SYBASE®

Server-Library/C Reference Manual

Open Server™

15.0

DOCUMENT ID: DC35400-01-1500-05

LAST REVISED: December 2008

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

Contents

About This Book

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

Audience

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

How to use this book

When writing an Open Server application, use the Open Server *Server-Library/C Reference Manual* as a source of reference information.

Chapter 1, "Introducing Open Server," contains a brief introduction to Open Server.

Chapter 2, "Topics,"contains information on how to accomplish specific programming tasks, such as using **Server-Library** routines to read a text or image value from the **server**. This chapter also contains information on Open Server structures, programming techniques, and error handling.

Chapter 3, "Routines," contains specific information about each Server-Library routine, such as what parameters the routine accepts and what values it returns.

Chapter 4, "System Registered Procedures,"contains information on the registered procedures that Server-Library automatically provides. It includes a description of parameters, results, and messages.

Glossary words appear in bold the first time they are used in the text of this manual.

Related documents

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

- The Open Server and SDK *New Features* for Microsoft Windows, Linux, and UNIX, which describes new features available for Open Server and the Software Developer's Kit. This document is revised to include new features as they become available.
- The Open Client™ *Client-Library/C Reference Manual* contains reference information for Client-Library™, a collection of routines for use in writing client applications.
- The Open Client *DB-Library/C Reference Manual* describes DB-Library™. Like Client-Library, **DB-Library** is a collection of routines for use in writing client applications.
- The Sybase Adaptive Server® Enterprise *Reference Manual* describes Transact-SQL®, the database language an application uses to create and manipulate Sybase Adaptive Server Enterprise database objects.
- The SDK and Open Server *Installation Guide* for Microsoft Windows and SDK and Open Server *Installation Guide* for UNIX explain how to install Open Server.
- The Open Client and Open Server *Common Libraries Reference Manual* contains reference information for:
  - CS-Library
  - Bulk-Library
- The Open Client and Open Server *Programmer's Supplement* for your platform contains platform-specific programming information, including information about:
  - Compiling and linking an application
  - The sample programs that are included with Open Client and Open Server products
  - Routines that have platform-specific behaviors
- The Open Client and Open Server *Configuration Guide* for your platform contains platform-specific configuration information, including information about:
  - The **interfaces file**
  - **Localization**

Other sources of information

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

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

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

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

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

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

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

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

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

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

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

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

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

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

2 Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

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

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

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

3 Select a product.

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

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

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

Conventions

Table 1: Syntax conventions

Key	Definition
command	Command names, command option names, utility names, utility flags, and other keywords are in sans serif font.
variable	Variables, or words that stand for values that you fill in, are in *italics*.
{ }	Curly braces indicate that you choose at least one of the enclosed options. Do not include braces in your option.

Key	Definition
[ ]	Brackets mean choosing one or more of the enclosed items is optional. Do not include brackets in your option.
( )	Parentheses are to be typed as part of the command.
|	The vertical bar means you can select only one of the options shown.
,	The comma means you can choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

Online help

Open Server version 15.0 includes a number of sample **Open Server application** programs. They are located in *$SYBASE/$SYBASE_OCS/sample/srvlibrary* for UNIX, and *%SYBASE%\%SYBASE_OCS%\sample\srvlib* for Microsoft Windows. The Open Client and Open Server *Programmer's Supplement* for your platform summarizes each sample program and describes the requirements for running each.

If you have access to a SQL Server version 10.0 or later, you can use sp-syntax, a Sybase system procedure, to retrieve the syntax of Server-Library routines. For information on how to install sp-syntax, see the *System Administration Guide* for your platform. For information on how to run sp-syntax, see its reference page in the Adaptive Server Enterprise *Reference Manual*.

Accessibility features

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

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

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

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

If you need help

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

CHAPTER 1

Introducing Open Server

This chapter contains the following topics:

Client/Server overview

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

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

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

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

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

Types of clients

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

- Sybase SQL Toolset™ products
- Standalone utilities provided with Adaptive Server Enterprise, such as isql and bcp
- Applications written using Open Client libraries
- Applications written using Embedded SQL™
- PowerBuilder® applications

Types of servers

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

- Adaptive Server Enterprise is a database server. An Adaptive Server Enterprise manages information stored in one or more databases.

- Open Server provides the tools and interfaces needed to create a custom server. A custom server built with Open Server is called an "Open Server application."

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

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

- Adaptive Server Enterprise and Open Server applications are both servers that respond to client requests.
- Clients communicate with both Adaptive Server Enterprise and Open Server applications through Open Client libraries.

But they also differ:

- An application programmer must create an Open Server application, using Open Server's building blocks and supplying custom code. Adaptive Server Enterprise is complete and does not require custom code.
- An Open Server application can be any kind of server, and can be written to understand any language. Adaptive Server Enterprise is a database server, and understands only Transact-SQL.
- An Open Server application can communicate with "foreign" applications and servers that are not based on Sybase's Tabular Data Stream™, or **TDS**, protocol. It can also communicate with Sybase applications and servers. Adaptive Server Enterprise can communicate directly only with Sybase applications and servers. To communicate with foreign applications and servers, Adaptive Server Enterprise must use an Open Server gateway application as an intermediary.

Open server configurations

An Open Server application's position in the client/server architecture depends on its function. Open Server applications fall into one of three functional categories:

- Standalone

- Auxiliary
- Gateway

Standalone open server application

A client can connect directly to a standalone Open Server application.

The client submits requests to the server using:

- Remote Procedure Calls (RPCs), which allow you to execute **registered procedures** on an Open Server application. Registered procedures are defined pieces of Open Server code stored by the Open Server application. They can be user-defined or system-defined procedures.
- A **cursor** command.
- Any other kind of client **command**.

The Open Server application programmer supplies code to process client commands.

