which is used to handle Adaptive Server numeric data. It can be converted to character or packed decimal. It is defined the same way as CS_NUMERIC.

Money

Open ClientConnect supports the following two money datatypes that are intended to hold 8-byte and 4-byte money values, respectively:

- CS_MONEY corresponds to the Adaptive Server datatype money.
  The range of legal CS_MONEY values is +/- $922,337,203,685,477.5807.
- CS_MONEY4 corresponds to the Adaptive Server datatype smallmoney.
  The range of legal CS_MONEY4 values is between -$214,748.3648 and +$214,748.3647.

Error and message handling

Description

All Open ClientConnect functions return success or failure indications in their RETCODE arguments. It is highly recommended that applications check these return codes for each call.

In addition, Open ClientConnect applications must handle three types of error and informational messages:

- Open ClientConnect messages, also known as "client messages," are generated by the functions documented in this book. They range in severity from informational messages to fatal errors.
- Operating system messages.
- Server messages—messages generated by the server. Server messages also range in severity from informational messages to fatal errors.

For a list of messages, see the Mainframe Connect Client Option and Server Option *Messages and Codes*.

Return codes

Client-Library return codes begin with "CS_". The codes returned to each Client-Library function are listed on the reference pages for that function, under "Return value."

Gateway-Library return codes all begin with "TDS_". However, on the mainframe, a few TDS_XXX return codes are returned to functions. Some are returned to both Gateway-Library and Client-Library functions.

TDS_XXX return codes that are returned to specific functions are only documented on the reference pages for those functions, under "Return value." Programs using these functions should check for these codes.

Some TDS_XXX return codes can be returned to any function under certain circumstances.

You can find a list of all TDS_XXX codes in the Mainframe Connect Client Option and Server Option *Messages and Codes*.

Messages

A Client-Library application uses the function ct_diag to handle messages in line. ct_diag returns message information to two structures defined within the application:

- The CS_CLIENTMSG structure for client messages
- The CS_SERVERMSG structure for server messages

An application calls ct_diag to initialize in-line message handling for a connection. ct_diag cannot be used at the context level.

Note Whenever a Client-Library function returns CS_FAIL, you must run ct_diag to determine what the error is.

An application can retrieve messages into SQLCA and SQLCODE structures. If the application uses these structures, it must set the property CS_EXTRA_INF to CS_TRUE, using ct_con_alloc.

Note This is because the SQL structures require information that Client-Library does not customarily return. If CS_EXTRA_INF is not set, a loss of information occurs. For more information, see "The CS_EXTRA_INF property" on page 37.

For additional information on the in-line method of handling function and server messages, see "CS_SERVERMSG structure" on page 52.

The CS_EXTRA_INF property

The CS_EXTRA_INF property is used by an application to determine the number of rows affected by the most recent command. If you want this extra information, you must set this property before you call the function.

An application can determine the number of rows in two ways:

- An application that is retrieving messages into a SQLCA or SQLCODE structure must set the property CS_EXTRA_INF to CS_TRUE, using ct_con_props. This is necessary because the SQLCA structure needs to know the number of rows affected, information that functions do not customarily return.

  Note If CS_EXTRA_INF is not set, a loss of information occurs.

- An application that is not using SQLCA or SQLCODE can also set CS_EXTRA_INF to CS_TRUE. In this case, the extra information is returned as standard Client-Library messages.

For further information on the use of the SQLCA and SQLCODE structures in Open ClientConnect, turn to "SQLCA structure" on page 54 and "SQLCODE structure" on page 55.

For three-tier processing, requests directed to a Adaptive Server or an Open Server application are sent to Mainframe ClientConnect (MCC), supplied with the Sybase DirectConnect product.

Note For two-tier processing, requests are sent directly to a Adaptive Server.

Mainframe ClientConnect (MCC) allows mainframe transactions to access the LAN-based environment in which Sybase servers operate. It logs into the target server, passing along login information, does the necessary conversions, and forwards the request. When results are ready, it passes them from the server to the client, again performing any necessary conversions.

MCC is transparent to the mainframe application. You specify the name of the server in the *servername* parameter of the ct_connect call, and MCC forwards the request to the specified server.

To learn how CICS or IMS TM determines the MCC and connections to use to connect to the target server, see the Mainframe Connect Client Option *Installation and Administration Guide*. The client program is not concerned with these details.

Note Requests to Open ServerConnect are sent directly from one transaction processing station to another and do not use Mainframe ClientConnect.

Nulls

Description

Client-Library allows parameters to have a "null" value, that is, to contain no information, not even blanks or zeroes.

NULL and unused parameters

There are several rules for assigning null values to arguments:

- For handles, an application assigns the following symbolic values to indicate a null:
  - CS_NULL_CONTEXT for context handles.
  - CS_NULL_CONHANDLE for connection handles.

    - CS_NULL_CMD for command handles.
- For output arguments:
    - Arguments that return a single integer are never null.
    - Arguments that return longer values can be null. A null value is indicated by a separate argument, indicating that the argument of interest should be treated as null.
- For arguments that have a corresponding length argument, assign the value CS_NULL_STRING to the corresponding length argument, if one is present, to indicate that the value of an argument should be treated as null.

Note For DATAFMT structures, you indicate a null field by setting maxlength to zero.

- For arguments that have NULL indicators, assign CS_PARAM_NULL to the indicator argument.
- For all other variables, assign CS_UNUSED to indicate that the argument should be ignored.

Padding with NULLs

The format field of the DATAFMT structure of ct_bind can be padded with either blanks or nulls. CS_FMT_PADNULL pads with binary zeroes; CS_FMT_PADBLANK pads with blanks.

Properties

Description

Properties define aspects of Open ClientConnect behavior.

- *Login properties* are used when logging into a server.

  These include CS_USERNAME, CS_PASSWORD, CS_PACKETSIZE and CS_NET_DRIVER (used with dynamic network drivers).
- *Negotiated properties* are used by servers when a property value must be changed.

That is, a server can change the values of some login properties during the login process. For example, if an application sets CS_PACKETSIZE to 2048 bytes and then logs into a server that cannot support this packet size, the server overwrites 2048 with a packet size it can support.

Setting and retrieving properties

An application calls ct_config, ct_con_props, and ct_cmd_props to set and retrieve properties at the context, connection, and command structure levels, respectively. An application calls ct_config to set and retrieve most context properties; it calls cs_config to set and retrieve global context properties.

Note When a context structure is allocated, its property values default to standard values.

When a connection structure is allocated, it picks up default property values from its parent context. For example, if CS_TEXTLIMIT is set to 16,000 at the context level, then any connection created within this context has a default text limit value of 16,000. Likewise, when a command structure is allocated, it picks up default property values from its parent connection.

An application can override a default property value by calling cs_config, ct_config, ct_con_props, or ct_cmd_props to change the value of the property.

Note Most property values can be either set or retrieved by an application, but some properties are retrieve only.

Summary of properties

Table 2-8 on page 40 lists the Open ClientConnect properties.

Table 2-8: Open ClientConnect properties

Property	Meaning	Values	Function set by	Notes
CS_APPNAME	The application name used when logging into the server.	A character string. The default is NULL.	ct_con_props	Login property. Takes effect only if set before the connection is established.
CS_CHARSETCNV	The conversion indicator. It indicates whether or not character set conversion is taking place.	CS_TRUE or CS_FALSE. A default is not applicable.	ct_con_props	Retrieve only, after the connection is established.

Property	Meaning	Values	Function set by	Notes
CS_COMMBLOCK	A pointer to a communication sessions block (EIB).	A pointer value. The default is NULL.	ct_con_props	Takes effect only if set before the connection is established.
CS_EXTRA_INF	The extra information indicator. It specifies whether or not to return the extra information that is required when processing messages in-line using a SQLCA or SQLCODE.	CS_TRUE or CS_FALSE. The default is CS_FALSE.	cs_config, ct_con_props	
CS_HOSTNAME	The host (server) machine name.	A character string. The default is NULL.	ct_con_props	Login property. Takes effect only if set before the connection is established.
CS_LOC_PROP	A pointer to a CS_LOCALE structure that defines localization information.	A pointer value. A connection picks up a default CS_LOC_PROP from its parent context.	ct_con_props	Login property.
CS_LOGIN_STATUS	The connection status indicator. It indicates whether or not the connection is open.	CS_TRUE or CS_FALSE. A default is not applicable.	ct_con_props	Retrieve only.
CS_LOGIN_TIMEOUT	The login timeout value.	An integer value. The default is 60 seconds. A value of CS_NO_LIMIT represents an infinite timeout period.	ct_config	Open ClientConnect ignores this property. This is the same value as the CICS RTIMEOUT.

Property	Meaning	Values	Function set by	Notes
CS_MAX_CONNECT	The maximum number of connections for this context.	An integer value. The default varies by platform. On mainframes, the default is 25 (an unlimited number of connections can be defined for a context).	ct_config	
CS_NET_DRIVER	The type of network driver in use.	CS_LU62, CS_TCPIP, CS_INTERLINK, or CS_NCPIC. Defaults for: CICS – CS_LU62 IMS – CS_LU62 MVS – CS_NCPIC	ct_con_props	
CS_NETIO	The sync/async indicator. It indicates whether network I/O is synchronous or asynchronous.	CS_SYNC_IO or CS_ASYNC_IO. The default is CS_SYNC_IO.	ct_config, ct_con_props	With Open ServerConnect this value is always CS_SYNC_IO.
CS_NOINTERRUPT	The interrupt indicator. It indicates whether or not the application can be interrupted.	CS_TRUE or CS_FALSE. The default is CS_FALSE, which means the application can be interrupted.	ct_config, ct_con_props	N/A for CICS. This property is included for compatibility with other Open Client libraries.
CS_PACKETSIZE	The TDS packet size.	An integer value. The default varies by platform. On UNIX and MVS platforms, the default is 512 bytes.	ct_con_props	Negotiated login property. Takes effect only if set before the connection is established.
CS_PASSWORD	The password used to log into the server.	A character string. The default is NULL.	ct_con_props	Login property. Takes effect only if set before the connection is established.

Property	Meaning	Values	Function set by	Notes
CS_TDS_VERSION	The version of the TDS protocol that the connection is using.	A symbolic version level. CS_TDS_VERSION defaults to the value of CS_VERSION.	ct_con_props	Negotiated login property. Takes effect only if set before the connection is established.
CS_TEXTLIMIT	The largest text or image value to be returned on this connection.	An integer value. The default is CS_NO_LIMIT.	ct_config, ct_con_props	
CS_TIMEOUT	The timeout value.	An integer value. The default is CS_NO_LIMIT.	ct_config	Not supported under CICS. CICS waits forever.
CS_TRANSACTION_ NAME	A transaction name.	A string value. The default is NULL.	ct_con_props	
CS_USERDATA	User–allocated data.	User–allocated data.	ct_con_props, ct_cmd_props	These are pointers to data that allow the customer to tie into the data.
CS_USERNAME	The name used to log into the server.	A character string. The default is NULL.	ct_con_props	Login property. Takes effect only if set before the connection is established.
CS_VERSION	The version of Client-Library used by this context.	CS_VERSION gets its value from a context ct_init call.	cs_config, ct_config	

About the properties

Application name

- CS_APPNAME defines the application name that a connection uses when connecting to a server.

Character set conversion

- CS_CHARSETCNV indicates whether or not the server is converting between the client and server character sets. This property is retrieve-only, after a connection is established.
- A value of CS_TRUE indicates that the server is converting between the client and server character sets; CS_FALSE indicates that conversion does not occur.

Communications session block

- CS_COMMBLOCK defines a pointer to a communications block (EIB).

Extra information

- CS_EXTRA_INF determines whether or not Open ClientConnect returns the extra information that ct_diag requires to fill in SQLCA or SQLCODE structures.
    - This extra information includes the number of rows affected by the most recent command.
    - If an application is not retrieving messages into a SQLCA or SQLCODE, the extra information is returned as ordinary Client-Library messages.

Host name

- CS_HOSTNAME is the name of the host machine, used when logging into a server.

Locale information

- CS_LOC_PROP defines a pointer to a CS_LOCALE structure, which contains localization information. Localization information includes a language, a character set, datetime, money, and numeric formats, and a collating sequence. This property must be set to 0.

Login status

- CS_LOGIN_STATUS is CS_TRUE if a connection is open, CS_FALSE if it is not. This property can only be retrieved.

- ct_connect is used to open a connection.

Login timeout

- CS_LOGIN_TIMEOUT defines the length of time, in seconds, that an application waits for a login response when making a connection attempt. Timeouts are not supported under CICS.

Maximum number of connections

- CS_MAX_CONNECT defines the maximum number of simultaneously open connections that a context can have. The default varies by platform. Negative and zero values are not allowed for CS_MAX_CONNECT.
    - On mainframes, CS_MAX_CONNECT has a default value of 25 (an unlimited number of connections can be defined for a context).
    - If ct_config is called to set a value for CS_MAX_CONNECT which is less than the number of currently open connections, ct_config generates an error and returns CS_FAIL without altering the value of CS_MAX_CONNECT.

Network driver

- CS_NET_DRIVER determines the type of dynamic network driver that is used. Possible values are:
    - CS_INTERLINK
    - CS_LU62
    - CS_NCPIC
    - CS_TCPIP

Note The default value for CICS and IMS is CS_LU62. The default value for MVS is CS_NCPIC.

Network I/O

- CS_NETIO determines whether a connection is synchronous or asynchronous.

- Because Open ClientConnect does not support asynchronous processing, this value is always CS_SYNC_IO.

No interrupt

- CS_NOINTERRUPT is not supported for Open ClientConnect. It is included for compatibility with other Open Client libraries.

Packet size

- CS_PACKETSIZE determines the packet size that Open ClientConnect uses when sending Tabular Data Stream (TDS) packets.
- If an application needs to send or receive large amounts of text, image, or bulk data, a larger packet size can improve efficiency. The default packet size is 512.

Password

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

TDS version

- CS_TDS_VERSION defines the version of the Tabular Data Stream (TDS) protocol that the connection is using.

  Because CS_TDS_VERSION is a negotiated login property, its value can change during the login process. An application can set CS_TDS_VERSION to request a TDS level before calling ct_connect. In this case, when ct_connect creates the connection, it looks for the requested TDS version. If the server cannot provide the requested TDS version, a new (lower) TDS version is negotiated. An application can retrieve the value of CS_TDS_VERSION after a connection is established to determine the actual version of TDS in use.

  Table 2-9 lists the symbolic values that CS_TDS_VERSION can have.

Table 2-9: Values for CS_TDS_VERSION

Value	Indicates	Features supported
CS_TDS_46	4.6 TDS	Registered procedures, TDS passthrough, negotiable TDS packet size, multi-byte character sets.

Value	Indicates	Features supported
CS_TDS_50	5.0 TDS	Accesses system Adaptive Server 10.0 and above. **Note** TDS 5.0 only works with a Adaptive Server 10.0 and above.

Text and image limit

- CS_TEXTLIMIT indicates the length, in bytes, of the longest text or image value that an application wants to receive. Open ClientConnect reads but ignores any part of a text or image value that goes over this limit.
  - The default value of CS_TEXTLIMIT is CS_NO_LIMIT, meaning the application reads and returns all data sent by the server.

Timeout

- CS_TIMEOUT controls the length of time, in seconds, that Client-Library waits for a server response when making a request. This value is ignored by Open ClientConnect.

Transaction name

- CS_TRANSACTION_NAME names a transaction. If the accessed server is a Gateway-Library application, this is the name of the transaction.
  - Calls to Adaptive Server do not require a transaction name.
  - All Client-Library applications can set CS_TRANSACTION_NAME. If a transaction name is not required, CS_TRANSACTION_NAME is ignored.

User data

- CS_USERDATA defines user-allocated data. This property allows an application to associate user data with a particular connection or command structure. An application allocates a data space from which it can get this data when needed.
- To associate user data with a context structure, an application calls cs_config.
- A program can use the Working Storage section to define this data.

User name

- CS_USERNAME defines the user login name that the connection uses to log into a server.

Version of Open ClientConnect

- CS_VERSION represents the version of Open ClientConnect behavior than an application requested via ct_init. The value of this property can only be retrieved.
- Connections allocated within a context pick up default CS_TDS_VERSION values from their parent context CS_VERSION level.

Remote procedure calls (RPCs)

Description

A client application can call a stored procedure on a Adaptive Server or an Open ServerConnect transaction running in a separate CICS or IMS region.

A client application can call a stored procedure or mainframe transaction in two ways:

- By executing a SQL language request (for example, "execute myproc").
- By making an RPC.

Comparing RPCs and *execute* statements

RPCs have a few advantages over execute statements:

- An RPC can execute a Adaptive Server stored procedure or any Open ServerConnect transaction.

  A SQL language request can only execute a Adaptive Server stored procedure or a specially written Open ServerConnect language transaction.

- When sending a request to a Adaptive Server, it is simpler and faster to accommodate stored procedure return parameters if the procedure is invoked with an RPC instead of a language request.

RPC routines

The following functions are related to RPCs:

- ct_remote_pwd sets and clears the passwords that are used when logging into a remote server (This feature is not available for calls to Open ServerConnect).

- ct_command initiates an RPC.
- ct_param defines parameters for an RPC.
- ct_send sends an RPC.
- ct_results, ct_bind, and ct_fetch process remote procedure results.

Executing remote procedures

A server can execute a stored procedure or transaction residing on another server. This might occur when a stored procedure being executed on one Adaptive Server contains an execute statement for a stored procedure on another Adaptive Server. The execute command causes the first server to log into the second server and execute the remote procedure. This is called a server-to-server RPC. It happens without any intervention from the application, although the application can specify the remote password that the first server uses to log into the second.

A server-to-server RPC also occurs when an application sends a request to execute a stored procedure that does not reside on the server to which it is directly connected.

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

RPC results

In addition to results generated by the SQL statements they contain, Adaptive Server stored procedures and Open ServerConnect transactions that are executed through an RPC:

- Can generate a return parameter result set
- Always generate a return status result set

All types of results—rows, status, and parameters—can be processed using ct_results, ct_bind, and ct_fetch.

Stored procedure return parameters

Adaptive Server stored procedures and mainframe server transactions can return values for specified return parameters. Changes made to the value of a return parameter inside the stored procedure or transaction are then available to the program that called the procedure or transaction. This is analogous to the “pass by reference” facility available in some programming languages.

In order for a parameter to function as a return parameter, it must be declared as such within the stored procedure. For example, Client-Library applications use the ct_param routine to indicate return parameters.

Processing RPC return parameters

Return parameter values are available to an application as a parameter result set only if the application invoked the stored procedure using an RPC.

ct_results returns CS_PARAM_RESULT if a parameter result set is available to be processed. Because stored procedure parameters are returned to an application as a single row, one call to ct_fetch copies all of the return parameters for a stored procedure into the program variables designated through ct_bind. However, an application should always call ct_fetch in a loop until it returns CS_END_DATA.

When executing a stored procedure, the server returns any parameter values immediately after returning all row results. Therefore, an application can fetch return parameters only after processing the stored procedure row results. A stored procedure can generate several sets of row results, one for each select it contains. An application must call ct_results and ct_fetch as many times as necessary to process these row results before calling ct_fetch to fetch the stored procedure return parameters.

Stored procedure return status

Adaptive Server, Open Server, and Open ServerConnect applications can all return a status.

All stored procedures that run on a Adaptive Server version 4.0 or later return a status. Stored procedures usually return 0 to indicate normal completion. For a list of Adaptive Server default return status values, see return in the Adaptive Server Enterprise *Reference Manual*, which is part of the basic Sybase documentation set. Open ServerConnect status values are documented under TDSNDDON and TDSTATUS in the Mainframe Connect Server Option *Programmer's Reference for C*.

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

When executing a stored procedure, Adaptive Server returns the status immediately after returning all other results. Therefore, an application can fetch a return status only after processing the stored procedure row and parameter results,
if any.

Open Server applications return the status after any row results, but either before or after return parameters.

Processing RPC return status

ct_results returns CS_STATUS_RESULT if a return status result set is available to be processed. Because a return status result set contains only a single value, one call to ct_fetch copies the status into the program variable designated through ct_bind. However, an application should always call ct_fetch in a loop until it returns CS_END_DATA.

Results

Description

When a client request executes a server procedure or transaction, it can generate the following types of result sets that are returned to the client application:

- Regular row results, which contain one or more rows of tabular data.
- Return parameter results, which contain a single row of return parameter data. Return parameters are values returned by stored procedures and transactions in the parameters (arguments) of the called function. For information on return parameters, see "Remote procedure calls (RPCs)" on page 48.
- Return status results, which contain a single return status value. For more information on a stored procedure return status, see "Remote procedure calls (RPCs)" on page 48.

Note These are the only result types supported by Open ClientConnect. Although additional result types are supported by Open Client for other platforms, they are not supported on the mainframe.

Result sets

Results are returned to an application in the form of result sets. A result set contains only a single type of result data. Regular row result sets can contain multiple rows of data, but other types of result sets contain at most a single row of data.

Processing results

An application processes results by calling ct_results, which indicates the type of result available by setting the *result_type* argument. The application calls ct_results once for each result row. ct_results returns CS_CMD_DONE in result_type to indicate that a result set processed completely.

Note Some requests, for example a language request containing a Transact-SQL update statement, do not generate results. ct_results returns CS_CMD_SUCCEED to indicate the success of a request that does not return results.

CS_SERVERMSG structure

Description

A CS_SERVERMSG (server message) structure contains information about an error or informational message returned by the server. This structure is defined within the application. ct_diag returns a message string and information about the message in this structure.

Client messages are returned to a CS_CLIENTMSG structure as described in "CS_CLIENTMSG structure" on page 26.

CS_CLIENTMSG and CS_SERVERMSG structures are part of the Mainframe ClientConnect (MCC) *CTPUBLIC include* file.

This structure contains information about all messages received by the client application, including MCC messages, messages returned by the remote transactions, and messages returned by the database, such as DB2 Access Module messages and Adaptive Server messages.

A CS_SERVERMSG structure is defined as follows:

```
typedef struct_cs_servermsg {
CS_MSGNUM msgnumber;
CS_INT state;
CS_INT severity;
CS_CHAR text[CS_MAX_MSG];
CS_INT textlen;
CS_CHAR svrname[CS_MAX_MSG];
CS_INT svrnlen;

 /*
**RPC's may be involved. If so, the
**following have info in them.

*/

CS_CHAR proc[CS_MAX_MSG];
CS_INT proclen;
CS_INT line;
CS_INT status;
} CS_SERVERMSG
```

Description of arguments in *CS_SERVERMSG* structure

- *msgnumber* is the server message number. This field corresponds to the *message_number* argument of the Gateway-Library function tdsndmsg.
- *state* is the message state. This field corresponds to the *error_state* argument of the Gateway-Library function tdsndmsg.
- *severity* is a symbolic value representing the severity of the message. Severity values are provided in the *CTPUBLIC include* file. This field corresponds to the *severity* argument of the Gateway-Library function tdsndmsg.

Table 2-10 lists the legal values for *severity*.

Table 2-10: Values for the CS_SERVERMSG severity field

Severity value	Meaning
CS_SV_INFORM (0)	No error occurred. The message is informational.
CS_SV_API_FAIL (1)	A Client-Library routine generated an error. This error is typically caused by a bad parameter or calling sequence. The server connection is probably salvageable.
CS_SV_RETRY_FAIL (2)	An operation failed, but it can be retried.
CS_SV_RESOURCE_FAIL (3)	A resource error occurred. This error is typically caused by an allocation error, a lack of file descriptors, or timeout error. The server connection is probably not salvageable.
CS_SV_CONFIG_FAIL (4)	A configuration error occurred.
CS_SV_COMM_FAIL (5)	An unrecoverable error in the server communication channel occurred. The server connection is not salvageable.
CS_SV_INTERNAL_FAIL (6)	An internal Client-Library error occurred.
CS_SV_FATAL (7)	A serious error occurred. All server connections are unusable.

- *text* is the text of the message string. This field corresponds to the *message_text* argument of the Gateway-Library function tdsndmsg.
- *textlen* is the length, in bytes, of *text*. If there is no message text, the value of *textlen* is 0. This field corresponds to the *message_length* argument of the Gateway-Library function tdsndmsg.
- *svrname* is the name of the server that generated the message. This is the server name from the Server Path Table.

  The Server Path Table contains the information the Client-Library programs need to route requests to a remote server, including the name of the server and the connections that can be used to access that server. This table is part of the Connection Router, described in the Mainframe Connect Server Option *Installation and Administration Guide*.
- *svrnlen* is the length, in bytes, of *svrname*.
- *proc* is the name of the remote procedure or transaction that returned the message. That is, the name of the Adaptive Server stored procedure or the transaction ID of the mainframe transaction. This field corresponds to the *transaction_id* argument of the Gateway-Library function tdsndmsg.
- *proclen* is the length, in bytes, of *proc*. This field corresponds to the *transaction_id_length* argument of the Gateway-Library function tdsndmsg.

- *line* is the line number in the called procedure or transaction where the error occurred. It may also be used for miscellaneous information. This field corresponds to the *line_id* argument of the Gateway-Library function tdsndmsg.
- *status* is reserved for future use.

SQLCA structure

Description

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

A SQLCA structure is defined as follows:

```
        /* SQL Communication Area - SQLCA */
    typedef struct sqlca
  {
    unsigned char  sqlcaid[8];      /* Eyecatcher = 'SQLCA' */
    long           sqlcabc;         /* SQLCA size in bytes = 136 */
    long           sqlcode;         /* SQL return code */
    short          sqlerrml;        /* Length for SQLERRMC */
    unsigned char  sqlerrmc[256];   /* Error message tokens */
    unsigned char  sqlerrp[8];      /* Diagnostic information */
    long           sqlerrd[6];      /* Diagnostic information */
    unsigned char  sqlwarn[8];      /* Warning flags */
    unsigned char  sqlext[8];       /* Reserved */
  } SQLCA;
```

Description of arguments in SQLCA structure

- *sqlcaid* is “SQLCA” (This value is automatically provided).
- *sqlcabc* is ignored.
- *sqlcode* is the server or Client-Library message number.
  For information on how Client-Library maps message numbers to SQLCODE, see “SQLCODE structure” on page 55. For a list of gateway messages, see the Mainframe Connect Client Option and Server Option *Messages and Codes* shipped with this product.
- *sqlerrml* is the length of the actual message text (not the length of the text placed in *sqlerrmc*).
- *sqlerrmc* is the null-terminated text of the message. If the message is too long for the array, Client-Library truncates it before appending the null terminator.

- *sqlerrp* is the first eight characters of the stored procedure, if any, being executed at the time of the error.
- *sqlerrd* is the number of rows successfully inserted, updated, or deleted before the error occurred.
- *sqlwarn* is an array of warnings:
  - If *sqlwarn0* is blank, all other *sqlwarn* variables are blank. If *sqlwarn0* is not blank, at least one other *sqlwarn* variable is set to “W”.
  - If *sqlwarn1* is W, Client-Library truncated at least one column’s value when storing it into a mainframe variable.
  - If *sqlwarn2* is W, at least one null value was eliminated from the argument set of a function.
  - If *sqlwarn3* is W, the number of mainframe variables specified in the into clause of a select statement is not equal to the number of result columns.
  - If *sqlwarn4* is W, a dynamic SQL update or delete statement did not include a where clause.
  - If *sqlwarn5* is W, a server conversion or truncation error occurred.
  - *sqlext* is ignored.

SQLCODE structure

Description

A SQLCODE structure can be used in conjunction with ct_diag to retrieve Client-Library and server error and informational messages. A SQLCODE structure can be located anywhere and mapped to SQLCA.

Client-Library always sets SQLCODE and the *sqlcode* field of the SQLCA structure identically (See “SQLCA structure” on page 54 for more information).

Note A SQLCODE structure is defined as a 4-byte integer.

Mapping server messages to SQLCODE

A server message number is mapped to a SQLCODE of 0 when it has a severity of 0.

Other server messages may be mapped to a SQLCODE of 0 as well.

Server message numbers are negated before being placed into SQLCODE. This ensures that SQLCODE is negative when an error occurs.

For a list of server messages returned by gateway products (Mainframe ClientConnect, Open ServerConnect, or OmniSQL Access Module for DB2), see the messages and codes for any of these products.

Mapping Client-Library messages to SQLCODE

The Client-Library message "No rows affected" is mapped to a SQLCODE of 100.

Client-Library messages with CS_SV_INFORM severities are mapped to a SQLCODE of 0.

Other Client-Library messages may be mapped to a SQLCODE of 0 as well.

Client-Library message numbers are negated before being placed into SQLCODE. This ensures that SQLCODE is negative when an error occurs.

For a list of Client-Library messages, see the Mainframe Connect Client Option and Server Option *Messages and Codes*.

Handles

Client-Library uses handles at three levels. Each handle defines and manages a particular environment. Each type of handle can have certain properties, listed below.

Note Most Client-Library functions include a handle argument. An application must allocate these handles before using them as arguments.

Types of handles

The following handles are used with Client-Library:

- *Context handle*. A context handle defines a particular application, context, or operating environment. The context handle is defined in the program's cs_ctx_alloc call.

  An application can have only one context.

  A context handle corresponds to the IHANDLE structure in the Open ServerConnect Gateway-Library.

  The context handle can have the following properties listed in Table 2-11.

Table 2-11: Context handle properties

Property	Set by
CS_EXTRA_INF	cs_config
CS_LOGIN_TIMEOUT	ct_config
CS_MAX_CONNECT	ct_config
CS_NETIO	ct_config
CS_NOINTERRUPT	ct_config
CS_TEXTLIMIT	ct_config
CS_TIMEOUT	ct_config
CS_VERSION	cs_config

- *Connection handle*. This is the handle for an individual client/server connection. The connection handle is defined in the program's ct_con_alloc call. If parallel sessions are used, there must be one connection handle for each session. An application can have up to 25 connections.

  Open ClientConnect uses a Connection Router program to define connections. Each connection handle corresponds to a connection defined with the Connection Router. For details about the Connection Router, see the Mainframe Connect Client Option *Installation and Administration Guide*.

  A connection handle can have the following properties listed in Table 2-12.

Table 2-12: Connection handle properties

Property	Set by
CS_APPNAME	ct_con_props
CS_CHARSETCNV	ct_con_props
CS_COMMBLOCK	ct_con_props
CS_EXTRA_INF	ct_con_props
CS_HOSTNAME	ct_con_props
CS_LOC_PROP	ct_con_props
CS_LOGIN_STATUS	ct_con_props
CS_NET_DRIVER	ct_con_props
CS_NETIO	ct_con_props
CS_NOINTERRUPT	ct_con_props
CS_PACKETSIZE	ct_con_props
CS_PASSWORD	ct_con_props
CS_TDS_VERSION	ct_con_props
CS_TEXTLIMIT	ct_con_props
CS_TRANSACTION_NAME	ct_con_props
CS_USERNAME	ct_con_props

- *Command handle*. A command handle defines a command space, which is used to send commands to a server over a connection and process the results. A command handle is defined in the program call ct_cmd_alloc. Each command handle is associated with a particular connection. A connection can have any number of command handles associated with it.

  A command handle and its associated connection handle correspond to the TDPROC handle in the Open ServerConnect Gateway-Library. A command handle can have the following property:

  - CS_USERDATA (set by ct_cmd_props)

Routines that affect handles

Table 2-13 lists the routines that allocate, use, and deallocate handles.

Table 2-13: Routines that manipulate hidden structures

Structure	Allocated and used by
CONTEXT	cs_ctx_alloc, ct_config, cs_config, cs_ctx_drop
CONNECTION	ct_con_alloc, ct_con_props, ct_con_drop
COMMAND	ct_cmd_alloc, ct_cmd_props, ct_cmd_drop

CHAPTER 3 Functions

This chapter describes the functions that are included with your Open ClientConnect software. Following Table 3-1 is a detailed description of each listed one.

Table 3-1: Functions included with Open ClientConnect

Function and location	***Description***
ct_bind (see ct_bind on page 61)	Binds a returned column or parameter to a program variable.
ct_cancel (see ct_cancel on page 69)	Cancels a request or the results of a request.
ct_close (see ct_close on page 72)	Closes a server connection.
ct_cmd_alloc (see ct_cmd_alloc on page 75)	Allocates a command handle.
ct_cmd_drop (see ct_cmd_drop on page 78)	Deallocates a command handle.
ct_cmd_props (see ct_cmd_props on page 81)	Sets, retrieves, or clears information about the current result set.
ct_command (see ct_command on page 83)	Initiates a language request or RPC.
ct_con_alloc (see ct_con_alloc on page 88)	Allocates a connection handle.
ct_con_drop (see ct_con_drop on page 93)	Deallocates a connection handle.
ct_config (see ct_config on page 96)	Sets or retrieves context properties.
ct_connect (see ct_connect on page 99)	Connects to a server.
ct_con_props (see ct_con_props on page 104)	Sets or retrieves connection handle properties.
ct_describe (see ct_describe on page 109)	Returns a description of result data.
ct_diag (see ct_diag on page 116)	Manages in-line error handling.
ct_exit (see ct_exit on page 131)	Exits the programming interface.
ct_fetch (see ct_fetch on page 134)	Fetches result data.
ct_get_format (see ct_get_format on page 140)	Returns the server-defined format for a result column.
ct_init (see ct_init on page 141)	Initializes the programming interface.
ct_param (see ct_param on page 145)	Defines a command parameter.
ct_remote_pwd (see ct_remote_pwd on page 152)	Defines or clears passwords to be used for server-to-server connections.
ct_res_info (see ct_res_info on page 154)	Returns result set information.
ct_results (see ct_results on page 160)	Sets up result data to be processed.

Function and location	*Description*
ct_send (see ct_send on page 165)	Sends a request to the server.
cs_config (see cs_config on page 171)	Sets or retrieves global context properties.
cs_convert (see cs_convert on page 174)	Converts a data value from one datatype to another.
cs_ctx_alloc (see cs_ctx_alloc on page 179)	Allocates a context structure.
cs_ctx_drop (see cs_ctx_drop on page 182)	Deallocates a context structure.

ct_bind

Description

Associates a returned column, parameter, or status with a program variable.

Syntax

```
CS_RETCODE ct_bind (command, item_num, datafmt,
               buffer, copied, indicator);
CS_COMMAND    *command;
CS_LONG       item_num;
CS_DATAFMT    *datafmt;
CS_BYTE       *buffer;
CS_INT        *copied;
CS_SMALLINT   *indicator;
```

Parameters

command

(I) Handle for this connection. This is the handle defined in the ct_cmd_alloc call for this connection. The command handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

item_num

(I) Ordinal number of the result column, return parameter, or return status value that is to be bound.

When binding a result column, *item_num* is the column number. For example, the first column in the select list of a SQL select statement is column number 1, the second is column number 2, and so forth.

When binding a return parameter, *item_num* is the ordinal rank of the return parameter. The first parameter returned by a procedure or parameter is number 1. Adaptive Server Enterprise stored procedure return parameters are returned in the order originally specified in the create procedure statement for the stored procedure. This is not necessarily the same order as specified in the RPC that invoked the stored procedure or transaction.

In determining what number to assign to *item_num*, do not count non-return parameters. For example, if the second parameter in a stored procedure is the only return parameter, its *item_num* is 1.

When binding a stored procedure return status, *item_num* must be 1. There is only one column and one row in a return status result set.

To clear all bindings, assign *item_num* a value of CS_UNUSED with *datafmt*, *buffer*, *copied*, and *indicator* as CS_NULL.

DATAFMT

(I) A structure that contains a description of the destination variable(s). This structure is also used by ct_describe, ct_param, and cs_convert and is explained in Chapter 2, "Topics", under "DATAFMT structure" on page 28.

Table 3-2 lists the fields in the *DATAFMT* structure, indicates whether they are used by ct_bind, and contains general information about each field. ct_bind ignores datafmt fields that it does not use.

Warning! You must initialize the entire *DATAFMT* structure to zeroes. Failure to do so causes addressing exceptions.

Table 3-2 lists the fields in the *DATAFMT* structure for ct_bind.

Table 3-2: Fields in the DATAFMT structure for ct_bind

Field	When used	Value represents
name	Not used (CS_FMT_UNUSED).	Not applicable.
namelen	Not used (CS_FMT_UNUSED).	Not applicable.
datatype	When binding all types of results.	The datatype of the destination variable (*buffer*). All datatypes listed under "DATAFMT structure" on page 28 are valid. ct_bind supports a wide range of datatype conversions, so datatype can be different from the datatype returned by the server. For instance, by specifying a datatype of CS_FLOAT, you can bind a CS_MONEY or CS_MONEY4 value to a float-type program variable. The appropriate data conversion happens automatically. A return status always has a datatype of CS_INT.
format	When binding results to character or binary destination variables. In all other cases, this field is unused (CS_FMT_UNUSED).	The destination format of character or binary data. For character-type destinations only: CS_FMT_PADBLANK—pads to the full length of the variable with blanks. For character or binary type destination variables: CS_FMT_PADNULL—pads to the full length of the variable with nulls.

Field	When used	Value represents
maxlength	When binding all types of results to non-fixed-length types. maxlength is ignored when binding to fixed-length datatypes.	The length of the destination variable, in bytes. If *buffer* has more than one element (that is, it is an array), maxlength is the length of one element. When binding to character or binary destinations, maxlength must describe the total length of the destination variable, including any space required for special terminating bytes, with this exception: when binding to a VARYCHAR-type destination such as DB2's VARCHAR, maxlength does not include the length of the "LL" length specification. To clear bind values, assign maxlength a value of 0. If the length specified in maxlengthis too small to hold a result data item, then, at fetch time, ct_fetch discards the result item that is too large, fetches any remaining items in the row, and returns CS_ROW_FAIL. If this occurs, the contents of *buffer* are undefined. When binding Sybase-numerical/decimal to char, use ct_describe to determine precision. maxlength should be precision + 2 in this case. When binding to packed-decimal, ct_bind calculates maxlength as (precision/2) + 1.
scale	Only when converting column results or return parameters to or from an Open ServerConnect packed decimal (CS_PACKED370), Sybase-decimal, and Sybase-numeric datatypes.	The number of digits to the right of the decimal point. If the source value is the same datatype as the destination value, set scaleto CS_SRC_VALUE to indicate that the destination variable should pick up the value for scale from the source data. scale must be less than or equal to precision and cannot be greater than 31. If the actual scale is greater than the scale specified in scale but not greater than 31, ct_bind truncates the results and issues a warning. If the actual scale is greater than 31, the ct_bind call fails. When binding sybase-numeric/decimal to char or packed-decimal use ct_describe to determine precision and scale.

Field	When used	Value represents
precision	Only when binding column results or return parameters to an Open ServerConnect packed decimal (CS_PACKED370), Sybase-decimal, and Sybase-numeric datatypes.	The total number of decimal digits in the destination variable. If the source data is the same datatype as the destination variable, setting precision to CS_SRC_VALUE instructs the destination variable to pick up its value for precision from the source data. If the precision of the value fetched exceeds the precision of the destination variable, ct_bind returns a warning message. precision must be greater than or equal to scale and cannot be less than 1 or greater than 31.
status	Not used (CS_FMT_UNUSED).	Not applicable.
count	When binding all types of results. Only regular row result sets ever contain multiple rows. Other types of results (for example, return parameters, status) are treated like a single row of results.	The number of result rows to be copied to program variables per ct_fetch call. If count is larger than the number of available rows, only the available rows are copied. count must have the same value for all columns in a result set: • If count is 0 or 1, 1 row is fetched; • If count is greater than 1, it represents the number of rows that are fetched. In this case, buffer must be an array. **Note** Only regular row result sets can contain multiple rows. Other types of results (such as return parameters and status) are treated like a single row of results.
usertype	Not used (CS_FMT_UNUSED).	Not applicable.
locale	Not used (CS_FMT_UNUSED).	Reserved for future use.

buffer

(I) Destination variable. A single field or an array of *n* elements where *n* is count. Each array element is of size maxlength.

buffer is the program variable to which ct_bind binds the server results. When the application calls ct_fetch to fetch the result data, it is copied into this space.

If you no longer want to store incoming data in this buffer, set maxlength to 0. This clears the binding.

copied

(O) Length of the incoming data. This can be a single field or, if *buffer* is an array, it can be an array of *n* elements where *n* is count. At fetch time, ct_fetch fills copied with the length(s) of the copied data.

indicator

(O) From 1 to count integer variables. At fetch time, ct_fetch uses each variable to indicate the following conditions about the fetched data:

Value	Integer value	Meaning
CS_NULLDATA	-1	There was no data to fetch. In this case, data is not copied to the destination variable.
CS_GOODDATA	0	The fetch was successful.

If *buffer* is an array, *indicator* is also an array.

Return value

ct_bind returns one of the following values listed in Table 3-3.

Table 3-3: return values

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.
TDS_CONNECTION_TERMINATED (-4997)	The connection is not active.
TDS_INVALID_DATAFMT_VALUE (-181)	datafmt field contains an illegal value.
TDS_INVALID_PARAMETER (-4)	A parameter was given an illegal value.
TDS_INVALID_VAR_ADDRESS (-175)	This value cannot be NULL.
TDS_NO_COMPUTES_ALLOWED (-60)	Compute results are not supported.
TDS_RESULTS_CANCELED (-49)	A cancel was sent to purge results.
TDS_SOS (-257)	Memory shortage. The operation failed.
TDS_WRONG_STATE (-6)	Program is in the wrong communication state to issue this call.

Examples

This code fragment demonstrates the use of ct_bind to bind returned data to program variables. It is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
/*****************************************************************/
/*                                                               */
/* Subroutine to bind each data                                  */
/*                                                               */
/*****************************************************************/
void   bind_columns (parm_cnt)

CS_INT        parm_cnt;
{
CS_INT        rc;
CS_INT        col_len;
CS_DATAFMT    datafmt;

      rc= ct_describe(cmd, parm_cnt, &datafmt);

      if (rc != CS_SUCCEED)
      {
         strncpy (msgstr, "CT_DESCRIBE failed", msg_size);
         no_errors_sw = FALSE ;
         error_out(rc);
      }

/*--------------------------------------------------------------*/
/* We need TO bind the data TO program variables. We don't    */
/* care about the indicator variable so we'll pass NULL for   */
/* that PARAMeter in OC_BIND().                               */
/*--------------------------------------------------------------*/

/*--------------------------------------------------------------*/
/* rows per fetch                                             */
/*--------------------------------------------------------------*/

      switch (datafmt.datatype)
      {
/*--------------------------------------------------------------*/
/* bind the first column, FIRSTNME defined as VARCHAR(12)     */
/*--------------------------------------------------------------*/
          case  CS_VARCHAR_TYPE:

             rc= ct_bind(cmd, parm_cnt, &datafmt, &col_firstnme,
                          &col_len, CS_NULL);

             if (rc != CS_SUCCEED)
```

```
                {
                   strncpy (msgstr, "CT_BIND CS_VARCHAR_TYPE failed",
                   msg_size);
                   no_errors_sw = FALSE ;
                   error_out(rc);
                }
                break;
    /*------------------------------------------------------------*/
    /* bind the second column, EDLEVEL defined as SMALLINT        */
    /*------------------------------------------------------------*/
             case  CS_SMALLINT_TYPE:

                rc= ct_bind(cmd, parm_cnt, &dtafmt, &col_edlevel,
                            &col_len, CS_NULL);

                if (rc != CS_SUCCEED)
                {
                   strncpy (msgstr, "CT_BIND CS_SMALLINT_TYPE failed",
                   msg_size);
                   no_errors_sw = FALSE ;
                   error_out(rc);
                }
                break;

             default:
                break;

         } /* end of switch (datatype) */

   }   /* end bind_columns */
```

Usage

- ct_bind associates ("binds") a column, parameter, or status returned by a server to a program variable. Once a result is bound to a variable, any information returned in that column or parameter, or any status returned during a ct_fetch call is copied to that variable.
- An application must call ct_bind once for each result column or return parameter.
- ct_bind can be used to bind a result column, a return parameter, or a stored procedure status value. When binding a result column, a single call to ct_bind can bind multiple rows of the column. When binding a return status, you must bind a single variable of type integer.

- An application calls ct_bind after ct_results and before ct_fetch. ct_results tells the application whether there are any results to be bound and if so, what kind; ct_fetch retrieves the results and copies them into the bound variable.
- ct_bind binds only the current result type. ct_results indicates the current result type via its *result_type* argument. For example, if ct_results returns CS_STATUS_RESULT, a return status is available for binding.
- An application can call ct_res_info to determine the number of items in the current result set, and can call ct_describe to get a description of each item.
- An application can only bind a result item to a single program variable. If an application binds a result item to multiple variables, only the last binding takes effect.
- Binding for a particular type of result remains in effect until ct_results returns CS_CMD_DONE to indicate that the results of a logical command are completely processed.
- If you no longer want to store incoming data in the program variable, call ct_bind with a zero-length *buffer* (for example, maxlength = 0).
- An application can rebind while actively fetching rows. That is, an application can call ct_bind inside a ct_fetch loop if it needs to change the binding of a result item (This action is not recommended).
- ct_bind supports CS_NUMERIC and CS_DECIMAL datatypes.
- Use ct_describe before ct_bind with decimal datatypes to get correct precision and scale.
- Table 3-4 lists the datatype conversions performed by ct_bind.

Table 3-4: ct_bind

Source type	Result type
CS_VARCHAR	CS_CHAR
CS_CHAR	CS_VARCHAR
CS_MONEY	CS_CHAR
CS_MONEY	CS_VARCHAR
CS_FLT4	CS_FLT8
CS_MONEY	CS_FLT8
CS_PACKED370	CS_FLT8
CS_FLT8	CS_FLT4
CS_CHAR	CS_PACKED370
CS_VARCHAR	CS_PACKED370

Source type	Result type
CS_MONEY	CS_PACKED370
CS_FLT8	CS_PACKED370
CS_NUMERIC	CS_CHAR
CS_DECIMAL	CS_CHAR
CS_PACKED370	CS_DECIMAL
CS_NUMERIC	CS_PACKED370
CS_DECIMAL	CS_PACKED370
CS_DATETIME	CS_CHAR

Array Binding

- Array binding is the act of binding a result column to an array of program variables. At fetch time, multiple rows' worth of the column are copied to the array of variables with a single ct_fetch call. An application indicates array binding by assigning count a value greater than 1.
- Array binding is only practical for regular row results. Other types of results are considered to be the equivalent of a single row.
- When binding columns to arrays in a single command, all ct_bind calls in the sequence of calls binding the columns must use the same value for count. For example, when binding three columns to arrays, it is an error to assign count a value of 5 in your first two ct_bind calls and 3 in the last.
- ct_bind supports CS_NUMERIC and CS_DECIMAL datatypes.
- Use ct_describe before ct_bind with decimal datatypes to get correct precision and scale.

See also

Related functions

- ct_describe on page 109
- ct_fetch on page 134
- ct_res_info on page 154
- ct_results on page 160

ct_cancel

Description

Cancels a request or the results of a request.

Syntax

```
CS_RETCODE ct_cancel(connection, command, type);
CS_CONNECTION *connection;
CS_COMMAND    *command;
CS_INT        type;
```

Parameters

connection

(I) Handle for this connection. This connection handle must already be allocated with ct_con_alloc. The connection handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

Either *connection* or *command* must be null. If *connection* is supplied and *command* is null, the cancel operation applies to all commands pending for this connection.

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call. The command handle also corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

Either *connection* or *command* must be null. If *command* is supplied and *connection* is null, the cancel operation applies only to the command pending for this command structure.

type

(I) Type of cancel requested. The following table lists the symbolic values that are legal for *type*:

Value	Meaning
CS_CANCEL_ALL (6001) or CS_CANCEL_ATTN (6002)	ct_cancel sends an attention to the server, instructing it to cancel the current request, and immediately discards all results generated by the request.

Return value

ct_cancel returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.
TDS_CONNECTION_TERMINATED (-4997)	The connection is not active.
TDS_INVALID_TDPROC (-18)	Specified command handle is invalid.
TDS_WRONG_STATE (-6)	Program is in the wrong communication state to issue this call.

Usage

- ct_cancel cancels the current result set.

- Canceling the current result set is equivalent to discarding the current set of results. Once results are discarded, they are no longer available to an application.
- In Open ClientConnect, CS_CANCEL_ALL and CS_CANCEL_ATTN function identically. Both immediately cancel the current request and discard all results generated by it.

Canceling a request

- To cancel the current request and all results generated by it, an application calls ct_cancel with *type* as CS_CANCEL_ATTN or CS_CANCEL_ALL. These calls tell Client-Library to:
    - Discard all results already generated by the request.
    - Send an attention to the server instructing it to halt execution of the current request.
- For example, suppose the current request is a Transact-SQL language request that contains the queries:

```
select * from titles
select * from authors
```

- If an application cancels the language request after the first query executes but before the second query executes:
    - All remaining results from the first query are discarded.
    - Execution of the second query is halted.

Note A call to ct_cancel with type as CS_CANCEL_ALL or CS_CANCEL_ATTN must be immediately followed by a ct_results call.

- In Client-Library, canceling with type as CS_CANCEL_ALL or CS_CANCEL_ATTN leaves the command structure in a “clean” state, available for use by another operation.
- For both the CS_CANCEL_ATTN and CS_CANCEL_ALL types of cancels, if no request is in progress, ct_cancel returns CS_SUCCEED immediately.
- If a request was initiated but not yet sent, a CS_CANCEL_ALL is rejected.

See also

Related functions

- ct_fetch on page 134

- ct_results on page 160

ct_close

Description

Closes a server connection.

Syntax

```
CS_RETCODE ct_close (connection, option);
CS_CONNECTION   *connection;
CS_INT          option;
```

Parameters

connection

(I) Handle for this SNA connection. This connection handle must already be allocated with ct_con_alloc. The connection handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

option

(I) Option, if any, to use for the close. The following table lists the symbolic values that are legal for *option*:

Value	Meaning
CS_UNUSED (-99999)	ct_close logs out and closes the connection. If the connection has results pending, ct_close returns CS_FAIL. This is the default behavior.
CS_FORCE_CLOSE (302)	ct_close closes the connection whether or not results are pending, and without notifying the server. This option is primarily for use when an application is hung waiting for a server response.
CS_KEEP_CON	This option is ignored. CICS treats it like CS_UNUSED.

Return value

ct_close returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed. The most common reason for a ct_close failure is pending results on the connection.
TDS_CONNECTION_TERMINATED (4997)	The connection is not active.
TDS_COMMAND_ACTIVE (-7)	A command is in progress.

Value	Meaning
TDS_RESULTS-STILL_ACTIVE (-50)	Some results are still pending.

Examples

The following code fragment demonstrates how ct_close is used with other functions at the end of a program to close the connection and return to CICS. It is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
  /*******************************************************************/
  /*                                                                 */
  /* Subroutine to perform drop command handler, close server        */
  /* connection, and deallocate Connection Handler.                  */
  /*                                                                 */
  /*******************************************************************/
void   close_connection ()

{
 CS_INT        rc;

/*------------------------------------------------------------------*/
/* drop the command handle                                          */
/*------------------------------------------------------------------*/

       rc = ct_cmd_drop (cmd);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CMD_DROP failed", msg_size);
          error_out (rc) ;
       }

/*------------------------------------------------------------------*/
/* close the server connection                                      */
/*------------------------------------------------------------------*/

       rc = ct_close (connection, (long) CS_UNUSED);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CLOSE failed", msg_size);
          error_out (rc) ;
       }

/*------------------------------------------------------------------*/
/* De_allocate the connection handle                                */
/*------------------------------------------------------------------*/
```

```
        rc = ct_con_drop (connection);

        if (rc == CS_FAIL)
        {
           strncpy (msgstr, "CT_CON_DROP failed", msg_size);
           error_out (rc) ;
        }
}    /* end close_connection */
```

Usage

- ct_close closes a server connection. All command handles associated with the connection are deallocated.
- To deallocate a connection handle, an application can call ct_con_drop after the connection successfully closes.
- The behavior of ct_close depends on the value of *option*, which determines the type of close. The following sections contain information on a type of close.

Default close behavior (*option* is CS_UNUSED)

- If the connection has any pending results, ct_close returns CS_FAIL. To correct the failure, use ct_close with the CS_FORCE_CLOSE option or read in all of your results.
- Before terminating the connection with the server, ct_close sends a logout message to the server and reads the response to this message. The contents of this message do not affect the behavior of ct_close.

Forced close behavior (*option* is CS_FORCE_CLOSE)

- The connection is closed whether or not it has pending results.
- Because this option sends no logout message to the server, the server cannot tell whether the close is intentional or whether it is the result of a lost connection or crashed client.

See also

Related functions

- ct_con_drop on page 93
- ct_connect on page 99
- ct_con_props on page 104

ct_cmd_alloc

Description
Allocates a command handle.

Syntax

```
CS_RETCODE ct_cmd_alloc(connection, command);
CS_CONNECTION *connection;
CS_COMMAND    **command;
CS_INT        type;
```

Parameters

connection

(I) Handle for this SNA connection. This connection handle must already be allocated with ct_con_alloc. The connection handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

command

(O) Variable where this newly-allocated command handle is returned. All subsequent client requests using this connection must use this same name in the *command* argument. The command handle also corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

In case of error, ct_cmd_alloc returns zeroes to this argument.

Return value

ct_cmd_alloc returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed. The most common reason for act_cmd_alloc failure is a lack of adequate memory.
TDS_SOS (-257)	Memory shortage. The mainframe subsystem was unable to allocate enough memory for the control block that ct_cmd_alloc was trying to create. The operation failed.

Examples

The following code fragment demonstrates how ct_cmd_alloc is used during program initialization. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/*------------------------------------------------------------*/
/* Open connection to the server or CICS region              */
/*------------------------------------------------------------*/

         rc = ct_connect (connection, servname, server_size);

         if (rc != CS_SUCCEED)
```

```
       {
          strncpy (msgstr, "CT_CONNECT failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
       }

/*-----------------------------------------------------------------*/
/* Invokes SEND_COMMAND routine                                    */
/*-----------------------------------------------------------------*/
       if (no_errors_sw)
          send_command ();

/*-----------------------------------------------------------------*/
/* Process the results of the command                              */
/*-----------------------------------------------------------------*/

       if (no_errors_sw)
       {
          while (no_more_results == FALSE)
             proces_results ();
       }
}   /* end proces_input */

/*******************************************************************/
/*                                                                 */
/* Subroutine to allocate, send, and process commands              */
/*                                                                 */
/*******************************************************************/
void send_command ()
 {
 CS_INT        rc;
 CS_INT        *outlen;
 CS_INT        buf_len;
 CS_CHAR       sql_cmd[45];

  /*-----------------------------------------------------------------*/
  /* Find out what the maximum number of connections is              */
  /*-----------------------------------------------------------------*/
        rc = ct_config (context, CS_GET, CS_MAX_CONNECT,
                        &maxconnect, CS_FALSE, outlen);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CONFIG failed", msg_size);
          strncpy (msgtext2, "Please press return to
          continue!", text_size);
```

```
            error_out(rc);

            /* reset program flags to move on with the task */
            print_once = TRUE;
            diag_msgs_initialized = TRUE;
            strncpy(msgtext2, "Press Clear To Exit", text_size);
         }
/*------------------------------------------------------------*/
/* Allocate a command handle                                  */
/*------------------------------------------------------------*/
         rc = ct_cmd_alloc (connection, &cmd);
         if (rc != CS_SUCCEED)
       {
            strncpy (msgstr, "CT_CMDALLOC failed", msg_size);
            no_errors_sw = FALSE ;
            error_out(rc);
       }
/*------------------------------------------------------------*/
/* Prepare the language request                               */
/*------------------------------------------------------------*/
       strcpy(sql_cmd,
               "SELECT FIRSTNME, EDUCLVL FROM SYBASE.SAMPLETB");
       buf_len = sizeof(sql_cmd);
       rc = ct_command(cmd, (long) CS_LANG_CMD, sql_cmd,
                       buf_len, (long) CS_UNUSED);
        if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_COMMAND failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
       }
 /*------------------------------------------------------------*/
 /* Send the language request                                  */
 /*------------------------------------------------------------*/
       rc = ct_send (cmd);

       if (rc != CS_SUCCEED)
       {
          strcpy (msgstr, "CT_SEND failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
       }
}   /* end send_command */
```

Usage

- ct_cmd_alloc allocates a command handle on a specified connection.
  A command handle is a control structure that a Client-Library application uses to send requests to a server and process the results. Together, command and connection handles perform the functions of the Open ServerConnect *TDPROC* structure.
- Before calling ct_cmd_alloc, an application must allocate a connection structure via the Client-Library routine ct_con_alloc.
- An application must call ct_cmd_alloc once for each logical command it issues. Each SQL statement is considered a separate logical command. For batched SQL, call ct_cmd_alloc once for each batch.

See also

Related functions

- ct_cmd_drop on page 78
- ct_cmd_props on page 81
- ct_command on page 83
- ct_con_alloc on page 88

Related documentation

- Mainframe Connect Client Option and Server Option *Messages and Codes*

ct_cmd_drop

Description

Deallocates a command handle.

Syntax

```
CS_RETCODE ct_cmd_drop(command);
CS_COMMAND *command;
```

Parameters

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call. The connection handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

Return value

ct_cmd_drop returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.

Value	Meaning
CS_FAIL (-2)	The routine failed. ct_cmd_drop returns CS_FAIL if the command handle has any results pending.
TDS_COMMAND_ACTIVE (-7)	A command is in progress.
TDS_RESULTS_STILL_ACTIVE (50)	Some results are still pending.

Examples

The following code fragment demonstrates how ct_cmd_drop is used with other routines at the end of a program after results have been processed. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/************************************************************************/
/*                                                                      */
/* Subroutine to perform drop command handler, close server             */
/* connection, and deallocate Connection Handler.                       */
/*                                                                      */
/************************************************************************/
  void   close_connection ()
    {
   CS_INT        rc;

/*--------------------------------------------------------------------*/
/* drop the command handle                                            */
/*--------------------------------------------------------------------*/
         rc = ct_cmd_drop (cmd);

         if (rc == CS_FAIL)
         {
            strncpy (msgstr, "CT_CMD_DROP failed", msg_size);
            error_out (rc) ;
         }
   /*-----------------------------------------------------------------*/
  /* close the server connection                                      */
  /*------------------------------------------------------------------*/
          rc = ct_close (connection, (long) CS_UNUSED);

         if (rc == CS_FAIL)
         {
            strncpy (msgstr, "CT_CLOSE failed", msg_size);
            error_out (rc) ;
         }
  /*-------------------------------------------------------------------*/
  /* De_allocate the connection handle                                 */
```

```
/*--------------------------------------------------------------------*/
        rc = ct_con_drop (connection);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CON_DROP failed", msg_size);
          error_out (rc) ;
       }
}   /* end close_connection */
```

Usage

- ct_cmd_drop deallocates a command handle.
- If ct_cmd_drop is called while a command is pending (which means all results have not been returned), it fails. Before deallocating a command structure, an application should process or cancel any pending results.
- Once a command handle has been deallocated, it cannot be reused. To allocate a new command handle, an application calls ct_cmd_alloc.

See also

Related functions

- ct_cmd_alloc on page 75
- ct_command on page 83

ct_cmd_props

Description

Sets, retrieves, or clears information about the current result set.

Syntax

```
CS_RETCODE ct_cmd_props(command, action, property,
                 buffer, buf_len, outlen);
CS_COMMAND   *command;
CS_INT       action;
CS_INT       property;
CS_VOID      *buffer;
CS_INT       buf_len;
CS_INT       *outlen
```

Parameters

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call. The command handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

action

(I) Action to be taken by this call. *action* is an integer variable that indicates the purpose of this call.
Assign *action* one of the following symbolic values:

Value	Meaning
CS_GET (33)	Retrieves the value of the property.
CS_SET (34)	Sets the value of the property.
CS_CLEAR (35)	Clears the value of the property by resetting the property to its Client-Library default value.

property

(I) Symbolic name of the property whose value is being set or retrieved. Client-Library properties are listed under “Remote procedure calls (RPCs)” on page 48, with descriptions, possible values, and defaults.

buffer

(I/O) Variable (buffer) that contains the specified property value.

If *action* is CS_SET, the buffer contains the value used by ct_cmd_props.

If *action* is CS_GET, ct_cmd_props returns the requested information to this buffer.

If *action* is CS_CLEAR, the buffer is reset to the default property value.

This argument is typically one of the following datatypes:

```
int buffer[n];
char buffer[n];
```

buf_len

(I) Length, in bytes, of the buffer.

If *action* is CS_SET and the value in the buffer is a fixed-length or symbolic value, *buf_len* should have a value of CS_UNUSED.

If *action* is CS_GET and *buffer* is too small to hold the requested information, ct_cmd_props sets *outlen* to the length of the requested information and returns CS_FAIL. To retrieve all the requested information, change the value of *buf_len* to the length returned in *outlen* and rerun the application.

If *action* is CS_CLEAR, set this value to CS_UNUSED.

outlen

(O) Length, in bytes, of the retrieved information. *outlen* is an integer variable where ct_cmd_props returns the length of the property value being retrieved.

When the retrieved information is larger than *buf_len* bytes, an application uses the value of *outlen* to determine how many bytes are needed to hold the information.

outlen is used only when *action* is CS_GET. When *action* is CS_CLEAR or CS_SET, this value is zero.

Return value

ct_cmd_props returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.
TDS_INVALID_PARAMETER (-4)	One or more arguments were given illegal values.
TDS_CANNOT_SET_VALUE (-43)	This property cannot be set by the application.

Usage

- ct_cmd_props sets or retrieves the values of properties of command handle structures.
- command handle properties affect the behavior of an application at the command structure level.
- Some command handle properties default to the value of the property in the parent context. To find out which ones, see "Remote procedure calls (RPCs)" on page 48.

See also

Related functions

- ct_cmd_alloc on page 75
- ct_config on page 96
- ct_con_props on page 104
- ct_res_info on page 154

Related topics

- "Buffers" on page 23
- "Remote procedure calls (RPCs)" on page 48

ct_command

Description

Initiates a language request or remote procedure call (RPC).

Syntax

```
CS_RETCODE (command, type, buffer, buf_len, option);
CS_COMMAND    *command;
CS_INT        type;
CS_BYTE       *buffer;
CS_INT        buf_len;
CS_INT        option;
```

Parameters

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call. The command handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

type

(I) Type of request to initiate. The following symbolic values are legal for *type*:

When type Is	ct_command Initiates	Buffer contains
CS_LANG_CMD (148)	A language request.	The text of the language request.
CS_RPC_CMD (149)	A remote procedure call.	The name of the remote procedure.

buffer

(I) Variable (buffer) that contains the language request or RPC name.

This argument is typically one of the following datatypes:

```
int buffer[n];
char buffer[n];
```

buf_len

(I) Length, in bytes, of the buffer.

If the value in the buffer is a fixed-length or symbolic value, assign *buf_len* a value of CS_UNUSED.

option

Option associated with this request, if any.

Currently, only RPCs take options. For language requests, assign *option* a value of CS_UNUSED.

The following symbolic values are legal for *option* when *type* is CS_RPC_CMD:

Value	Meaning
CS_RECOMPILE (188)	Recompile the stored procedure before executing it.
CS_NORECOMPILE (189)	Do not recompile the stored procedure before executing it.
CS_UNUSED (-99999)	No options are assigned.

Return value

ct_command returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.
TDS_CONNECTION_TERMINATED (-4997)	The connection is not active.
TDS_INVALID_PARAMETER (-4)	A parameter contains an illegal value.
TDS_WRONG_STATE (-6)	Program is in the wrong communication state to issue this call.

Examples

The following code fragment demonstrates the use of ct_command. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/*------------------------------------------------------------------*/
/* Open connection to the server or CICS region                     */
/*------------------------------------------------------------------*/

         rc = ct_connect (connection, servname, server_size);

         if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CONNECT failed", msg_size);
```

```
          no_errors_sw = FALSE ;
          error_out (rc);
       }
/*-------------------------------------------------------------*/
/* Invokes SEND_COMMAND routine                                 */
/*-------------------------------------------------------------*/
       if (no_errors_sw)
          send_command ();

/*-------------------------------------------------------------*/
/* Process the results of the command                           */
/*-------------------------------------------------------------*/
       if (no_errors_sw)
       {
          while (no_more_results == FALSE)
             proces_results ();
       }
}   /* end proces_input */
/*********************************************************************/
/*                                                                   */
/* Subroutine to allocate, send, and process commands                */
/*                                                                   */
/*********************************************************************/
void send_command ()
{
 CS_INT        rc;
 CS_INT        *outlen;
 CS_INT        buf_len;
 CS_CHAR       sql_cmd[45];

/*-------------------------------------------------------------*/
/* Find out what the maximum number of connections is          */
/*-------------------------------------------------------------*/
        rc = ct_config (context, CS_GET, CS_MAX_CONNECT,
                       &maxconnect, CS_FALSE, outlen);
        if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CONFIG failed", msg_size);
          strncpy (msgtext2, "Please press return to
          continue!",text_size);
          error_out(rc);

          /* reset program flags to move on with the task */
          print_once = TRUE;
          diag_msgs_initialized = TRUE;
          strncpy(msgtext2, "Press Clear To Exit", text_size);
```

```
        }
    /*------------------------------------------------------------*/
    /* Allocate a command handle                                  */
    /*------------------------------------------------------------*/
          rc = ct_cmd_alloc (connection, &cmd);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_CMDALLOC failed", msg_size);
           no_errors_sw = FALSE ;
           error_out(rc);
        }

     /*------------------------------------------------------------*/
     /* Prepare the language request                               */
     /*------------------------------------------------------------*/
        strcpy(sql_cmd,
                "SELECT FIRSTNME, EDUCLVL FROM SYBASE.SAMPLETB");
          buf_len = sizeof(sql_cmd);
          rc = ct_command(cmd, (long) CS_LANG_CMD, sql_cmd,
                        buf_len, (long) CS_UNUSED);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_COMMAND failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }

    /*------------------------------------------------------------*/
    /* Send the language request                                  */
    /*------------------------------------------------------------*/

        rc = ct_send (cmd);

        if (rc != CS_SUCCEED)
        {
           strcpy (msgstr, "CT_SEND failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
    }   /* end send_command */
```

Usage

- ct_command initiates a language request or RPC. Initiating a request is the first step in sending it to a server.

- Sending a request to a server is a three step process. To send a request to a server, an application must:
  a. Call ct_command to initiate the request. ct_command sets up internal structures that are used in developing a request stream to send to the server.
  b. Call ct_param to pass parameters for the request. An application must call ct_param once for each parameter in the request.
  c. Call ct_send to send the request to the server.

Language requests

- Language requests contain character strings that represent requests in a server's own language. For example, language requests to Adaptive Server Enterprise can include any legal Transact-SQL command.
- A language request can be in any language, as long as the server to which it is directed can understand it. For example, Adaptive Server Enterprise understands Transact-SQL, but requests to DB2 must use the DB2 version of SQL.
- If the language request string contains variables, an application can pass values for these variables by calling ct_param once for each variable that the language string contains. A language request can have up to 255 parameters.
- Transact-SQL request variables must begin with a colon (:).

Remote Procedure Calls (RPCs)

- An RPC instructs a server to execute a stored procedure or transaction on either itself or a remote server.
- If an application is using an RPC to execute a stored procedure or transaction that requires parameters, the application calls ct_param once for each parameter the stored procedure or transaction requires.
- After sending an RPC with ct_send, an application can process the stored procedure or transaction results with ct_results and ct_fetch. The functions ct_results and ct_fetch are used to process both the result rows generated by the stored procedure or transaction and the return parameters and status, if any.

See also

Related functions

- ct_cmd_alloc on page 75
- ct_param on page 145

- ct_send on page 165

Related topics

- "Remote procedure calls (RPCs)" on page 48

ct_con_alloc

Description

Allocates a connection handle.

Syntax

```
CS_RETCODE cs_con_alloc(context, connection);
CS_CONTEXT     *context;
CS_CONNECTION  **connection;
```

Parameters

context

(I) A context structure. The context structure is defined in the program call cs_ctx_alloc. The context structure corresponds to the *IHANDLE* structure in the Open ServerConnect Gateway-Library.

If this value is invalid or nonexistent, ct_con_alloc fails.

connection

(O) Handle for this connection. All subsequent Client-Library calls using this connection must use this same name in their connection argument. The connection handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

This is the same value used to define the connection to the Open ClientConnect Connection Table.

In case of error, ct_con_alloc returns zeroes to this argument.

Return value

ct_con_alloc returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed. The most common reason for a ct_con_alloc failure is a lack of adequate memory.
TDS_SOS (-257)	Memory shortage. The mainframe subsystem was unable to allocate enough memory for the control block that ct_con_alloc was trying to create. The operation failed.
TDS_GWLIB_NO_STORAGE (-17)	Could not get DSA for Gateway-Library.

Examples

The following code fragment demonstrates the use of ct_con_alloc. It is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
/********************************************************************/
/*                                                                  */
/* Subroutine to process input data                                 */
/*                                                                  */
/********************************************************************/
  void   proces_input ()
  {
   CS_INT        rc;
   CS_INT        *outlen;
   CS_INT        buf_len;
   CS_INT        msglimit;
   CS_INT        netdriver;

    /*------------------------------------------------------------*/
    /* Allocate a connection to the server                        */
    /*------------------------------------------------------------*/
         rc = ct_con_alloc (context, &maxconnect);

         if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CONALLOC failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
          }
  /*------------------------------------------------------------*/
  /* Alter properties of the connection for user-id             */
  /*------------------------------------------------------------*/

         buf_len  = user_size;
         rc = ct_con_props (connection, (long)CS_SET,
                            (long)CS_USERNAME, username,
                            buf_len, outlen);
         if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CON_PROPS for user-id failed",
            msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
          }
  /*------------------------------------------------------------*/
  /* Alter properties of the connection for password            */
  /*------------------------------------------------------------*/
```

```
        buf_len = pwd_size;
        rc = ct_con_props (connection, (long)CS_SET,
                           (long)CS_PASSWORD, pwd, buf_len, outlen);
         if (rc != CS_SUCCEED)
         {
           strncpy (msgstr, "CT_CON_PROPS for password failed",
           msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
         }
 /*------------------------------------------------------------*/
 /* Alter properties of the connection for transaction         */
 /*------------------------------------------------------------*/

        buf_len = tran_size;
        rc = ct_con_props (connection, (long)CS_SET,
                           (long)CS_TRANSACTION_NAME,
                           tran, buf_len, outlen);
        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_CON_PROPS for transaction failed",
           msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
 /*------------------------------------------------------------*/
 /* Alter properties of the connection for network driver      */
 /*------------------------------------------------------------*/

        netdriver = 9999;   /* default value for non-regconized
                               driver name                     */

        /* if no netdriver entered, default is LU62  */
        if (strncmp(driver,"         ",9) == 0  ??
            strncmp(driver,"LU62",4) == 0)
           netdriver = CS_LU62;
        else if (strncmp(driver,"INTERLINK",8) == 0)
           netdriver = CS_INTERLINK;
        else if (strncmp(driver,"IBMTCPIP",8) == 0)
           netdriver = CS_TCPIP;
        else if (strncmp(driver,"CPIC",4) == 0)
           netdriver = CS_NCPIC;
        rc = ct_con_props (connection, (long)CS_SET,
                           (long)CS_NET_DRIVER, (long) netdriver,
                           CS_UNUSED, outlen);
        if (rc != CS_SUCCEED)
```

```
        {
           strncpy (msgstr, "CT_CON_PROPS for network driver failed",
           msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
/*------------------------------------------------------------*/
/* Setup retrieval of All Messages                            */
/*------------------------------------------------------------*/

        rc = ct_diag (connection, CS_INIT,
                      CS_UNUSED, CS_UNUSED, CS_NULL);
        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_DIAG CS_INIT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
/*------------------------------------------------------------*/
/* Set the upper limit of number of messages                  */
/*------------------------------------------------------------*/
        msglimit = 5 ;
        rc = ct_diag (connection, CS_MSGLIMIT, CS_ALLMSG_TYPE,
                      CS_UNUSED, &msglimit);
        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_DIAG CS_MSGLIMIT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
/*------------------------------------------------------------*/
/* Open connection to the server or CICS region               */
/*------------------------------------------------------------*/

        rc = ct_connect (connection, servname, server_size);
        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_CONNECT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
/*------------------------------------------------------------*/
/* Invokes SEND_COMMAND routine                               */
/*------------------------------------------------------------*/
        if (no_errors_sw)
           send_command ();
```

Usage

- ct_con_alloc allocates a connection handle to a Mainframe ClientConnect or another processing region (three-tier processing), or an Adaptive Server Enterprise if using two-tier (gateway-less) processing.
- Before calling ct_con_alloc, an application must:
    - Call ct_ctx_alloc to allocate a context structure.
    - Call ct_init to initialize Client-Library.
- Connecting to a server is a three-step process. To connect to a server, an application:

    a Calls ct_con_alloc to obtain a connection handle.

    b Calls ct_con_props to set the values of connection-specific properties, if desired.

    c Calls ct_connect to create the connection and log into the server.
- All connections created within a context pick up default property values from the parent context. An application can override these default values by calling ct_con_props to set property values at the connection level.
- An application can have multiple connections to one or more servers at the same time.

    For example, an application can simultaneously have two connections to the server "mars," one connection to the server "venus," and one connection to a separate CICS region named CICX3. The context property CS_MAX_CONNECT, set by ct_config, determines the maximum number of connections allowed per context.

    Each server connection requires a separate connection handle.
- In order to send requests to a server, one or more command handles must be allocated for a connection. ct_cmd_alloc allocates a command handle.

See also

Related functions

- cs_ctx_alloc on page 179
- ct_close on page 72
- ct_cmd_alloc on page 75
- ct_connect on page 99
- ct_con_props on page 104

ct_con_drop

Description

Deallocates a connection handle.

Syntax

```
CS_RETCODE cs_con_drop(context, connection);
CS_CONTEXT     *context;
CS_CONNECTION  *connection;
```

Parameters

connection

(I) Handle for this connection. This must be the same value specified in the ct_con_alloc call that initialized this connection.

Return value

ct_con_drop returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed. The most common reason for a ct_con_drop failure is that the connection is still open.
TDS_CONNECTION_TERMINATED(-4997)	The connection is not active.