The standalone Open Server application makes no external requests to respond to a client request.

Auxiliary open server application

An auxiliary Open Server application can support Adaptive Server Enterprise by processing RPCs:

The client connects directly to Adaptive Server Enterprise and uses Transact-SQL for its language requests. To execute a registered procedure on the Open Server application, the client prefixes the procedure name with the name of the Open Server application in the Transact-SQL statement, which causes Adaptive Server Enterprise to initiate an RPC. For example, this client statement causes the procedure “print_calls” to be executed on the Open Server application named “OpnSrv211”:

```
exec OpnSrv211...print_calls
```

An RPC is the only type of client command that can be sent to an Open Server application directly from an Adaptive Server Enterprise. You can initiate the RPC calls by using stored procedures, triggers, or threshold management in Adaptive Server Enterprise. RPCs give you access to:

- Operating system functionality, such as sending e-mail and printing.
- Whatever functions you have defined in your Open Server application code.

The Open Server application can return information to the Adaptive Server Enterprise, or back to the client through Adaptive Server Enterprise.

Using server-to-server RPCs, an Open Server application can perform specialized calculations, provide access to real-time data, and permit Adaptive Server Enterprise to access services such as electronic mail.

Gateway Open Server application

A gateway server enables a client to access a server that may or may not be able to accept the client connection directly. The gateway does not have to connect to an Adaptive Server Enterprise or, for that matter, to any DBMS server. It could connect to a file system or an application program that can act as a server.

An Open Server application that accesses an Adaptive Server Enterprise or another Open Server application includes both Client-Library and Server-Library routines. It assumes both client and server roles. In the server role, it uses Open Server to interface with clients. In the client role, it uses Client-Library routines to send requests to, and receive results from, an Adaptive Server Enterprise or another Open Server. See “Gateway applications” on page 98 for details.

The gateway above connects clients to an Adaptive Server Enterprise. The dotted lines in the illustration indicate that this particular gateway uses “TDS passthrough mode,” a low-overhead method of passing requests and results between Sybase clients and Sybase servers. See “Passthrough mode” on page 129 for details.

Open Server

Open Server provides the tools and interfaces needed to create custom server applications.

Broadly speaking, Open Server contains a programming interface, in the form of libraries of functions, and network services.

The Open Server libraries

The libraries that make up the Open Server programming interface are:

- Server-Library, a collection of routines for use in writing server applications. Server-Library includes routines that:
  - Listen for commands from clients
  - Return results to clients
  - Set application attributes
  - Handle error conditions
  - Schedule interactions with clients
  - Provide a variety of information about client connections
- CS-Library, a collection of utility routines that are useful to both client and server applications. All Server-Library programs must include at least one call to CS-Library, because Server-Library routines use a structure that is allocated in CS-Library.

  Both Open Client and Open Server use CS-Library, which contains utility routines for both client and server applications.

Standalone and auxiliary Open Server applications include calls to Server-Library and CS-Library. Gateway applications include calls to Server-Library, CS-Library, and Client-Library.

Open Server also contains a set of header files that define structures, types, and values used by Server-Library routines. They are:

- *ospublic.h*
- *oserror.h*
- *oscompat.h*

Network services

Open Server network services are, in most cases, transparent to Open Server developers and end users of Open Server applications. On PC platforms, however, networking services are externalized.

Network services include Net-Library, which provides support for specific network protocols, such as TCP/IP.

Using Open Server

You write an Open Server application by using calls to Server-Library and CS-Library routines to set up structures, listen for connection requests from clients and other servers, process client requests, and clean up memory. A gateway application also includes calls to Client-Library routines.

An Open Server application program is compiled in the same way as any other C language program. On most UNIX platforms, you need to include these libraries when you compile and link your program (file names or extensions may vary by platform):

- *libsybsrv.a*
- *libsybcs.a*
- *libsybcomn.a*
- *libsybtcl.a*
- *libsybintl.a*
- *libsybblk.a* – if you are using bulk copy routines
- *libsybct.a* – if you are using a gateway

The library files are located in the *$SYBASE/$SYBASE_OCS/lib* directory.

The CS_CONTEXT structure

An Open Server application requires a CS_CONTEXT structure, which defines a particular application "context," or operating environment. A CS_CONTEXT structure contains localization information, as well as server-wide control information. The first step in any Open Server application program is to call cs_ctx_alloc to allocate a CS_CONTEXT structure.

An application programmer shapes an application's behavior and attributes by manipulating the contents of the application's CS_CONTEXT structure. See "Properties" on page 139 for more information.

Steps in a simple program

On most platforms, creating a simple Open Server application program involves these steps:

Step	Function	Routines
1	Set up the Open Server operating environment by allocating structures and setting global attributes, known as **properties**.	cs_ctx_alloc srv_version srv_props
2	Define error handling. Applications may install an error handling routine, which Open Server calls when it detects an error. Applications may also call the srv_sendinfo routine on an ad hoc basis to send error messages to the client, or srv_log to write to the log file. See "Errors" on page 89 for details.	srv_props(SRV_S_ERRHANDLE)
3	Initialize the server.	srv_init
4	Install event-handling routines, which Open Server calls when client commands trigger Open Server events. An Open Server application does most of its work inside its event-handling routines. Refer to "Open Server events" on page 13 for more information.	srv_handle
5	Start the server running. In this state, the server simply listens for client requests.	srv_run
6	Clean up and exit.	cs_ctx_drop

The sample program in the following section demonstrates all but step 4; it does not install user-defined event handlers. Therefore, the default handlers will execute instead.