Examples

The following code fragment demonstrates how ct_con_drop and other functions at the end of a program close the connection and return to CICS. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/********************************************************************/
/*                                                                  */
/* Subroutine to perform drop command handler, close server        */
/* connection, and deallocate Connection Handler.                  */
/*                                                                  */
/********************************************************************/
void   close_connection ()
{
CS_INT        rc;

/*------------------------------------------------------------------*/
/* drop the command handle                                          */
/*------------------------------------------------------------------*/
     rc = ct_cmd_drop (cmd);
     if (rc == CS_FAIL)
     {
        strncpy (msgstr, "CT_CMD_DROP failed", msg_size);
        error_out (rc) ;
     }
/*------------------------------------------------------------------*/
/* close the server connection                                      */
```

```
/*------------------------------------------------------------------*/
      rc = ct_close (connection, (long) CS_UNUSED);
      if (rc == CS_FAIL)
      {
         strncpy (msgstr, "CT_CLOSE failed", msg_size);
         error_out (rc) ;
      }
/*------------------------------------------------------------------*/
/* De_allocate the connection handle                                */
/*------------------------------------------------------------------*/
      rc = ct_con_drop (connection);
      if (rc == CS_FAIL)
      {
         strncpy (msgstr, "CT_CON_DROP failed", msg_size);
         error_out (rc) ;
      }
}   /* end close_connection */

/********************************************************************/
/*                                                                  */
/* Subroutine to perform exit client library and deallocate context */
/* structure.                                                       */
/*                                                                  */
/********************************************************************/
void   quit_client_library ()
{
CS_INT        rc;
/*------------------------------------------------------------------*/
/* Exit the Client Library                                          */
/*------------------------------------------------------------------*/
      rc = ct_exit (context, (long) CS_UNUSED);
      if (rc == CS_FAIL)
      {
         strncpy (msgstr, "CT_EXIT failed", msg_size);
         error_out(rc) ;
       }
/*------------------------------------------------------------------*/
/* De-allocate the context structure                                */
/*------------------------------------------------------------------*/
      rc = cs_ctx_drop (context);
      if (rc == CS_FAIL)
      {
         strncpy (msgstr, "CT_CTX_DROP failed", msg_size);
         error_out(rc) ;
      }
      EXEC CICS RETURN ;
```

```
    }   /* end quit_client_library */
```

Usage

- ct_con_drop deallocates a connection handle and all command handles associated with that connection.
- Once a connection handle is deallocated, it cannot be reused. To allocate a new connection handle, an application calls ct_con_alloc.
- An application cannot deallocate a connection handle until the connection it represents successfully closes. To close a connection, an application calls ct_close.

See also

Related functions

- ct_close on page 72
- ct_con_alloc on page 88
- ct_connect on page 99
- ct_con_props on page 104

ct_config

Description

Sets or retrieves context properties.

Syntax

```
CS_RETCODE (context, action, property, buffer,
        buf_len, outlen);
CS_CONTEXT    *context;
CS_INT        action;
CS_INT        property;
CS_BYTE       *buffer;
CS_INT        buf_len;
CS_INT        *outlen;
```

Parameters

context

(I) A context structure. The context structure is defined in the program call cs_ctx_alloc. It corresponds to the *IHANDLE* structure in the Open ServerConnect Gateway-Library.

If this value is invalid or nonexistent, ct_config fails.

action

(I) Action this call takes. *action* is an integer variable that indicates the purpose of this call.

Assign *action* one of the following symbolic values:

Value	Meaning
CS_GET (33)	Retrieves the value of the property.
CS_SET (34)	Sets the value of the property.
CS_CLEAR (35)	Clears the value of the property by resetting the property to its Client-Library default value.

property

(I) Symbolic name of the property for which the value is being set or retrieved. Client-Library properties are listed under "Remote procedure calls (RPCs)" on page 48, with descriptions, possible values, and defaults.

buffer

(I/O) Variable (buffer) that contains the specified property value.

If *action* is CS_SET, the buffer contains the value used by ct_config.

If *action* is CS_GET, ct_config returns the requested information to this buffer.

If *action* is CS_CLEAR, the buffer is reset to the default property value.

This argument is typically one of the following datatypes:

```
int buffer[n]; char buffer[n]
```

buf_len

(I) Length, in bytes, of the buffer.

If *action* is CS_SET and the value in the buffer is a fixed-length or symbolic value, *buf_len* should have a value of CS_UNUSED.

If *action* is CS_GET and *buffer* is too small to hold the requested information, ct_cmd_props sets *outlen* to the length of the requested information and returns CS_FAIL. To retrieve all the requested information, change the value of *buf_len* to the length returned in *outlen* and rerun the application.

If *action* is CS_CLEAR, set this value to zeroes.

outlen

(O) Length, in bytes, of the retrieved information. *outlen* is an integer variable where ct_config returns the length of the property value being retrieved.

When the retrieved information is larger than *buf_len* bytes, an application uses the value of *outlen* to determine how many bytes it needs to hold the information.

outlen is used only when *action* is CS_GET.If *action* is CS_SET or CS_CLEAR, this value is zero.

Return value

ct_config returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.

Examples

The following code fragment demonstrates how to use ct_config to determine the maximum number of connections. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/*------------------------------------------------------------*/
/* Find out what the maximum number of connections is         */
/*------------------------------------------------------------*/
     rc = ct_config (context, CS_GET, CS_MAX_CONNECT,
                     &maxconnect, CS_FALSE, outlen);
      if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_CONFIG failed", msg_size);
        strncpy (msgtext2, "Please press return to
        continue!", text_size);
        error_out(rc);
        /* reset program flags to move on with the task */
```

```
            print_once = TRUE;
            diag_msgs_initialized = TRUE;
            strncpy(msgtext2, "Press Clear To Exit", text_size);
    }
```

Usage

- cs_config sets or retrieves the values of CS_EXTRA_INF and CS_VERSION. All other context properties are set or reset by ct_config. Context properties define aspects of Client-Library behavior at the context level.
- All connections created within a context pick up default property values from the parent context. An application can override these default values by calling ct_con_props to set property values at the connection level.
- If an application changes context property values after allocating connections for the context, the existing connections do not pick up the new property values. Only new connections allocated after the new context property values are set use the new values as defaults.

See also

Related functions

- ct_cmd_props on page 81
- ct_connect on page 99
- ct_con_props on page 104
- ct_init on page 141

Related topics

- “Buffers” on page 23
- ““Remote procedure calls (RPCs)” on page 48”

ct_connect

Description

Connects to a server.

Syntax

```
CS_RETCODE ct_connect(connection, servername,
                      servername_len);
CS_CONNECTION   *connection;
CS_CHAR         *servername;
CS_INT          servername_len
```

Parameters

connection

(I) Handle for this SNA connection. This connection handle must be allocated with ct_con_alloc. The connection handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

servername

(I) Name of the connected server. For clients running SNA, this is the name by which the server is known to the Open ClientConnect Server Path Definition Table. For clients running TCP/IP without a gateway, this is the actual name of the Adaptive Server Enterprise in the LAN interfaces file.

You must assign a value to this argument. If a server name is not specified, ct_connect fails.

servername_len

(I) Length, in bytes, of *servername*.

Return value

ct_connect returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.
TDS-CRTABLE-UNAVAILABLE (-31)	The connection router table cannot be loaded.

Examples

The following code fragment demonstrates the use of ct_connect. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/********************************************************************/
/*                                                                  */
/* Subroutine to process input data                                 */
/*                                                                  */
/********************************************************************/
   void   proces_input ()
{
CS_INT        rc;
CS_INT        *outlen;
CS_INT        buf_len;
```

```
CS_INT        msglimit;
CS_INT        netdriver;
/*------------------------------------------------------------*/
/* Allocate a connection to the server                        */
/*------------------------------------------------------------*/
rc = ct_con_alloc (context, &connection);

     if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_CONALLOC failed", msg_size);
        no_errors_sw = FALSE ;
        error_out (rc);
      }
/*------------------------------------------------------------*/
/* Alter properties of the connection for user-id             */
/*------------------------------------------------------------*/

     buf_len  = user_size;
     rc = ct_con_props (connection, (long)CS_SET,
                         (long)CS_USERNAME, username,
                         buf_len, outlen);

     if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_CON_PROPS for user-id failed",
        msg_size);
        no_errors_sw = FALSE ;
        error_out (rc);
      }
/*------------------------------------------------------------*/
/* Alter properties of the connection for password            */
/*------------------------------------------------------------*/

     buf_len = pwd_size;
     rc = ct_con_props (connection, (long)CS_SET,
                        (long)CS_PASSWORD, pwd, buf_len, outlen);

      if (rc != CS_SUCCEED)
      {
        strncpy (msgstr, "CT_CON_PROPS for password failed",
        msg_size);
        no_errors_sw = FALSE ;
        error_out (rc);
      }

/*------------------------------------------------------------*/
```

```
/* Alter properties of the connection for transaction         */
/*------------------------------------------------------------*/

      buf_len = tran_size;
      rc = ct_con_props (connection, (long)CS_SET,
                         (long)CS_TRANSACTION_NAME,
                         tran, buf_len, outlen);

      if (rc != CS_SUCCEED)
      {
         strncpy (msgstr, "CT_CON_PROPS for transaction failed",
         msg_size);
         no_errors_sw = FALSE ;
         error_out (rc);
      }
/*------------------------------------------------------------*/
/* Alter properties of the connection for network driver      */
/*------------------------------------------------------------*/

      netdriver = 9999;   /* default value for non-regconized
                             driver name                     */

      /* if no netdriver entered, default is LU62  */
      if (strncmp(driver,"         ",9) == 0  ??
          strncmp(driver,"LU62",4) == 0)
         netdriver = CS_LU62;

      else if (strncmp(driver,"INTERLINK",8) == 0)
         netdriver = CS_INTERLINK;

      else if (strncmp(driver,"IBMTCPIP",8) == 0)
         netdriver = CS_TCPIP;

      else if (strncmp(driver,"CPIC",4) == 0)
         netdriver = CS_NCPIC;

      rc = ct_con_props (connection, (long)CS_SET,
                         (long)CS_NET_DRIVER, (long) netdriver,
                         CS_UNUSED, outlen);

      if (rc != CS_SUCCEED)
      {
         strncpy (msgstr, "CT_CON_PROPS for network driver failed",
         msg_size);
         no_errors_sw = FALSE ;
         error_out (rc);
```

```
     }
/*------------------------------------------------------------*/
/* Setup retrieval of All Messages                            */
/*------------------------------------------------------------*/

     rc = ct_diag (connection, CS_INIT,
                   CS_UNUSED, CS_UNUSED, CS_NULL);

     if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_DIAG CS_INIT failed", msg_size);
        no_errors_sw = FALSE ;
        error_out (rc);
     }
/*------------------------------------------------------------*/
/* Set the upper limit of number of messages                  */
/*------------------------------------------------------------*/
      msglimit = 5 ;

     rc = ct_diag (connection, CS_MSGLIMIT, CS_ALLMSG_TYPE,
                   CS_UNUSED, &msglimit);

     if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_DIAG CS_MSGLIMIT failed", msg_size);
        no_errors_sw = FALSE ;
        error_out (rc);
     }
/*------------------------------------------------------------*/
/* Open connection to the server or CICS region               */
/*------------------------------------------------------------*/
     rc = ct_connect (connection, servname, server_size);

     if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_CONNECT failed", msg_size);
        no_errors_sw = FALSE ;
        error_out (rc);
     }
/*------------------------------------------------------------*/
/* Invokes SEND_COMMAND routine                               */
/*------------------------------------------------------------*/
     if (no_errors_sw)
        send_command ();
```

Usage

- ct_connect establishes a connection between a mainframe transaction processing region and a remote server. Information about the connection is stored in a connection handle, which uniquely identifies the connection.
- The remote server can be another transaction processing region or server (Adaptive Server Enterprise, Open Server, and so on). For clients running SNA, the name in the Server Path Definition Table is the name of the remote region or server. For clients running TCP/IP, SYGWHOST (server name and IP address) is used in conjunction with either the MVS-side information file for the specific drive or the CICS partner table.
- When it establishes a connection, ct_connect sets up communication with the server, forwards login information, and communicates any connection-specific property information to the server.
- Because creating a connection involves sending login information, an application must define login parameters (server user ID and password) before calling ct_connect. An application calls ct_con_props to define login parameters.
- The maximum number of open connections per context is determined by the CS_MAX_CONNECT property (set by ct_config). The default maximum is 25 connections.
- An attempt to establish a connection can fail in two ways (assuming that the system is correctly configured):
  - If the specified server machine (the machine on which the server resides) is running correctly and the network is running correctly, but no server is listening on the specified port, the specified server machine signals the client, through a network error, that the connection cannot be formed. Regardless of the login time-out value, the connection fails.
  - If the machine on which the server resides is down, the server does not respond. Because "no response" is not considered to be an error, the network does not signal the client that an error occurred. However, if a login time-out period is set, a time-out error occurs when the client fails to receive a response within the set period.
- To close a connection, an application calls ct_close.

See also

Related functions

- ct_close on page 72
- ct_con_alloc on page 88

- ct_con_drop on page 93
- ct_config on page 96
- ct_con_props on page 104
- ct_remote_pwd on page 152

Related topics

- "Remote procedure calls (RPCs)" on page 48

Related documentation

- Mainframe Connect Client Option *Installation and Administration Guide*

ct_con_props

Description

Sets or retrieves connection handle properties.

Syntax

```
CS_RETCODE ct_con_props(connection, action, property,
                        buffer, buf_len, outlen);
CS_CONNECTION  *connection;
CS_INT         action;
CS_INT         property;
CS_BYTE          *buffer;
CS_INT         buf_len;
CS_INT         *outlen;
```

Parameters

connection

(I) Handle for this connection. This connection handle must already be allocated with ct_con_alloc. The connection handle corresponds to the *TDPROC* handle in the Open ServerConnect Gateway-Library.

action

(I) Action to be taken by this call. *action* is an integer variable that indicates the purpose of this call.

Assign *action* one of the following symbolic values:

Value	Meaning
CS_GET (33)	Retrieves the value of the property.
CS_SET (34)	Sets the value of the property.
CS_CLEAR (35)	Clears the value of the property by resetting the property to its Client-Library default value.

property

(I) Symbolic name of the property for which the value is being set or retrieved. Client-Library properties are listed under "Remote procedure calls (RPCs)" on page 48, with description, possible values, and defaults.

buffer

(I/O) Variable (buffer) that contains the specified property value.

If *action* is CS_SET, the buffer contains the value ct_cmd_props uses.

If *action* is CS_GET, ct_cmd_props returns the requested information to this buffer.

If *action* is CS_CLEAR, the buffer is reset to the default property value.

This argument is typically one of the following datatypes:

```
int buffer;
char buffer[n];
```

buf_len

(I/O) Length, in bytes, of the buffer.

If *action* is CS_SET and the value in the buffer is a fixed-length or symbolic value, *buf_len* should have a value of CS_UNUSED.

If *action* is CS_GET and *buffer* is too small to hold the requested information, ct_cmd_props sets *outlen* to the length of the requested information and returns CS_FAIL. To retrieve all the requested information, change the value of *buf_len* to the length returned in *outlen* and rerun the application.

Note If *action* is GS_GET, you can not use CS_UNUSED for *buf_len*. You must provide some value for *buf_len*.

If *action* is CS_CLEAR, this value is zero.

outlen

(O) Length, in bytes, of the retrieved information. *outlen* is an integer variable where ct_con_props returns the length of the property value being retrieved.

If the retrieved information is larger than *buf_len* bytes, an application uses the value of *outlen* to determine how many bytes are needed to hold the information.

outlen is used only when *action* is CS_GET. If *action* is CS_CLEAR or CS_SET, this value is zero.

Return value

ct_con_props returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.
TDS_CANNOT_SET_VALUE (-43)	This property cannot be set by the application.
TDS_INVALID_PARAMETER (-4)	One or more arguments contain illegal values.

Examples

The following code fragment demonstrates how to use ct_con_props. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/********************************************************************/
/*                                                                  */
/* Subroutine to process input data                                 */
/*                                                                  */
/********************************************************************/
void   proces_input ()
{
CS_INT        rc;
CS_INT        *outlen;
CS_INT        buf_len;
CS_INT        msglimit;
CS_INT        netdriver;
/*------------------------------------------------------------*/
/* Allocate a connection to the server                        */
/*------------------------------------------------------------*/
     rc = ct_con_alloc (context, &connection);
     if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_CONALLOC failed", msg_size);
        no_errors_sw = FALSE ;
        error_out (rc);
      }
/*------------------------------------------------------------*/
/* Alter properties of the connection for user-id             */
/*------------------------------------------------------------*/
       buf_len  = user_size;
       rc = ct_con_props (connection, (long)CS_SET,
                          (long)CS_USERNAME, username,
                          buf_len, outlen);
       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CON_PROPS for user-id failed",
```

```
            msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
          }
     /*-----------------------------------------------------------*/
     /* Alter properties of the connection for password          */
     /*-----------------------------------------------------------*/
            buf_len = pwd_size;
            rc = ct_con_props (connection, (long)CS_SET,
                              (long)CS_PASSWORD, pwd, buf_len, outlen);
             if (rc != CS_SUCCEED)
             {
               strncpy (msgstr, "CT_CON_PROPS for password failed",
               msg_size);
               no_errors_sw = FALSE ;
               error_out (rc);
             }
     /*-----------------------------------------------------------*/
     /* Alter properties of the connection for transaction       */
     /*-----------------------------------------------------------*/
            buf_len = tran_size;
            rc = ct_con_props (connection, (long)CS_SET,
                              (long)CS_TRANSACTION_NAME,
                              tran, buf_len, outlen);
            if (rc != CS_SUCCEED)
            {
               strncpy (msgstr, "CT_CON_PROPS for transaction failed",
               msg_size);
               no_errors_sw = FALSE ;
               error_out (rc);
            }
     /*-----------------------------------------------------------*/
     /* Alter properties of the connection for network driver    */
     /*-----------------------------------------------------------*/
            netdriver = 9999;   /* default value for non-regconized
                                   driver name                     */
            /* if no netdriver entered, default is LU62  */
            if (strncmp(driver,"         ",9) == 0  &&
                 strncmp(driver,"LU62",4) == 0)
               netdriver = CS_LU62;
            else if (strncmp(driver,"INTERLINK",8) == 0)
               netdriver = CS_INTERLINK;
            else if (strncmp(driver,"IBMTCPIP",8) == 0)
               netdriver = CS_TCPIP;
            else if (strncmp(driver,"CPIC",4) == 0)
               netdriver = CS_NCPIC;
```

```
rc = ct_con_props (connection, (long)CS_SET,
                  (long)CS_NET_DRIVER, (long) netdriver,
                  CS_UNUSED, outlen);
if (rc != CS_SUCCEED)
{
   strncpy (msgstr, "CT_CON_PROPS for network driver failed",
   msg_size);
   no_errors_sw = FALSE ;
   error_out (rc);
}
```

Usage

- ct_con_props sets or retrieves the values of properties for a connection handle. Connection properties define aspects of Client-Library behavior at the connection level.
- All command structures allocated for a connection pick up default property values from the parent connection. An application can override these default values by calling ct_cmd_props at the command structure level.
- If an application changes connection property values after allocating command structures for the connection, the existing command structures do not pick up the new property values. New command structures allocated for the connection use the new property values as defaults.
- Some connection properties only take effect if they are set before an application calls ct_connect to establish the connection.
- An application can use ct_con_props to set or retrieve the following properties:
  - CS_APPNAME
  - CS_CHARSETCNV
  - CS_COMMBLOCK
  - CS_EXTRA_INF
  - CS_HOSTNAME
  - CS_LOGIN_STATUS
  - CS_NET_DRIVER
  - CS_NETIO
  - CS_NOINTERRUPT
  - CS_PACKETSIZE

- CS_PASSWORD
- CS_TDS_VERSION
- CS_TRANSACTION_NAME
- CS_USERDATA

See also

Related functions

- ct_cmd_props on page 81
- ct_config on page 96
- ct_connect on page 99
- ct_init on page 141

Related topics

- "Buffers" on page 23
- "Remote procedure calls (RPCs)" on page 48"

ct_describe

Description

Returns a description of result data.

Syntax

```
CS_RETCODE ct_describe(command, item_num, datafmt);
CS_COMMAND     *command;
CS_INT         item_num;
CS_DATAFMT     *datafmt;
```

Parameters

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call.

item_num

(I) Ordinal number of the column, parameter, or status being returned. This value is an integer.

When describing a column, *item_num* is the column number. For example, the first column in the select list of a SQL select statement is column number 1, the second is column number 2, and so forth.

When describing a return parameter, *item_num* is the ordinal rank of the parameter. The first parameter returned by a procedure or transaction is number 1. Adaptive Server Enterprise stored procedure return parameters are returned in the order originally specified in the create procedure statement for the stored procedure. This is not necessarily the same order as specified in the RPC that invoked the stored procedure or transaction.

In determining what number to assign to *item_num*, do not count non-return parameters. For example, if the second parameter in a stored procedure or transaction is the only return parameter, its *item_num* is 1.

When describing a stored procedure return status, *item_num* must be 1, as there can be only a single status in a return status result set.

To clear all bindings, assign *item_num* a value of CS_UNUSED.

DATAFMT

(O) A structure that contains a description of the result data item referenced by *item_num*. This structure is also used by ct_bind, ct_param and cs_convert and is explained in the Topics chapter, under "DATAFMT structure" on page 28.

Warning! You must initialize *DATAFMT* to zeroes. Failure to do so causes addressing exceptions.

The *DATAFMT* structure contains the following fields listed in Table 3-5.

Table 3-5: Fields in the DATAFMT structure for ct_describe

When this field	Is used with these result items	ct_describe sets the field to
name	Regular columns, return parameters	The null-terminated name of the data item, if any. To indicate a name does not exist, set namelen to 0.
namelen	Regular columns, return parameters	The actual length, in bytes, of *name*, not including the null terminator. A zero value here indicates no name.

When this field	Is used with these result items	ct_describe sets the field to
datatype	Regular columns, return parameters, return status	The datatype of the data item. All items listed in "DATAFMT structure" on page 28 are valid. A return status always has a datatype of CS_INT.
format	Not used (CS_FMT_UNUSED)	Not applicable.
maxlength	Regular columns, return parameters	The maximum possible length, of the data for the column or parameter being described.
scale	Regular columns and return parameters with a datatype of packed decimal (CS_PACKED370), or Sybase-decimal/numeric	The number of digits to the right of the decimal point.
precision	Regular columns and return parameters whose datatype is packed decimal (CS_PACKED370), or Sybase-decimal/numeric	The total number of decimal digits in the result data item.
status	Regular columns only	One or more of the following symbolic values, added together: • CS_CANBENULL to indicate a column that was tagged "nullable" by the server. • CS_NODATA to indicate that no data is associated with the column.
count	Regular columns, return parameters, return status	The number of rows copied to destination variables per ct_fetch call. ct_describe initializes count as 1 to provide a default value in case an application uses the ct_describe return datafmt structure as the ct_bind input datafmt structure. This value is always 1 for return parameters and status results.
usertype	Regular columns, return parameters	The user-defined datatype of the column or parameter, if any. usertype is set in addition to (not instead of) DATATYPE. **Note** This field is used for datatypes defined at the server, not for Open Client user-defined datatypes.
locale	Reserved for future use (CS_FORMAT_UNUSED)	Reserved for future use.

Return value

ct_describe returns one of the following values listed in Table 3-6.

Table 3-6: return values

Value	Meaning
CS-SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed. ct_describe returns CS_FAIL if *item_num* does not represent a valid result data item.
TDS_CANCEL_RECEIVED (-12)	Operation canceled. The remote partner issued a cancel. The current operation failed.
TDS_CONNECTION_TERMINATED (-4997)	The connection is not active.
TDS_NO_COMPUTES_ALLOWED (-60)	Compute results are not supported.
TDS_RESULTS_CANCELED (-49)	A cancel was sent to purge results.
TDS_WRONG_STATE (-6)	Program is in the wrong communication state to issue this call.

Examples

The following code fragment demonstrates the use of ct_describe. It is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
  /********************************************************************/
  /*                                                                  */
  /* Subroutine to process result rows                                */
  /*                                                                  */
  /********************************************************************/
  void   result_row_processing ()
  {
  CS_INT        rc;
  CS_INT        col_len;
  CS_INT        &&numcol;
  CS_INT        parm_cnt;
  char          msg1[40] = "The maximum number of connections is   ";
  char          msg2[25] = "The number of columns is ";
  char          wrk_str [4];
  char          period = '.';
/*------------------------------------------------------------*/
/* We need to bind the data to program variables. We don't    */
/* care about the indicator variable so we'll pass NULL for   */
/* that parameter in OC_BIND().                               */
/*------------------------------------------------------------*/

       rc = ct_res_info(cmd,CS_NUMDATA,&&numcol,
       sizeof(&numcol),&col_len);

       if (rc != CS_SUCCEED)
       {
```

```
          strncpy (msgstr, "CT_RES_INFO failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
       }
/*-------------------------------------------------------------*/
/* display the number of connections                           */
/*-------------------------------------------------------------*/
        row_num = row_num + 1;
        strncpy (RS[row_num].rsltno, msg1, msg_size);

        cvtleft  = 4;           /* Digits to the left */
        cvtright = 0;           /* Digits to the right */
        SYCVTD(maxconnect, wrk_str,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

        strncat (RS[row_num].rsltno, wrk_str, 4);
        row_num = row_num + 2;
/*-------------------------------------------------------------*/
/* display the number of columns                               */
/*-------------------------------------------------------------*/

        strncpy (RS[row_num].rsltno, msg2, sizeof(msg2));
        cvtleft  = 4;           /* Digits to the left */
        cvtright = 0;           /* Digits to the right */
        SYCVTD(&numcol, wrk_str,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

        strncat (RS[row_num].rsltno, wrk_str, 4);
        row_num = row_num + 2;

        if (&numcol != 2)
        {
          strncpy (msgstr, "CT_RES_INFO returned wrong # of parms",
                  msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
        }
/*-------------------------------------------------------------*/
/* Setup column headings                                       */
/*-------------------------------------------------------------*/

        strncpy (RS[row_num].rsltno,
                 "FirstName    EducLvl", text_size);
        row_num = row_num + 1;
        strncpy (RS[row_num].rsltno,
                 "==========   =======", text_size);
```

```
        for (parm_cnt = 1; parm_cnt <= &numcol; ++parm_cnt)
            bind_columns (parm_cnt);

}   /* end result_row_processing */

/*********************************************************************/
/*                                                                   */
/* Subroutine to bind each data                                      */
/*                                                                   */
/*********************************************************************/
void   bind_columns (parm_cnt)

CS_INT        parm_cnt;
{
 CS_INT        rc;
 CS_INT        col_len;
 CS_DATAFMT    datafmt;

       rc= ct_describe(cmd, parm_cnt, &dtafmt);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_DESCRIBE failed", msg_size);
          no_errors_sw = FALSE ;
          error_out(rc);
       }
/*------------------------------------------------------------*/
/* We need TO bind the data TO program variables. We don't    */
/* care about the indicator variable so we'll pass NULL for   */
/* that PARAMeter in OC_BIND().                               */
/*------------------------------------------------------------*/
/*------------------------------------------------------------*/
/* rows per fetch                                             */
/*------------------------------------------------------------*/

       switch (datafmt.datatype)
       {
/*------------------------------------------------------------*/
/* bind the first column, FIRSTNME defined as VARCHAR(12)     */
/*------------------------------------------------------------*/
           case  CS_VARCHAR_TYPE:

              rc= ct_bind(cmd, parm_cnt, &dtafmt, &col_firstnme,
                          &col_len, CS_NULL);
```

```
                if (rc != CS_SUCCEED)
                {
                   strncpy (msgstr, "CT_BIND CS_VARCHAR_TYPE failed",
                   msg_size);
                   no_errors_sw = FALSE ;
                   error_out(rc);
                }
                break;
/*-------------------------------------------------------------*/
/* bind the second column, EDLEVEL defined as SMALLINT        */
/*-------------------------------------------------------------*/
             case  CS_SMALLINT_TYPE:

                rc= ct_bind(cmd, parm_cnt, &dtafmt, &col_edlevel,
                            &col_len, CS_NULL);

                if (rc != CS_SUCCEED)
                {
                   strncpy (msgstr, "CT_BIND CS_SMALLINT_TYPE failed",
                   msg_size);
                   no_errors_sw = FALSE ;
                   error_out(rc);
                }
                break;

             default:
                break;

         } /* end of switch (datatype) */

}   /* end bind_columns */
```

Usage

- ct_describe returns a complete description of a result data item in the current result set. Result data items include regular result columns, return parameters, and stored procedure return status values.
- An application can call ct_res_info to find out how many result items are present in the current result set.
- An application generally needs to call ct_describe to describe a result data item after it establishes a connection and sends a request, and before it binds the result item to a program variable using ct_bind.
- ct_describe also indicates when the client issues a cancel.

See also

Related functions

- ct_bind on page 61

- ct_fetch on page 134
- ct_res_info on page 154
- ct_results on page 160,

Related topics

- "DATAFMT structure" on page 28
- "Results" on page 51

ct_diag

Description

Manages in-line error handling.

Syntax

```
CS_RETCODE ct_diag(connection, compiler, operation,
               msgtype, index, buffer);
CS_CONNECTION    *connection;
CS_INT           operation
CS_INT           msgtype
CS_INT           index
CS_BYTE            *buffer;
```

Parameters

connection

(I) Handle for this connection. This connection handle must already be allocated with ct_con_alloc.

operation

(I) Operation to perform. Assign this argument one of the following values:

Value	Meaning
CS_GET (33)	Retrieves a specific message.
CS_CLEAR (35)	Clears message information for this connection.
CS_INIT (36)	Initializes in-line error handling.
CS_STATUS (37)	Returns the current number of stored messages.
CS_MSGLIMIT (38)	Sets the maximum number of messages to store.

msgtype

(I) Type of message or structure on which the operation is to be performed. One of the following symbolic values:

Value	Meaning
CS_CLIENTMSG_TYPE (4700)	A CS_CLIENTMSG structure. Indicates Client-Library messages.

Value	Meaning
CS_SERVERMSG_TYPE (4701)	A CS_SERVERMSG structure. Indicates messages sent by Mainframe ClientConnect or another server.
CS_ALLMSG (4702)	Operation is performed on both Client-Library and server messages.
SQLCA_TYPE (4703)	A SQLCA structure.
SQLCODE_TYPE (4704)	A SQLCODE structure.

index

(I) Index number of the message being retrieved. Messages are numbered sequentially: the first message has an index of 1, the second an index of 2, and so forth.

If *msgtype* is CS_CLIENTMSG_TYPE, then *index* refers to Client-Library messages only.

If *msgtype* is CS_SERVERMSG_TYPE, then *index* refers to server messages only.

If *msgtype* is CS_ALLMSG_TYPE, then *index* refers to both Client-Library and server messages.

index should be initialized to 1.

buffer

(I/O) An integer or a variable (“buffer”) that contains the message. See Table 3-7 below, to learn the relationship between *buffer* and other arguments.

This argument is typically either CS_CHAR, a SQLCA structure, or a CS_CLIENTMSG or CS_SERVERMSG structure.

Note It is the responsibility of the programmer to provide a buffer large enough to hold the largest possible message.

Return value

ct_diag returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed. Common reasons for a ct_diag failure include: • Invalid *connection*. • Inability to allocate memory. • Invalid parameter (for example, parameter is not allowed for operation). • Invalid parameter combination.

Value	Meaning
CS_NOMSG (-207)	The application attempted to retrieve a message for which the index number is greater than the number of messages in the queue. For example, the application attempted to retrieve message number 3, when only 2 messages are queued.

Examples

Example 1

The following example uses ct_diag to prepare to receive messages.
This example is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
    /******************************************************************/
    /*                                                                */
    /* Subroutine to process input data                               */
    /*                                                                */
    /******************************************************************/
    void   proces_input ()
    {
    CS_INT        rc;
    CS_INT        *outlen;
    CS_INT        buf_len;
    CS_INT        msglimit;
    CS_INT        netdriver;
   /*------------------------------------------------------------*/
   /* Allocate a connection to the server                        */
   /*------------------------------------------------------------*/
 rc = ct_con_alloc (context, &connection);
 if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CONALLOC failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
          }

    /*------------------------------------------------------------*/
    /* Alter properties of the connection for user-id             */
    /*------------------------------------------------------------*/

         buf_len  = user_size;
         rc = ct_con_props (connection, (long)CS_SET,
                           (long)CS_USERNAME, username,
                           buf_len, outlen);
 if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CON_PROPS for user-id failed",
```

```
             msg_size);
             no_errors_sw = FALSE ;
             error_out (rc);
           }
     /*------------------------------------------------------------*/
     /* Alter properties of the connection for password            */
     /*------------------------------------------------------------*/
   buf_len = pwd_size;
            rc = ct_con_props (connection, (long)CS_SET,
                              (long)CS_PASSWORD, pwd, buf_len, outlen);
   if (rc != CS_SUCCEED)
           {
             strncpy (msgstr, "CT_CON_PROPS for password failed",
             msg_size);
             no_errors_sw = FALSE ;
             error_out (rc);
           }
   /*------------------------------------------------------------*/
   /* Alter properties of the connection for transaction         */
   /*------------------------------------------------------------*/
   buf_len = tran_size;
            rc = ct_con_props (connection, (long)CS_SET,
                              (long)CS_TRANSACTION_NAME,
                              tran, buf_len, outlen);
   if (rc != CS_SUCCEED)
          {
             strncpy (msgstr, "CT_CON_PROPS for transaction failed",
             msg_size);
             no_errors_sw = FALSE ;
             error_out (rc);
          }

      /*------------------------------------------------------------*/
      /* Alter properties of the connection for network driver      */
      /*------------------------------------------------------------*/
   netdriver = 9999;   /* default value for non-regconized
                                     driver name                   */
   /* if no netdriver entered, default is LU62  */
            if (strncmp(driver,"         ",9) == 0  ??
                 strncmp(driver,"LU62",4) == 0)
                netdriver = CS_LU62;
   else if (strncmp(driver,"INTERLINK",8) == 0)
                netdriver = CS_INTERLINK;
   else if (strncmp(driver,"IBMTCPIP",8) == 0)
                netdriver = CS_TCPIP;
   else if (strncmp(driver,"CPIC",4) == 0)
```

```
            netdriver = CS_NCPIC;
rc = ct_con_props (connection, (long)CS_SET,
                           (long)CS_NET_DRIVER, (long) netdriver,
                           CS_UNUSED, outlen);
if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CON_PROPS for network driver failed",
            msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }
/*------------------------------------------------------------*/
/* Setup retrieval of All Messages                            */
/*------------------------------------------------------------*/
rc = ct_diag (connection, CS_INIT,
                       CS_UNUSED, CS_UNUSED, CS_NULL);
if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_DIAG CS_INIT failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }
/*------------------------------------------------------------*/
/* Set the upper limit of number of messages                  */
  /*------------------------------------------------------------*/
msglimit = 5 ;
rc = ct_diag (connection, CS_MSGLIMIT, CS_ALLMSG_TYPE,
                       CS_UNUSED, &msglimit);
if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_DIAG CS_MSGLIMIT failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }
/*------------------------------------------------------------*/
/* Open connection to the server or CICS region               */
/*------------------------------------------------------------*/
rc = ct_connect (connection, servname, server_size);
if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CONNECT failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }
```

Example 2

The following example uses ct_diag to retrieve diagnostic messages. This example is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
/******************************************************************/
/*                                                                */
/* Subroutine to retrieve any diagnostic messages                 */
/*                                                                */
/*******************************************************************/
 void   get_diag_messages()
{
   CS_SMALLINT   cnt;
   CS_INT        num_of_msgs = 0;
   CS_INT        rc;
/*----------------------------------------------------------------*/
/* Disable calls to this subroutine                               */
/*----------------------------------------------------------------*/
diag_msgs_initialized = FALSE ;

    /*----------------------------------------------------------------*/
    /* First, get client messages                                     */
    /*----------------------------------------------------------------*/
         rc = ct_diag (connection, CS_STATUS, CS_CLIENTMSG_TYPE,
                       CS_UNUSED, #_of_msgs);
if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_DIAG CS_STATUS CLIENTMSG_TYPE failed",
                     msg_size);
            error_out(rc) ;
          }
          else if (num_of_msgs > 0)
          {
               for (cnt = 1; cnt <= num_of_msgs; ++cnt)
                   get_client_msgs ();
          }
  /*----------------------------------------------------------------*/
  /* Then, get server messages                                      */
  /*----------------------------------------------------------------*/
         rc = ct_diag (connection, CS_STATUS, CS_SERVERMSG_TYPE,
                       CS_UNUSED, #_of_msgs);
         if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_DIAG CS_STATUS SERVERMSG_TYPE failed",
                     msg_size);
            error_out(rc) ;
```

```
          }
          else if (num_of_msgs > 0)
          {
               for (cnt = 1; cnt <= num_of_msgs; ++cnt)
                   get_server_msgs ();
          }
  }   /* end get_diag_messages  */

   /*****************************************************************/
   /*                                                               */
   /* Subroutine to retrieve diagnostic messages from client        */
   /*                                                               */
   /*****************************************************************/
   void   get_client_msgs()
   {
   CS_INT        rc;
   CS_INT        i;
   CS_CHAR      *txtpos;
   CS_INT        textleft;
   CS_INT        msgno = 1;
   CS_CHAR       blank_13[13] = "             ";
   CS_CLIENTMSG  clientmsg;
   struct {
           char    msgno_hdr[13];
           char    msgno_data[8];
           char    severity_hdr[13];
           char    severity_data[6];
          } client_msg;
         rc = ct_diag (connection, CS_GET, CS_CLIENTMSG_TYPE,
                          msgno, &clientmsg);
if (rc !=  CS_SUCCEED)
         {
            strncpy (msgstr, "CT_DIAG CS_GET CS_CLIENTMSG_TYPE failed",
                          msg_size);
            error_out(rc) ;
         }
  /*--------------------------------------------------------------*/
  /* display message text                                         */
  /*--------------------------------------------------------------*/
         i = 1 ;
         strncpy (RS[i].rsltno, "Client Message:");
         i = 3 ;
memset(&client_msg, ' ', sizeof(client_msg));
         strcpy (client_msg.msgno_hdr, "  OC MsgNo:  ");
         strcpy (client_msg.severity_hdr, "  Severity:  ");
cvtleft  = 8;            /* Digits to the left */
```

```
          cvtright = 0;            /* Digits to the right */
          SYCVTD(clientmsg.msgnumber, client_msg.msgno_data,
                  cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
          cvtleft  = 6;            /* Digits to the left */
          SYCVTD(clientmsg.severity, client_msg.severity_data,
                  cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
memcpy (RS[i].rsltno, &client_msg, sizeof(client_msg));
          i += 1 ;
/* get number of Client msgs */
if (clientmsg.msgnumber != 0)
          {
             strcpy (RS[i].rsltno, "  OC MsgTx:  ");
             strncat (RS[i].rsltno, clientmsg.msgstring, 66);
             i += 1 ;
             txtpos = clientmsg.msgstring + 66;
             textleft = clientmsg.msgstringlen - 66;
             while (textleft > 0)
             {
                strncpy (RS[i].rsltno, blank_13, 13);
                strncat (RS[i].rsltno, txtpos, 66);
                i += 1;
                txtpos += 66;
                textleft -= 66;
             }
          }
          else
          {
             strncpy (RS[i].rsltno, "  OC MsgTx:  No Message!",
             text_size);
             i += 1 ;
          }
/* get number of Server msgs */
memset(&client_msg, ' ', sizeof(client_msg));
          strcpy (client_msg.msgno_hdr, "  OS MsgNo:  ");
          cvtleft  = 8;            /* Digits to the left */
          cvtright = 0;            /* Digits to the right */
          SYCVTD(clientmsg.osnumber, client_msg.msgno_data,
                  cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
memcpy (RS[i].rsltno, &client_msg, sizeof(client_msg));
          i += 1 ;
if (clientmsg.osnumber != 0)
          {
             strcpy (RS[i].rsltno, "  OS MsgTx:  ");
             strncat (RS[i].rsltno, clientmsg.osstring, 66);
             i += 1 ;
             txtpos = clientmsg.osstring + 66;
```

```
            textleft = clientmsg.osstringlen - 66;
            while (textleft > 0)
            {
               strncpy (RS[i].rsltno, blank_13, 13);
               strncat (RS[i].rsltno, txtpos, 66);
               i += 1;
               txtpos += 66;
               textleft -= 66;
            }
         }
         else
         {
            strncpy (RS[i].rsltno, "  OS MsgTx:  No Message!",
            text_size);
            i += 1 ;
         }

   }   /* end get_client_msgs */

    /*------------------------------------------------------------------*/
    /*                                                                  */
    /* Subroutine to retrieve diagnostic messages from server           */
    /*                                                                  */
    /*------------------------------------------------------------------*/
    void   get_server_msgs()
    {
    CS_INT        rc;
    CS_INT        i;
    CS_CHAR      *txtpos;
    CS_INT        textleft;
    CS_INT        msgno = 1;
    CS_CHAR       blank_13[13] = "             ";
    CS_CHAR       proc_id_data[66];
    CS_CHAR       svrname_data[66];
    CS_SERVERMSG  servermsg;

    struct {
           char    msg_no_hdr[13];
           char    msg_no_data[6];
           char    severity_hdr[14];
           char    severity_data[6];
           char    state_hdr[14];
           char    state_data[4];
           char    line_no_hdr[13];
           char    line_no_data[4];
          } serv_msg;
```

```
        memset(&serv_msg, ' ', sizeof(serv_msg));

        rc = ct_diag (connection, CS_GET, CS_SERVERMSG_TYPE,
                      msgno, &servermsg);
         if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_DIAG CS_GET CS_SERVERMSG_TYPE failed",
                    msg_size);
           error_out (rc) ;
        }
/*------------------------------------------------------------------*/
/* display message text                                             */
/*------------------------------------------------------------------*/

        strcpy (serv_msg.msg_no_hdr, "  Message#:  ");
        strcpy (serv_msg.severity_hdr, ",  Severity:  ");
        strcpy (serv_msg.state_hdr, ",  State No:  ");
        strcpy (serv_msg.line_no_hdr, "  Line  No:  ");

        cvtleft  = 6;              /* Digits to the left */
        cvtright = 0;              /* Digits to the right */
        SYCVTD(servermsg.msgnumber, serv_msg.msg_no_data,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
        SYCVTD(servermsg.severity, serv_msg.severity_data,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

        cvtleft  = 4;              /* Digits to the left */
        SYCVTD(servermsg.state, serv_msg.state_data,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
        SYCVTD(servermsg.line, serv_msg.line_no_data,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

        if (servermsg.svrnlen > 66)
        {
             strncpy (svrname_data, servermsg.svrname, 63);
             strcat (svrname_data, "...");
        }
        else
             strncpy (svrname_data, servermsg.svrname, 66);

        if (servermsg.proclen > 66)
        {
             strncpy (proc_id_data, servermsg.proc, 63);
             strcat (proc_id_data, "...");
```

```
            }
            else
                 strncpy (proc_id_data, servermsg.proc, 66);

            strncpy (RS[1].rsltno, "Server Message:", text_size);
            memcpy (RS[3].rsltno, &serv_msg, sizeof(serv_msg));
            strcpy (RS[5].rsltno, "  Serv Nam:  ");
            strcat (RS[5].rsltno, svrname_data);
            strcpy (RS[6].rsltno, "  Proc ID :  ");
            strcat (RS[6].rsltno, proc_id_data);
            strcpy (RS[7].rsltno, "  Message :  ");
            strncat (RS[7].rsltno, servermsg.text, 66);

            i = 8 ;
            txtpos = servermsg.text + 66;
            textleft = servermsg.textlen - 66;
            while (textleft > 0)
               {
                  strncpy (RS[i].rsltno, blank_13, 13);
                  strncat (RS[i].rsltno, txtpos, 66);
                  i += 1;
                  txtpos += 66;
                  textleft -= 66;
               }
   }   /* end get_server_msgs */
```

Usage

Table 3-7 lists a summary of arguments for ct_diag.

Table 3-7: Summary of arguments

Operation	ct_diag action	msgtype value	index value	buffer value
CS_INIT	Initializes in-line error handling. An application must callct_diag with a CS_INIT operation before it can process error messages in line.	CS_UNUSED	CS_UNUSED	Ignored.

Operation	ct_diag action	msgtype value	index value	buffer value
CS_CLEAR	Clears message information for this connection. If buffer is not zeroes and *msgtype* is not CS_ALLMSG_ TYPE, ct_diag clears the buffer by initializing it with blanks or zeroes.	One of the legal *msgtype* values. • If *msgtype* is CS_CLIENTMSG_TYPE, ct_diag clears Client-Library messages only. • If *msgtype* is CS_SERVERMSG_TYPE, ct_diag clears server messages only. • If *msgtype* has any other legal value, ct_diag clears both Client-Library and server messages.	CS_UNUSED	A buffer whose type is defined by *msgtype*.
CS_GET	Retrieves a specific message.	Any legal *msgtype* value except CS_ALLMSG_TYPE. • If *msgtype* is CS_CLIENTMSG_TYPE, ct_diag retrieves a Client-Library message into a CS_CLIENTMSG structure. • If *msgtype* is CS_SERVERMSG_TYPE, ct_diag retrieves a server message into a CS_SERVERMSG structure. • If *msgtype* has any other value, ct_diag retrieves either a server message or a Client-Library message.	The index number of the message to retrieve.	A buffer whose type is defined by *msgtype*.
CS_MSGLIMIT	Sets the maximum number of messages to store.	CS_CLIENTMSG_TYPE to limit Client-Library messages only. CS_SERVERMSG_TYPE to limit server messages only. CS_ALLMSG_TYPE to limit the total number of Client-Library and server messages combined.	CS_UNUSED	An integer value.

Operation	ct_diag action	msgtype value	index value	buffer value
CS_STATUS	Returns the current number of stored messages.	CS_CLIENTMSG_TYPE to retrieve the number of Client-Library messages. CS_SERVERMSG_TYPE to retrieve the number of server messages. CS_ALLMSG_TYPE to retrieve the total number of Client-Library and server messages combined.	CS_UNUSED	An integer variable.

- ct_diag manages in-line message handling for a specific connection. If an application has more than one connection, it must make separate ct_diag calls for each connection.
- Open ClientConnect applications always use ct_diag to handle Client-Library and server messages. Applications built with Open Client can offer alternative message-handling facilities.
- An application can perform operations on Client-Library messages, server messages, or both.

  For example, an application can clear Client-Library messages without affecting server messages:

  ```
  rc = ct_diag (connection, CS_UNUSED, CS_CLEAR,
  CS_CLIENTMSG, CS_UNUSED, MSGBUFFER);
  ```

- ct_diag allows an application to retrieve message information into standard Client-Library structures (CS_CLIENTMSG, CS_SERVERMSG, SQLCA or SQLCODE).

  When retrieving messages, ct_diag assumes that *buffer* points to a structure of the type indicated by *msgtype*.

  - An application that retrieves messages into a SQLCA or SQLCODE structure must set the Client-Library property CS_EXTRA_INF to CS_TRUE. This is because the SQL structures require information that is not ordinarily returned by the Client-Library error handling mechanism.

    Use ct_con_props or cs_config to set CS_EXTRA_INF.

  - An application that does not use the SQLCA or SQLCODE structures can also set CS_EXTRA_INF to CS_TRUE. In this case, the extra information returns as standard Client-Library messages.

For more about CS_EXTRA_INF, see “Remote procedure calls (RPCs)” on page 48 and the reference pages for ct_con_props and cs_config.

Warning! If ct_diag does not have sufficient internal storage space in which to save a new message, it throws away all unread messages and stops saving messages. The next time it is called with *operation* as CS_GET, it returns a message to indicate the space problem. After returning this message, ct_diag starts saving messages again.

Initializing in-line error handling

- An application must initialize in-line error handling before it can retrieve any errors. To initialize in-line error handling, call ct_diag with *operation* as CS_INIT.
- Generally, if a connection uses in-line error handling, the application should call ct_diag to initialize in-line error handling for a connection immediately after allocating it with ct_con_alloc.

Clearing messages

- To clear message information for a connection, an application calls *operation* as CS_CLEAR.
    - To clear Client-Library messages only, set *msgtype* to CS_CLIENTMSG_TYPE.
    - To clear server messages only, set *msgtype* to CS_SERVERMSG_TYPE.
    - To clear both Client-Library and server messages, set *msgtype* to SQLCA, SQLCODE, or CS_ALLMSG_TYPE.
- If *operation* is CS_CLEAR and *msgtype* is not CS_ALLMSG_TYPE:
    - ct_diag assumes that *buffer* is a structure of type *msgtype*.
    - ct_diag clears the buffer by setting it to blanks or zeroes, as appropriate.
- Message information is not cleared until an application explicitly calls ct_diag with *operation* as CS_CLEAR. Retrieving a message does not remove it from the message queue.

Retrieving messages

- To retrieve message information, an application calls ct_diag with *operation* as CS_GET, *msgtype* as the type of structure in which to retrieve the message, *index* as the index number of the message of interest, and *buffer* as an integer or a variable, as appropriate.
- If *msgtype* is CS_CLIENTMSG_TYPE, *index* refers only to Client-Library messages.
- If *msgtype* is CS_SERVERMSG_TYPE, *index* refers only to server messages.
- If *msgtype* has any other value, *index* refers to the collective "queue" of both types of messages combined.
- ct_diag creates a message queue in the buffer and fills the buffer with message information. It returns messages to the client in the order in which they are received.
- If an application attempts to retrieve a message for which the index is higher than the highest valid index, ct_diag returns CS_NOMSG to indicate that no message is available.

Limiting messages

- The Client-Library default behavior is to save an unlimited number of messages. Applications running on platforms with limited memory may want to limit the number of messages that Client-Library saves. The default for MVS is 25.

  An application can limit the number of saved Client-Library messages, the number of saved server messages, and the total number of saved messages.
- To limit the number of saved messages, an application calls ct_diag with *operation* as CS_MSGLIMIT and *msgtype* as CS_CLIENTMSG_TYPE, CS_SERVERMSG_TYPE, or CS_ALLMSG_TYPE:
  - If *msgtype* is CS_CLIENTMSG_TYPE, the number of Client-Library messages is limited.
  - If *msgtype* is CS_SERVERMSG_TYPE, the number of server messages is limited.
  - If *msgtype* is CS_ALLMSG_TYPE, the total number of Client-Library and server messages combined is limited.

- When a specific message limit is reached, Client-Library discards any new messages of that type. When a combined message limit is reached, Client-Library discards any new messages.

Retrieving the number of messages

- To find out how many messages were retrieved, an application calls ct_diag with *operation* as CS_STATUS and *msgtype* as the type of message of interest.

See also

Related topics

- "Error and message handling" on page 35
- "CS_CLIENTMSG structure" on page 26
- "CS_SERVERMSG structure" on page 52
- "SQLCA structure" on page 54

ct_exit

Description

Exits Client-Library.

Syntax

```
CS_RETCODE ct_exit (context, option);
CS_CONTEXT   *context
CS_INT       option;
```

Parameters

context

(I) A context structure. The context structure is defined in the program call cs_ctx_alloc. This value identifies the Client-Library context being exited.

If this value is invalid or nonexistent, ct_exit fails.

option

(O) Indicator specifying whether or not ct_exit closes connections for which results are pending.

ct_exit can behave in different ways, depending on the value specified for *option*. The following table lists the symbolic values that are legal for *option*:

OPTION value	ct_exit action
CS_UNUSED (-99999)	Closes all open connections for which no results are pending and terminates Client-Library for this context. If results are pending on one or more connections, ct_exit returns CS_FAIL and does not terminate Client-Library.
CS_FORCE_EXIT (300)	Closes all open connections for this context, whether or not any results are pending, and terminates Client-Library for this context.

Return value

ct_exit returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.
TDS_INVALID_PARAMETER (-4)	A parameter contains an illegal value.
TDS_RESULTS_STILL_ACTIVE (-50)	Some results are still pending.
TDS_CONNECTION_TERMINATED (4997)	The connection is not active.
TDS_WRONG_STATE (-6)	Program is in the wrong communication state to issue this call.

Examples

The following code fragment demonstrates the use of ct_exit. It is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
  /********************************************************************/
  /*                                                                  */
  /* Subroutine to perform exit client library and deallocate context */
  /* structure.                                                       */
  /*                                                                  */
  /********************************************************************/
  void   quit_client_library ()
{
 CS_INT        rc;
/*------------------------------------------------------------------*/
/* Exit the Client Library                                          */
/*------------------------------------------------------------------*/
```

```
        rc = ct_exit (context, (long) CS_UNUSED);

        if (rc == CS_FAIL)
        {
           strncpy (msgstr, "CT_EXIT failed", msg_size);
           error_out(rc) ;
         }
/*------------------------------------------------------------------*/
/* De-allocate the context structure                                */
/*------------------------------------------------------------------*/
        rc = cs_ctx_drop (context);

        if (rc == CS_FAIL)
        {
           strncpy (msgstr, "CT_CTX_DROP failed", msg_size);
           error_out(rc) ;
        }

        EXEC CICS RETURN ;

}   /* end quit_client_library */
```

Usage

- ct_exit terminates a Client-Library context. It closes all open connections, deallocates internal data space and cleans up any platform-specific initialization.
- ct_exit must be the last Client-Library routine called within a Client-Library context.
- If an application needs to call Client-Library routines after it calls ct_exit, it can re-initialize Client-Library by calling ct_init again.
- If results are pending on any of the context connections and *option* is not passed as CS_FORCE_EXIT, ct_exit returns CS_FAIL. This means that Client-Library is not correctly terminated. The application must handle the pending results before calling ct_exit, or it can call ct_exit again, specifying CS_FORCE_EXIT.
- To close a single connection, an application calls ct_close.
- If ct_init is called for a context, the application must call ct_exit before it calls cs_ctx_drop to deallocate the context.

See also

Related functions

- ct_close on page 72
- ct_init on page 141

ct_fetch

Description

Fetches result data.

Syntax

```
CS_RETCODE  ct_fetch(command, type, offset, option,
                rows_read);
CS_COMMAND  *command;
CS_INT      type;
CS_INT      offset;
CS_INT      option;
CS_INT      *rows_read;
```

Parameters

command

(I) Handle for this client/server operation. This command handle must already be allocated with ct_cmd_alloc.

type

(I) This argument is currently unused and should be passed as CS_UNUSED in order to ensure compatibility with future versions of Client-Library.

offset

(I) This argument is currently unused and should be passed as CS_UNUSED in order to ensure compatibility with future versions of Client-Library.

option

(I) This argument is currently unused and should be passed as CS_UNUSED in order to ensure compatibility with future versions of Client-Library.

rows_read

(I) Variable where the number of result rows is returned. This variable is of type integer. ct_fetch sets *rows_read* to the number of rows read by the ct_fetch call. This argument is required.

Return value

ct_fetch returns one of the following values listed in Table 3-8.

Table 3-8: return values

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully. ct_fetch places the total number of rows read in *rows_read*.
CS_FAIL (-2)	The routine failed. ct_fetch places the number of rows fetched before the failure occurred in *rows_read*. A common reason for a ct_fetch failure is that a program variable specified through ct_bind is too small to hold a fetched data item.
CS_CANCELLED (-202)	The operation was canceled. ct_fetch places the number of rows fetched before the cancel occurred in *rows_read*.

Value	Meaning
CS_ROW_FAIL (-203)	A recoverable error occurred while fetching a row. Recoverable errors include memory allocation failures and conversion errors that occur while copying row values to program variables. An application can continue calling ct_fetch to continue retrieving rows, or can call ct_cancel to cancel the remaining results. ct_fetch places the number of rows fetched before the error occurred in *rows_read*, then continues by fetching the row after the error.
CS_END_DATA (-204)	No more rows are available in this result set (Note that this is also a successful completion).
TDS_INVALID_PARAMETER (-4)	One of the ct_fetch arguments contains an illegal value. The most likely cause of this code is assigning a value other than CS_UNUSED to one or more of the reserved arguments, *type*, *offset*, and *option*.
TDS_WRONG_STATE (-6)	Program is in the wrong communication state to issue this call. It is in Send state instead of Receive state.

Examples

The following example shows a typical use of ct_fetch. It is taken from the SYCTSAA6 sample program in Appendix A, “Sample Language Application.”

```
/********************************************************************/
/*                                                                  */
/* Subroutine to fetch row processing                               */
/*                                                                  */
/********************************************************************/
void   fetch_row_processing ()
{
CS_INT        rows_read;
CS_INT        rc;
CS_INT        col_len;
CS_INT        max_screen_rows = 10;
CS_SMALLINT   nomore_rows    = 0;
CS_DATAFMT    datafmt;
CS_DATAFMT    datafmt2;
struct {
       char  first[12];
       char  space2[2];
       char  edlevel[4];
      } output_row;
   while (nomore_rows == FALSE)
   {
     strcpy(col_firstnme.str, "            ");
     memset (&output_row, ' ', sizeof(output_row));
```

```
        rc = ct_fetch (cmd, (long) CS_UNUSED,
                       (long) CS_UNUSED,
                       (long) CS_UNUSED,
                       &rows_read);

        switch (rc)
        {
            case  CS_SUCCEED:
                nomore_rows         = FALSE ;
                datafmt.datatype     = CS_VARCHAR_TYPE;
                datafmt.maxlength    = sizeof(col_firstnme);
                datafmt2.datatype    = CS_CHAR_TYPE;
                datafmt2.maxlength   = 12;
/*------------------------------------------------------------*/
/* convert the first column from VARCHAR to CHAR for display  */
/*------------------------------------------------------------*/
               if (cs_convert(context, datafmt, col_firstnme, datafmt2,
                              &output_row.first, &col_len) !=CS_SUCCEED)
               {
                  strncpy (msgstr,
                    "CS_CONVERT CS_CHAR_TYPE failed", msg_size);
                  no_errors_sw = FALSE ;
                  error_out(rc);
               }
 /*------------------------------------------------------------*/
 /* save ROW RESULTS for later display                         */
 /*------------------------------------------------------------*/

                cvtleft  = 4;           /* Digits to the left */
                cvtright = 0;           /* Digits to the right */
                SYCVTD(col_edlevel, output_row.edlevel,
                        cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

                if (row_num > max_screen_rows)
                {
                   strncpy (msgtext1, "Please press return to
                   continue!",text_size);
                   memset (msgtext2, ' ', text_size);

                   disp_data ();

                   /* re-init output lines */
                   for (row_num = 0; row_num < 14; ++row_num)
                       memset (RS[row_num].rsltno, ' ', text_size);
```

```
                row_num = 1;
                page_cnt = page_cnt + 1 ;
/*------------------------------------------------------------*/
/* Setup column headings                                      */
/*------------------------------------------------------------*/
                strncpy (RS[row_num].rsltno, "FirstName    EducLvl",
                         text_size);
                row_num += 1;
                strncpy (RS[row_num].rsltno, "=========   =======",
                         text_size);
                row_num += 1;
             }  /* if row_num > 10 */

             row_num += 1;
             memcpy (RS[row_num].rsltno, &output_row,
                     sizeof(output_row));

             break; /* end of CS_SUCCEED */

         case  CS_END_DATA:
             nomore_rows = TRUE ;
             strncpy (msgtext1, "All rows processing completed!",
                      text_size);
             strncpy (msgtext2, "Press Clear To Exit", text_size);
             disp_data ();
             break; /* end of CS_END_DATA  */

         case  CS_FAIL:
             nomore_rows = TRUE ;
             no_errors_sw = FALSE ;
             strncpy (msgstr, "CT_FETCH returned CS_FAIL ret_code");
             error_out(rc);
             break; /* end of CS_FAIL  */

         case  CS_ROW_FAIL:
             nomore_rows = TRUE ;
             no_errors_sw = FALSE ;
             strncpy (msgstr, "CT_FETCH returned CS_ROW_FAIL
             ret_code");
             error_out(rc);
             break; /* end of CS_ROW_FAIL  */

         case  CS_CANCELLED:
             nomore_rows = TRUE ;
             no_errors_sw = FALSE ;
             strncpy (msgstr, "CT_FETCH returned CS_CANCELLED
```

```
                    ret_code");
                    error_out(rc);
                    break; /* end of CS_CANCELLED  */

                default:
                    nomore_rows = TRUE ;
                    no_errors_sw = FALSE ;
                    strncpy (msgstr, "CT_FETCH returned Unknown ret_code");
                    error_out(rc);
                    break; /* end of OTHERWISE */

            }  /* end of switch (rc) */
          }   /* end of while nomore_rows false */

     }  /* end fetch_row_processing */
```

Usage

- ct_fetch fetches result data. "Result data" is an umbrella term for the various types of data that a server can return to an application. These types of data include:
  - Regular rows.
  - Return parameters, including both message parameters and RPC return parameters.
  - Stored procedure status results.

  ct_fetch is used to fetch all of these types of data.
- Conceptually, result data is returned to an application in the form of one or more rows that make up a "result set".

  Regular row result sets can contain more than one row. For example, a regular row result set might contain a hundred rows. If array binding is specified for the data items in a regular row result set, then multiple rows can be fetched with a single call to ct_fetch. The number of rows fetched is returned in the *rows_read* argument.

  Return parameters and status results, however, only contain a single row. For this reason, even if array binding is specified, only a single row of data is fetched.
- ct_results specifies the type of result available in the *result_type* variable. ct_results must indicate a result type of CS_ROW_RESULT, CS_PARAM_RESULT, or CS_STATUS_RESULT before an application calls ct_fetch.
- After calling ct_results, an application can:

- Process the result set by binding the result items and fetching the data, using ct_fetch (optionally preceded by ct_describe and ct_bind), or
- Discard the result set, using ct_cancel.

- If an application does not cancel a result set, it must completely process the result set by repeatedly calling ct_fetch as long as ct_fetch continues to indicates that rows are available.

  The simplest way to do this is in a loop that terminates when ct_fetch fails to return either CS_SUCCEED or CS_ROW_FAIL. After the loop terminates, an application can check the ct_fetch final return code to find out what caused the termination.

Fetching regular rows

- Regular rows can be fetched from the server one row at a time, or several rows at once.
- When fetching multiple rows, the number of rows to be fetched is indicated by the count field in the datafmt structures used to bind the data items in the result set. Note that the count field must have the same value for all ct_bind calls for a result set.

  If count is 0 or 1, ct_fetch fetches one row.

Fetching return parameters

- A return parameter result set contains either stored procedure return parameters or message parameters.
- A return parameter result set consists of a single row with a number of columns equal to the number of return parameters.

Fetching a return status

- A stored procedure return status result set consists of a single row with a single column, containing the status.

See also

Related functions

- ct_bind on page 61
- ct_describe on page 109
- ct_results on page 160

Related documentation

- Mainframe Connect Client Option and Server Option *Messages and Codes*

ct_get_format

Description

Returns the user-defined format for a result column.

Note This function is used with requests to Adaptive Server Enterprise only.

Syntax

```
CS_RETCODE cs_get_format (command, column_num,
                          buffer, buf_len, outlen);
CS_COMMAND   *command;
CS_INT       column_num;
CS_BYTE      *buffer;
CS_INT       buf_len;
CS_INT       *outlen;
```

Parameters

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call.

column_num

(I) Number of the column for which the user-specified format is desired.

column_num refers to the select-list ID of the column. The first column in the select list of a select statement is column number 1, the second is column number 2, and so forth.

buffer

(O) Variable ("buffer") in which ct_get_format places the requested information.

This argument is typically:

```
char buffer[n];
```

buf_len

(I) Length, in bytes, of the buffer.

If *buf_len* is too small to hold the requested information, ct_get_format sets *outlen* to the length of the requested information, and returns CS_FAIL.

outlen

(O) Length, in bytes, of the format string. *outlen* is an integer variable where ct_get_format returns the total number of bytes being retrieved.

When the format string is larger than *buf_len* bytes, an application uses this value to determine how many bytes are needed to hold the string.

If no format string is associated with the specified column, ct_get_format sets *outlen* to 0.

Return value

ct_get_format returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.
TDS_INVALID_PARAMETER(-4)	One of ct_get_format arguments contains an illegal value.

Usage

- ct_get_format returns the user-defined format, if any, for a result column. It indicates how the field should be formatted on screen.
- An application can call ct_get_format after ct_results indicates results of type CS_ROW_RESULT.
- For a description of how to add user-defined formats to Adaptive Server Enterprise databases or Open Servers, see the Adaptive Server Enterprise and Open Server documentation.

See also

Related functions

- ct_bind on page 61
- ct_describe on page 109

ct_init

Description

Initializes Client-Library.

Syntax

```
CS_RETCODE ct_init (connection, version);
CS_CONTEXT  *context;
CS_INT      version;
```

Parameters

context

(I) A context structure. The context structure is defined in the program call cs_ctx_alloc. If this value is invalid or nonexistent, ct_init fails.

version

(I) Version of Client-Library behavior that the application expects. The following table lists the symbolic values that are legal for *version*.

Value	Meaning	Supported features
CS_VERSION_46	Application communicates with a version 4.6 SQL Server.	RPCs.

Value	Meaning	Supported features
CS_VERSION_50	Application communicates with a version 10.0 SQL Server and above.	RPCs.

Return value

ct_init returns one of the following values listed in Table 3-9.

Table 3-9: return values

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_MEM_ERROR (-3)	The routine failed due to a memory allocation error.
CS_FAIL (-2)	The routine failed for other reasons. **Note** ct_init returns CS_FAIL if Client-Library cannot provide version-level behavior.
TDS_INVALID_PARAMETER (-4)	A parameter contains an illegal value. The most likely cause is an erroneous version number.
TDS_WRONG_STATE (-6)	Program is in the wrong communication state to issue this call. The most likely cause is that this context was already initiated.

Examples

The following code fragment demonstrates how ct_init is used with other functions to initialize a program. It is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
   /*------------------------------------------------------------------*/
   /* program initialization                                           */
   /*------------------------------------------------------------------*/
        memset (msgtext1, ' ', text_size);
        strncpy (msgtext2, "Press Clear To Exit", text_size);
        page_cnt  = page_cnt + 1;
        memset(servname, ' ', 30);
        memset(username, ' ', 8);
        memset(tran, ' ', 8);
        memset(pwd, ' ', 8);
        memset(driver, ' ', 9);

        for (i = 0; i < 14; ++i)
           memset (RS[i].rsltno, ' ', text_size); /* init output
lines  */

        /*  get system time   */
        EXEC CICS ASKTIME ABSTIME(UTIME);
```

```
        EXEC CICS FORMATTIME
                  ABSTIME(UTIME)
                  DATESEP('/')
                  MMDDYY(TMP_DATE)
                  TIME(TMP_TIME)
                  TIMESEP ;

        display_initial_screen ();
        get_input_data ();
 /*------------------------------------------------------------------*/
 /* Allocate a context structure                                     */
 /*------------------------------------------------------------------*/

     if (no_input == FALSE)
     {
        version = CS_VERSION_50;

        rc = cs_ctx_alloc (version, &context);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CSCTXALLOC failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
           EXEC CICS RETURN ;
        }

 /*------------------------------------------------------------------*/
 /* Initialize the Client-Library                                    */
 /*------------------------------------------------------------------*/

        /* context allocated, it's now OK to use ct_diag for message
           retrieving                                                */
        diag_msgs_initialized = 1;

        rc = ct_init (context, version);

        if (rc == CS_SUCCEED)
        {
           proces_input ();
        }
        else
        {
           strncpy (msgstr, "CT_INIT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
```

```
            }
            close_connection ();
            quit_client_library ();
         }  /* process input data entered */

         else  /* no input data received */
            EXEC CICS RETURN ;

     }   /* end main */
```

Usage

- ct_init initializes Client-Library. It sets up internal control structures and defines the version of Client-Library behavior that an application expects. Client-Library provides the requested behavior, regardless of the actual version of Client-Library in use.
- ct_init must be the first Client-Library routine call after cs_ctx_alloc. Other Client-Library routines fail if they are called before ct_init.
- Because an application calls ct_init before it sets up error handling, an application must check the ct_init return code to detect failure.
- It is not an error for an application to call ct_init multiple times. Some applications cannot guarantee which of several modules executes first. In such a case, each module should contain a call to ct_init.

See also

Related functions

- cs_ctx_alloc on page 179
- ct_exit on page 131

ct_param

Description

Defines a command parameter.

Syntax

```
CS_RETCODE ct_param (command, datafmt, data, datalen,
                indicator);
CS_COMMAND     *command;
CS_DATAFMT     *datafmt;
CS_BYTE        *data;
CS_INT         datalen;
CS_SMALLINT    indicator;
```

Parameters

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call.

DATAFMT

(I) A structure that contains a description of the parameter. This structure is also used by ct_bind, ct_describe and cs_convert and is explained in "DATAFMT structure" on page 28.

Table 3-10 lists the fields in the *DATAFMT* structure, indicates whether or when they are used by ct_param, and contains general information about the fields.

Table 3-10 on page 146 contains specific information on how to set these fields when defining a parameter for a particular kind of command.

Note The programmer is responsible for adhering to these rules. Client-Library does not enforce them.

Table 3-10: Fields in the DATAFMT structure for ct_param

When this field	Is used in this condition	Set the field to
name	When defining parameters for all supported commands.	The name of the parameter being defined. If namelen is 0, the parameter is considered to be unnamed. Unnamed parameters are interpreted positionally. It is an error to mix named and unnamed parameters in a single command. **Note** When sending parameters to an Adaptive Server Enterprise, name must begin with the "@" symbol, which prefixes all Adaptive Server Enterprise stored procedure parameter names. When sending parameters with language requests, this must be the variable name as it appears in the language string. Transact-SQL names begin with the colon (:) symbol.
namelen	When defining parameters for all supported commands.	The length, in bytes, of name. If namelen is 0, the parameter is considered to be unnamed.
datatype	When defining parameters for all supported commands.	The datatype of the parameter value. All datatypes listed under "Datatypes" on page 32 are valid.
format	Not used (CS_FMT_UNUSED).	Not applicable.
maxlength	When defining non-fixed-length return parameters for RPCs; otherwise CS_UNUSED.	The maximum length, in bytes, of the data returned in this parameter. For character or binary data, maxlength must represent the total length of the return parameter, including any space required for special terminating bytes, with this exception: when the parameter is a VARYCHAR datatype such as the DB2 VARCHAR, maxlength does not include the length of the "LL" length specification. For Sybase-decimal and Sybase-numeric, set *maxlength* to 35. If the parameter is non-return, if datatype is fixed-length, or if the application does not need to restrict the length of return parameters, set maxlength to CS_UNUSED.
scale	Used for packed decimal, Sybase-numeric, and Sybase-decimal datatypes.	The number of digits after the decimal point.

When this field	Is used in this condition	Set the field to
precision	Used for packed decimal, Sybase-numeric, and Sybase-decimal datatypes.	The total number of digits before and after the decimal point.
status	When defining parameters for all types of commands except message commands.	The type of parameter being defined. One of the following values: • CS_INPUTVALUE - The parameter is an input parameter value for a non-return RPC parameter or a language request parameter. • CS_RETURN - The parameter is a return parameter.
count	Not used (CS_FMT_UNUSED).	Not applicable.
usertype	Only when defining a parameter that has an Adaptive Server Enterprise user-defined datatype; otherwise CS-UNUSED.	The user-defined datatype of the parameter, if any. usertype is set in addition to (not instead of) *datatype*. **Note** This field is used for datatypes defined at the server, not for Open Client user-defined datatypes.
locale	Not used (CS_FMT_UNUSED).	Zeroes.

data
Variable that contains the parameter data.

To indicate a null parameter (zeroes), assign *indicator* a value of -1.

If *indicator* is -1, *data* and *data_len* are ignored. For example, an application might pass empty parameters to a stored procedure or transaction that assigns default values to empty input parameters.

data_len
The length, in bytes, of the parameter data. For Sybase-numeric and Sybase-decimal, set *data_len* to 35.

indicator
An integer variable used to indicate an empty parameter. To indicate that a parameter is empty, assign *indicator* a value of -1. If *indicator* is -1, *data* and *data_len* are ignored.

Return value

ct_param returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.

Examples

The following code fragment illustrates the use of ct_param. It is taken from the sample program SYCTSAR6 in Appendix B, “Sample RPC Application.”

```
/*------------------------------------------------------------*/
/* Prepare the command (an RPC request)                       */
/*------------------------------------------------------------*/

     buf_len = 4;

     rc = ct_command(cmd, (long) CS_RPC_CMD, rpc_cmd,
                     buf_len, (long) CS_UNUSED);

     if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_COMMAND failed", msg_size);
        no_errors_sw = FALSE ;
        error_out (rc);
     }
/*------------------------------------------------------------*/
/*                                                            */
/* Setup a return parameter for NUM_OF_ROWS                   */
/*                                                            */
/* Describe the first parameter (NUM_OF_ROWS)                 */
/*                                                            */
/*------------------------------------------------------------*/

     strcpy (datafmt.name, "@parm1");
     datafmt.namelen      = 6;
     datafmt.datatype     = CS_INT_TYPE;
     datafmt.format       = CS_FMT_UNUSED;
     datafmt.maxlength    = CS_UNUSED;
     datafmt.status       = CS_RETURN;
     datafmt.usertype     = CS_UNUSED;

     buf_len   = sizeof(param1);
     rc = ct_param (cmd, datafmt, param1, buf_len, nullind);

     if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_PARAM CS_INT_TYPE parm1 failed",
        msg_size) ;
        no_errors_sw = FALSE ;
```

```
            error_out (rc);
         }
   /*------------------------------------------------------------*/
   /*                                                            */
   /* Describe the second parameter (DEPTNO)                     */
   /*                                                            */
   /*------------------------------------------------------------*/

         strcpy (datafmt.name, "@parm2");
         datafmt.namelen      = 6;
         datafmt.datatype     = CS_VARCHAR_TYPE;
         datafmt.format       = CS_FMT_UNUSED;
         datafmt.maxlength    = CS_UNUSED;
         datafmt.status       = CS_INPUTVALUE;
         datafmt.usertype     = CS_UNUSED;

         buf_len   = sizeof(param2);

         rc = ct_param (cmd, datafmt, param2, buf_len, nullind);

         if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_PARAM CS_VARCHAR_TYPE parm2 failed",
            msg_size)
  ;
            no_errors_sw = FALSE ;
            error_out (rc);
         }
   /*------------------------------------------------------------*/
   /* Send the command                                           */
   /*------------------------------------------------------------*/

         rc = ct_send (cmd);

         if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_SEND failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }
   }   /* end send_command */
```

Usage

Table 3-11 lists a summary of arguments for ct_parm.

Table 3-11: Summary of arguments

Command	Status value	Data, data_len value
Language request	CS_INPUTVALUE	The parameter value and length.
RPC (return parameters)	CS_RETURN	The parameter value and length.
RPC (non-return parameters)	CS_INPUTVALUE	The parameter value and length.

- An application calls ct_command to initiate a language request, RPC or message command.
- An application calls ct_param once for each parameter that is sent with the current RPC. It describes each parameter. That description is forwarded to the procedure or transaction called.
- ct_param defines parameters for the following types of commands:
  - Language requests
  - RPCs
- A language request requires input parameter values when the text of the language request contains host variables.
- Parameters must be described by ct_param in the order they are sent to the server. The first ct_param call describes the first parameter, the second ct_param call describes the second parameter, and so on, until all parameters are described and sent.

Defining arguments for language requests

- An application calls ct_param with status as CS_INPUTVALUE to define a parameter value for a language request containing variables.
- A language request can have up to 255 parameters.
- The following fields in the datafmt structure take special values when describing a parameter for a language request. These are listed in Table 3-12.

Table 3-12: Fields in DATAFMT for language request parameters with ct_param

Field	Value
name	The variable name as it appears in the language string. Transact-SQL names begin with the colon (:) character.
status	CS_INPUTVALUE
All other fields	Standard ct_param values.

Defining arguments for RPCs

- An application calls ct_param with status as CS_RETURN to define a return parameter for an RPC, and calls ct_param with status as CS_INPUTVALUE to define a non-return parameter.
- An application can call a stored procedure or transaction in two ways: (1) by sending a language request, or (2) by issuing an RPC. See “Remote procedure calls (RPCs)” on page 48 for a discussion of the differences between these techniques.
- To send an RPC, a Client-Library application performs the following steps:

  a Calls ct_command to initiate the request.

  b Calls ct_param once for each parameter that is being passed to the remote procedure.

  c Calls ct_send to send the request to the server. One ct_send forwards the RPC with all defined parameters; the application does not issue rc = ct_send separately for each parameter.
- An RPC can have up to 255 parameters.
- The following fields in the datafmt structure take special values when describing an RPC parameter. These are listed in Table 3-13.

Table 3-13: Fields in datafmt for RPC parameters with ct_param

Field	Value
name	When sending parameters to an Adaptive Server Enterprise, name must begin with the “@” symbol, which prefixes all Adaptive Server Enterprise stored procedure parameter names.
maxlength	The maximum length of data to be returned by the server. Set to CS_UNUSED if the parameter is non-return, if datatype is fixed-length, or if the application does not need to restrict the length of return parameters.
status	CS_RETURN to indicate that the parameter is a return parameter. CS_INPUTVALUE to indicate that the parameter is not a return parameter.
All other fields	Standard ct_param values.

See also

Related functions

- ct_command on page 83
- ct_send on page 165

Related topics

- “DATAFMT structure” on page 28

ct_remote_pwd

Description

Defines or clears passwords to be used for server-to-server connections.

Syntax

```
CS_RETCODE ct_remote_pwd(connection, action,
servername, srvlen,
                    passwd, pwd_len);
CS_CONNECTION   *connection;
CS_INT         action;
CS_CHAR          *servername;
CS_INT         srvnamelen;
CS_CHAR          passwd;
CS_INT         pwd_len;
```

Parameters

connection

(I) Handle for this connection. This connection handle must already be allocated with ct_con_alloc.

Remote passwords can only be defined for a connection that is not open. Passwords defined after a connection is open are ignored.

action

(I) Action to be taken by this call. *action* is an integer variable that indicates the purpose of this call. *action* can be any of the following symbolic values:

Value	Meaning
CS_SET (34)	Sets the remote password.
CS_CLEAR (35)	Clears all remote passwords specified for this connection.

servername

(I) Name of the server for which the password is being defined. This is the name by which the server is known in the Server Path Table.

If *action* is CS_CLEAR, set *servername* to zeroes.

If *servername* is blank, the specified password is considered a “universal” password, to be used with any server that does not have a password explicitly specified for it.

srvnamelen

(I) Length, in bytes, of *servername*. To use the default “universal” password, assign CS_NULL_STRING to this argument.

passwd

(I) Password being installed for remote logins to the server named in *servername*.

If *action* is CS_CLEAR, set *passwd* to zeroes, and the password defaults to the one set for this connection in ct_con_props, if any.

passwd_len
(I) Length, in bytes, of *passwd*.

Return value

ct_remote_pwd returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	Results are available for processing.
CS_FAIL (-2)	The routine failed.
TDS_INVALID_PARAMETER (-4)	One or more of the ct_remote_pwd arguments contains an illegal value. Likely causes for this code are: • Erroneous value for *action*. *action* cannot be CS_GET for ct_remote_pwd. • Erroneous value for a length argument. Length values cannot be negative numbers.
TDS_SOS (-257)	Memory shortage. The operation failed.

Usage

- ct_remote_pwd defines the password that a server uses when logging into another server. An application can call ct_remote_pwd to clear remote passwords for a connection at any time.
- A Transact-SQL language command or stored procedure or transaction running on one server can call a stored procedure or transaction located on another server. To accomplish this server-to-server communication, the first server, to which an application connects through ct_connect, actually logs into the second, remote server, performing a server-to-server remote procedure call.

  ct_remote_pwd allows an application to specify the password to use when the first server logs into the remote server.
- Multiple passwords can be specified, one for each server that the first server might need to log into. Each password must be defined with a separate call to ct_remote_pwd.
- If an application does not specify a remote password for a particular server, the password defaults to the password set for this connection through ct_con_props, if any. If a password is not defined, the password is set to zeroes. If an application user generally has the same password on different servers, this default behavior can be sufficient.

- Remote passwords are stored in an internal buffer, which is only 255 bytes long. Each password entry in the buffer consists of the password itself, the associated server name, and two extra bytes. If the addition of a password to this buffer would cause overflow, ct_remote_pwd returns CS_FAIL and generates a Client-Library error message that indicates the problem.
- Define remote passwords before calling ct_connect to create an active connection. It is an error to call ct_remote_pwd to define a remote password for a connection that is already open.

See also

Related functions

- ct_connect on page 99
- ct_con_props on page 104

ct_res_info

Description

Returns result set information.

Syntax

```
CS_RETCODE ct_res_info(command, result_type, buffer,
                       buf_len, outlen);
CS_COMMAND    *command;
CS_INT        result_type;
CS_BYTE       *buffer;
CS_INT        buf_len;
CS_INT        *outlen;
```

Parameters

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call.

result_type

(I) Type of information to return. Assign this argument one of the following values:

Value	Meaning
CS_ROW_COUNT (800)	The number of rows affected by the current command.
CS_CMD_NUMBER (801)	The number of the command that generated the current result set.
CS_NUMDATA (803)	The number of items in the current result set.

buffer
(O) Variable ("buffer") where ct_res_info returns the requested information. At present, this is always an integer value.

buf_len
(I) Length, in bytes, of the buffer.

If the returned value is longer than *buf_len*, ct_res_info sets *outlen* to the length of the requested information and returns CS_FAIL.

outlen
(O) Length, in bytes, of the retrieved information. *outlen* is an integer variable where ct_res_info returns the length of the information being retrieved.

If the retrieved information is larger than *buf_len* bytes, an application uses the value of *outlen* to determine how many bytes are needed to hold the information.

Return value

ct_res_info returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	Results are available for processing.
CS_FAIL (-2)	The routine failed. ct_res_info returns CS_FAIL if the requested information is larger than *buf_len* bytes.

Examples

The following code fragment demonstrates the use of ct_res_info. It is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
/********************************************************************/
/*                                                                  */
/* Subroutine to process result rows                                */
/*                                                                  */
/********************************************************************/
void   result_row_processing ()
{
CS_INT        rc;
CS_INT        col_len;
CS_INT        &numcol;
CS_INT        parm_cnt;
char          msg1[40] = "The maximum number of connections is   ";
char          msg2[25] = "The number of columns is ";
char          wrk_str [4];
char          period = '.';
/*------------------------------------------------------------*/
```

```
    /* We need to bind the data to program variables. We don't    */
    /* care about the indicator variable so we'll pass NULL for   */
    /* that parameter in OC_BIND().                               */
    /*------------------------------------------------------------*/
  rc = ct_res_info(cmd,CS_NUMDATA,&&numcol,
           sizeof(&numcol),&col_len);
  if (rc != CS_SUCCEED)
           {
              strncpy (msgstr, "CT_RES_INFO failed", msg_size);
              no_errors_sw = FALSE ;
              error_out (rc);
           }
    /*------------------------------------------------------------*/
    /* display the number of connections                          */
    /*------------------------------------------------------------*/
            row_num = row_num + 1;
            strncpy (RS[row_num].rsltno, msg1, msg_size);
  cvtleft  = 4;           /* Digits to the left */
            cvtright = 0;           /* Digits to the right */
            SYCVTD(maxconnect, wrk_str,
                    cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
  strncat (RS[row_num].rsltno, wrk_str, 4);
            row_num = row_num + 2;
    /*------------------------------------------------------------*/
    /* display the number of columns                              */
    /*------------------------------------------------------------*/
  strncpy (RS[row_num].rsltno, msg2, sizeof(msg2));
            cvtleft  = 4;           /* Digits to the left */
            cvtright = 0;           /* Digits to the right */
            SYCVTD(&numcol, wrk_str,
                    cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
  strncat (RS[row_num].rsltno, wrk_str, 4);
            row_num = row_num + 2;
  if (&numcol != 2)
            {
              strncpy (msgstr, "CT_RES_INFO returned wrong # of parms",
                       msg_size);
              no_errors_sw = FALSE ;
              error_out (rc);
            }
```

Usage

The following arguments listed in Table 3-14 are returned to *result_type* after ct_results indicates that results are present.