Basic Open Server program

This code illustrates the basic framework of an Open Server application program:

```
/*
 **    This program demonstrates the minimum steps necessary
 **    to initialize and start up an Open Server application.
 **    No user-defined event handlers are installed, therefore
 **    the default handlers will execute instead.
 */

/*
 ** Include the required Open Server header files.
```

```
 **
 **    ospublic.h: Public Open Server structures, typedefs,
 **    defines, and function prototypes.
 **
 **    oserror.h:   Open Server error number #defines. This header
 **    file is only required if the Open Server application wants
 **    to detect specific errors inside the Open Server error
 **    handler.
 */

#include        <ospublic.h>
 #include        <oserror.h>

/*
 **    Include the operating system specific header files required
 **    by this Open Server application.
 */
 #include        <stdio.h>

 /*
 ** Local defines.
 **
 **    OS_ARGCOUNT    Expected number of command line arguments
 */
 #define        OS_ARGCOUNT    2


 /*
 **    This Open Server application expects the following
 **    command line arguments:
 **
 **    servername: The name of the Open Server application.
 **
 **    This name must exist in the interfaces file defined by
 **    the SYBASE environment variable.
 **
 ** Returns:
 **    0        Open Server exited successfully.
 **    1        An error was detected during initialization.
 */


 int    main(argc, argv)
 int    argc;
 char    *argv[];
 {
     CS_CONTEXT    *cp;              /* Context structure */
     CS_CHAR       *servername;      /* Open Server name */
```

```
CS_CHAR       logfile[512];     /* Log file name */
CS_BOOL       ok;               /* Error control flag */
SRV_SERVER    *ssp;             /* Server control structure*/

/* Initialization.      */
ok = CS_TRUE;

/*
** Read the command line options.  There must be one
** argument specifying the server name.
*/
if(argc != OS_ARGCOUNT)
{
    (CS_VOID)fprintf(stderr, "Invalid number of
    arguments(%d)\n",argc);


    (CS_VOID)fprintf(stderr, "Usage: <program>
    <server name>\n");
    exit(1);
}

/*
** Initialize 'servername' to the command line argument
** provided.
*/


servername = (CS_CHAR *)argv[1];

/*
** Allocate a CS-Library context structure to define the
** default localization information.  Open Server
** also stores global state information in this structure
** during initialization.
*/
if(cs_ctx_alloc(CS_VERSION_110, &cp) != CS_SUCCEED)
{
    (CS_VOID)fprintf(stderr, "%s: cs_ctx_alloc failed",
    servername);
    exit(1);
}

/*
** Default Open Server localization information can be
** changed here before calling srv_version, using cs_config
```

```
** and cs_locale.
*/

/*
** Set the Open Server version and context information
*/
if(srv_version(cp, CS_VERSION_110) != CS_SUCCEED)
{
    /*
    ** Release the context structure already allocated.
    */
    (CS_VOID)cs_ctx_drop(cp);

    (CS_VOID)fprintf(stderr, "%s: srv_version failed",
    servername);
    exit(1);
}

/*
** There is no error handler installed in this sample
** Open Server application.  Any errors detected by Open
** Server are written to the Open Server log file
** configured below.  A real Open Server application would
** install an error handler after calling srv_version, using
** srv_props(SRV_S_ERRHANDLE). Then, any subsequent errors
** will be detected by the Open Server application code.
*/

/*
** Default Open Server global properties can be changed here
** before calling srv_init.  We choose just to change the
** default log file name to use the name of this Open
** Server application.
*/

/*
** Build a new Open Server log file name using 'servername'
*/
(CS_VOID)sprintf(logfile, "%s.log", servername);

/*
** Set the new log file name using the global SRV_S_LOGFILE
** property.
*/
if(srv_props(cp, CS_SET, SRV_S_LOGFILE, logfile,
CS_NULLTERM,(CS_INT *)NULL) != CS_SUCCEED)
```

```
{
    /*
    ** Release the context structure already allocated.
    */
    (CS_VOID)cs_ctx_drop(cp);

    (CS_VOID)fprintf(stderr,"%s: srv_props(SRV_S_LOGFILE)
    failed\n",servername);
    exit(1);
}

/*
** Initialize Open Server. This causes Open Server to
** allocate internal control structures based on the global
** properties set above. Open Server also looks up
** the application name in the interfaces file.
*/
if((ssp = srv_init((SRV_CONFIG *)NULL, servername,
CS_NULLTERM))== (SRV_SERVER *)NULL)
{
    /*
    ** Release the context structure already allocated
    */
    (CS_VOID)cs_ctx_drop(cp);

    (CS_VOID)fprintf(stderr, "%s: srv_init failed\n",
    servername);
    exit(1);
}

/*
** Start the Open Server application running.  We don't
** install any event handlers in this simple example.  This
** causes Open Server to use the default event handlers.
**
** The call to srv_run does not return until a fatal error is
** detected by this Open Server application, or a SRV_STOP
** event is queued.  Since we haven't installed any event
** handlers, the only way to stop this Open Server
** application is to kill the operating system process in
** which it is running.
*/
if(srv_run((SRV_SERVER *)NULL) == CS_FAIL)
{
    (CS_VOID)fprintf(stderr, "%s: srv_run failed\n",
    servername);
```

```
        ok = CS_FALSE;
    }

    /*
    ** Release all allocated control structures and exit.
    */
    (CS_VOID)srv_free(ssp);
    (CS_VOID)cs_ctx_drop(cp);
    exit(ok ? 0 : 1);
}
```

Open Server events

The requests a client sends to an Open Server application trigger **events** in the server. This causes the client's server process, known as a **thread**, to execute a routine that processes the event. This routine is called an **event handler**.

There are many types of standard events defined internally by Server-Library, the most common of which are shown in this table:

Client request	Event type	Open Server event
ct_command(CS_LANG_CMD) ct_send	Language	SRV_LANGUAGE
ct_command(CS_RPC_CMD) ct_send	RPC	SRV_RPC
ct_cancel	Attention	SRV_ATTENTION
ct_connect	Connect	SRV_CONNECT
ct_close ct_exit	Disconnect	SRV_DISCONNECT
Non client-initiated	Start	SRV_START
Non client-initiated	Stop	SRV_STOP

For more information, see "Events" on page 92.

Default event handlers

Default event handlers exist for most of the standard events, but usually you will replace these with your own coded event handlers. Most of the default event handlers simply echo the request. For example, the default language event handler returns the message:

```
No language handler installed.
```

Installing an event handler automatically overrides the default event handler.

Non client-initiated events

Some events cannot be directly triggered by client programs:

- User-defined events
- SRV_STOP, which is triggered by calling srv_event in the Open Server code
- SRV_START, which occurs as a part of the start-up process

Registered procedures

A registered procedure is a piece of Open Server/C code identified by a name. When an application registers a procedure, it maps the procedure name to a routine, so that when Open Server detects this procedure name in an incoming RPC datastream, it can call a specific routine immediately without raising a SRV_RPC event.

When an Open Server application receives an RPC, it looks up the procedure name in the list of registered procedures. If the name is registered, the runtime system executes the routine associated with the registered procedure. If the procedure name is not found in the list of registered procedures, Open Server calls the SRV_RPC event handler.

System registered procedures are built-in procedures that are internal to all Open Server applications. See Chapter 4, “System Registered Procedures” for a detailed description of each system registered procedure.

See “Registered procedures” on page 162 for details on registered procedures.

Returning results to clients

This section describes the types and order of result data that can be sent and returned to clients.

Types of result data

An Open Server application can send results to a client as:

- Messages
- Rows of data
- Result parameters
- Status values

A single client request can obtain more than one set of results. After sending the first result set, call srv_senddone with a status of SRV_DONE_MORE if there are more result sets for the request. Call srv_senddone with a status of SRV_DONE_FINAL if there are no more results. Calling srv_senddone with a SRV_DONE_FINAL status is the minimum response to a client request. The client waits until it receives srv_senddone(SRV_DONE_FINAL) before proceeding.

Messages

An application can send error messages to clients with srv_sendinfo. Client-Library programs process messages with a message handler routine. These routines typically display the message information on the user's terminal. If the message is an **error message**, the client program can attempt to recover from the error or exit.

Data rows

Open Server can return rows of data to clients just as Adaptive Server Enterprise returns the results of SQL queries. A row consists of one or more columns of data. See "Processing parameter and row data" on page 134 for details.

Parameters

Parameters are data that is passed using client commands between clients and the Open Server application.

Status values

An application can call srv_sendstatus to return an optional status value to a client application. The status is a CS_INT value that has an application-specific meaning. CS_INT is an Open Server data type; see “Types” on page 199 for more information. There can be only one status value for each set of results.

Order of results

The order in which you return results to clients is important:

- Do not interrupt a set of data rows with other kinds of results. Data rows must be sent one after another until the entire set has been sent to the client. For example, you cannot send a few rows, then send a message, then send more rows.
- After you have sent all of the data rows (if any), you can send messages and status information to the client in any order.
- At the end of a set of results, call srv_senddone to signal the end of the results.

Error handling

One of the first actions to take in an Open Server application is to install an error handler with srv_props. If no error handler has been installed, Open Server writes the error messages to the log file. See “Errors” on page 89 for details.

Multithread programming

Open Server employs a multithread architecture. This architecture allows application developers to create multithread servers. A multithread server is a collection of threads, each executing routines to accomplish its specific task. For example, each client uses a thread that manages its connection and executes the event handlers and procedures that fulfill its requests. The Open Server runtime system employs several threads that manage server activities such as delivering messages, handling network communications, and scheduling tasks in the server. You can "spawn" threads for other nonclient activities.

See "Multithread programming" on page 109 for details.

Summary of changes for version 15.0

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

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

  CS_OPT_HIDE_VCC instructs the Adaptive Server to hide Virtual Computed Columns (VCC), while CS_OPT_SHOW_FI adds columns for each Functional Index (FI).
- Large identifiers: Limits on lengths of identifiers have been reduced. This is now 255 bytes for identifiers.
- Unilib® support: Unicode Infrastructure Library (Unilib), an independent library of Unicode-based routines, has been included to facilitate character-set conversion.
- ASE default packet size support: You can now configure packet size centrally on the server, with the default set to 8192 bytes.

- Clusters support: A cluster of servers can now perform load balancing for all client connections coming into the cluster.
- Scrollable cursors: You can now set the position of a cursor anywhere in the cursor result set.
- Table 1-1 lists the new datatypes introduced in this version:

Table 1-1: New datatypes

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

CHAPTER 2

Topics

This chapter contains information on:

- Open Server programming topics, such as processing parameter and row data, and support for text and image
- How to use Open Server routines to accomplish specific programming tasks, such as responding to cursor requests and handling errors
- Open Server properties, datatypes, and structures

This chapter contains the following topics:

Attention events

When a client application cancels a request through a dbcancel or ct_cancel command, it triggers an Open Server SRV_ATTENTION event. Open Server then calls the Open Server application's SRV_ATTENTION event handler. Once the SRV_ATTENTION event handler returns, Open Server resumes processing where it left off when the attention event was detected.

Interrupt-level activity

A SRV_ATTENTION event handler is the only event handler that runs at interrupt level. An Open Server application can only issue the following Server-Library calls from inside a SRV_ATTENTION handler:

- srv_wakeup with the wakeflags argument set to SRV_M_WAKE_INTR
- srv_ucwakeup with the wakeflags argument set to SRV_M_WAKE_INTR
- srv_thread_props with the cmd argument set to CS_GET
- srv_props with the cmd argument set to CS_GET
- srv_event_deferred

No other Server-Library routines can be called from the SRV_ATTENTION event handler, or from other interrupt-level code.