Table 3-14: Summary of arguments

RESULT_TYP	ct_res_info returns	Buffer value
CS_ROW_COUNT (800)	The number of rows affected by the current command.	An integer value.
CS_CMD_NUMBER (801)	The number of the command that generated the current result set.	An integer value.
CS_NUMDATA (803)	The number of items in the current result set.	An integer value.

- ct_res_info returns information about the current result set or the current command. The current command is defined as the request that generated the current result set.
- A result set is a collection of a single type of result data. Result sets are generated by requests. For more information on result sets, see ct_results on page 160 and "Results" on page 51.

Retrieving the command number for the current result set

- To determine the number of the command that generated the current result set, call ct_res_info with *result_type* as CS_CMD_NUMBER.
- Client-Library keeps track of the command number by counting the number of times ct_results returns CS_CMD_DONE.

  An application's first call to ct_results following a ct_send call sets the command number to 1. The command number remains 1 until ct_results returns CS_CMD_DONE. The next time the application calls ct_results, the command number is incremented to 2. The command number continues to be incremented each time ct_results is called after returning CS_CMD_DONE.
- CS_CMD_NUMBER is useful in the following cases:
  - To identify the SQL command within a language request that generated the current result set.
  - To identify the select command in a stored procedure or transaction that generated the current result set.
- A language request contains a string of text. This text represents one or more SQL commands or other language request statements. If the application is sending a language request, "command number" refers to the number of the statement in the language request.

  For example, the following Transact-SQL string represents three Transact-SQL commands—two select statements and one insert:

```
select * from authors
select * from titles
insert newauthors
select * from authors
where city = "San Francisco"
```

The two select statements can generate result sets. In this case, the command number that ct_res_info returns can be from 1 to 3, depending on when ct_res_info is called.

Note When you send SQL strings to DB2, remember to use semicolons (;) to separate SQL statements.

- Inside stored procedure or transactions, only select statements cause the command number to be incremented. If a stored procedure or transaction contains seven SQL commands, three of which are select statements, the command number that ct_res_info returns can be any integer from 1 to 3, depending on which select statement generated the current result set.

Retrieving the number of result data items

- To determine the number of result data items in the current result set, call ct_res_info with *result_type* as CS_NUMDATA.
- Result sets contain result data items. Row result sets contain columns, a parameter result set contains parameters, and a status result set contains a status. The columns, parameters, and status are known as result data items.

Retrieving the number of rows for the current command

- To determine the number of rows affected by the current command, call ct_res_info with *result_type* as CS_ROW_COUNT.
- If the current command is one that does not return rows—for example, a language command containing an insert statement—an application can get the row count immediately after ct_results returns CS_CMD_SUCCEED.
- If the current command does return rows:
  - An application can get a total row count after processing all of the rows.
  - An application can get an intermediate row count any time after ct_results indicates that results are available. An intermediate row count is the number of rows fetched so far.

- If the command is one that executes a stored procedure or transaction, for example a Transact-SQL exec language command or a remote procedure call, ct_res_info returns either the number of rows returned by the latest select statement executed by the stored procedure or transaction, or CS_NO_COUNT if the stored procedure or transaction does not execute any select statements. A stored procedure or transaction that does not contain a select statement can execute a select by calling another stored procedure or transaction that contains a select.
- ct_res_info returns CS_NO_COUNT if any of the following are true:
  - The SQL command fails for any reason, such as a syntax error.
  - The command is one that *never* affects rows, such as a Transact-SQL print command.
  - The command executes a stored procedure or transaction that does not execute any select statements.

See also

Related functions

- ct_cmd_props on page 81
- ct_con_props on page 104
- ct_results on page 160

Related topics

- “Results” on page 51

ct_results

Description

Sets up result data to be processed.

Syntax

```
CS_RETCODE    ct_results(command, result_type);
CS_COMMAND    *command;
CS_INT        *result_type;
```

Parameters

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call.

result_type

(O) Variable containing the result type. ct_results returns to this variable a symbolic value that indicates the type of result returned by the current request. The result type can be any of the following symbolic values listed in Table 3-15.

Table 3-15: Values for result_type (ct_results)

Value	Meaning	Result produced
CS_ROW_RESULT (4040)	Regular row results arrived.	One or more rows of tabular data.
CS_PARAM_RESULT (4042)	Return parameter results arrived.	A single row of return parameters.
CS_STATUS_RESULT (4043)	Stored procedure return status results arrived.	A single row containing a single status.
CS_CMD_DONE (4046)	The results of the request processed completely.	Not applicable.
CS_CMD_SUCCEED (4047)	A request that returns no data, such as a language request containing an insert statement, processed successfully.	No results.
CS_CMD_FAIL (4048)	The server encountered an error while executing the request. This value can indicate that the connection failed or was terminated.	No results.

Return value

ct_results returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	A result set is available for processing.
CS_END_RESULTS (-205)	No more result sets are available for processing.
CS_FAIL (-2)	The routine failed.
CS_CANCELLED (-202)	Results were canceled.

Examples

The following code fragment demonstrates how ct_results can describe a result row for a language request. It is taken from the sample program SYCTSAA6 in Appendix A, "Sample Language Application."

```
/******************************************************************/
/*                                                                */
/* Subroutine to process the result                               */
/*                                                                */
/******************************************************************/
void   proces_results ()

{
CS_INT        rc;

/*------------------------------------------------------------*/
/* Set up the results data                                    */
/*------------------------------------------------------------*/

       rc = ct_results (cmd, &results_type);

/*------------------------------------------------------------*/
/* Determine the outcome of the comand execution              */
/*------------------------------------------------------------*/

       switch (rc)
       {
          case CS_SUCCEED:

/*--------------------------------------------------------------*/
/* Determine the type of result returned by the current request */
/*--------------------------------------------------------------*/

               switch (results_type)
               {

/*------------------------------------------------------------*/
/* Process row results                                        */
/*------------------------------------------------------------*/

                  case CS_ROW_RESULT:
                       result_row_processing ();
                       fetch_row_processing ();
                       break;
/*------------------------------------------------------------*/
/* Process parameter results --- there should be no parameter */
/* to process                                                 */
/*------------------------------------------------------------*/
```

```
                  case CS_PARAM_RESULT:
                       no_more_results = FALSE;
                       break;

/*---------------------------------------------------------------*/
/* process status results --- the stored procedure status      */
/* result will not be processed in this example                */
/*---------------------------------------------------------------*/
                  case CS_STATUS_RESULT:
                       no_more_results = FALSE;
                       break;
/*---------------------------------------------------------------*/
/* print an error message if the server encountered an error   */
/* while executing the request                                 */
/*---------------------------------------------------------------*/

                  case CS_CMD_FAIL:
                       no_errors_sw = FALSE ;
                       strncpy (msgstr,
                         "CT_RESUL returned CS_CMD-FAIL restype",
                       msg_size);
                       error_out (rc);
                       break;
/*---------------------------------------------------------------*/
/* print a message for successful commands that returned no    */
/* data( optional )                                            */
/*---------------------------------------------------------------*/
                  case CS_CMD_SUCCEED:
                       strncpy (msgstr,
                         "CT_RESUL returned CS_CMD_SUCCEED restype",
                       msg_size);
                       break;
/*---------------------------------------------------------------*/
/* print a message for requests that have been processed       */
/* successfully( optional )                                    */
/*---------------------------------------------------------------*/
                  case CS_CMD_DONE:
                       strncpy (msgstr,
                       "CT_RESUL returned CS_CMD_DONE restype",
                       msg_size);
                       break;

                  default:
                       no_more_results = TRUE ;
                       no_errors_sw    = FALSE ;
                       strncpy (msgstr,
```

```
                            "CT_RESUL returned UNKNOWN restype",
                          msg_size);
                          error_out (rc);
                          break;

                  }   /* end of switch (result_type) */
                  break;   /* case of CS_SUCCEED  */
/*------------------------------------------------------------*/
/* print an error message if the CTBRESULTS call failed       */
/*------------------------------------------------------------*/
             case CS_FAIL:
                  no_more_results = TRUE ;
                  no_errors_sw    = FALSE ;
                  strncpy (msgstr, "CT_RESULTS returned CS_FAIL ret_code",
                           msg_size);
                  error_out (rc);
                  break;
/*------------------------------------------------------------*/
/* drop out of the results loop if no more result sets are    */
/* available for processing or if the results were cancelled  */
/*------------------------------------------------------------*/
             case CS_END_RESULTS:
             case CS_CANCELLED:
                  no_more_results = TRUE ;
                  break;

             default:
                  no_more_results = TRUE ;
                  no_errors_sw    = FALSE ;
                  strncpy (msgstr, "CT_RESULTS returned unknown ret_code",
                           msg_size);
                  error_out (rc);
                  break;

        }  /* end of switch (rc) */

}   /* end process_results */
```

Usage

- ct_results tells the application what kind of results returned and sets up result data for processing. An application calls ct_results after sending a request to the server through ct_send and before binding and retrieving the results of that request (if any) with ct_bind and ct_fetch.
- “Result data” is an umbrella term for all the types of data that a server can return to an application:

- Regular rows
- Return parameters
- Stored procedure return status

  ct_results sets up all of these types of results for processing.

- Result data is returned to an application in the form of result sets. A result set includes only a single type of result data. For example, a regular row result set contains only regular rows, and a return parameter result set contains only return parameters.

The *ct_results* loop

- Because a request can generate multiple result sets, an application must call ct_results as long as it continues to return CS_SUCCEED, indicating that results are available. The simplest way to do this is in a loop that terminates when ct_results fails to return CS_SUCCEED. After the loop, an application can test the ct_results final return code to determine why the loop terminated.
- Results are returned to an application in the order in which they are produced. However, this order is not always easy to predict. For example, when an application calls a stored procedure or transaction that in turn calls another stored procedure or transaction, the application might receive a number of row result sets, as well as a return parameter and a return status result set. The order in which these results are returned depends on how the called stored procedure or transaction is written.

  The *result_type* argument indicates what type of result data the result set contains.

When are the results of a command completely processed?

- ct_results sets the result type to CS_CMD_DONE to indicate that the results of a logical command processed completely.

  A logical command is defined as any command defined through ct_command, with the following rules:

  - Each Transact-SQL select statement inside a stored procedure is a logical command. Other Transact-SQL statements inside stored procedures do not count as logical commands.
  - Each Transact-SQL statement in a language request is a logical command.

- A result type of CS_CMD_SUCCEED or CS_CMD_FAIL is immediately followed by a result type of CS_CMD_DONE.

Canceling results

- To cancel remaining results from a request (and eliminate the need to continue calling ct_results until it fails to return CS_SUCCEED), call ct_cancel.

ct_results and stored procedures

- A run-time error on a language request that contains an execute statement returns CS_CMD_FAIL. However, a run-time error on a statement inside a stored procedure or transaction does not return CS_CMD_FAIL. For example, if a called stored procedure or transaction contains an insert statement and the user does not have insert permission on the database table, the insert statement fails, but ct_results still returns CS_SUCCEED.

  To check for run-time errors inside stored procedures or transactions, examine the procedure return status. This value returns as a return status result set immediately following the row and parameter results, if any. If the error generates a server message, the message is also available to the application.

  If results are coming from Open ServerConnect, a return status of TDS_DONE_ERROR indicates an error.

See also

Related functions

- ct_bind on page 61
- ct_command on page 83
- ct_describe on page 109
- ct_fetch on page 134
- ct_send on page 165

Related topics

- “Remote procedure calls (RPCs)” on page 48
- “Results” on page 51

ct_send

Description

Sends a request to the server.

Syntax

```
CS_RETCODE ct_send (command);
CS_COMMAND    *command;
```

Parameters

command

(I) Handle for this client/server operation. This handle is defined in the associated ct_cmd_alloc call.

Return value

ct_send returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed. This result can indicate that SNA sessions cannot be established.
CS_CANCELLED (-202)	The routine was canceled. **Note** This value is returned by SNA sessions only, and is never returned when sending a request to another CICS region.

Examples

Example 1

The following code fragment demonstrates the use of ct_send. It is taken from the sample program SYCTSAR6 in Appendix B, “Sample RPC Application.”

```
/*------------------------------------------------------------*/
/* Prepare the command (an RPC request)                       */
/*------------------------------------------------------------*/
     buf_len = 4;
     rc = ct_command(cmd, (long) CS_RPC_CMD, rpc_cmd,
                     buf_len, (long) CS_UNUSED);
     if (rc != CS_SUCCEED)
     {
        strncpy (msgstr, "CT_COMMAND failed", msg_size);
        no_errors_sw = FALSE ;
        error_out (rc);
     }
/*--------------------------------------------------------------*/
/*                                                              */
/* Setup a return parameter for NUM_OF_ROWS                     */
/* Describe the first parameter (NUM_OF_ROWS)                   */
/*                                                              */
/*--------------------------------------------------------------*/
       strcpy (datafmt.name, "@parm1");
       datafmt.namelen      = 6;
       datafmt.datatype     = CS_INT_TYPE;
       datafmt.format       = CS_FMT_UNUSED;
```

```
        datafmt.maxlength    = CS_UNUSED;
        datafmt.status       = CS_RETURN;
        datafmt.usertype     = CS_UNUSED;

        buf_len   = sizeof(param1);
        rc = ct_param (cmd, datafmt, param1, buf_len, nullind);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_PARAM CS_INT_TYPE parm1 failed",
           msg_size) ;
           no_errors_sw = FALSE ;
           error_out (rc);
        }
 /*-------------------------------------------------------------*/
 /*                                                             */
 /* Describe the second parameter (DEPTNO)                      */
 /*                                                             */
 /*-------------------------------------------------------------*/
        strcpy (datafmt.name, "@parm2");
        datafmt.namelen      = 6;
        datafmt.datatype     = CS_VARCHAR_TYPE;
        datafmt.format       = CS_FMT_UNUSED;
        datafmt.maxlength    = CS_UNUSED;
        datafmt.status       = CS_INPUTVALUE;
        datafmt.usertype     = CS_UNUSED;

        buf_len   = sizeof(param2);

        rc = ct_param (cmd, datafmt, param2, buf_len, nullind);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_PARAM CS_VARCHAR_TYPE parm2 failed",
           msg_size)
;
           no_errors_sw = FALSE ;
           error_out (rc);
        }
 /*-------------------------------------------------------------*/
 /* Send the command                                            */
 /*-------------------------------------------------------------*/
        rc = ct_send (cmd);

        if (rc != CS_SUCCEED)
        {
```

```
            strncpy (msgstr, "CT_SEND failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }
}   /* end send_command */
```

Example 2

The following code fragment demonstrates the use of ct_send. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
  /*------------------------------------------------------------*/
  /* Open connection to the server or CICS region              */
  /*------------------------------------------------------------*/

       rc = ct_connect (connection, servname, server_size);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CONNECT failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
       }
/*------------------------------------------------------------*/
/* Invokes SEND_COMMAND routine                               */
/*------------------------------------------------------------*/
       if (no_errors_sw)
          send_command ();
/*------------------------------------------------------------*/
/* Process the results of the command                         */
/*------------------------------------------------------------*/
       if (no_errors_sw)
       {
          while (no_more_results == FALSE)
             proces_results ();
       }
}   /* end proces_input */
/******************************************************************/
/*                                                                */
/* Subroutine to allocate, send, and process commands             */
/*                                                                */
/******************************************************************/
void send_command ()

{
 CS_INT        rc;
```

```
 CS_INT        *outlen;
 CS_INT        buf_len;
 CS_CHAR       sql_cmd[45];

/*------------------------------------------------------------*/
/* Find out what the maximum number of connections is         */
/*------------------------------------------------------------*/

      rc = ct_config (context, CS_GET, CS_MAX_CONNECT,
                      &maxconnect, CS_FALSE, outlen);

      if (rc != CS_SUCCEED)
      {
         strncpy (msgstr, "CT_CONFIG failed", msg_size);
         strncpy (msgtext2, "Please press return to
         continue!", text_size);
         error_out(rc);

         /* reset program flags to move on with the task */
         print_once = TRUE;
         diag_msgs_initialized = TRUE;
         strncpy(msgtext2, "Press Clear To Exit", text_size);
      }

/*-------------------------------------------------------------*/
/* Allocate a command handle                                   */
/*-------------------------------------------------------------*/
      rc = ct_cmd_alloc (connection, &cmd);

      if (rc != CS_SUCCEED)
      {
         strncpy (msgstr, "CT_CMDALLOC failed", msg_size);
         no_errors_sw = FALSE ;
         error_out(rc);
      }
/*------------------------------------------------------------*/
/* Prepare the language request                               */
/*------------------------------------------------------------*/
      strcpy(sql_cmd,
              "SELECT FIRSTNME, EDUCLVL FROM SYBASE.SAMPLETB");
      buf_len = sizeof(sql_cmd);
      rc = ct_command(cmd, (long) CS_LANG_CMD, sql_cmd,
                      buf_len, (long) CS_UNUSED);

      if (rc != CS_SUCCEED)
      {
```

```
            strncpy (msgstr, "CT_COMMAND failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }
/*------------------------------------------------------------*/
/* Send the language request                                  */
/*------------------------------------------------------------*/
         rc = ct_send (cmd);

         if (rc != CS_SUCCEED)
         {
            strcpy (msgstr, "CT_SEND failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }
}    /* end send_command */
```

Usage

- ct_send signals the end of the data to be sent to a server (no more parameters, data, messages) and sends a request to the server.
- Sending a request to a server is a three-step process. To send a request to a server, an application:
  - Initiates the request by calling ct_command, which initiates a language request, RPC, or message stream to send to the server.
  - Describes parameters for the request, using ct_param.

    Not all requests require parameters. For example, an RPC may or may not require parameters, depending on the stored procedure or transaction being called.
  - Calls ct_send to send the request stream to the server.
- ct_send does not wait for a response from the server. An application must call ct_results to verify the success of the request and to set up the results for processing.

See also

Related functions

- ct_command on page 83
- ct_fetch on page 134
- ct_param on page 145

cs_config

Description

Sets or retrieves context structure properties.

Syntax

```
CS_RETCODE cs_config (context, action,property,
                  buffer, buf_len, outlen);
CS_CONTEXT    *context;
CS_INT        action;
CS_INT        property;
CS_BYTE       *buffer;
CS_INT        buf_len;
CS_INT        *outlen;
```

Parameters

context

(I) A context structure for which the properties are being set or retrieved. The context structure is defined in the program call cs_ctx_alloc.

action

(I) Action this call takes. *action* is an integer variable that indicates the purpose of this call. Assign *action* one of the following symbolic values:

Value	Meaning
CS_GET (33)	Retrieves the value of the property.
CS_SET (34)	Sets the value of the property.
CS_CLEAR (35)	Clears the value of the property by resetting the property to its default value.

property

(I) Symbolic name of the property for which the value is being set or retrieved. Client-Library properties are listed under "Remote procedure calls (RPCs)" on page 48, with descriptions, possible values, and defaults.

Table 3-16 lists the properties that can be set or retrieved by cs_config.

Table 3-16: Values for property (cs_config)

Application action	Property	Indicates
Set, retrieve, or clear	CS_EXTRA_INF	Whether to return the extra information required when processing messages in line, using the SQLCA or SQLCODE structures.
Retrieve only	CS_VERSION	The version number of Open Client currently in use.

buffer

(I/O) Variable (buffer) that contains the specified property value.

If *action* is CS_SET, cs_config takes the value from this buffer.

If *action* is CS_GET, cs_config returns the requested information to this buffer.

If *action* is CS_CLEAR, this value is zeroes.

This argument is typically one of the following datatypes:

```
CS_INT buffer;
CS_CHAR buffer[n];
```

buf_len

(I) Length, in bytes, of the buffer.

If *action* is CS_SET and the value in the buffer is a fixed-length or symbolic value, *buf_len* should have a value of CS_UNUSED.

If *action* is CS_GET and *buffer* is too small to hold the requested information, cs_config sets *outlen* to the length of the requested information and returns CS_FAIL. To retrieve all the requested information, change the value of *buf_len* to the length returned in *outlen* and rerun the application.

If *action* is CS_CLEAR, this value is zeroes.

outlen

(O) Length, in bytes, of the retrieved information. *outlen* is an integer variable where cs_config returns the length of the property value being retrieved.

When the retrieved information is larger than *buf_len* bytes, an application uses the value of *outlen* to determine how many bytes are needed to hold the information.

outlen is used only when *action* is CS_GET. When *action* is CS_SET or CS_CLEAR, this value is zeroes.

Return value

cs_config returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed.
TDS_INVALID_PARAMETER (-4)	One of the cs_config arguments contains an illegal value.

Usage

Note cs_config and ct_config both set and retrieve context properties. cs_config is used with global context properties; ct_config is used with Client-Library properties.

- cs_config can be used to set and retrieve the value of CS_EXTRA_INF and to retrieve the version number of Open Client currently in use.
- Use ct_config to set and retrieve the values of Client-Library-specific context properties. Properties set through ct_config affect only Client-Library behaviors.

Global Context Properties

- *Extra information*
  - CS_EXTRA_INF determines whether or not Client-Library returns the extra information that is required to fill in a SQLCA or SQLCODE structure.
  - If an application is not retrieving messages into a SQLCA or SQLCODE structure, the extra information is returned as ordinary Client-Library messages.
- *Version level*
  - The CS_VERSION property represents the version of Client-Library behavior that an application requests through cs_ctx_alloc.
  - An application can only retrieve the value of CS_VERSION; it cannot assign a value to CS_VERSION.
- *User data*
  - The CS_USERDATA property defines user-allocated data. This property allows an application to associate user data with a particular connection or command structure. An application allocates a data space from which it can get this data when needed.

See also

Related functions

- cs_ctx_alloc on page 179
- ct_con_props on page 104
- ct_config on page 96
- ct_init on page 141

cs_convert

Description

Converts a data value from one datatype to another.

Syntax

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

Parameters

context

(I) A context structure. The context structure is defined in the program call cs_ctx_alloc.

srcfmt

(I) A structure that describes the variable(s) that contain the source data. cs_convert ignores *srcfmt* fields that it does not use.

Table 3-17 lists the fields in the *srcfmt* structure and indicates whether and how they are used by cs_convert. For a general discussion of this structure, see "DATAFMT structure" on page 28.

Table 3-17: Fields in the srcfmt structure for cs_convert

Field	When used	Value represents
name	Not used (CS_FMT_UNUSED).	Not applicable.
namelen	Not used (CS_FMT_UNUSED).	Not applicable.
type	For all datatype conversions.	The datatype of the source data. cs_convert converts this datatype to the datatype specified for the destination variable (*dest_type*).
format	Not used (CS_FMT_UNUSED).	Not applicable.
maxlen	When converting non-fixed-length source datatypes to any destination type. src_maxlen is ignored when converting fixed-length types.	The length of the source variable, in bytes. If srcdata is an array, src_maxlen is the length of an element in the array. When converting character or binary datatypes, src_maxlen must describe the total length of the source variable, including any space required for special terminating bytes, with this exception: when converting a VARYCHAR-type source such as the DB2 VARCHAR, src_maxlen does not include the length of the "LL" length specification. In case of Sybase-numeric, Sybase-decimal or packed decimal this value is the actual length.

Field	When used	Value represents
scale	Only when converting to or from numeric, Sybase-decimal, or packed-decimal	Number of digits that follow the decimal point in the source data. src_scale must be less than or equal to src_precis and cannot be greater than 31.
precis	Only when converting to or from packed- decimal, numeric and Sybase-decimal datatypes.	The total number of digits in the source data. src_precis must be greater than or equal to src_scale and cannot be less than 1 or greater than 31.
status	Not used (CS_FMT_UNUSED).	Not applicable.
count	Not used (CS_FMT_UNUSED).	Not applicable.
usertype	Not used (CS_FMT_UNUSED).	Not applicable.
locale	Not used (CS_FMT_UNUSED).	Not applicable.

srcdata
(I) Name of the source variable that contains the data to be converted. This is the variable described in the *srcfmt* structure.

destfmt
(I) A structure that contains a description of the variable(s) that contain destination (converted) data. cs_convert ignores *destfmt* fields that it does not use.

Table 3-18 lists the fields in the *destfmt* structure and indicates whether and how they are used by cs_convert. For a general discussion of this structure, see "DATAFMT structure" on page 28.

Table 3-18: Fields in the datafmt structure for cs_convert

Field	When used	Value represents
name	Not used (CS_FMT_UNUSED).	Not applicable.
namelen	Not used (CS_FMT_UNUSED).	Not applicable.
type	For all datatype conversions.	The datatype of the destination variable. cs_convert converts the datatype specified for the source data (srctype) to this datatype.
format	Not used (CS_FMT_UNUSED).	Not applicable.

Field	When used	Value represents
maxlen	When converting all source datatypes to non-fixed-length datatypes. dest_maxlen is ignored when converting to fixed-length datatypes.	The length of the destination variable, in bytes. If destdata is an array, dest_maxlen is the length of an element in the array. When converting character or binary datatypes, dest_maxlen must describe the total length of the destination variable, including any space required for special terminating bytes, with this exception: when converting to a VARYCHAR-type destination such as the DB2 VARCHAR, dest_maxlen does not include the length of the "LL" length specification. dest_maxlen = 35 when converting to numeric or Sybase-decimal.
scale	Only when converting to or from numeric, Sybase-decimal, or packed decimal datatypes.	Number of digits that follow the decimal point in the destination variable. dest_scale must be less than or equal to dest_precis and cannot be greater than 31. Use the same value as in src_scale.
precis	Only when converting to or from numeric, Sybase-decimal, or packed decimal datatypes.	The total number of digits in the destination data. dest_precis must be greater than or equal to dest_scale and cannot be less than 1 or greater than 31. Use the same value as in src_precis
status	Not used (CS_FMT_UNUSED).	Not applicable.
count	Not used (CS_FMT_UNUSED).	Not applicable.
usertype	Not used (CS_FMT_UNUSED).	Not applicable.
locale	Not used (CS_FMT_UNUSED).	Not applicable.

destdata

(O) Name of the variable that contains the converted data. This is the variable described in the *destdata* structure.

outlen

(O) Actual length, in bytes, of the data placed in *destdata*. If the conversion fails, cs_convert sets *outlen* to CS_UNUSED.

Return value

cs_convert returns one of the following values listed in Table 3-19 on page 176.

Table 3-19: Values returned

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.

Value	Meaning
CS_FAIL (-2)	The routine failed.
TDS_INVALID_DATAFMT_VALUE (-181)	A srcfmt or destfmt field contains an illegal value—probably an illegal datatype value.
TDS_INVALID_VAR_ADDRESS (-175)	This value cannot be NULL.
TDS_MONEY_CONVERSION_ERROR (-22)	Converting TDSMONEY4 failed, possibly because the TDS version is not 4.2 or above.
TDS_INVALID_DATA_CONVERSION (-172)	This value cannot be NULL.
TDS_INVALID_LENGTH(-173)	Converting TDSMONEY4 failed, possibly because the TDS version is not 4.2 or above.

Examples

The following code fragment illustrates the use of cs_convert to convert a column in result rows. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/**********************************************************************/
/*                                                                    */
/* Subroutine to fetch row processing                                 */
/*                                                                    */
/**********************************************************************/
void   fetch_row_processing ()

{
CS_INT          rows_read;
CS_INT          rc;
CS_INT          col_len;
CS_INT          max_screen_rows = 10;
CS_SMALLINT     nomore_rows     = 0;
CS_DATAFMT      datafmt;
CS_DATAFMT      datafmt2;
struct {
        char  first[12];
        char  space2[2];
        char  edlevel[4];
       } output_row;


    while (nomore_rows == FALSE)
    {
      strcpy(col_firstnme.str, "            ");
      memset (&output_row, ' ', sizeof(output_row));

      rc = ct_fetch (cmd, (long) CS_UNUSED,
                     (long) CS_UNUSED,
                     (long) CS_UNUSED,
```

```
                        &rows_read);

            switch (rc)
            {
                case  CS_SUCCEED:
                    nomore_rows         = FALSE ;
                    datafmt.datatype      = CS_VARCHAR_TYPE;
                    datafmt.maxlength     = sizeof(col_firstnme);
                    datafmt2.datatype     = CS_CHAR_TYPE;
                    datafmt2.maxlength    = 12;

      /*------------------------------------------------------------*/
      /* convert the first column from VARCHAR to CHAR for display  */
      /*------------------------------------------------------------*/
                    if (cs_convert(context, datafmt, col_firstnme, datafmt2,
                                   &output_row.first, &col_len) !=
     CS_SUCCEED)
                    {
                       strncpy (msgstr,
                         "CS_CONVERT CS_CHAR_TYPE failed", msg_size);
                       no_errors_sw = FALSE ;
                       error_out(rc);
                    }
```

Usage

- A client application can use this function to convert the datatype of RPC return parameters to the datatype of the target server, and to convert the datatype of a retrieved value to a datatype that can be used by Open ClientConnect. This function converts a single variable each time it executes.
- If converting columns, an application must issue a separate cs_convert call for each column to be converted. If several rows of data need converting, the application must issue a separate cs_convert call for every column that needs conversion in each row.
- Table 3-20 on page 178 lists the conversions you can perform with cs_convert.

Table 3-20: Conversions performed by cs_convert

Source type	Result type	Notes
CS_VARCHAR	CS_CHAR	Does EBCDIC to ASCII conversion; pads with blanks.
CS_CHAR	CS_VARCHAR	
CS_MONEY	CS_CHAR	
CS_MONEY	CS_VARCHAR	
CS_REAL	CS_FLOAT	Truncates low order digits.

Source type	Result type	Notes
CS_MONEY	CS_FLOAT	
CS_PACKED370	CS_FLOAT	
CS_FLOAT	CS_REAL	Pads with zeroes.
CS_MONEY	CS_PACKED370	
CS_CHAR	CS_PACKED370	
CS_VARCHAR	CS_PACKED370	
CS_FLOAT	CS_PACKED370	
CS_NUMERIC	CS_PACKED370	
CS_DECIMAL	CS_PACKED370	
CS_NUMERIC	CS_CHAR	
CS_DECIMAL	CS_CHAR	
CS_DATETIME	CS_CHAR	
CS_CHAR	CS_NUMERIC	
CS_CHAR	CS_DECIMAL	
CS_PACKED370	CS_NUMERIC	
CS_PACKED370	CS_DECIMAL	
CS_PACKED370	CS_CHAR	

Warning! Converting CS_MONEY or CS_CHAR values to CS_FLOAT can result in a loss of precision. Converting a CS_FLOAT value to a character type can also result in a loss of precision.

See also

Related functions

- ct_fetch on page 134

Related topics

- “Datatypes” on page 32
- “DATAFMT structure” on page 28

cs_ctx_alloc

Description

Allocates a context structure.

Syntax

```
CS_RETCODE cs_ctx_alloc(version, context);
CS_INT     version;
CS_CONTEXT **context;
```

Parameters

version

(I) Version of Client-Library behavior that the application expects. Table 3-21 lists the symbolic values that are legal for *version*.

Table 3-21: Values of version with cs_ctx_alloc

Value	Indicates	Supported features
CS_VERSION_46	Communicates with SQL Server release 4.6.	RPCs. **Note** This is the initial version of Client-Library.
CS_VERSION_50	Communicates with SQL Server release 10.0 and above.	RPCs.

context

(O) Variable where a pointer to this newly-allocated context structure is returned. This is the name used by ct_init when it initializes Client-Library.

Return value

cs_ctx_alloc returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed. The most common cause for a cs_ctx_alloc failure is a lack of available memory.