Coding recommendations for attention events

Attention events are problematic if they arrive while noninterrupt-level handler code is executing. An application may do work it no longer needs to do because the client has cancelled a request.

It is the application's responsibility to check for attention event periodically if it is performing a time-consuming I/O task or compute-intensive work at the noninterrupt level. The application code should periodically check for attention events using srv_thread_props, with cmd set to CS_GET and property set to SRV_T_GOTATTENTION.

Once it detects an attention event, the Open Server application code can continue to send results, but clients ignore them. The simplest way the application can respond to an attention event is to send a SRV_DONE_FINAL to the client and return.

An attention event can arrive while the Client-Library portion of the gateway application code is executing. The application can call ct_command with the type argument set to CS_CANCEL_ATTN in its SRV_ATTENTION event handler to force the Client-Library routine to return to noninterrupt-level code. Because this command does not take effect unless an attention event arrives, a gateway application should call it routinely.

All gateway calls performing client I/O should check for attention events with srv_thread_props before calling ct_send. This ensures that a **query** will not be sent to a remote server once the client has already cancelled it.

Handling disconnects

If an Open Server application is in the middle of returning results to a client and the client abruptly disconnects, the application continues to return results until it detects that the connection has been closed. Open Server subsequently calls the SRV_DISCONNECT event handler. In this scenario, the application continues to send results to a client that can no longer receive them. An abrupt client disconnect can occur if:

- A client calls ct_close before handling all the results the server is sending it.

- The client process dies suddenly.
- The machine goes down.

To avoid this situation, an application can request that Open Server first calls the application's SRV_ATTENTION event handler in response to a client disconnect, and then calls the SRV_DISCONNECT event handler. For Open Server to handle disconnects in this fashion, an application must use srv_props to set the SRV_S_DISCONNECT property to CS_TRUE. The SRV_DISCONNECT event handler is still called in the usual way, but it is called after the SRV_ATTENTION handler. The SRV_S_DISCONNECT property defaults to CS_FALSE.

The SRV_ATTENTION handler initiates the appropriate steps to terminate the I/O activity and stop the return of results from the routine that was executing at the time of the disconnect. An application can thus respond to disconnects in the same way that it would to attentions.

Using its SRV_ATTENTION event handler, an application can determine which event triggered the handler—an attention or a disconnect—by calling srv_props with cmd set to CS_GET and property set to SRV_S_ATTNREASON.

Example

The sample *ctos.c* includes attention handling code.

Browse mode

Note Browse mode is included to provide compatibility with Open Client libraries older than version 11.1. Sybase discourages its use in Open Server Server-Library applications, because cursors provide the same functionality in a more portable and flexible manner. Additionally, browse mode is Sybase-specific and is not suited for use in a heterogeneous environment.

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

Because a row being browsed is not the actual row residing in the database but a copy residing in program variables, the program must update the original database row with changes made to the variables' values. In multiuser situations, the program must ensure that updates made to the database by one user do not overwrite recent updates made by another user. Such overwrites occur because a client application typically selects a number of rows from a database to update at one time, but the application's users browse and update the database one row at a time. A timestamp column in browsable tables provides the information necessary to regulate this type of multiuser updating.

Client applications that permit users to enter ad hoc browse mode queries must update underlying database tables if a user command alters a table's contents. Consequently, these applications may need information about the underlying structure of a browse mode command.

Open Server includes two routines that provide such information, srv_tabname and srv_tabcolname:

- srv_tabname provides the name and number of each table involved in the browse mode command.
- srv_tabcolname returns a variety of information about result columns through a CS_BROWSEDESC structure. For more information, see "CS_BROWSEDESC structure" on page 52.

An Open Server application that receives browse mode requests can call these two routines, along with the standard data binding routines, to return browse mode information. The specific steps are:

1 Call srv_tabname once for each table that is the source of a result row.

2 Call srv_descfmt followed by srv_tabcolname once for each column in the result row.

   If the Open Server application has set the status field of the CS_BROWSEDESC structure to CS_RENAMED, this means that the client application's browse mode select statement renamed the column. The Open Server application must fill in the original name of the column in the database, and the length of its name, in the orignamе and orignlen fields in the CS_BROWSEDESC structure prior to calling srv_tabcolname.

3 Bind and transfer the column data using the srv_bind and srv_xferdata routines, respectively.

Note Because srv_tabcolname requires information returned by srv_tabname—the unique table number—srv_tabname must precede a call to srv_tabcolname.

For more information on browse mode, see the Sybase Open Client *Client-Library/C Reference Manual*.

Example

The sample program *ctos.c* includes code to process browse mode information.

Capabilities

An Open Server application and a client must agree on what requests the client can issue and what responses the Open Server application will return. For example, a client may want to issue language requests, but the Open Server application may not be equipped with a parser to process such requests. Similarly, a client may not want the Open Server application to return text or image data if the client is not equipped to handle it. A client/server connection's **capabilities** determine the types of client requests and server responses permitted for that connection.

The Open Server application ultimately determines which capabilities are valid for the connection. If the client does not accept these capabilities, its only option is to close the connection.

There are two types of capability negotiation: *transparent* and *explicit*. In transparent negotiations, the Open Server application assigns a **default** set of possible client requests and Open Server responses. In explicit negotiations, the Open Server application includes code to negotiate capabilities, using the srv_capability_info routine.

Transparent negotiation is part of both Open Server and Open Client's default behavior. Therefore, an Open Server application must call srv_capability_info if it wants to support something other than the default set of capabilities.

Request capabilities

Table 2-1 describes each request capability:

Table 2-1: Request capabilities

CS_REQUEST capability	Meaning	Capability relates to
CS_CAP_ EXTENDEDFAILOVER	Extended HA failover	Connections
CS_CON_INBAND	In-band (non-expedited) attentions	Connections
CS_CON_OOB	Out-of-band (expedited) attentions	Connections
CS_CSR_ABS	Fetch of specified absolute cursor row	Cursors
CS_CSR_FIRST	Fetch of first cursor row	Cursors
CS_CSR_LAST	Fetch of last cursor row	Cursors
CS_CSR_MULTI	Multi-row cursor fetch	Cursors
CS_CSR_PREV	Fetch previous cursor row	Cursors
CS_CSR_REL	Fetch specified relative cursor row	Cursors
CS_DATA_BIN	Binary datatype	Datatypes
CS_DATA_VBIN	Variable-length binary type	Datatypes
CS_DATA_LBIN	Long variable-length binary datatype	Datatypes
CS_DATA_BIT	Bit datatype	Datatypes
CS_DATA_BITN	Nullable bit datatype	Datatypes
CS_DATA_BOUNDARY	Boundary datatype	Datatypes
CS_DATA_CHAR	Character datatype	Datatypes
CS_DATA_VCHAR	Variable-length character datatype	Datatypes
CS_DATA_LCHAR	Long variable-length character datatype	Datatypes
CS_DATA_DATE	Date datatype	Datatype
CS_DATA_DATE4	Short datetime datatype	Datatypes
CS_DATA_DATE8	Datetime datatype	Datatypes
CS_DATA_DATETIMEN	Null datetime values	Datatypes
CS_DATA_DEC	Decimal datatype	Datatypes
CS_DATA_FLT4	4-byte float datatype	Datatypes
CS_DATA_FLT8	8-byte float datatype	Datatypes
CS_DATA_FLTN	Nullable float datatype	Datatypes
CS_DATA_IMAGE	Image datatype	Datatypes
CS_DATA_INT1	Tiny integer datatype	Datatypes
CS_DATA_INT2	Small integer datatype	Datatypes
CS_DATA_INT4	Integer datatype	Datatypes
CS_DATA_INT8	Big integer datatype	Datatypes
CS_DATA_INTN	Null integers	Datatypes
CS_DATA_MNY4	Short money datatype	Datatypes

CS_REQUEST capability	Meaning	Capability relates to
CS_DATA_MNY8	Money datatype	Datatypes
CS_DATA_MONEYN	Null money values	Datatypes
CS_DATA_NUM	Numeric datatype	Datatypes
CS_DATA_SENSITIVITY	Sensitivity datatype	Datatypes
CS_DATA_TEXT	Text datatype	Datatypes
CS_DATA_TIME	Time datatype	Datatypes
CS_DATA_UCHAR	2-byte character datatype	Datatypes
CS_DATA_UNITEXT	Unitext datatype	Datatypes
CS_DATA_XML	XML datatype	Datatypes
CS_OPTION_GET	Current option values	Datatypes
CS_PROTO_DYNAMIC	Use TDS DESCIN/OUT protocol	Commands
CS_PROTO_DYNPROC	Add “create proc” in the front of dynamic prepares	Commands
CS_REQ_BCP	Bulk copy requests	Commands
CS_REQ_CURSOR	Cursor requests	Commands
CS_REQ_DBRPC2	Large RPC name requests	Commands
CS_REQ_DYN	Dynamic SQL requests	Commands
CS_REQ_LANG	Language requests	Commands
CS_REQ_LARGEIDENT	Large identifier requests	Commands
CS_REQ_MIGRATE	Migration requests	Connection
CS_REQ_MSG	Message data	Commands
CS_REQ_MSTMT	Multiple server commands per Client-Library request	Connection
CS_REQ_NOTIF	Event notifications	Connection
CS_REQ_SRVPKTSIZE	Server-specified packetsize	Connection
CS_REQ_PARAM	Parameter data	Commands
CS_REQ_RPC	Remote procedure requests	Commands
CS_REQ_URGNOTIF	Use 5.0 event notification protocol	Commands
CS_WIDETABLES	Wider and increased number of columns per table	Commands

Response capabilities

Table 2-2 describes each response capability.

Note Response capabilities indicate the kinds of responses the client does *not* want to receive.

Table 2-2: Response capabilities

CS_RESPONSE capability	Meaning	Capability relates to
CS_CON_NOINBAND	No in-band (non-expedited) attentions	Connections
CS_CON_NOOOB	No out-of-band (expedited) attentions	Connections
CS_NO_SRVPKTSIZE	No server-specified packetsize	Connections
CS_DATA_NOBIN	No binary datatype	Datatypes
CS_DATA_NOVBIN	No variable-length binary type	Datatypes
CS_DATA_NOLBIN	No long variable-length binary datatype	Datatypes
CS_DATA_NOBIT	No bit datatype	Datatypes
CS_DATA_NOBOUNDARY	No boundary datatype	Datatypes
CS_DATA_NOCHAR	No character datatype	Datatypes
CS_DATA_NOVCHAR	No variable-length character datatype	Datatypes
CS_DATA_NOLCHAR	No long variable-length character datatype	Datatypes
CS_DATA_NODATE	No date datatype	Datatypes
CS_DATA_NODATE4	No short datetime datatype	Datatypes
CS_DATA_NODATE8	No datetime datatype	Datatypes
CS_DATA_NODATETIMEN	No null datetime values	Datatypes
CS_DATA_NODEC	No decimal datatype	Datatypes
CS_DATA_NOFLT4	No 4-byte float datatype	Datatypes
CS_DATA_NOFLT8	No 8-byte float datatype	Datatypes
CS_DATA_NOIMAGE	No image datatype	Datatypes
CS_DATA_NOINT1	No tiny integer datatype	Datatypes
CS_DATA_NOINT2	No small integer datatype	Datatypes
CS_DATA_NOINT4	No integer datatype	Datatypes
CS_DATA_NOINT8	No big integer datatype	Datatypes
CS_DATA_NOINTN	No null integers	Datatypes
CS_DATA_NOMNY4	No short money datatype	Datatypes
CS_DATA_NOMNY8	No money datatype	Datatypes
CS_DATA_NOMONEYN	No null money values	Datatypes
CS_DATA_NONUM	No numeric datatype	Datatypes
CS_DATA_NOSENSITIVITY	No sensitivity datatype	Datatypes
CS_DATA_NOTEXT	No text datatype	Datatypes
CS_DATA_NOTIME	No time datatype	Datatype