Examples

The following code fragment demonstrates how cs_ctx_alloc works with other functions to initialize a program. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/*------------------------------------------------------------------*/
/* program initialization                                           */
/*------------------------------------------------------------------*/
     memset (msgtext1, ' ', text_size);
     strncpy (msgtext2, "Press Clear To Exit", text_size);
     page_cnt  = page_cnt + 1;
     memset(servname, ' ', 30);
     memset(username, ' ', 8);
     memset(tran, ' ', 8);
     memset(pwd, ' ', 8);
     memset(driver, ' ', 9);

     for (i = 0; i < 14; ++i)
        memset (RS[i].rsltno, ' ', text_size); /* init output
```

```
lines  */

        /*  get system time   */
        EXEC CICS ASKTIME ABSTIME(UTIME);

        EXEC CICS FORMATTIME
                  ABSTIME(UTIME)
                  DATESEP('/')
                  MMDDYY(TMP_DATE)
                  TIME(TMP_TIME)
                  TIMESEP ;

        display_initial_screen ();
        get_input_data ();

 /*------------------------------------------------------------------*/
 /* Allocate a context structure                                     */
 /*------------------------------------------------------------------*/

     if (no_input == FALSE)
     {
        version = CS_VERSION_50;

        rc = cs_ctx_alloc (version, &context);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CSCTXALLOC failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
           EXEC CICS RETURN ;
        }

 /*------------------------------------------------------------------*/
 /* Initialize the Client-Library                                    */
 /*------------------------------------------------------------------*/

        /* context allocated, it's now OK to use ct_diag for message
           retrieving                                                */
        diag_msgs_initialized = 1;
      rc = ct_init (context, version);

        if (rc == CS_SUCCEED)
        {
           proces_input ();
        }
```

```
          else
          {
             strncpy (msgstr, "CT_INIT failed", msg_size);
             no_errors_sw = FALSE ;
             error_out (rc);
          }
          close_connection ();
          quit_client_library ();
       }  /* process input data entered */

       else  /* no input data received */
          EXEC CICS RETURN ;

    }   /* end main */
```

Usage

- cs_ctx_alloc allocates a context structure.
- A context structure contains information that describes an application context. For example, a context structure defines the version of Client-Library that is in use.
- Allocating a context structure is the first step in any Client-Library application.
- After allocating a context structure, a Client-Library application typically customizes the context by calling cs_config and/or ct_config, then sets up one or more connections within the context.
- To deallocate a context structure, an application calls cs_ctx_drop.

See also

Related functions

- ct_config on page 96
- ct_con_alloc on page 88
- ct_config on page 96
- cs_ctx_drop on page 182

cs_ctx_drop

Description

Deallocates a context structure.

Syntax

CS_RETCODE cs_ctx_drop(context);
cs_context *context;

Parameters

context
(I) A pointer to a context structure.

Return value

cs_ctx_drop returns one of the following values:

Value	Meaning
CS_SUCCEED (-1)	The routine completed successfully.
CS_FAIL (-2)	The routine failed. The most common cause for a cs_ctx_drop failure is that the context contains an open connection.

Examples

The following code fragment demonstrates how to use cs_ctx_drop with other functions at the end of a program to close the connection and return to CICS. It is taken from the sample program SYCTSAA6 in Appendix A, “Sample Language Application.”

```
/********************************************************************/
/*                                                                  */
/* Subroutine to perform drop command handler, close server         */
/* connection, and deallocate Connection Handler.                   */
/*                                                                  */
/********************************************************************/
void   close_connection ()

{
CS_INT        rc;

/*------------------------------------------------------------------*/
/* drop the command handle                                          */
/*------------------------------------------------------------------*/

      rc = ct_cmd_drop (cmd);

      if (rc == CS_FAIL)
      {
         strncpy (msgstr, "CT_CMD_DROP failed", msg_size);
         error_out (rc) ;
      }


/*------------------------------------------------------------------*/
/* close the server connection                                      */
/*------------------------------------------------------------------*/
```

```
       rc = ct_close (connection, (long) CS_UNUSED);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CLOSE failed", msg_size);
          error_out (rc) ;
       }

/*-----------------------------------------------------------------*/
/* De_allocate the connection handle                               */
/*-----------------------------------------------------------------*/

       rc = ct_con_drop (connection);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CON_DROP failed", msg_size);
          error_out (rc) ;
       }
}   /* end close_connection */

/******************************************************************/
/*                                                                */
/* Subroutine to perform exit client library and deallocate context */
/* structure.                                                     */
/*                                                                */
/******************************************************************/
void   quit_client_library ()

{
 CS_INT        rc;
/*-----------------------------------------------------------------*/
/* Exit the Client Library                                         */
/*-----------------------------------------------------------------*/

       rc = ct_exit (context, (long) CS_UNUSED);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_EXIT failed", msg_size);
          error_out(rc) ;
        }
/*-----------------------------------------------------------------*/
/* De-allocate the context structure                               */
/*-----------------------------------------------------------------*/
```

```
            rc = cs_ctx_drop (context);

            if (rc == CS_FAIL)
            {
               strncpy (msgstr, "CT_CTX_DROP failed", msg_size);
               error_out(rc) ;
            }

            EXEC CICS RETURN ;

      }   /* end quit_client_library */
```

Usage

- cs_ctx_drop deallocates a context structure.
- A context structure describes a particular context, or operating environment, for a set of server connections.
- After a context deallocates, it cannot be reused. To allocate a new context, an application calls cs_ctx_alloc.
- A Client-Library application cannot call cs_ctx_drop to deallocate a context structure until it calls ct_exit to clean up Client-Library space associated with the context.
- cs_ctx_drop fails if the context contains an open connection.

See also

Related functions

- cs_ctx_alloc on page 179
- ct_close on page 72
- ct_exit on page 131

APPENDIX A Sample Language Application

This appendix contains a sample Open ClientConnect application, SYCTSAA6, that sends a language request to an Adaptive Server Enterprise. It retrieves information from a table, SYBASE.SAMPLETB, on the target server.

The purpose of this sample application is to demonstrate the use of Client-Library functions, particularly those designed to send language requests. In some cases, one Client-Library function is used for demonstration purposes when another function would be more efficient. To best illustrate the flow of processing, the program does not do extensive error checking.

This sample application is part of the Open ClientConnect package, on the API tape. The Transaction Router Service (TRS) administrator can create the table SYBASE.SAMPLETB on that server with a script provided with TRS.

An additional sample application, SYCTSAL6, is provided as part of the Open ClientConnect package on the API tape. It demonstrates how to send a language request to either Open ServerConnect or Adaptive Server Enterprise. The table SYBASE.SAMPLETB is provided on the Open ServerConnect API tape.

Sample program – SYCTSAA6

```
/*        @(#) syctsaa6.c   11.3 10/14/96     */

 /****** SYCTSAA6 - CLIENT LANGUAGE REQUEST APPL - C - CICS **********/
 /*                                                                   */
 /*  CICS TRANID:  SYA6                                               */
 /*                                                                   */
 /*  PROGRAM:     SYCTSAA6                                            */
 /*                                                                   */
 /*  PURPOSE:  Demonstrates Open Client for CICS CALLs.               */
 /*                                                                   */
 /*  FUNCTION: Illustrates how to send a language request with        */
 /*            parameters to:                                         */
 /*                                                                   */
 /*              - A SQL Server                                       */
 /*                                                                   */
 /*            SQL Server:                                            */
 /*                                                                   */
 /*              If the request is sent to a SQL Server it            */
 /*              executes the SQL statement:                          */
 /*                                                                   */
 /*                  SELECT  FIRSTNME, EDUCLVL                        */
 /*                    FROM  SYBASE.SAMPLETB                          */
 /*                                                                   */
 /*                                                                   */
 /*  PREREQS:  Before running SYCTSAA6, make sure that the server     */
 /*            you wish to access has an entry in the Connection      */
 /*            Router Table for that Server and the MCG(s) that       */
 /*            you wish to use.                                       */
 /*                                                                   */
 /*                                                                   */
 /* INPUT:    On the input screen, make sure to enter the Server      */
 /*           name, user id, and password for the target              */
 /*           TRAN NAME is not used for LAN servers.                  */
 /*                                                                   */
 /*                                                                   */
 /*  Open Client CALLs used in this sample:                           */
 /*                                                                   */
 /*    cs_convert    convert a datatype from one value to another     */
 /*    cs_ctx_alloc  allocate a context                               */
 /*    cs_ctx_drop   drop a context                                   */
 /*    ct_bind       bind a column variable                           */
 /*    ct_close      close a server connection                        */
 /*    ct_config     set or retrieve context properties               */
 /*    ct_cmd_alloc  allocate a command                               */
```

```
/*    ct_cmd_drop   drop a command                                        */
/*    ct_command    initiate remote procedure call                        */
/*    ct_con_alloc  allocate a connection                                 */
/*    ct_con_drop   drop a connection                                     */
/*    ct_con_props  alter properties of a connection                      */
/*    ct_connect    open a server connection                              */
/*    ct_describe   return a description of result data                   */
/*    ct diag       retrieve SQLCODE messages                             */
/*    ct_exit       exit client library                                   */
/*    ct_fetch      fetch result data                                     */
/*    ct_init       init client library                                   */
/*    ct_results    set up result data                                    */
/*    ct_res_info   return result set info                                */
/*    ct_send       send a request to the server                          */
/*                                                                        */
/* History:                                                               */
/*                                                                        */
/* Date    BTS#   Descrition                                              */
/* ======= ====== ======================================================= */
/* Sept 96        Create                                                  */
/*                                                                        */
/*                                                                        */
/**************************************************************************/
#pragma csect(code,"SYCTSAA6")
#pragma linkage(SYCVTD,  OS)

/*------------------------------------------------------------------------*/
/* CLIENT LIBRARY C COPY BOOK                                             */
/*------------------------------------------------------------------------*/

#include "ctpublic.h"

/*------------------------------------------------------------------------*/
/* CICS BMS DEFINITIONS C COPY BOOK                                       */
/*------------------------------------------------------------------------*/

#include "syctba6.h"

/*------------------------------------------------------------------------*/
/* DEFINES                                                                */
/*------------------------------------------------------------------------*/

#define FALSE          0
#define TRUE           1
#define IN  a6panel.a6paneli
#define OUT a6panel.a6panelo
```

```
#define RS  a6panel.a6panelo.rsltne


/*------------------------------------------------------------------*/
/* GLOBALS - CLIENT COMMON WORK AREA                                */
/*------------------------------------------------------------------*/

   /*
   **  Open Client variables
   */

   CS_CONTEXT    *context;          /* pointer to context handle    */
   CS_CONNECTION *connection;       /* pointer to connection handle */
   CS_COMMAND    *cmd;              /* pointer to command proc      */


   CS_CHAR     servname[30];        /* Server name        */
   CS_CHAR     username[8];         /* User login name    */
   CS_CHAR     pwd[8];              /* Password           */
   CS_CHAR     tran[8];             /* Transaction name   */
   CS_CHAR     driver[9];           /* Network driver     */
   CS_INT      server_size;
   CS_INT      user_size;
   CS_INT      pwd_size;
   CS_INT      tran_size;

   CS_CHAR     msgstr[40];          /* message string     */
   CS_INT      msg_size = 40;       /* error message size */
   CS_CHAR     msgtext1[79];
   CS_CHAR     msgtext2[79];
   CS_INT      text_size = 79;
   CS_INT      page_cnt = 0;
   CS_INT      row_num = 0;

   CS_INT      version;             /* CT version #       */
   CS_VARCHAR  col_firstnme;
   CS_SMALLINT col_edlevel;
   CS_INT      maxconnect = 0;
   CS_INT      results_type;

   /*
   **  Variables to control program flow
   */
   CS_SMALLINT   no_input = 1;
   CS_SMALLINT   no_errors_sw = 1;
   CS_SMALLINT   no_more_results = 0;
```

```
   CS_SMALLINT   diag_msgs_initialized = 0;
   CS_INT        print_once = 1;

   /*
   **  Variables for conversion to display
   */
   CS_INT      cvtleft;
   CS_INT      cvtright;
   double      cvtdbl;
   char        cvtwork[17];

   /*
   **  CICS variables for
   **  System date and time
   */
   char        TMP_DATE[8] = "        ";
   char        TMP_TIME[8] = "        ";
   char UTIME[8];

/*------------------------------------------------------------------*/
/* Standard CICS Attribute and Print Control Chararcter List        */
/*------------------------------------------------------------------*/

#include <dfhbmsca.h>                  /* BMS definitions          */

/*------------------------------------------------------------------*/
/* CICS Standard Attention Identifiers C Copy Book                  */
/*------------------------------------------------------------------*/

 #include <dfhaid.h>                   /* PFK definitions          */

 /*------------------------------------------------------------------*/
 /* Local Functions Prototypes                                       */
 /*------------------------------------------------------------------*/

 void   proces_input();
 void   bind_columns();
 void   send_command();
 void   proces_results();
 void   result_row_processing();
 void   fetch_row_processing();
 void   error_out();
 void   get_diag_messages();
 void   get_client_msgs();
 void   get_server_msgs();
 void   close_connection();
```

```
 void   quit_client_library();
 void   display_initial_screen();
 void   get_input_data();
 void   disp_data();

 /*********************************************************************/
 /* Main routine                                                      */
 /*********************************************************************/

 main ()
 {
  CS_INT        rc;
  CS_INT        i;


  EXEC CICS ADDRESS EIB(dfheiptr);

 /*-------------------------------------------------------------------*/
 /* program initialization                                            */
 /*-------------------------------------------------------------------*/
        memset (msgtext1, ' ', text_size);
        strncpy (msgtext2, "Press Clear To Exit", text_size);
        page_cnt  = page_cnt + 1;
        memset(servname, ' ', 30);
        memset(username, ' ', 8);
        memset(tran, ' ', 8);
        memset(pwd, ' ', 8);
        memset(driver, ' ', 9);

        for (i = 0; i < 14; ++i)
           memset (RS[i].rsltno, ' ', text_size); /* init output
lines  */

        /*  get system time   */
        EXEC CICS ASKTIME ABSTIME(UTIME);

        EXEC CICS FORMATTIME
                  ABSTIME(UTIME)
                  DATESEP('/')
                  MMDDYY(TMP_DATE)
                  TIME(TMP_TIME)
                  TIMESEP ;

        display_initial_screen ();
        get_input_data ();
```

```
 /*------------------------------------------------------------------*/
 /* Allocate a context structure                                     */
 /*------------------------------------------------------------------*/

     if (no_input == FALSE)
     {
        version = CS_VERSION_50;

        rc = cs_ctx_alloc (version, &context);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CSCTXALLOC failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
           EXEC CICS RETURN ;
        }

/*------------------------------------------------------------------*/
/* Initialize the Client-Library                                    */
/*------------------------------------------------------------------*/

        /* context allocated, it's now OK to use ct_diag for message
           retrieving                                                 */
        diag_msgs_initialized = 1;

        rc = ct_init (context, version);

        if (rc == CS_SUCCEED)
        {
           proces_input ();
        }
        else
        {
           strncpy (msgstr, "CT_INIT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
        close_connection ();
        quit_client_library ();
     }  /* process input data entered */

     else  /* no input data received */
        EXEC CICS RETURN ;

 }   /* end main */
```

```
/*****************************************************************/
/*                                                               */
/* Subroutine to display the initial screen                      */
/*                                                               */
/*****************************************************************/
void   display_initial_screen ()

{
        strncpy (OUT.sdateo, TMP_DATE, 8);
        strncpy (OUT.stimeo, TMP_TIME, 8);
        strncpy (OUT.prognmo, "SYCTSAA6", 8);
        strncpy (OUT.msg1o, msgtext1, text_size);
        strncpy (OUT.msg2o, msgtext2, text_size);
        strncpy (OUT.spageo, "0001", 4);

        EXEC CICS SEND MAP("SYCTBA6")
                         FROM(a6panel)
                         CURSOR(252)
                         FRSET
                         ERASE
                         FREEKB ;

}   /* end display_initial_screen  */

/*****************************************************************/
/*                                                               */
/* Subroutine to get input data                                  */
/*                                                               */
/*****************************************************************/
void   get_input_data ()
{
 CS_SMALLINT   enter_data_sw = 0;
 CS_CHAR       blank30[30] = "                              ";
 CS_CHAR       blank8[8] = "        ";
 int           i;


        EXEC CICS RECEIVE MAP("a6panel")
                          MAPSET("SYCTBA6")
                          ASIS ;

        if (IN.serverl == 0)
        {
           if (strncmp (servname, blank30, 30) == 0)
           {
              IN.serverl    = -1 ;        /* set the cursor position */
```

```
               strncpy (msgtext1, "Please Enter Server Name",
text_size);
               enter_data_sw  = TRUE ;
            }
         }
         else if (IN.serverl > 0)
         {
            server_size = IN.serverl ;
            strncpy (servname, IN.serveri, server_size);
            no_input = 0;
         }

         if (IN.userl == 0)
         {
            if (strncmp (username, blank8, 8) == 0)
            {
               IN.userl       = -1 ;   /* set the cursor position */
               if (enter_data_sw == FALSE) /* not overlaid  msgtext1 */
                  strncpy (msgtext1, "Please Enter User-ID", text_size);
               enter_data_sw  = TRUE ;
            }
         }
         else
            {
               user_size = IN.userl ;
               strncpy (username, IN.useri, user_size);
            }

         if (IN.pswdl != 0)
            {
               pwd_size = IN.pswdl ;
               strncpy (pwd, IN.pswdi, pwd_size);
            }

         if (IN.tranl != 0)
            {
               tran_size = IN.tranl ;
               strncpy (tran, IN.trani, tran_size);
            }

         if (IN.netdrvl != 0)
            {
               strncpy (driver, IN.netdrvi, 9);
               for (i=0; i<9; ++i)
                   driver[i] = toupper (driver[i]);
            }
```

```
        if (enter_data_sw == TRUE)      /* bad input data  */
           {
              enter_data_sw = FALSE ;
              display_initial_screen ();/* request for input again */
              get_input_data ();
           }

}   /*  end get_input_data  */

/********************************************************************/
/*                                                                  */
/* Subroutine to process input data                                 */
/*                                                                  */
/********************************************************************/
void   proces_input ()

{
 CS_INT         rc;
 CS_INT         *outlen;
 CS_INT         buf_len;
 CS_INT         msglimit;
 CS_INT         netdriver;

/*------------------------------------------------------------*/
/* Allocate a connection to the server                        */
/*------------------------------------------------------------*/

       rc = ct_con_alloc (context, &connection);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CONALLOC failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
        }

/*------------------------------------------------------------*/
/* Alter properties of the connection for user-id             */
/*------------------------------------------------------------*/

       buf_len  = user_size;
       rc = ct_con_props (connection, (long)CS_SET,
                         (long)CS_USERNAME, username,
                         buf_len, outlen);
```

```
        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_CON_PROPS for user-id failed",
           msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
         }

  /*---------------------------------------------------------------*/
  /* Alter properties of the connection for password             */
  /*---------------------------------------------------------------*/

        buf_len = pwd_size;
        rc = ct_con_props (connection, (long)CS_SET,
                           (long)CS_PASSWORD, pwd, buf_len, outlen);

         if (rc != CS_SUCCEED)
         {
           strncpy (msgstr, "CT_CON_PROPS for password failed",
           msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
         }

  /*---------------------------------------------------------------*/
  /* Alter properties of the connection for transaction          */
  /*---------------------------------------------------------------*/

        buf_len = tran_size;
        rc = ct_con_props (connection, (long)CS_SET,
                           (long)CS_TRANSACTION_NAME,
                          tran, buf_len, outlen);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_CON_PROPS for transaction failed",
           msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }

  /*---------------------------------------------------------------*/
  /* Alter properties of the connection for network driver       */
  /*---------------------------------------------------------------*/

        netdriver = 9999;   /* default value for non-regconized
```

```
                              driver name                       */

       /* if no netdriver entered, default is LU62  */
       if (strncmp(driver,"         ",9) == 0  ??
           strncmp(driver,"LU62",4) == 0)
          netdriver = CS_LU62;

       else if (strncmp(driver,"INTERLINK",8) == 0)
          netdriver = CS_INTERLINK;

       else if (strncmp(driver,"IBMTCPIP",8) == 0)
          netdriver = CS_TCPIP;

       else if (strncmp(driver,"CPIC",4) == 0)
          netdriver = CS_NCPIC;

       rc = ct_con_props (connection, (long)CS_SET,
                         (long)CS_NET_DRIVER, (long) netdriver,
                         CS_UNUSED, outlen);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CON_PROPS for network driver failed",
          msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
       }

/*------------------------------------------------------------*/
/* Setup retrieval of All Messages                            */
/*------------------------------------------------------------*/

       rc = ct_diag (connection, CS_INIT,
                     CS_UNUSED, CS_UNUSED, CS_NULL);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_DIAG CS_INIT failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
       }

/*------------------------------------------------------------*/
/* Set the upper limit of number of messages                  */
/*------------------------------------------------------------*/
```

```
        msglimit = 5 ;

        rc = ct_diag (connection, CS_MSGLIMIT, CS_ALLMSG_TYPE,
                      CS_UNUSED, &msglimit);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_DIAG CS_MSGLIMIT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }

/*------------------------------------------------------------*/
/* Open connection to the server or CICS region               */
/*------------------------------------------------------------*/

        rc = ct_connect (connection, servname, server_size);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_CONNECT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }

/*------------------------------------------------------------*/
/* Invokes SEND_COMMAND routine                               */
/*------------------------------------------------------------*/
        if (no_errors_sw)
           send_command ();

/*------------------------------------------------------------*/
/* Process the results of the command                         */
/*------------------------------------------------------------*/

        if (no_errors_sw)
        {
           while (no_more_results == FALSE)
              proces_results ();
        }
}   /* end proces_input */
/******************************************************************/
/*                                                                */
/* Subroutine to allocate, send, and process commands             */
/*                                                                */
/******************************************************************/
```

```
void send_command ()

{
 CS_INT         rc;
 CS_INT         *outlen;
 CS_INT         buf_len;
 CS_CHAR        sql_cmd[45];

/*------------------------------------------------------------*/
/* Find out what the maximum number of connections is         */
/*------------------------------------------------------------*/

       rc = ct_config (context, CS_GET, CS_MAX_CONNECT,
                       &maxconnect, CS_FALSE, outlen);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CONFIG failed", msg_size);
          strncpy (msgtext2, "Please press return to continue!",
                   text_size);
          error_out(rc);

          /* reset program flags to move on with the task */
          print_once = TRUE;
          diag_msgs_initialized = TRUE;
          strncpy(msgtext2, "Press Clear To Exit", text_size);
       }

/*------------------------------------------------------------*/
/* Allocate a command handle                                  */
/*------------------------------------------------------------*/

       rc = ct_cmd_alloc (connection, &cmd);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CMDALLOC failed", msg_size);
          no_errors_sw = FALSE ;
          error_out(rc);
       }

/*------------------------------------------------------------*/
/* Prepare the language request                               */
/*------------------------------------------------------------*/
       strcpy(sql_cmd,
              "SELECT FIRSTNME, EDUCLVL FROM SYBASE.SAMPLETB");
```

```
        buf_len = sizeof(sql_cmd);

        rc = ct_command(cmd, (long) CS_LANG_CMD, sql_cmd,
                        buf_len, (long) CS_UNUSED);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_COMMAND failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }

/*-------------------------------------------------------------*/
/* Send the language request                                   */
/*-------------------------------------------------------------*/

        rc = ct_send (cmd);

        if (rc != CS_SUCCEED)
        {
           strcpy (msgstr, "CT_SEND failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
}    /* end send_command */

/*******************************************************************/
/*                                                                 */
/* Subroutine to process the result                                */
/*                                                                 */
/*******************************************************************/
void   proces_results ()

{
 CS_INT        rc;

/*-------------------------------------------------------------*/
/* Set up the results data                                     */
/*-------------------------------------------------------------*/

        rc = ct_results (cmd, &results_type);

/*-------------------------------------------------------------*/
/* Determine the outcome of the comand execution               */
/*-------------------------------------------------------------*/
```

```
        switch (rc)
        {
           case CS_SUCCEED:

  /*-----------------------------------------------------------------*/
  /* Determine the type of result returned by the current request */
  /*-----------------------------------------------------------------*/

                switch (results_type)
                {

  /*---------------------------------------------------------------*/
  /* Process row results                                           */
  /*---------------------------------------------------------------*/

                   case CS_ROW_RESULT:
                        result_row_processing ();
                        fetch_row_processing ();
                        break;

 /*---------------------------------------------------------------*/
 /* Process parameter results --- there should be no parameter */
 /* to process                                                     */
 /*---------------------------------------------------------------*/

                   case CS_PARAM_RESULT:
                        no_more_results = FALSE;
                        break;

  /*---------------------------------------------------------------*/
  /* process status results --- the stored procedure status     */
  /* result will not be processed in this example               */
  /*---------------------------------------------------------------*/

                   case CS_STATUS_RESULT:
                        no_more_results = FALSE;
                        break;

  /*---------------------------------------------------------------*/
  /* print an error message if the server encountered an error  */
  /* while executing the request                                */
  /*---------------------------------------------------------------*/

                   case CS_CMD_FAIL:
                        no_errors_sw = FALSE ;
                        strncpy (msgstr,
```

```
                          "CT_RESUL returned CS_CMD-FAIL restype",
                        msg_size);
                        error_out (rc);
                        break;

/*------------------------------------------------------------*/
/* print a message for successful commands that returned no   */
/* data( optional )                                           */
/*------------------------------------------------------------*/

                   case CS_CMD_SUCCEED:
                        strncpy (msgstr,
                          "CT_RESUL returned CS_CMD_SUCCEED restype",
                        msg_size);
                        break;

/*------------------------------------------------------------*/
/* print a message for requests that have been processed      */
/* successfully( optional )                                   */
/*------------------------------------------------------------*/

                   case CS_CMD_DONE:
                        strncpy (msgstr,
                        "CT_RESUL returned CS_CMD_DONE restype",
                        msg_size);
                        break;

                   default:
                        no_more_results = TRUE ;
                        no_errors_sw    = FALSE ;
                        strncpy (msgstr,
                          "CT_RESUL returned UNKNOWN restype",
                        msg_size);
                        error_out (rc);
                        break;

                }   /* end of switch (result_type) */
                break;   /* case of CS_SUCCEED  */

/*------------------------------------------------------------*/
/* print an error message if the CTBRESULTS call failed       */
/*------------------------------------------------------------*/

          case CS_FAIL:
                no_more_results = TRUE ;
                no_errors_sw    = FALSE ;
```

```
                strncpy (msgstr, "CT_RESULTS returned CS_FAIL ret_code",
                         msg_size);
                error_out (rc);
                break;

/*------------------------------------------------------------*/
/* drop out of the results loop if no more result sets are    */
/* available for processing or if the results were cancelled  */
/*------------------------------------------------------------*/

           case CS_END_RESULTS:
           case CS_CANCELLED:
                no_more_results = TRUE ;
                break;

           default:
                no_more_results = TRUE ;
                no_errors_sw    = FALSE ;
                strncpy (msgstr, "CT_RESULTS returned unknown ret_code",
                         msg_size);
                error_out (rc);
                break;

       }  /* end of switch (rc) */

}   /* end process_results */

/*******************************************************************/
/*                                                                 */
/* Subroutine to process result rows                               */
/*                                                                 */
/*******************************************************************/
void   result_row_processing ()

{
 CS_INT         rc;
 CS_INT         col_len;
 CS_INT         numcol;
 CS_INT         parm_cnt;
 char           msg1[40] = "The maximum number of connections is   ";
 char           msg2[25] = "The number of columns is ";
 char           wrk_str [4];
 char           period = '.';
```

```
 /*------------------------------------------------------------*/
 /* We need to bind the data to program variables. We don't    */
 /* care about the indicator variable so we'll pass NULL for   */
 /* that parameter in OC_BIND().                               */
 /*------------------------------------------------------------*/

        rc =
ct_res_info(cmd,CS_NUMDATA,&numcol,sizeof(numcol),&col_len);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_RES_INFO failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }

/*------------------------------------------------------------*/
/* display the number of connections                          */
/*------------------------------------------------------------*/
         row_num = row_num + 1;
         strncpy (RS[row_num].rsltno, msg1, msg_size);

         cvtleft  = 4;            /* Digits to the left */
         cvtright = 0;            /* Digits to the right */
         SYCVTD(maxconnect, wrk_str,
                 cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

         strncat (RS[row_num].rsltno, wrk_str, 4);
         row_num = row_num + 2;

 /*------------------------------------------------------------*/
 /* display the number of columns                              */
 /*------------------------------------------------------------*/

         strncpy (RS[row_num].rsltno, msg2, sizeof(msg2));
         cvtleft  = 4;            /* Digits to the left */
         cvtright = 0;            /* Digits to the right */
         SYCVTD(numcol, wrk_str,
                 cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

         strncat (RS[row_num].rsltno, wrk_str, 4);
         row_num = row_num + 2;

         if (numcol != 2)
         {
           strncpy (msgstr, "CT_RES_INFO returned wrong # of parms",
```

```
                 msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
        }

  /*------------------------------------------------------------*/
  /* Setup column headings                                      */
  /*------------------------------------------------------------*/

        strncpy (RS[row_num].rsltno,
                 "FirstName    EducLvl", text_size);
        row_num = row_num + 1;
        strncpy (RS[row_num].rsltno,
                 "=========    =======", text_size);

        for (parm_cnt = 1; parm_cnt <= numcol; ++parm_cnt)
            bind_columns (parm_cnt);

  }   /* end result_row_processing */

/*******************************************************************/
/*                                                                 */
/* Subroutine to bind each data                                    */
/*                                                                 */
/*******************************************************************/
 void   bind_columns (parm_cnt)

 CS_INT        parm_cnt;
 {
  CS_INT        rc;
  CS_INT        col_len;
  CS_DATAFMT    datafmt;

       rc= ct_describe(cmd, parm_cnt, &datafmt);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_DESCRIBE failed", msg_size);
          no_errors_sw = FALSE ;
          error_out(rc);
       }
 /*------------------------------------------------------------*/
 /* We need to bind the data to program variables. We don't    */
 /* care about the indicator variable so we'll pass NULL for   */
 /* that parameter in OC_BIND().                               */
 /*------------------------------------------------------------*/
```

```
 /*-------------------------------------------------------------*/
 /* rows per fetch                                              */
 /*-------------------------------------------------------------*/

        switch (datafmt.datatype)
        {
 /*-------------------------------------------------------------*/
 /* bind the first column, FIRSTNME defined as VARCHAR(12)     */
 /*-------------------------------------------------------------*/
            case  CS_VARCHAR_TYPE:

               rc= ct_bind(cmd, parm_cnt, &datafmt, &col_firstnme,
                           &col_len, CS_NULL);

               if (rc != CS_SUCCEED)
               {
                  strncpy (msgstr, "CT_BIND CS_VARCHAR_TYPE failed",
                  msg_size);
                  no_errors_sw = FALSE ;
                  error_out(rc);
               }
               break;
/*-------------------------------------------------------------*/
/* bind the second column, EDLEVEL defined as SMALLINT        */
/*-------------------------------------------------------------*/
            case  CS_SMALLINT_TYPE:

               rc= ct_bind(cmd, parm_cnt, &datafmt, &col_edlevel,
                           &col_len, CS_NULL);

               if (rc != CS_SUCCEED)
               {
                  strncpy (msgstr, "CT_BIND CS_SMALLINT_TYPE failed",
                  msg_size);
                  no_errors_sw = FALSE ;
                  error_out(rc);
               }
               break;

            default:
               break;

        } /* end of switch (datatype) */

 }   /* end bind_columns */
```

```
/******************************************************************/
/*                                                                */
/* Subroutine to fetch row processing                             */
/*                                                                */
/******************************************************************/
void   fetch_row_processing ()

{
 CS_INT        rows_read;
 CS_INT        rc;
 CS_INT        col_len;
 CS_INT        max_screen_rows = 10;
 CS_SMALLINT   nomore_rows    = 0;
 CS_DATAFMT    datafmt;
 CS_DATAFMT    datafmt2;
 struct {
         char  first[12];
         char  space2[2];
         char  edlevel[4];
        } output_row;


     while (nomore_rows == FALSE)
     {
       strcpy(col_firstnme.str, "            ");
       memset (&output_row, ' ', sizeof(output_row));

       rc = ct_fetch (cmd, (long) CS_UNUSED,
                      (long) CS_UNUSED,
                      (long) CS_UNUSED,
                      &rows_read);

       switch (rc)
       {
           case  CS_SUCCEED:
               nomore_rows         = FALSE ;
               datafmt.datatype     = CS_VARCHAR_TYPE;
               datafmt.maxlength    = sizeof(col_firstnme);
               datafmt2.datatype    = CS_CHAR_TYPE;
               datafmt2.maxlength   = 12;

/*------------------------------------------------------------*/
/* convert the first column from VARCHAR to CHAR for display  */
/*------------------------------------------------------------*/
               if (cs_convert(context, datafmt, col_firstnme, datafmt2,
                              &output_row.first, &col_len) !=
```

```
CS_SUCCEED)
                {
                   strncpy (msgstr,
                     "CS_CONVERT CS_CHAR_TYPE failed", msg_size);
                   no_errors_sw = FALSE ;
                   error_out(rc);
                }

 /*------------------------------------------------------------*/
 /* save ROW RESULTS for later display                         */
 /*------------------------------------------------------------*/

                cvtleft  = 4;           /* Digits to the left */
                cvtright = 0;           /* Digits to the right */
                SYCVTD(col_edlevel, output_row.edlevel,
                        cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

                if (row_num > max_screen_rows)
                {
                   strncpy (msgtext1, "Please press return to
                   continue!", text_size);
                   memset (msgtext2, ' ', text_size);

                   disp_data ();

                   /* re-init output lines */
                   for (row_num = 0; row_num < 14; ++row_num)
                       memset (RS[row_num].rsltno, ' ', text_size);

                   row_num = 1;
                   page_cnt = page_cnt + 1 ;

 /*------------------------------------------------------------*/
 /* Setup column headings                                      */
 /*------------------------------------------------------------*/
                   strncpy (RS[row_num].rsltno, "FirstName    EducLvl",
                            text_size);
                   row_num += 1;
                   strncpy (RS[row_num].rsltno, "==========   =======",
                            text_size);
                   row_num += 1;
                }  /* if row_num > 10 */

                row_num += 1;
                memcpy (RS[row_num].rsltno, &output_row,
                        sizeof(output_row));
```

```
                break; /* end of CS_SUCCEED */

            case  CS_END_DATA:
                nomore_rows = TRUE ;
                strncpy (msgtext1, "All rows processing completed!",
                         text_size);
                strncpy (msgtext2, "Press Clear To Exit", text_size);
                disp_data ();
                break; /* end of CS_END_DATA  */

            case  CS_FAIL:
                nomore_rows = TRUE ;
                no_errors_sw = FALSE ;
                strncpy (msgstr, "CT_FETCH returned CS_FAIL ret_code");
                error_out(rc);
                break; /* end of CS_FAIL  */

            case  CS_ROW_FAIL:
                nomore_rows = TRUE ;
                no_errors_sw = FALSE ;
                strncpy (msgstr, "CT_FETCH returned CS_ROW_FAIL
                ret_code");
                error_out(rc);
                break; /* end of CS_ROW_FAIL  */

            case  CS_CANCELLED:
                nomore_rows = TRUE ;
                no_errors_sw = FALSE ;
                strncpy (msgstr, "CT_FETCH returned CS_CANCELLED
                ret_code");
                error_out(rc);
                break; /* end of CS_CANCELLED  */

            default:
                nomore_rows = TRUE ;
                no_errors_sw = FALSE ;
                strncpy (msgstr, "CT_FETCH returned Unknown ret_code");
                error_out(rc);
                break; /* end of OTHERWISE */

        }  /* end of switch (rc) */
      }   /* end of while nomore_rows false */

  }  /* end fetch_row_processing */
```

```
/********************************************************************/
/*                                                                  */
/* Subroutine to print output messages.                             */
/*                                                                  */
/********************************************************************/

void   error_out (rc)
 CS_INT        rc;

{
   /*
   **  Display Message
   */
   struct {
           char        test_case[9] ;
           char        samp_lit[5] ;
           char        samp_rc[4] ;
           char        rest_lit[15] ;
           char        rest_type[4] ;
           char        filler[2] ;
           char        msg[40];
   } disp_msg;


        memset(&disp_msg, ' ', sizeof(disp_msg));
        strcpy(disp_msg.test_case, "SYCTSAA6");
        strcpy(disp_msg.samp_lit, "rc = ");
        strcpy(disp_msg.rest_lit, "  Result Type: ");

        cvtleft  = 4;           /* Digits to the left */
        cvtright = 0;           /* Digits to the right */

        SYCVTD(rc, disp_msg.samp_rc,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
        SYCVTD(results_type, disp_msg.rest_type,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

        if (diag_msgs_initialized)
           get_diag_messages ();

/*------------------------------------------------------------------*/
/* display error messages                                           */
/*------------------------------------------------------------------*/

        strncpy (disp_msg.msg, msgstr, msg_size);
        memcpy (msgtext1, &disp_msg, sizeof(disp_msg));
```

```
         if (print_once)
           {
             disp_data ();
             print_once = FALSE ;
           }

 }   /* end error_out */

 /********************************************************************/
 /*                                                                  */
 /* Subroutine to retrieve any diagnostic messages                   */
 /*                                                                  */
 /********************************************************************/
 void   get_diag_messages()

 {
  CS_SMALLINT   cnt;
  CS_INT        num_of_msgs = 0;
  CS_INT        rc;

/*------------------------------------------------------------------*/
/* Disable calls to this subroutine                                 */
/*------------------------------------------------------------------*/

         diag_msgs_initialized = FALSE ;

 /*------------------------------------------------------------------*/
 /* First, get client messages                                       */
 /*------------------------------------------------------------------*/
        rc = ct_diag (connection, CS_STATUS, CS_CLIENTMSG_TYPE,
                      CS_UNUSED, #_of_msgs);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_DIAG CS_STATUS CLIENTMSG_TYPE failed",
                    msg_size);
           error_out(rc) ;
         }
         else if (num_of_msgs > 0)
         {
              for (cnt = 1; cnt <= num_of_msgs; ++cnt)
                  get_client_msgs ();
         }
```

```
/*-----------------------------------------------------------------*/
/* Then, get server messages                                       */
/*-----------------------------------------------------------------*/
        rc = ct_diag (connection, CS_STATUS, CS_SERVERMSG_TYPE,
                      CS_UNUSED, #_of_msgs);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_DIAG CS_STATUS SERVERMSG_TYPE failed",
                    msg_size);
           error_out(rc) ;
         }
         else if (num_of_msgs > 0)
         {
              for (cnt = 1; cnt <= num_of_msgs; ++cnt)
                  get_server_msgs ();
         }
 }   /* end get_diag_messages  */

 /*******************************************************************/
 /*                                                                 */
 /* Subroutine to retrieve diagnostic messages from client          */
 /*                                                                 */
 /*******************************************************************/
 void   get_client_msgs()
 {
  CS_INT        rc;
  CS_INT        i;
  CS_CHAR      *txtpos;
  CS_INT        textleft;
  CS_INT        msgno = 1;
  CS_CHAR       blank_13[13] = "             ";
  CS_CLIENTMSG  clientmsg;

  struct {
          char    msgno_hdr[13];
          char    msgno_data[8];
          char    severity_hdr[13];
          char    severity_data[6];
         } client_msg;


        rc = ct_diag (connection, CS_GET, CS_CLIENTMSG_TYPE,
                      msgno, &clientmsg);

        if (rc !=  CS_SUCCEED)
```

```
       {
          strncpy (msgstr, "CT_DIAG CS_GET CS_CLIENTMSG_TYPE failed",
                   msg_size);
          error_out(rc) ;
       }
/*------------------------------------------------------------------*/
/* display message text                                             */
/*------------------------------------------------------------------*/
       i = 1 ;
       strncpy (RS[i].rsltno, "Client Message:");
       i = 3 ;

       memset(&client_msg, ' ', sizeof(client_msg));
       strcpy (client_msg.msgno_hdr, "  OC MsgNo:  ");
       strcpy (client_msg.severity_hdr, "  Severity:  ");

       cvtleft  = 8;            /* Digits to the left */
       cvtright = 0;            /* Digits to the right */
       SYCVTD(clientmsg.msgnumber, client_msg.msgno_data,
               cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
       cvtleft  = 6;            /* Digits to the left */
       SYCVTD(clientmsg.severity, client_msg.severity_data,
               cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

       memcpy (RS[i].rsltno, &client_msg, sizeof(client_msg));
       i += 1 ;