CS_RESPONSE capability	Meaning	Capability relates to
CS_DATA_NOUCHAR	No 2-byte character datatype	Datatypes
CS_DATA_NOUNITEXT	No Unitext datatype	Datatypes
CS_DATA_NOXML	No XML datatype	Datatypes
CS_RES_NOEED	No extended error results	Results
CS_RES_NOMSG	No message results	Results
CS_RES_NOPARAM	No result parameters	Results
CS_RES_NOTDSDEBUG	No TDS debug token	Results
CS_NO_LARGEIDENT	No large identifiers	Commands
CS_NOWIDETABLES	No increase in column size or number of columns per table	Commands

Note When an Open Server application defines the client data format using the srv_descfmt routine, Open Server verifies that the response capability for the relevant datatype is *not* set. If it is set, either the client has requested the server not to send results pertaining to that datatype or the TDS version of the client connection does not support that datatype. In such cases, Open Server raises an error and srv_descfmt returns CS_FAIL.

Transparent negotiation

Open Server includes a set of default capability values. For a list of defaults, see “Server-wide defaults” on page 31. These defaults are server-wide; they apply to all client connections. When the defaults are used, all capabilities Open Server supports are turned on.

An Open Server application can change the server-wide default values during initialization by calling the srv_props routine. See srv_props on page 334.

When a DB-Library or Client-Library client logs in to an Open Server application, it sends a list of desired capabilities in its login record. In transparent negotiation, Open Server finds the intersection of its default values and the client values. The resulting values are the capabilities supported on that connection.

When does transparent negotiation take place?

Transparent negotiation takes place when:

- An Open Server application does not have a SRV_CONNECT handler other than the default handler.
- An Open Server application does not explicitly include code in its custom SRV_CONNECT event handler to override default capabilities.

Note In passthrough mode, srv_getloginfo and srv_setloginfo handle capability negotiation transparently.

Server-wide defaults

Table 2-3 indicates the default setting for each request capability by TDS version. A *1* indicates that the capability is supported in the TDS version. A *0* indicates that the capability is not supported.

Table 2-3: Request capabilities by TDS version

CS_REQUEST capability	4.0	4.0.2	4.2	4.6	5.0
CS_CAP_EXTENDEDFAILOVER	0	0	0	0	1
CS_CON_INBAND	0	0	0	0	1
CS_CON_OOB	1	1	1	1	0
CS_CSR_ABS	0	0	0	0	0
CS_CSR_FIRST	0	0	0	0	0
CS_CSR_LAST	0	0	0	0	0
CS_CSR_MULTI	0	0	0	0	0
CS_CSR_PREV	0	0	0	0	0
CS_CSR_REL	0	0	0	0	0
CS_DATA_BIN	1	1	1	1	1
CS_DATA_BIT	1	1	1	1	1
CS_DATA_BITN	0	0	0	0	0
CS_DATA_BOUNDARY	0	0	0	0	0
CS_DATA_CHAR	1	1	1	1	1
CS_DATA_DATE	0	0	0	0	1
CS_DATA_DATE4	0	0	1	1	1
CS_DATA_DATE8	1	1	1	1	1
CS_DATA_DATETIME	1	1	1	1	1
CS_DATA_DEC	0	0	0	0	0
CS_DATA_FLT4	0	0	1	1	1
CS_DATA_FLT8	1	1	1	1	1
CS_DATA_FLTN	1	1	1	1	1
CS_DATA_IMAGE	1	1	1	1	1
CS_DATA_INT1	1	1	1	1	1
CS_DATA_INT2	1	1	1	1	1
CS_DATA_INT4	1	1	1	1	1
CS_DATA_INT8	0	0	0	0	1
CS_DATA_INTN	1	1	1	1	1
CS_DATA_LBIN	0	0	0	0	0
CS_DATA_LCHAR	0	0	0	0	0
CS_DATA_MNY4	0	0	1	1	1
CS_DATA_MNY8	1	1	1	1	1
CS_DATA_MONEYN	1	1	1	1	1
CS_DATA_NUM	0	0	0	0	0
CS_DATA_SENSITIVITY	0	0	0	0	0
CS_DATA_TEXT	1	1	1	1	1

CS_REQUEST capability	4.0	4.0.2	4.2	4.6	5.0
CS_DATA_TIME	0	0	0	0	1
CS_DATA_UCHAR	0	0	0	0	1
CS_DATA_UNITEXT	0	0	0	0	1
CS_DATA_VBIN	1	1	1	1	1
CS_DATA_VCHAR	1	1	1	1	1
CS_DATA_XML	0	0	0	0	1
CS_OPTION_GET	0	0	0	0	0
CS_PROTO_DYNAMIC	0	0	0	0	0
CS_PROTO_DYNPROC	0	0	0	0	0
CS_REQ_BCP	1	1	1	1	1
CS_REQ_CURSOR	0	0	0	0	0
CS_REQ_DBRPC2	0	0	0	0	1
CS_REQ_DYN	0	0	0	0	0
CS_REQ_LANG	1	1	1	1	1
CS_REQ_LARGEIDENT	0	0	0	0	1
CS_REQ_MIGRATE	0	0	0	0	1
CS_REQ_MSG	0	0	0	0	0
CS_REQ_MSTMT	0	0	0	0	0
CS_REQ_NOTIF	0	0	0	1	1
CS_REQ_PARAM	0	0	0	0	0
CS_REQ_RPC	1	1	1	1	1
CS_REQ_SRVPKTSIZE	0	0	0	0	1
CS_REQ_URGNOTIF	0	0	0	0	0
CS_WIDETABLES	0	0	0	0	1

Table 2-4 describes the default setting for each response capability by TDS version.