       /* get number of Client msgs */

       if (clientmsg.msgnumber != 0)
       {
          strcpy (RS[i].rsltno, "  OC MsgTx:  ");
          strncat (RS[i].rsltno, clientmsg.msgstring, 66);
          i += 1 ;
          txtpos = clientmsg.msgstring + 66;
          textleft = clientmsg.msgstringlen - 66;
          while (textleft > 0)
          {
             strncpy (RS[i].rsltno, blank_13, 13);
             strncat (RS[i].rsltno, txtpos, 66);
             i += 1;
             txtpos += 66;
             textleft -= 66;
          }
       }
       else
```

```
        {
           strncpy (RS[i].rsltno, "  OC MsgTx:  No Message!",
           text_size);
           i += 1 ;
        }

        /* get number of Server msgs */

        memset(&client_msg, ' ', sizeof(client_msg));
        strcpy (client_msg.msgno_hdr, "  OS MsgNo:  ");
        cvtleft  = 8;            /* Digits to the left */
        cvtright = 0;            /* Digits to the right */
        SYCVTD(clientmsg.osnumber, client_msg.msgno_data,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

        memcpy (RS[i].rsltno, &client_msg, sizeof(client_msg));
        i += 1 ;

        if (clientmsg.osnumber != 0)
        {
           strcpy (RS[i].rsltno, "  OS MsgTx:  ");
           strncat (RS[i].rsltno, clientmsg.osstring, 66);
           i += 1 ;
           txtpos = clientmsg.osstring + 66;
           textleft = clientmsg.osstringlen - 66;
           while (textleft > 0)
           {
              strncpy (RS[i].rsltno, blank_13, 13);
              strncat (RS[i].rsltno, txtpos, 66);
              i += 1;
              txtpos += 66;
              textleft -= 66;
           }
        }
        else
        {
           strncpy (RS[i].rsltno, "  OS MsgTx:  No Message!",
           text_size);
           i += 1 ;
        }
   }   /* end get_client_msgs */
/*-------------------------------------------------------------------*/
/*                                                                   */
/* Subroutine to retrieve diagnostic messages from server           */
/*                                                                   */
/*-------------------------------------------------------------------*/
```

```
void   get_server_msgs()
{
 CS_INT        rc;
 CS_INT        i;
 CS_CHAR      *txtpos;
 CS_INT        textleft;
 CS_INT        msgno = 1;
 CS_CHAR       blank_13[13] = "             ";
 CS_CHAR       proc_id_data[66];
 CS_CHAR       svrname_data[66];
 CS_SERVERMSG  servermsg;

 struct {
         char    msg_no_hdr[13];
         char    msg_no_data[6];
         char    severity_hdr[14];
         char    severity_data[6];
         char    state_hdr[14];
         char    state_data[4];
         char    line_no_hdr[13];
         char    line_no_data[4];
        } serv_msg;


       memset(&serv_msg, ' ', sizeof(serv_msg));

       rc = ct_diag (connection, CS_GET, CS_SERVERMSG_TYPE,
                     msgno, &servermsg);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_DIAG CS_GET CS_SERVERMSG_TYPE failed",
                   msg_size);
          error_out (rc) ;
       }

/*------------------------------------------------------------------*/
/* display message text                                             */
/*------------------------------------------------------------------*/

       strcpy (serv_msg.msg_no_hdr, "  Message#:  ");
       strcpy (serv_msg.severity_hdr, ",  Severity:  ");
       strcpy (serv_msg.state_hdr, ",  State No:  ");
       strcpy (serv_msg.line_no_hdr, "  Line  No:  ");

       cvtleft  = 6;              /* Digits to the left */
```

```
cvtright = 0;              /* Digits to the right */
SYCVTD(servermsg.msgnumber, serv_msg.msg_no_data,
        cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
SYCVTD(servermsg.severity, serv_msg.severity_data,
        cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

cvtleft  = 4;              /* Digits to the left */
SYCVTD(servermsg.state, serv_msg.state_data,
        cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
SYCVTD(servermsg.line, serv_msg.line_no_data,
        cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

if (servermsg.svrnlen > 66)
{
     strncpy (svrname_data, servermsg.svrname, 63);
     strcat (svrname_data, "...");
}
else
     strncpy (svrname_data, servermsg.svrname, 66);

if (servermsg.proclen > 66)
{
     strncpy (proc_id_data, servermsg.proc, 63);
     strcat (proc_id_data, "...");
}
else
     strncpy (proc_id_data, servermsg.proc, 66);

strncpy (RS[1].rsltno, "Server Message:", text_size);
memcpy (RS[3].rsltno, &serv_msg, sizeof(serv_msg));
strcpy (RS[5].rsltno, "  Serv Nam:  ");
strcat (RS[5].rsltno, svrname_data);
strcpy (RS[6].rsltno, "  Proc ID :  ");
strcat (RS[6].rsltno, proc_id_data);
strcpy (RS[7].rsltno, "  Message :  ");
strncat (RS[7].rsltno, servermsg.text, 66);

i = 8 ;
txtpos = servermsg.text + 66;
textleft = servermsg.textlen - 66;
while (textleft > 0)
   {
      strncpy (RS[i].rsltno, blank_13, 13);
      strncat (RS[i].rsltno, txtpos, 66);
      i += 1;
      txtpos += 66;
```

```
            textleft -= 66;
         }

}   /* end get_server_msgs */

/********************************************************************/
/*                                                                  */
/* Subroutine to perform drop command handler, close server         */
/* connection, and deallocate Connection Handler.                   */
/*                                                                  */
/********************************************************************/
void   close_connection ()

{
 CS_INT        rc;

/*------------------------------------------------------------------*/
/* drop the command handle                                          */
/*------------------------------------------------------------------*/

       rc = ct_cmd_drop (cmd);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CMD_DROP failed", msg_size);
          error_out (rc) ;
       }

/*------------------------------------------------------------------*/
/* close the server connection                                      */
/*------------------------------------------------------------------*/

       rc = ct_close (connection, (long) CS_UNUSED);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CLOSE failed", msg_size);
          error_out (rc) ;
       }

/*------------------------------------------------------------------*/
/* De_allocate the connection handle                                */
/*------------------------------------------------------------------*/

       rc = ct_con_drop (connection);
```

```
        if (rc == CS_FAIL)
        {
           strncpy (msgstr, "CT_CON_DROP failed", msg_size);
           error_out (rc) ;
        }
 }   /* end close_connection */

/*******************************************************************/
/*                                                                 */
/* Subroutine to perform exit client library and deallocate context */
/* structure.                                                      */
/*                                                                 */
/*******************************************************************/
 void   quit_client_library ()

 {
  CS_INT        rc;
 /*-----------------------------------------------------------------*/
 /* Exit the Client Library                                         */
 /*-----------------------------------------------------------------*/

        rc = ct_exit (context, (long) CS_UNUSED);

        if (rc == CS_FAIL)
        {
           strncpy (msgstr, "CT_EXIT failed", msg_size);
           error_out(rc) ;
         }

 /*-----------------------------------------------------------------*/
 /* De-allocate the context structure                               */
 /*-----------------------------------------------------------------*/

        rc = cs_ctx_drop (context);

        if (rc == CS_FAIL)
        {
           strncpy (msgstr, "CT_CTX_DROP failed", msg_size);
           error_out(rc) ;
        }

        EXEC CICS RETURN ;

 }   /* end quit_client_library */

 /*******************************************************************/
```

```
/*                                                                    */
/* Subroutine to display output                                       */
/*                                                                    */
/******************************************************************-*/
void   disp_data()

{
   CS_SMALLINT    QF_LEN;
   CS_SMALLINT    QF_MAXLEN;
   CS_CHAR        QF_ANSWER;
   CS_SMALLINT    CICS_RESPONSE;


        strncpy (OUT.stimeo, TMP_TIME, 8);
        strncpy (OUT.sdateo, TMP_DATE, 8);
        strncpy (OUT.prognmo, "SYCTSAA6", 8);

       switch (page_cnt)
       {
          case 1:
                 strcpy (OUT.spageo, "0001");  break;
          case 2:
                 strcpy (OUT.spageo, "0002");  break;
          case 3:
                 strcpy (OUT.spageo, "0003");  break;
          case 4:
                 strcpy (OUT.spageo, "0004");  break;
          case 5:
                 strcpy (OUT.spageo, "0005");  break;
          case 6:
                 strcpy (OUT.spageo, "0006");  break;
          case 7:
                 strcpy (OUT.spageo, "0007");  break;
          case 8:
                 strcpy (OUT.spageo, "0008");  break;
          case 9:
                 strcpy (OUT.spageo, "0009");  break;
          default:
                 strcpy (OUT.spageo, "9999");
       }

        OUT.servera = DFHBMPRO;
        strncpy (OUT.servero, servname, server_size);

        OUT.usera   = DFHBMPRO;
        strncpy (OUT.usero, username, user_size);
```

```
        OUT.pswda   = DFHBMDAR;
        strncpy (OUT.pswdo, pwd, pwd_size);

        OUT.trana   = DFHBMPRO;
        strncpy (OUT.trano, tran, tran_size);

        OUT.netdrva = DFHBMPRO;
        strncpy (OUT.netdrvo, driver, 9);

        memcpy (OUT.msg1o, msgtext1, text_size);
        strncpy (OUT.msg2o, msgtext2, text_size);



/*------------------------------------------------------------------*/
/* DISPLAY THE DATA                                                 */
/*------------------------------------------------------------------*/

        EXEC CICS SEND MAP("SYCTBA6")
                       FROM(a6panel)
                       CURSOR
                       FRSET
                       ERASE
                       FREEKB ;

        EXEC CICS RECEIVE INTO(QF_ANSWER)
                          LENGTH(QF_LEN)
                          MAXLENGTH(QF_MAXLEN)
                          RESP(CICS_RESPONSE) ;

}   /* end disp_data */
```

APPENDIX B Sample RPC Application

This appendix contains a sample Open ClientConnect application program, SYCTSAR6, that sends an RPC to either an Adaptive Server Enterprise or Open ServerConnect running in a CICS/IMS region.

The purpose of this sample program is to demonstrate the use of Client-Library functions, particularly those designed to send language requests. In some cases, one Client-Library function is used for demonstration purposes when another function would be more efficient. To best illustrate the flow of processing, the program does not do extensive error checking.

The remote procedure or transaction initiated by this RPC is called SYR2, which uses data from the sample table SYBASE.SAMPLETB.

- If your server is an Adaptive Server Enterprise, the remote procedure and table must be created by Transaction Router Service (TRS). A script is provided with TRS that the TRS administrator can use to create SYR2 and the sample table on Adaptive Server Enterprise.
- If your remote server is Open ServerConnect, SYR2 and SYBASE.SAMPLETB are provided on the product API tape.

SYCTSAR6 is provided as part of the Open ClientConnect package on the API tape.

The following additional sample applications are provided as part of the Open ClientConnect package on the API tape:

- SYCTSAD6, which demonstrates how to send an RPC request with parameters to an Adaptive Server Enterprise or Open ServerConnect running in a CICS region.
- SYCTSAV6, which demonstrates how to send an RPC request with parameters to an Adaptive Server Enterprise or Open ServerConnect running in a CICS/IMS region.

Sample program – SYCTSAR6

```
/*        @(#) syctsar6.c   1.1 10/14/96   */

 /****** SYCTSAR6 - Client RPC Request APPL - C - CICS ***************/
 /*                                                                   */
 /*  CICS TRANID:    SYR6                                             */
 /*                                                                   */
 /*  PROGRAM:       SYCTSAR6                                          */
 /*                                                                   */
 /*  PURPOSE:  Demonstrates Open Client for CICS CALLs.                */
 /*                                                                   */
 /*  FUNCTION: Illustrates how to send an RPC request with            */
 /*            parameters to:                                         */
 /*                                                                   */
 /*              - A SQL Server                                       */
 /*              - An Open Server running in a CICS/IMS region.       */
 /*                                                                   */
 /*            SQL Server:                                            */
 /*                                                                   */
 /*              If the request is sent to a SQL Server it            */
 /*              intitiates the stored procedure "SYR2".              */
 /*                                                                   */
 /*               Note: The Net-Gateway/MCG product includes a script*/
 /*                     that creates this procedure in a target SQL  */
 /*                     server.                                       */
 /*                                                                   */
 /*            Open Server/CICS or Open Server/IMS:                   */
 /*                                                                   */
 /*              If the request is sent to an Open Server/CICS or    */
 /*              IMS region, it invokes the transaction SYR2.        */
 /*                                                                   */
 /*              Note: The Open Server/CICS and IMS products include */
 /*                    the sample transaction SYR2. This is the      */
 /*                    server side transaction invoked by this       */
 /*                    program.                                       */
 /*                                                                   */
 /*  PREREQS:  Before running SYCTSAR6, make sure that the server    */
 /*            you wish to access has an entry in the Connection     */
 /*            Router Table for that Server and the MCG(s) that      */
 /*            you wish to use.                                       */
 /*                                                                   */
 /*  INPUT:    On the input screen, make sure to enter the Server    */
 /*            name, user id, and password for the target server     */
 /*            TRAN NAME is not used for LAN servers.                 */
 /*            Enter SYR2 in the TRAN NAME field if the server is    */
```

```
/*            in a CICS or IMS region.                              */
/*                                                                  */
/*  Open Client CALLs used in this sample:                          */
/*                                                                  */
/*    cs_ctx_alloc  allocate a context                              */
/*    cs_ctx_drop   drop a context                                  */
/*    ct_bind       bind a column variable                          */
/*    ct_close      close a server connection                       */
/*    ct_cmd_alloc  allocate a command                              */
/*    ct_cmd_drop   drop a command                                  */
/*    ct_command    initiate remote procedure call                  */
/*    ct_con_alloc  allocate a connection                           */
/*    ct_con_drop   drop a connection                               */
/*    ct_con_props  alter properties of a connection                */
/*    ct_connect    open a server connection                        */
/*    ct_describe   return a description of result data             */
/*    ct_diag       retrieve SQLCODE messages                       */
/*    ct_exit       exit client library                             */
/*    ct_fetch      fetch result data                               */
/*    ct_init       init client library                             */
/*    ct_param      define a command parameter                      */
/*    ct_results    sets up result data                             */
/*    ct_send       send a request to the server                    */
/*                                                                  */
/* History:                                                         */
/*                                                                  */
/* Date    BTS#   Descrition                                        */
/* ======= ====== ================================================= */
/* Sept 96        Create                                            */
/*                                                                  */
/*                                                                  */
/********************************************************************/
#pragma csect(code,"SYCTSAR6")
#pragma linkage(SYCVTD,  OS)
#pragma runopts(env(IMS),plist(IMS))
#pragma runopts(NOSPIE,NOSTAE)

/*------------------------------------------------------------------*/
/* CLIENT LIBRARY C COPY BOOK                                       */
/*------------------------------------------------------------------*/

#include "ctpublic.h"

/*------------------------------------------------------------------*/
/* CICS BMS DEFINITIONS C COPY BOOK                                 */
/*------------------------------------------------------------------*/
```

```
#include "syctba6.h"

/*------------------------------------------------------------------*/
/* DEFINES                                                          */
/*------------------------------------------------------------------*/

#define FALSE          0
#define TRUE           1
#define IN  a6panel.a6paneli
#define OUT a6panel.a6panelo
#define RS  a6panel.a6panelo.rsltne


/*------------------------------------------------------------------*/
/* GLOBALS - CLIENT COMMON WORK AREA                                */
/*------------------------------------------------------------------*/
   /*
   ** Open Client variables
   */

 CS_CONTEXT    *context;          /* pointer to context handle    */
 CS_CONNECTION *connection;       /* pointer to connection handle */
 CS_COMMAND    *cmd;              /* pointer to command proc      */

   CS_INT      version;           /* CT version #       */
   CS_CHAR     servname[30];      /* Server name        */
   CS_CHAR     username[8];       /* User login name    */
   CS_CHAR     pwd[8];            /* Password           */
   CS_CHAR     tran[8];           /* Transaction name   */
   CS_CHAR     driver[9];         /* Network driver     */
   CS_INT      server_size;
   CS_INT      user_size;
   CS_INT      pwd_size;
   CS_INT      tran_size;

   CS_CHAR     msgstr[40];        /* message string     */
   CS_INT      msg_size = 40;     /* error message size */
   CS_CHAR     msgtext1[79];
   CS_CHAR     msgtext2[79];
   CS_INT      text_size = 79;
   CS_INT      page_cnt = 0;
   CS_INT      row_num  = 0;
   CS_INT      results_type;

   /*
```

```
   ** Column variables for returned data
   */
   CS_VARCHAR  col_firstnme;
   CS_VARCHAR  col_lastname;
   CS_SMALLINT col_educlvl;
   CS_CHAR     col_jobcode[8];
   CS_CHAR     col_salary[12];
   CS_INT      ret_parm1;

   /*
   **  Variables to control program flow
   */
   CS_SMALLINT   no_input = 1;
   CS_SMALLINT   no_errors_sw = 1;
   CS_SMALLINT   no_more_results = 0;
   CS_SMALLINT   diag_msgs_initialized = 0;
   CS_INT        print_once = 1;

   /*
   **  Variables for convertion to display
   */
   CS_INT      cvtleft;
   CS_INT      cvtright;
   double      cvtdbl;
   char        cvtwork[17];

   /*
   **  CICS variables for
   **  System date and time
   */
   char        TMP_DATE[8] = "        ";
   char        TMP_TIME[8] = "        ";
   char           UTIME[8];


/*-------------------------------------------------------------------*/
/* Standard CICS Attribute and Print Control Chararcter List        */
/*-------------------------------------------------------------------*/

#include <dfhbmsca.h>                  /* BMS definitions            */

/*-------------------------------------------------------------------*/
/* CICS Standard Attention Identifiers C Copy Book                   */
/*-------------------------------------------------------------------*/

#include <dfhaid.h>                    /* PFK definitions            */
```

```
/*------------------------------------------------------------------*/
/* Local Functions Prototypes                                       */
/*------------------------------------------------------------------*/

void   proces_input();
void   bind_columns();
void   send_command();
void   proces_results();
void   result_row_processing();
void   fetch_row_processing();
void   result_param_processing ();
void   fetch_param_processing ();
void   error_out();
void   get_diag_messages();
void   get_client_msgs();
void   get_server_msgs();
void   close_connection();
void   quit_client_library();
void   display_initial_screen();
void   get_input_data();
void   disp_data();

/********************************************************************/
/* Main routine                                                     */
/********************************************************************/

main ()

{
 CS_INT         rc;
 CS_INT         i;


 EXEC CICS ADDRESS EIB(dfheiptr);

/*------------------------------------------------------------------*/
/* CICS Aid Handler -- not support in C language                    */
/*------------------------------------------------------------------*/

/*      EXEC CICS HANDLE AID ANYKEY(NO_INPUT)
                             CLEAR(GETOUT) ; */

/*------------------------------------------------------------------*/
/* Program initialization                                           */
/*------------------------------------------------------------------*/
```

```
        strncpy(msgtext1, " ", text_size);
        strncpy(msgtext2, "Press Clear To Exit", text_size);
        page_cnt = page_cnt + 1;
        memset(servname, ' ', 30);
        memset(username, ' ', 8);
        memset(pwd, ' ', 8);
        memset(tran, ' ', 8);
        memset(driver, ' ', 9);

        for (i = 0; i < 14; ++i)
           memset (RS[i].rsltno, ' ', text_size); /* init output lines */

        /*  get system time   */
         EXEC CICS ASKTIME ABSTIME(UTIME);

         EXEC CICS FORMATTIME
                   ABSTIME(UTIME)
                   DATESEP('/')
                   MMDDYY(TMP_DATE)
                   TIME(TMP_TIME)
                   TIMESEP ;

        display_initial_screen ();
        get_input_data ();

/*---------------------------------------------------------------------*/
/* Allocate a context structure                                        */
/*---------------------------------------------------------------------*/

      if (no_input == FALSE)
      {
        version = CS_VERSION_50;

        rc = cs_ctx_alloc (version, &context);

        if (rc != CS_SUCCEED)
        {
           strncpy(msgstr,"CSCTXALLOC failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
           EXEC CICS RETURN ;
        }
/*---------------------------------------------------------------------*/
/* Initialize the Client-Library                                       */
/*---------------------------------------------------------------------*/
```

```
/* context allocated, it's now OK to use ct_diag for message
            retrieving                                           */
         diag_msgs_initialized = 1;

         rc = ct_init (context, version);

         if (rc == CS_SUCCEED)
         {
            proces_input ();
         }
         else
         {
            strncpy (msgstr, "CT_INIT failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }
         close_connection ();
         quit_client_library ();
      }  /* process input data */

      else  /* no input data */
         EXEC CICS RETURN ;

  }   /* end main */


/********************************************************************/
/*                                                                  */
/* Subroutine to display the initial screen                         */
/*                                                                  */
/********************************************************************/
  void   display_initial_screen ()

  {
          strncpy (OUT.sdateo, TMP_DATE, 8);
          strncpy (OUT.stimeo, TMP_TIME, 8);
          strncpy (OUT.prognmo, "SYCTSAR6", 8) ;
          strncpy (OUT.msg1o, msgtext1, text_size);
          strncpy (OUT.msg2o, msgtext2, text_size);
          strncpy (OUT.spageo, "0001", 4) ;

          EXEC CICS SEND MAP("SYCTBA6")
                         FROM(a6panel)
                         CURSOR(252)
                         FRSET
                         ERASE
```

```
                        FREEKB ;

}   /* end display_initial_screen  */

/*********************************************************************/
/*                                                                   */
/* Subroutine to get input data                                      */
/*                                                                   */
/*********************************************************************/
void   get_input_data ()
{
 CS_SMALLINT   enter_data_sw = 0;
 CS_CHAR       blank30[30] = "                              ";
 CS_CHAR       blank8[8] = "        ";
 int           i;


        EXEC CICS RECEIVE MAP("a6panel")
                          MAPSET("SYCTBA6")
                          ASIS ;

        if (IN.serverl == 0)
        {
           if (strncmp (servname, blank30, 30) == 0)
           {
              IN.serverl    = -1 ;        /* set the cursor position */
              strncpy (msgtext1, "Please Enter Server Name", text_size);
              enter_data_sw  = TRUE ;
           }
        }
        else if (IN.serverl > 0)
        {
           server_size = IN.serverl ;
           strncpy (servname, IN.serveri, server_size);
           no_input = 0;
        }

        if (IN.userl == 0)
        {
           if (strncmp (username, blank8, 8) == 0)
           {
              IN.userl       = -1 ;        /* set the cursor position */
              if (enter_data_sw == FALSE)  /* not overlaid msgtext1   */
                 strncpy (msgtext1, "Please Enter User-ID", text_size);
              enter_data_sw  = TRUE ;
           }
```

```
          }
          else
          {
             user_size = IN.userl ;
             strncpy (username, IN.useri, user_size);
          }

          if (IN.pswdl != 0)
             {
                pwd_size = IN.pswdl ;
                strncpy (pwd, IN.pswdi, pwd_size);
             }

          if (IN.tranl != 0)
             {
                tran_size = IN.tranl ;
                strncpy (tran, IN.trani, tran_size);
             }

          if (IN.netdrvl != 0)
             {
                strncpy (driver, IN.netdrvi, 9);
                for (i=0; i<9; ++i)
                    driver[i] = toupper (driver[i]);
             }

          if (enter_data_sw == TRUE)              /* bad input data  */
             {
                enter_data_sw = FALSE ;
                display_initial_screen ();       /* request for input again */
                get_input_data ();
             }

  }   /*  end get_input_data  */
/**********************************************************************/
/*                                                                    */
/* Subroutine to process input data                                   */
/*                                                                    */
/**********************************************************************/

  void   proces_input ()

  {
   CS_INT        rc;
   CS_INT        *outlen;
   CS_INT        buf_len;
```

```
 CS_INT        msglimit;
 CS_INT        netdriver;

/*------------------------------------------------------------*/
/* Allocate a connection to the server                        */
/*------------------------------------------------------------*/

       rc = ct_con_alloc (context, &connection);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CONALLOC failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
        }

/*------------------------------------------------------------*/
/* Alter properties of the connection for user-id             */
/*------------------------------------------------------------*/

       buf_len  = user_size;
       rc = ct_con_props (connection, (long)CS_SET,
                          (long)CS_USERNAME, username,
                          buf_len, outlen);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_CON_PROPS for user-id failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
        }

/*------------------------------------------------------------*/
/* Alter properties of the connection for password            */
/*------------------------------------------------------------*/

       buf_len = pwd_size;
       rc = ct_con_props (connection, (long)CS_SET,
                         (long)CS_PASSWORD, pwd, buf_len, outlen);

        if (rc != CS_SUCCEED)
        {
          strncpy (msgstr, "CT_CON_PROPS for password failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
        }
```

```
/*------------------------------------------------------------*/
/* Alter properties of the connection for transaction         */
/*------------------------------------------------------------*/

         buf_len = tran_size;
         rc = ct_con_props (connection, (long)CS_SET,
                            (long)CS_TRANSACTION_NAME,
                            tran, buf_len, outlen);

         if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CON_PROPS for transaction failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
         }

  /*------------------------------------------------------------*/
  /* Alter properties of the connection for network driver      */
  /*------------------------------------------------------------*/

         netdriver = 9999;   /* default value for non-regconized
                                driver name                     */

         /* if no netdriver entered, default is LU62  */
         if (strncmp(driver,"         ",9) == 0  ??
             strncmp(driver,"LU62",4) == 0)
            netdriver = CS_LU62;

         else if (strncmp(driver,"INTERLINK",8) == 0)
            netdriver = CS_INTERLINK;

         else if (strncmp(driver,"IBMTCPIP",8) == 0)
            netdriver = CS_TCPIP;

         else if (strncmp(driver,"CPIC",4) == 0)
            netdriver = CS_NCPIC;

         rc = ct_con_props (connection, (long)CS_SET,
                            (long)CS_NET_DRIVER, (long) netdriver,
                            CS_UNUSED, outlen);

         if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CON_PROPS for network driver failed",
 msg_size);
            no_errors_sw = FALSE ;
```

```
            error_out (rc);
        }

/*------------------------------------------------------------*/
/* Setup retrieval of All Messages                            */
/*------------------------------------------------------------*/

        rc = ct_diag (connection, CS_INIT,
                      CS_UNUSED, CS_UNUSED, CS_NULL);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_DIAG CS_INIT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }

/*------------------------------------------------------------*/
/* Set the upper limit of number of messages                  */
/*------------------------------------------------------------*/

        msglimit = 5 ;

        rc = ct_diag (connection, CS_MSGLIMIT, CS_ALLMSG_TYPE,
                      CS_UNUSED, &msglimit);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_DIAG CS_MSGLIMIT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }

/*------------------------------------------------------------*/
/* Open connection to the server or CICS region               */
/*------------------------------------------------------------*/

        rc = ct_connect (connection, servname, server_size);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_CONNECT failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
```

```
/*-------------------------------------------------------------*/
/* Invokes SEND_COMMAND routine                                */
/*-------------------------------------------------------------*/

         if (no_errors_sw)
            send_command ();

  /*-------------------------------------------------------------*/
  /* Process the results of the command                          */
  /*-------------------------------------------------------------*/

         if (no_errors_sw)
         {
            while (no_more_results == FALSE)
               proces_results ();
         }
  }   /* end proces_input */

  /*********************************************************************/
  /*                                                                   */
  /* Subroutine to allocate, send, and process commands                */
  /*                                                                   */
  /*********************************************************************/
  void send_command ()

  {
   CS_INT         rc;
   CS_INT         *outlen;
   CS_INT         buf_len;
   CS_SMALLINT    nullind = 0;
   CS_CHAR        rpc_cmd[4] = "SYR2";
   CS_INT         param1 = 0;
   CS_CHAR        param2[3] = "D11";
   CS_DATAFMT     datafmt;

  /*-------------------------------------------------------------*/
  /* Allocate a command handle                                   */
  /*-------------------------------------------------------------*/

         rc = ct_cmd_alloc (connection, &cmd);

         if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_CMDALLOC failed", msg_size);
            no_errors_sw = FALSE ;
            error_out (rc);
```

```
        }

  /*-----------------------------------------------------------*/
  /* Prepare the command (an RPC request)                      */
  /*-----------------------------------------------------------*/

        buf_len = 4;

        rc = ct_command(cmd, (long) CS_RPC_CMD, rpc_cmd,
                        buf_len, (long) CS_UNUSED);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_COMMAND failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
  /*-----------------------------------------------------------*/
  /*                                                           */
  /* Setup a return parameter for NUM_OF_ROWS                  */
  /*                                                           */
  /* Describe the first parameter (NUM_OF_ROWS)                */
  /*                                                           */
  /*-----------------------------------------------------------*/
        strcpy (datafmt.name, "@parm1");
        datafmt.namelen      = 6;
        datafmt.datatype     = CS_INT_TYPE;
        datafmt.format       = CS_FMT_UNUSED;
        datafmt.maxlength    = CS_UNUSED;
        datafmt.status       = CS_RETURN;
        datafmt.usertype     = CS_UNUSED;

        buf_len   = sizeof(param1);
        rc = ct_param (cmd, datafmt, param1, buf_len, nullind);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_PARAM CS_INT_TYPE parm1 failed", msg_size) ;
           no_errors_sw = FALSE ;
           error_out (rc);
        }
/*-----------------------------------------------------------*/
/*                                                           */
/* Describe the second parameter (DEPTNO)                    */
/*                                                           */
/*-----------------------------------------------------------*/
```

```
        strcpy (datafmt.name, "@parm2");
        datafmt.namelen      = 6;
        datafmt.datatype     = CS_VARCHAR_TYPE;
        datafmt.format       = CS_FMT_UNUSED;
        datafmt.maxlength    = CS_UNUSED;
        datafmt.status       = CS_INPUTVALUE;
        datafmt.usertype     = CS_UNUSED;

        buf_len   = sizeof(param2);

        rc = ct_param (cmd, datafmt, param2, buf_len, nullind);

        if (rc != CS_SUCCEED)
        {
          strncpy (msgstr, "CT_PARAM CS_VARCHAR_TYPE parm2 failed", msg_size)
;
           no_errors_sw = FALSE ;
           error_out (rc);
        }

 /*------------------------------------------------------------*/
 /* Send the command                                           */
 /*------------------------------------------------------------*/

        rc = ct_send (cmd);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_SEND failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }
 }   /* end send_command */
 /******************************************************************/
 /*                                                                */
 /* Subroutine to process the result                               */
 /*                                                                */
 /******************************************************************/
 void   proces_results ()

 {
  CS_INT        rc;
 /*------------------------------------------------------------*/
 /* Set up the results data                                    */
 /*------------------------------------------------------------*/
```

```
       rc = ct_results (cmd, &results_type);

/*------------------------------------------------------------*/
/* Determine the outcome of the comand execution              */
/*------------------------------------------------------------*/

       switch (rc)
       {
          case CS_SUCCEED:

/*--------------------------------------------------------------*/
/* Determine the type of result returned by the current request */
/*--------------------------------------------------------------*/

               switch (results_type)
               {

/*------------------------------------------------------------*/
/* Process row results                                        */
/*------------------------------------------------------------*/

                  case CS_ROW_RESULT:
                       result_row_processing ();
                       fetch_row_processing ();
                       break;

/*------------------------------------------------------------*/
/* Process parameter results --- there should be no parameter */
/* to process                                                 */
/*------------------------------------------------------------*/

                  case CS_PARAM_RESULT:
                       no_more_results = FALSE;
                       result_param_processing ();
                       fetch_param_processing ();
                       break;

/*------------------------------------------------------------*/
/* Process status results --- the stored procedure status     */
/* result will not be processed in this example               */
/*------------------------------------------------------------*/

                  case CS_STATUS_RESULT:
                       no_more_results = FALSE;
                       break;
```

```
 /*------------------------------------------------------------*/
 /* Print an error message if the server encountered an error  */
 /* while executing the request                                 */
 /*------------------------------------------------------------*/

                  case CS_CMD_FAIL:
                       no_errors_sw = FALSE ;
                       strncpy (msgstr,
                         "CT_RESUL returned CS_CMD_FAIL restype", msg_size);
                       error_out (rc);
                       break;

/*------------------------------------------------------------*/
/* Print a message for successful commands that returned no   */
/* data( optional )                                            */
/*------------------------------------------------------------*/

                  case CS_CMD_SUCCEED:
                       strncpy (msgstr,
                         "CT_RESUL returned CS_CMD_SUCCEED restype",
msg_size);
                       break;

 /*------------------------------------------------------------*/
 /* Print a message for requests that have been processed      */
 /* successfully( optional )                                    */
 /*------------------------------------------------------------*/

                  case CS_CMD_DONE:
                       strncpy (msgstr,
                         "CT_RESUL returned CS_CMD_DONE restype", msg_size);
                       break;

                  default:
                       no_more_results = TRUE ;
                       no_errors_sw    = FALSE ;
                       strncpy (msgstr,
                         "CT_RESUL returned UNKNOWN restype", msg_size);
                       error_out (rc);

               }   /* end of switch (result_type) */
               break;
 /*------------------------------------------------------------*/
 /* Print an error message if the CTBRESULTS call failed       */
 /*------------------------------------------------------------*/
```

```
          case CS_FAIL:
               no_more_results = TRUE ;
               no_errors_sw    = FALSE ;
               strncpy (msgstr, "CT_RESULTS returned CS_FAIL ret_code",
                        msg_size);
               error_out (rc);
               break;

/*------------------------------------------------------------*/
/* Drop out of the results loop if no more result sets are    */
/* available for processing or if the results were cancelled  */
/*------------------------------------------------------------*/

          case CS_END_RESULTS:
          case CS_CANCELLED:
               no_more_results = TRUE ;
               break;

          default:
               no_more_results = TRUE ;
               no_errors_sw    = FALSE ;
               strncpy (msgstr, "CT_RESULTS returned unknown ret_code",
                        msg_size);
               error_out (rc);
               break;

       }  /* end of switch (rc) */

}   /* end process_results */

/******************************************************************/
/*                                                                */
/* Subroutine to process result rows                              */
/*                                                                */
/******************************************************************/
void   result_row_processing()

{
 CS_INT        rc;
 CS_INT        numcol = 5;
 CS_INT        parm_cnt;

        for (parm_cnt = 1; parm_cnt <= numcol; ++parm_cnt)
            bind_columns (parm_cnt);
```

```
}    /* end result_row_processing */

/********************************************************************/
/*                                                                  */
/* Subroutine to bind each data                                     */
/*                                                                  */
/********************************************************************/

void   bind_columns (parm_cnt)

CS_INT        parm_cnt;
{
 CS_INT        rc;
 CS_INT        col_len;
 CS_DATAFMT    datafmt;


/*------------------------------------------------------------*/
/* bind the first column, FIRSTNME defined as VARCHAR(12)     */
/* and the second column, LASTNAME defined as VARCHAR(15)     */
/*------------------------------------------------------------*/
       memset (&datafmt, '\0', sizeof(datafmt));

       switch (parm_cnt)
       {
           case 1:
           case 2:

              datafmt.datatype    = CS_VARCHAR_TYPE;
              datafmt.format      = CS_FMT_PADBLANK;

              if (parm_cnt == 1)
              {
                 datafmt.maxlength   = 12;
                 rc= ct_bind(cmd, parm_cnt, &datafmt, &col_firstnme,
                             &col_len, CS_NULL);
              }
              else
              {
                 datafmt.maxlength   = 15;
                 rc= ct_bind(cmd, parm_cnt, &datafmt, &col_lastname,
                             &col_len, CS_NULL);
              }

              if (rc != CS_SUCCEED)
              {
```

```
                 strncpy (msgstr, "CT_BIND CS_VARCHAR_TYPE failed",
msg_size);
                 no_errors_sw = FALSE ;
                 error_out (rc);
              }
              break;

 /*------------------------------------------------------------*/
 /* bind the third column, EDUCLVL defined as SMALLINT         */
 /*------------------------------------------------------------*/

           case 3:

              datafmt.datatype    = CS_SMALLINT_TYPE;
              datafmt.maxlength   = 2;

              rc= ct_bind(cmd, parm_cnt, &datafmt, &col_educlvl,
                          &col_len, CS_NULL);

              if (rc != CS_SUCCEED)
              {
                 strncpy (msgstr, "CT_BIND CS_SMALLINT_TYPE failed",
msg_size);
                 no_errors_sw = FALSE ;
                 error_out(rc);
              }
              break;

 /*------------------------------------------------------------*/
 /* bind the fourth column, JOBCODE as CHAR                    */
 /* bind the fifth column, SALARY as CHAR. It will be          */
 /* converted from float8 or money.                            */
 /*------------------------------------------------------------*/

           case 4:
           case 5:

              datafmt.datatype    = CS_CHAR_TYPE;
              datafmt.format      = CS_FMT_UNUSED;
              datafmt.scale       = CS_SRC_VALUE;
              datafmt.precision   = CS_SRC_VALUE;

              if (parm_cnt == 4)
              {
                 datafmt.maxlength   = sizeof(col_jobcode);
                 rc= ct_bind(cmd, parm_cnt, &datafmt, &col_jobcode,
```

```
                              &col_len, CS_NULL);
               }
               else
               {
                  datafmt.maxlength   = sizeof(col_salary);
                  rc= ct_bind(cmd, parm_cnt, &datafmt, &col_salary,
                              &col_len, CS_NULL);
               }

               if (rc != CS_SUCCEED)
               {
                  strncpy (msgstr,"CT_BIND CS_CHAR failed", msg_size);
                  no_errors_sw = FALSE ;
                  error_out (rc);
               }
               break;

            default:
               break;

         } /* end of switch (parm_cnt) */

  }   /* end bind_columns */

  /*********************************************************************/
  /*                                                                   */
  /* Subroutine to fetch row processing                                */
  /*                                                                   */
  /*********************************************************************/
  void   fetch_row_processing ()

  {
   CS_INT         rows_read;
   CS_INT         rc;
   CS_INT         col_len;
   CS_INT         max_screen_rows = 10;
   CS_SMALLINT    nomore_rows = 0;
   struct {
           char  first[14];
           char  last[17];
           char  educ[10];
           char  jobcode[8];
           char  salary[12];
           } output_row;
```

```
      while (nomore_rows == FALSE)
      {
        rc = ct_fetch (cmd, (long) CS_UNUSED,
                       (long) CS_UNUSED,
                       (long) CS_UNUSED,
                       &rows_read);

        switch (rc)
        {
            case  CS_SUCCEED:
                nomore_rows = FALSE ;
                if (row_num == 0)
                {
/*------------------------------------------------------------*/
/* Setup column headings for the 1st screen                   */
/*------------------------------------------------------------*/
                   row_num += 1;
                   strcpy (RS[row_num].rsltno, "FirstName     ");
                   strcat (RS[row_num].rsltno, "LastName         ");
                   strcat (RS[row_num].rsltno, "EducLvl   ");
                   strcat (RS[row_num].rsltno, "JobCode    ");
                   strcat (RS[row_num].rsltno, "Salary  ");

                   row_num += 1;
                   strcpy (RS[row_num].rsltno, "==========    ");
                   strcat (RS[row_num].rsltno, "=============    ");
                   strcat (RS[row_num].rsltno, "========  ");
                   strcat (RS[row_num].rsltno, "========   ");
                   strcat (RS[row_num].rsltno, "========= ");
                }

                row_num += 1;

                if (row_num > max_screen_rows)
                {
                   strncpy (msgtext1, "Please press return to continue!",
                            text_size);
                   memset (msgtext2, ' ', text_size);
                   disp_data ();

                   /* re-init output lines */
                   for (row_num = 0; row_num < 14; ++row_num)
                       strncpy (RS[row_num].rsltno, " ", text_size);

                   row_num = 1;
                   page_cnt = page_cnt + 1 ;
```

```
                strcpy (RS[row_num].rsltno, "FirstName     ");
                strcat (RS[row_num].rsltno, "LastName         ");
                strcat (RS[row_num].rsltno, "EducLvl   ");
                strcat (RS[row_num].rsltno, "JobCode    ");
                strcat (RS[row_num].rsltno, "Salary  ");

                row_num += 1;
                strcpy (RS[row_num].rsltno, "==========    ");
                strcat (RS[row_num].rsltno, "=============    ");
                strcat (RS[row_num].rsltno, "========  ");
                strcat (RS[row_num].rsltno, "========   ");
                strcat (RS[row_num].rsltno, "========= ");

                row_num += 1;
             }

/*------------------------------------------------------------*/
/* Display results                                            */
/*------------------------------------------------------------*/


             memset (&output_row, ' ', sizeof(output_row));
             strcpy (output_row.first, col_firstnme.str);
             strcpy (output_row.last, col_lastname.str);

             cvtleft  = 4;           /* Digits to the left  */
             cvtright = 0;           /* Digits to the right */
             SYCVTD(col_educlvl, output_row.educ,
                     cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

             strcpy (output_row.jobcode, col_jobcode);
             strcpy (output_row.salary, col_salary);

             memcpy (RS[row_num].rsltno, &output_row, sizeof(output_row));

             break; /* end of CS_SUCCEED */

         case  CS_END_DATA:
             nomore_rows = TRUE ;
             strncpy (msgtext1, "All rows processing completed!",
                      text_size);
             strncpy (msgtext2, "Press Clear To Exit");
             break; /* end of CS_END_DATA  */

         case  CS_FAIL:
```

```
                nomore_rows = TRUE ;
                no_errors_sw = FALSE ;
                strncpy (msgstr, "CT_FETCH returned CS_FAIL ret_code");
                error_out(rc);
                break; /* end of CS_FAIL  */

            case  CS_ROW_FAIL:
                nomore_rows = TRUE ;
                no_errors_sw = FALSE ;
                strncpy (msgstr, "CT_FETCH returned CS_ROW_FAIL ret_code");
                error_out(rc);
                break; /* end of CS_ROW_FAIL  */

            case  CS_CANCELLED:
                nomore_rows = TRUE ;
                no_errors_sw = FALSE ;
                strncpy (msgstr, "CT_FETCH returned CS_CANCELLED ret_code");
                error_out(rc);
                break; /* end of CS_CANCELLED  */

            default:
                nomore_rows = TRUE ;
                no_errors_sw = FALSE ;
                strncpy (msgstr, "CT_FETCH returned Unknown ret_code");
                error_out(rc);
                break; /* end of OTHERWISE */

        }  /* end of switch (rc) */
      }   /* end of while nomore_rows false */

}  /* end fetch_row_processing */

/********************************************************************/
/*                                                                  */
/* Subroutine to describe the returned parameters                   */
/*                                                                  */
/********************************************************************/
void   result_param_processing ()

{
 CS_INT        rc;
 CS_INT        parm = 1;
 CS_INT        col_len;
 CS_DATAFMT    datafmt;

/*------------------------------------------------------------*/
```

```
/*  Get desription of return parameter                           */
/*--------------------------------------------------------------*/

       rc= ct_describe(cmd, parm, &datafmt);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_DESCRIBE failed", msg_size);
          no_errors_sw = FALSE ;
          error_out (rc);
       }

/*--------------------------------------------------------------*/
/* bind the return parameter, NUM_OF_ROWS defined as INTEGER  */
/*--------------------------------------------------------------*/

        datafmt.datatype   = CS_INT_TYPE;
        datafmt.format     = CS_FMT_UNUSED;
        datafmt.maxlength  = CS_UNUSED;

        rc= ct_bind(cmd, parm, &datafmt, &ret_parm1,
                    &col_len, CS_NULL);

        if (rc != CS_SUCCEED)
        {
           strncpy (msgstr, "CT_BIND return parameter failed", msg_size);
           no_errors_sw = FALSE ;
           error_out (rc);
        }

}   /* end result_param_processing */

/*******************************************************************/
/*                                                                 */
/* Subroutine to fetch return parameter                            */
/*                                                                 */
/*******************************************************************/

void   fetch_param_processing ()

{
 CS_INT        rc;
 CS_INT        rows_read;
 struct {
    char  paren;
    char  col_ret[4];
```

```
    char  or2_msg[17];
        } output_row;


       memset (&output_row, ' ', sizeof(output_row));
       strcpy (output_row.or2_msg, " row(s) affected)");
       output_row.paren = '(';

       rc = ct_fetch (cmd, (long) CS_UNUSED,
                      (long) CS_UNUSED,
                      (long) CS_UNUSED,
                      &rows_read);

       if (rc != CS_SUCCEED)
         {
            strncpy (msgstr, "CT_FETCH return parameter failed", msg_size);
            no_errors_sw = FALSE ;
            error_out(rc);
         }

       if (ret_parm1 > 0)
         {
           cvtleft  = 4;       /* Digits to the left */
           cvtright = 0;       /* Digits to the right */
           SYCVTD(ret_parm1, output_row.col_ret,
                  cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

           row_num  += 2;
           memcpy (RS[row_num].rsltno, &output_row, sizeof(output_row));
           disp_data ();
         }

}   /* end fetch_param_processing  */

/********************************************************************/
/*                                                                  */
/* Subroutine to print output messages.                             */
/*                                                                  */
/********************************************************************/

void   error_out (rc)
 CS_INT        rc;

{
   /*
   **  Display Message
```

```
    */
    struct {
            char        test_case[9] ;
            char        samp_lit[5] ;
            char        samp_rc[4] ;
            char        rest_lit[15] ;
            char        rest_type[4] ;
            char        filler[2] ;
            char        msg[40];
    } disp_msg;


         memset(&disp_msg, ' ', sizeof(disp_msg));
         strcpy(disp_msg.test_case, "SYCTSAR6");
         strcpy(disp_msg.samp_lit, "rc = ");
         strcpy(disp_msg.rest_lit, "  Result Type: ");

         cvtleft  = 4;           /* Digits to the left */
         cvtright = 0;           /* Digits to the right */
         SYCVTD(rc, disp_msg.samp_rc,
                 cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
         SYCVTD(results_type, disp_msg.rest_type,
                 cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

         if (diag_msgs_initialized)
            get_diag_messages ();

  /*------------------------------------------------------------------*/
  /* display error messages                                           */
  /*------------------------------------------------------------------*/

         strcpy (disp_msg.msg, msgstr);
         memcpy (msgtext1, &disp_msg, sizeof(disp_msg));

         if (print_once)
           {
             disp_data ();
             print_once = FALSE ;
           }

  }   /* end error_out */

  /********************************************************************/
  /*                                                                  */
  /* Subroutine to retrieve any diagnostic messages                   */
  /*                                                                  */
```

```
/*******************************************************************/

void   get_diag_messages()

{
 CS_SMALLINT   cnt;
 CS_INT        num_of_msgs = 0;
 CS_INT        rc;

/*-----------------------------------------------------------------*/
/* Disable calls to this subroutine                                */
/*-----------------------------------------------------------------*/

        diag_msgs_initialized = FALSE ;



/*-----------------------------------------------------------------*/
/* First, get client messages                                      */
/*-----------------------------------------------------------------*/
       rc = ct_diag (connection, CS_STATUS, CS_CLIENTMSG_TYPE,
                     CS_UNUSED, #_of_msgs);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_DIAG CS_STATUS CLIENTMSG_TYPE failed",
                   msg_size);
          error_out(rc) ;
        }
        else if (num_of_msgs > 0)
        {
             for (cnt = 1; cnt <= num_of_msgs; ++cnt)
                 get_client_msgs ();
        }

/*-----------------------------------------------------------------*/
/* Then, get server messages                                       */
/*-----------------------------------------------------------------*/

       rc = ct_diag (connection, CS_STATUS, CS_SERVERMSG_TYPE,
                     CS_UNUSED, #_of_msgs);

       if (rc != CS_SUCCEED)
       {
          strncpy (msgstr, "CT_DIAG CS_STATUS SERVERMSG_TYPE failed",
                   msg_size);
```

```
          error_out(rc) ;
        }
        else if (num_of_msgs > 0)
        {
             for (cnt = 1; cnt <= num_of_msgs; ++cnt)
                 get_server_msgs ();
        }
}   /* end get_diag_messages  */

/*********************************************************************/
/*                                                                   */
/* Subroutine to retrieve diagnostic messages from client            */
/*                                                                   */
/*********************************************************************/
void   get_client_msgs()
{
 CS_INT        rc;
 CS_INT        i;
 CS_CHAR      *txtpos;
 CS_INT        textleft;
 CS_INT        msgno = 1;
 CS_CHAR       blank_13[13] = "             ";
 CS_CLIENTMSG  clientmsg;

 struct {
         char    msgno_hdr[13];
         char    msgno_data[8];
         char    severity_hdr[13];
         char    severity_data[6];
        } client_msg;


       rc = ct_diag (connection, CS_GET, CS_CLIENTMSG_TYPE,
                     msgno, &clientmsg);

       if (rc !=  CS_SUCCEED)
       {
          strncpy (msgstr, "CT_DIAG CS_GET CS_CLIENTMSG_TYPE failed",
                   msg_size);
          error_out(rc) ;
       }

/*-------------------------------------------------------------------*/
/* Display message text                                              */
/*-------------------------------------------------------------------*/
       i = 1 ;
```

```
strncpy (RS[i].rsltno, "Client Message:");
i = 3 ;

memset(&client_msg, ' ', sizeof(client_msg));
strcpy (client_msg.msgno_hdr, "  OC MsgNo:  ");
strcpy (client_msg.severity_hdr, "  Severity:  ");

cvtleft  = 8;           /* Digits to the left */
cvtright = 0;           /* Digits to the right */
SYCVTD(clientmsg.msgnumber, client_msg.msgno_data,
        cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
cvtleft  = 6;           /* Digits to the left */
SYCVTD(clientmsg.severity, client_msg.severity_data,
        cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

memcpy (RS[i].rsltno, &client_msg, sizeof(client_msg));
i += 1 ;

/* get number of Client msgs */

if (clientmsg.msgnumber != 0)
{
   strcpy (RS[i].rsltno, "  OC MsgTx:  ");
   strncat (RS[i].rsltno, clientmsg.msgstring, 66);
   i += 1 ;
   txtpos = clientmsg.msgstring + 66;
   textleft = clientmsg.msgstringlen - 66;
   while (textleft > 0)
   {
      strncpy (RS[i].rsltno, blank_13, 13);
      strncat (RS[i].rsltno, txtpos, 66);
      i += 1;
      txtpos += 66;
      textleft -= 66;
   }
}
else
{
   strncpy (RS[i].rsltno, "  OC MsgTx:  No Message!", text_size);
   i += 1 ;
}

/* get number of Server msgs */

memset(&client_msg, ' ', sizeof(client_msg));
strcpy (client_msg.msgno_hdr, "  OS MsgNo:  ");
```

```
        cvtleft  = 8;            /* Digits to the left */
        cvtright = 0;            /* Digits to the right */
        SYCVTD(clientmsg.osnumber, client_msg.msgno_data,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

        memcpy (RS[i].rsltno, &client_msg, sizeof(client_msg));
        i += 1 ;

        if (clientmsg.osnumber != 0)
        {
           strcpy (RS[i].rsltno, "  OS MsgTx:  ");
           strncat (RS[i].rsltno, clientmsg.osstring, 66);
           i += 1 ;
           txtpos = clientmsg.osstring + 66;
           textleft = clientmsg.osstringlen - 66;
           while (textleft > 0)
           {
              strncpy (RS[i].rsltno, blank_13, 13);
              strncat (RS[i].rsltno, txtpos, 66);
              i += 1;
              txtpos += 66;
              textleft -= 66;
           }
        }
        else
        {
           strncpy (RS[i].rsltno, "  OS MsgTx:  No Message!", text_size);
           i += 1 ;
        }

 }   /* end get_client_msgs */

/*********************************************************************/
/*                                                                   */
/* Subroutine to retrieve diagnostic messages from server           */
/*                                                                   */
/*********************************************************************/
void   get_server_msgs()
{
 CS_INT        rc;
 CS_INT        i;
 CS_CHAR      *txtpos;
 CS_INT        textleft;
 CS_INT        msgno = 1;
 CS_CHAR       blank_13[13] = "             ";
```

```
 CS_CHAR       proc_id_data[66];
 CS_CHAR       svrname_data[66];
 CS_SERVERMSG  servermsg;

 struct {
        char    msg_no_hdr[13];
        char    msg_no_data[6];
        char    severity_hdr[14];
        char    severity_data[6];
        char    state_hdr[14];
        char    state_data[4];
        char    line_no_hdr[13];
        char    line_no_data[4];
       } serv_msg;


      memset(&serv_msg, ' ', sizeof(serv_msg));

      rc = ct_diag (connection, CS_GET, CS_SERVERMSG_TYPE,
                    msgno, &servermsg);

      if (rc != CS_SUCCEED)
      {
         strncpy (msgstr, "CT_DIAG CS_GET CS_SERVERMSG_TYPE failed",
                  msg_size);
         error_out(rc) ;
      }

/*------------------------------------------------------------------*/
/* Display message text                                             */
/*------------------------------------------------------------------*/

      strcpy (serv_msg.msg_no_hdr, "  Message#:  ");
      strcpy (serv_msg.severity_hdr, ",  Severity:  ");
      strcpy (serv_msg.state_hdr, ",  State No:  ");
      strcpy (serv_msg.line_no_hdr, "  Line  No:  ");

      cvtleft  = 6;              /* Digits to the left */
      cvtright = 0;              /* Digits to the right */
      SYCVTD(servermsg.msgnumber, serv_msg.msg_no_data,
              cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
      SYCVTD(servermsg.severity, serv_msg.severity_data,
              cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

      cvtleft  = 4;              /* Digits to the left */
      SYCVTD(servermsg.state, serv_msg.state_data,
```

```
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);
        SYCVTD(servermsg.line, serv_msg.line_no_data,
                cvtleft, cvtright, cvtwork, cvtdbl, CS_TRUE);

        if (servermsg.svrnlen > 66)
        {
             strncpy (svrname_data, servermsg.svrname, 63);
             strcat (svrname_data, "...");
        }
        else
             strncpy (svrname_data, servermsg.svrname, 66);

        if (servermsg.proclen > 66)
        {
             strncpy (proc_id_data, servermsg.proc, 63);
             strcat (proc_id_data, "...");
        }
        else
             strncpy (proc_id_data, servermsg.proc, 66);

        strncpy (RS[1].rsltno, "Server Message:", text_size);
        memcpy (RS[3].rsltno, &serv_msg, sizeof(serv_msg));
        strcpy (RS[5].rsltno, "  Serv Nam:  ");
        strcat (RS[5].rsltno, svrname_data);
        strcpy (RS[6].rsltno, "  Proc ID :  ");
        strcat (RS[6].rsltno, proc_id_data);
        strcpy (RS[7].rsltno, "  Message :  ");
        strncat (RS[7].rsltno, servermsg.text, 66);

        i = 8 ;
        txtpos = servermsg.text + 66;
        textleft = servermsg.textlen - 66;
        while (textleft > 0)
           {
              strncpy (RS[i].rsltno, blank_13, 13);
              strncat (RS[i].rsltno, txtpos, 66);
              i += 1;
              txtpos += 66;
              textleft -= 66;
           }

  }   /* end get_server_msgs */


 /********************************************************************/
```

```
/*                                                                  */
/* Subroutine to perform drop command handler, close server        */
/* connection, and deallocate Connection Handler.                   */
/*                                                                  */
/********************************************************************/
void   close_connection ()

{
 CS_INT        rc;

/*------------------------------------------------------------------*/
/* Drop the command handle                                          */
/*------------------------------------------------------------------*/

       rc = ct_cmd_drop (cmd);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CMD_DROP failed", msg_size);
          error_out(rc) ;
       }

/*------------------------------------------------------------------*/
/* Close the server connection                                      */
/*------------------------------------------------------------------*/

       rc = ct_close (connection, (long) CS_UNUSED);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CLOSE failed", msg_size);
          error_out(rc) ;
       }

/*------------------------------------------------------------------*/
/* De_allocate the connection handle                                */
/*------------------------------------------------------------------*/

       rc = ct_con_drop (connection);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CON_DROP failed", msg_size);
          error_out(rc) ;
       }
}   /* end close_connection */
```

```
/******************************************************************/
/*                                                                */
/* Subroutine to perform exit client library and deallocate context */
/* structure.                                                     */
/*                                                                */
/******************************************************************/
void   quit_client_library ()

{
 CS_INT        rc;

/*----------------------------------------------------------------*/
/* Exit the Client Library                                        */
/*----------------------------------------------------------------*/

       rc = ct_exit (context, (long) CS_UNUSED);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_EXIT failed", msg_size);
          error_out(rc) ;
        }

/*----------------------------------------------------------------*/
/* De-allocate the context structure                              */
/*----------------------------------------------------------------*/

       rc = cs_ctx_drop (context);

       if (rc == CS_FAIL)
       {
          strncpy (msgstr, "CT_CTX_DROP failed", msg_size);
          error_out(rc) ;
       }

       EXEC CICS RETURN ;

}   /* end quit_client_library */

/******************************************************************/
/*                                                                */
/* Subroutine to display output                                   */
/*                                                                */
```

```
/*******************************************************************/
void   disp_data()

{
   CS_SMALLINT    QF_LEN;
   CS_SMALLINT    QF_MAXLEN;
   CS_CHAR        QF_ANSWER;
   CS_SMALLINT    CICS_RESPONSE;


        strncpy (OUT.sdateo, TMP_DATE, 8);
        strncpy (OUT.stimeo, TMP_TIME, 8);
        strncpy (OUT.prognmo, "SYCTSAR6", 8);

       switch (page_cnt)
       {
          case 1:
                 strcpy (OUT.spageo, "0001");  break;
          case 2:
                 strcpy (OUT.spageo, "0002");  break;
          case 3:
                 strcpy (OUT.spageo, "0003");  break;
          case 4:
                 strcpy (OUT.spageo, "0004");  break;
          case 5:
                 strcpy (OUT.spageo, "0005");  break;
          case 6:
                 strcpy (OUT.spageo, "0006");  break;
          case 7:
                 strcpy (OUT.spageo, "0007");  break;
          case 8:
                 strcpy (OUT.spageo, "0008");  break;
          case 9:
                 strcpy (OUT.spageo, "0009");  break;
          default:
                 strcpy (OUT.spageo, "9999");
       }

        OUT.servera = DFHBMPRO;
        strncpy (OUT.servero, servname, server_size);

        OUT.usera   = DFHBMPRO;
        strncpy (OUT.usero, username, user_size);

        OUT.pswda   = DFHBMDAR;
        strncpy (OUT.pswdo, pwd, pwd_size);
```

```
        OUT.trana   = DFHBMPRO;
        strncpy (OUT.trano, tran, tran_size);

        OUT.netdrva = DFHBMPRO;
        strncpy (OUT.netdrvo, driver, 9);

        memcpy (OUT.msg1o, msgtext1, text_size);
        strncpy (OUT.msg2o, msgtext2, text_size);
     /*------------------------------------------------------------------*/
     /* DISPLAY THE DATA                                                 */
     /*------------------------------------------------------------------*/
        EXEC CICS SEND MAP("SYCTBA6")
                       FROM(a6panel)
                       CURSOR
                       FRSET
                       ERASE
                       FREEKB ;

        EXEC CICS RECEIVE INTO(QF_ANSWER)
                          LENGTH(QF_LEN)
                          MAXLENGTH(QF_MAXLEN)
                          RESP(CICS_RESPONSE) ;

 }   /* end disp_data */
```

APPENDIX C Sybase Product Documentation

This appendix summarizes MainframeConnect documentation by content and by audience.

This appendix includes the following topics:

- Publications by content
- Publications by audience

Note For instructions on ordering documentation, go to the Sybase web site at http://www.sybase.com.

Publications by content

Table C-1 includes a synopsis of each publication in the current documentation set.

Table C-1: Documentation description

Title	Contents
Mainframe Connect Client Option and Server Option *Messages and Codes*	Provides details on messages that Mainframe Connect components return.
Mainframe Connect Server Option for CICS *Installation and Administration Guide*	Describes configuring the Mainframe Connect network, installing Open ServerConnect, setting up security, and troubleshooting for an MVS-CICS environment.
Mainframe Connect Server Option for IMS and MVS *Installation and Administration Guide*	Describes configuring the Mainframe Connect network, setting up APPC communications, installing Open ServerConnect, setting up security, and troubleshooting for an IMS or MVS environment.

Title	Contents
Mainframe Connect Server Option *Programmer's Reference for COBOL*	Provides reference material for writing Open ServerConnect programs that call COBOL Gateway-Library functions. This guide contains reference pages for Gateway-Library routines and descriptions of the underlying concepts for COBOL programmers.
Mainframe Connect Server Option *Programmer's Reference for PL/1*	Provides reference material for writing Open ServerConnect programs that call PL/I Gateway-Library functions. This guide contains reference pages for Gateway-Library routines and descriptions of the underlying concepts for PL/I programmers.
Mainframe Connect Server Option *Programmer's Reference for RSPs*	Provides information for anyone who designs, codes, and tests remote stored procedures (RSPs).
Mainframe Connect Client Option for CICS *Installation and Administration Guide*	Describes installing and configuring Open ClientConnect, routing requests to a server, and using Sybase isql. This manual also contains instructions for using the connection router and the mainframe-based isql utility.
Mainframe Connect Client Option for IMS and MVS *Installation and Administration Guide*	Describes installing Open ClientConnect, routing requests to a server, and using Sybase isql. This manual also contains instructions for using mainframe-based isql utility.
Mainframe Connect Client Option *Programmer's Reference for COBOL*	Describes writing Open ClientConnect programs that call COBOL Client-Library functions. This guide contains reference pages for Client-Library routines and descriptions of the underlying concepts for COBOL programmers.
Mainframe Connect Client Option *Programmer's Reference for PL/1*	Describes writing Open ClientConnect programs that call PL/I Client-Library functions. This guide contains reference pages for Client-Library routines and descriptions of the underlying concepts for PL/I programmers.
Mainframe Connect Client Option *Programmer's Reference for C*	Describes writing Open ClientConnect programs that call C Client-Library functions. This guide contains reference pages for Client-Library routines and descriptions of the underlying concepts for C programmers.
Mainframe Connect Client Option *Programmer's Reference for CSAs*	Provides information for anyone who designs, codes, and tests client services applications (CSAs).
Mainframe Connect DB2 UDB Option for CICS *Installation and Administration Guide*	Describes configuring the mainframe, installing MainframeConnect, setting up security, and troubleshooting for a CICS environment.

Title	Contents
Enterprise Connect Data Access and Mainframe Connect *Server Administration Guide* for DirectConnect	Describes administration of the DirectConnect server. Information about administering specific service libraries and services is provided in other DirectConnect publications.
Mainframe Connect DirectConnect for z/OS Option *Installation Guide*	Describes installing a DirectConnect server and service libraries.
Mainframe Connect DirectConnect for z/OS Option *User's Guide for Transaction Router Services*	Describes configuring, controlling, and monitoring DirectConnect Transaction Router Service Library, as well as setting up security.
Mainframe Connect DirectConnect for z/OS Option *User's Guide for DB2 Access Services* (for use with MainframeConnect for DB2 UDB)	Describes configuring, controlling, and monitoring DirectConnect for OS/390 Access Service, as well as setting up security.

Publications by audience

Table C-2 lists the publications in the documentation set and shows the intended audience for each book. The symbols used in the table are:

- R = required for this role
- O = optional (can be useful for this role)

Table C-2: Documentation by audience

Title	Mainframe Systems Support	Mainframe Application Developer	DirectConnect 3.0.x Admin.	Workstation Application Developer
Mainframe Connect Client Option and Server Option *Messages and Codes*	R	R	R	R
Mainframe Connect Server Option for CICS *Installation and Administration Guide*	R		O	
Mainframe Connect Server Option for IMS and MVS *Installation and Administration Guide*	R		O	
Mainframe Connect Server Option *Programmer's Reference for COBOL*	R	R		
Mainframe Connect Server Option *Programmer's Reference for PL/1*	R	R		
Mainframe Connect Server Option *Programmer's Reference for RSPs*	R	R		
Mainframe Connect Client Option for CICS *Installation and Administration Guide*	R		O	
Mainframe Connect Client Option for IMS and MVS *Installation and Administration Guide*	R		O	
Mainframe Connect Client Option *Programmer's Reference for COBOL*	R	R		
Mainframe Connect Client Option *Programmer's Reference for PL/1*	R	R		
Mainframe Connect Client Option *Programmer's Reference for C*	R	R		
Mainframe Connect Client Option *Programmer's Reference for CSAs*	R	R		
Enterprise Connect Data Access and Mainframe Connect *Server Administration Guide* for DirectConnect	O		R	
Mainframe Connect DirectConnect for z/OS Option *Installation Guide*	O		R	

Title	Mainframe Systems Support	Mainframe Application Developer	DirectConnect 3.0.x Admin.	Workstation Application Developer
Mainframe Connect DirectConnect for z/OS Option *User's Guide for Transaction Router Services*	O		R	R
Mainframe Connect DirectConnect for z/OS Option *User's Guide for DB2 Access Services*	O		R	R

Index

A

Adaptive Server Enterprise
  messages 35
APPC 11
Application name
  property 40, 43
Arguments
  assigning NULL to 38
Array binding
  description 69

B

Binding
  array binding 69
buf_len
  what to do when too short 24

C

Character datatypes
  list of 33
Character set conversion
  property 40, 43
Choosing
  dynamic network drivers 9, 12, 13
  network drivers 9, 12, 13
CICS and LU 6.2
  discussion of 12
CICS operating environment 13
Client message structure
  definition 26
  severity values 26
Client messages
  description 35
Client-Library
  datatypes 31
  error handling 35
  functions 15, 18
  initializing 141
  programs 16, 18, 19, 21
  properties 39
  setting up environment 18
  version 141, 179
Closing
  server connections 72
Columns
  result columns 61
Command handles
  allocating 74
  assigning NULL to 38
  de-allocating 78
  definition 58
  Gateway-Library equivalent 78
  properties 81
  routines that affect 58
Command parameters
  defining 145
Command structures
  allocating 18
  deallocating 18, 21
  definition 16
Commands
  language commands 87
  sending 18
  sending a command 86
  sending to a server 165
  steps in sending a command 170
Common Programming Interface
  discussion of 10
Communications sessions block
  property 44
Compatibility, Sybase mainframe access components 14
Connection handles
  allocating 88
  assigning NULL to 38

de-allocating 93
deallocating 93
definition 57
Gateway-Library equivalent 75
properties 58
retrieving properties 104
routines that affect 58
setting properties 104
Connection Router
function of 20
Connection Router table 8
Connection structures
allocating 19
deallocating 18, 21
description 19
Connections
closing 18, 21, 72
establishing 18
forcing a close 74
max number of 42
opening 20
setting maximum number of 45
Context handles
allocating 179
assigning NULL to 38
deallocating 182
definition 56
properties 56
retrieving properties 96
routines that affect 58
setting properties 96
Context structures
allocating 19
deallocating 21
definition 16
description 19
Control structures. See structures, control 16
Conversion
character set 40
CPI
discussion of 10
CS_APPNAME
description 40, 43
CS_CHARSETCNV
description 40
CS_CLEAR
when action is 24
CS_CLIENTMSG structure
data definition 26
description 26
severity values 26
cs_config
description 173
cs_convert 28
conversions performed by 179
description 173
cs_ctx_alloc
description 19, 179
used in a program 18, 19
cs_ctx_drop
description 182, 185
used in a program 21
CS_EXTRA_INF
description 41
CS_GET
when action is 24
CS_HOSTNAME 41
description 44
CS_LOC_PROP 41
description 44
CS_LOGIN_STATUS
description 41, 44
CS_LOGIN_TIMEOUT
description 41, 45
CS_MAX_CONNECT
description 42, 45
CS_NET_DRIVER
description 42
CS_NETIO
description 42, 45
CS_NO_COUNT
meaning 159
CS_NOINTERRUPT
description 42, 46
CS_PACKETSIZE
description 46
TDS packet size property 42
CS_PASSWORD
description 42, 46
CS_SERVERMSG structure
data definition 52
description 54

severity values 53
CS_SET
when action is 24
CS_TDS_VERSION
description 43, 46
CS_TEXTLIMIT
description 43, 47
CS_TIMEOUT
description 43, 47
CS_TRANSACTION_NAME
description 43, 47
CS_USERDATA
description 47, 173
CS_USERNAME
description 43, 48
CS_VERSION
description 48
ct_bind
DATAFMT structure 28
description 61
used in a program 18, 20, 21
ct_cancel
description 72
ct_close
description 72, 74
used in a program 18, 21
ct_cmd_alloc
description 75, 78
used in a program 18, 20
ct_cmd_drop
description 78, 80
used in a program 18, 21
ct_cmd_props
description 81, 83
ct_command
description 83, 88
used in a program 18, 20
ct_con_alloc
description 88, 92
used in a program 18, 19
ct_con_drop
description 95
used in a program 18, 21
ct_con_props
description 104, 109
used in a program 18
ct_config
description 95, 98
max number of connections 45
ct_connect
description 98
TDS version 46
used in a program 18, 20
ct_describe
DATAFMT structure 28
description 109, 116
ct_diag
CS_EXTRA_INF structure 44
description 116, 131
ct_exit
description 131, 133
used in a program 21
ct_fetch
description 134, 139
ct_get_format
description 139, 141
ct_init
description 141, 144
used in a program 18
ct_param
DATAFMT structure 28
description 145, 151
used in a program 21
ct_remote_pwd
description 151, 154
ct_res_info
description 154, 159
used in a program 18, 20
ct_results
coding hints 164
description 159, 165
loop 20
used in a program 18, 20
ct_send
description 165
used in a program 18, 20
Customization
items used by Gateway-Library 27

D

Data
- descriptions 28
- padding 30

DATAFMT structure
- data definition 28
- destination format 30
- fields in 28, 31
- functions used by 31
- general description 28

Datatypes 35
- character 33
- converted by cs_convert 178
- correspondences 31, 35
- datetime 34
- DB2 LONG VARCHAR 27
- decimal 34
- discussion 31
- float 34
- integer 34
- list of supported 31, 35
- money 35
- numeric 34
- packed decimal 31
- real 34

Datetime datatypes
- list of 34

Decimal datatype
- description 34

Defining
- command parameters 145
- dynamic network drivers 9
- network drivers 9

Dynamic network driver 19
- choosing 9, 12, 13
- defining 9
- invoking 9
- loading 9
- network type and environment 12
- operating environment 12

E

EIB
- pointer to 44

Environment
- gateway-enabled 3
- gateway-less 3
- three-tier 3
- two-tier 3

Error handling
- description 35, 38
- in-line 116, 117, 129
- using SQLCA and SQLCODE 37
- with ct_diag 36, 116

Error messages
- all 35
- client 26, 35
- server 36

Errors
- SQLCODE used with 55

Extra information
- property 41

F

Fetching
- result columns 134
- result data 134
- return parameters 134
- return status 134

Float datatype
- description 34

format
- supported values 31
- symbolic values 30

G

Gateway-enabled 3

Gateway-less 3

H

Handles
- command 74, 75, 78, 81
- connection 88, 93, 104
- context 96, 179

Host name
property 44

I

IHANDLE
Client-Library equivalent 88, 96
Image data
maximum length 43
Integer datatypes
list of 34
interfaces file
oc_connect 99
Interrupt indicator 42
Invoking
dynamic network drivers 9
network drivers 9
isql utility 8

L

Language commands
defining parameters for 150
Language request
initiating 83
Length
buffer 24
Loading
dynamic network drivers 9
network drivers 9
Locale
property 41
Locale information
property 44
Login name
server 43
Login properties 39
Login status 41
property 44
Login timeout
property 41, 45
LOGOUT
closing the connection 72
LU 6.2
discussion of 11
LU 6.2 and CICS
discussion of 12

M

Macros
SYGWDRIV 10
maxlength
meaning 30
MCC
description 38
Messages
clearing 129
client 26
discussion 35, 38
limiting 130
number of 131
retrieving 130
SQLCA used with 54
Money datatypes
list of 35

N

Name
server login 43
National languages
set during customization 27
Negotiated properties 39
Network communication definitions
choosing a network driver 10
overview 10
Network driver
choosing 9, 12, 13
defining 9
invoking 9
loading 9
network type and environment 12
operating environment 12
type of 42
Network type and environment
dynamic network driver 12
network driver 12

No interrupt
  property 46
Nulls
  definition 38
  discussion 38, 39

O

Open Client for CICS
  determining version of 48
Open ClientConnect
  communication 3, 8
  communication at the mainframe 8
  communication at the server 8
  security 9
Operating environment
  CICS 13
  dynamic network driver 12
  network driver 12
outlen
  using to determine buffer length 24

P

Packed decimal
  specifying Precision 31
Packets
  packet size 46
Parameter conventions 26
Parameter results
  binding to program variables 61
Parameters
  data descriptions 28
  defining 145
  fetching 134
  processing parameter results 20
  return 49
  unused parameters 38
Passwords
  clearing 151
  client 27
  defining 151
  password property 46
  server 42
Precision
  of datatypes 31
  specifying for packed decimal data 31
  when used 31
Programs
  basic steps in 18
  mixed-mode 16
  setting up the environment 18
Properties
  application name 40
  character set conversion 40, 43
  command handle properties 81
  communications sessions block 44
  connection status 41
  CS_APPNAME 43
  defined at context level 19
  discussion 39, 48
  extra information 41
  host name 41, 44
  interrupt indicator 42
  locale information 44
  localization 41
  login name 43
  login properties 39
  login status 44
  login timeout 41, 45
  maximum length of image data 43
  maximum length of text data 43
  maximum number of connections 42, 45
  negotiated properties 39
  no interrupt 46
  Open Client for CICS version 48
  packet size 46
  password 42, 46
  server name 41
  summary of properties table 40
  synchronous I/O 42
  TDS version 46
  text and image limit 47
  timeout 43
  transaction name 43, 47
  type of network driver 42
  user data 47
  user name 47

R

Real datatype
    description 34
Requests
    initiating 83
    sending to a server 165
Result set
    definition 51
result_type
    used in a program 21
Results
    binding to program variables 61
    cancelling 165
    data descriptions 28
    description 50
    determining when completely processed 164
    processing 20, 21
    processing results 51
    result types 160
    retrieving information about 154, 157, 158, 159
    run-time errors 165
    setting up 160, 165
    types of 160
Return status
    fetching 134
Return status results
    binding to program variable 61
Returning
    command handle information 81
Rows
    processing result rows 20
RPCs
    defining parameters for 150
    discussion 48, 50
    initiating 83
    results 49
    routines used with 48

S

SAA 10
Server message structure
    definition 52
    severity values 53
Server messages 36
Server name property 41
Server-Host Mapping table 8
Servers
    closing a server connection 72
Severity level
    in CS_CLIENTMSG values 26
    in CS_SERVERMSG values 53
SNA 11
SQLCA structure
    data definition 54
    description 54, 55
SQLCODE structure
    description 55
status
    processing result status 21
    return 50
    symbolic values of 31
stored procedures
    results 49, 50
    two ways to execute 48
structures
    commands 16
    connection 19
    context 16, 19
    control 16
    discussion 56
    SQLCA 54
SYGWDRIV macro 10
SYGWXPCH 10
Synchronous I/O indicator
    property 42
System Application Architecture 10
Systems Network Architecture 11

T

TDPROC
    Client-Library equivalent 75, 78
TDS
    packet size property 42
    TDS version property 46
    version 43
TDS version
    symbolic values for 47
Text and image

limiting text and image values 47
text and image limit property 47
Text data
maximum length 43
Three-tier 3
Timeout
property 43
Timeouts
login timeout property 45
Transaction name
property 43, 47
Two-tier 3
Types. See Datatypes 31

U

User data
property 47
User name
property 47
User-defined datatypes
definition 31

V

Variables
data descriptions 28
Version
of Client-Library 173, 179
Open Client for CICS 48
TDS 43