- *1* indicates that the capability is not supported in the TDS version.
- *0* indicates that the capability is supported.

Table 2-4: Response capabilities by TDS version

CS_RESPONSE capability	4.0	4.0.2	4.2	4.6	5.0
CS_CON_NOINBAND	1	1	1	1	1
CS_CON_NOOOB	0	0	0	0	0
CS_DATA_NOBIN	0	0	0	0	0
CS_DATA_NOBIT	0	0	0	0	0
CS_DATA_NOBOUNDARY	1	1	1	1	1
CS_DATA_NOCHAR	0	0	0	0	0
CS_DATA_NODATE4	1	1	0	0	0
CS_DATA_NODATE8	0	0	0	0	0
CS_DATA_NODATETIME	0	0	0	0	0
CS_DATA_NODEC	1	1	1	1	1
CS_DATA_NOFLT4	1	1	0	0	0
CS_DATA_NOFLT8	0	0	0	0	0
CS_DATA_NOIMAGE	0	0	0	0	0
CS_DATA_NOINT1	0	0	0	0	0
CS_DATA_NOINT2	0	0	0	0	0
CS_DATA_NOINT4	0	0	0	0	0
CS_DATA_NOINT8	1	1	1	1	0
CS_DATA_NOINTN	0	0	0	0	0
CS_DATA_NOLBIN	1	1	1	1	1
CS_DATA_NOLCHAR	1	1	1	1	1
CS_DATA_NOMNY4	1	1	0	0	0
CS_DATA_NOMNY8	0	0	0	0	0
CS_DATA_NOMONEY	0	0	0	0	0
CS_DATA_NONUM	1	1	1	1	1
CS_DATA_NOSENSITIVITY	1	1	1	1	1
CS_DATA_NOSINT1	1	1	1	1	0
CS_DATA_NOTEXT	0	0	0	0	0
CS_DATA_NOUCHAR	1	1	1	1	0
CS_DATA_NOUNITEXT	1	1	1	1	0
CS_DATA_NOVBIN	0	0	0	0	0
CS_DATA_NOVCHAR	0	0	0	0	0
CS_DATA_NOXML	1	1	1	1	0
CS_RES_NOEED	1	1	1	1	1
CS_RES_NOMSG	1	1	1	1	1
CS_RES_NOPARAM	1	1	1	1	1
CS_RES_NOTDSDEBUG	1	1	1	1	1

CS_RESPONSE capability	4.0	4.0.2	4.2	4.6	5.0
CS_NO_LARGEIDENT	1	1	1	1	0
CS_NO_SRVPKTSIZE	1	1	1	1	0
CS_NOWIDETABLES	1	1	1	1	0

Explicit negotiation

Explicit negotiation takes place at connect time, from within the SRV_CONNECT event handler. The Open Server application retrieves the list of request capabilities sent by the client and returns the list of request capabilities it will accept. The process is repeated, this time with the list of response capabilities a client does *not* want to receive or those the Open Server application cannot return.

An application can retrieve and send capabilities one at a time or can retrieve and send an entire bitmask of capabilities at once. Open Server provides macros to test, clear, and set bits in a capability mask. For more information, see “Capability macros” on page 36.

Negotiating capabilities one at a time

To negotiate request capabilities one at a time, an application must make the following calls *for each capability* you want to negotiate:

1 Call srv_capability_info with the cmd argument set to CS_GET, the type argument set to CS_CAP_REQUEST, and the capability argument set to the capability of interest. If the *valp argument contains CS_TRUE, the client will request this type of capability. If *valp contains CS_FALSE, the client will not.

2 Call srv_capability_info with the cmd argument set to CS_SET, the type argument set to CS_CAP_REQUEST, and the capability argument set to the capability of interest, and *valp set to a Boolean value. The application sets *valp to CS_TRUE to support this type of capability and CS_FALSE to decline it.

An application negotiates response capabilities in a similar fashion, except that it must set the type argument to CS_CAP_RESPONSE.

An Open Server application only needs to call srv_capability_info for the request and response capabilities that it negotiates explicitly. The default values are used for all the other capabilities.

Negotiating using a capability bitmask

To negotiate request capabilities using a capability bitmask, an application must:

1 Read in the entire bitmask by calling srv_capability_info with the cmd argument set to CS_GET, the type argument set to CS_CAP_REQUEST, the capability argument set to CS_ALL_CAPS, and valp pointing to the CS_CAP_TYPE structure that will contain the bitmask.

2 Test, set, or clear particular bits in the bitmask using the CS_TST_CAPMASK, CS_SET_CAPMASK and CS_CLR_CAPMASK macros.

An application negotiates response capabilities in a similar fashion, except that it must set the type argument to CS_CAP_RESPONSE.

Gateway applications should use the mask method to negotiate capabilities. As the following diagram illustrates, the gateway calls srv_capability_info to retrieve the remote client's capability mask and sends those capabilities to the remote server by calling ct_capability prior to calling ct_connect. Once the remote connection has been established, the gateway can retrieve the capability masks that the remote server has sent using ct_capability and then define them on the remote client connection, using srv_capability_info.

Capability macros

Table 2-5 describes the macros that an application can use to manipulate a capability bitmask:

Table 2-5: Capability macros

Macro name	Function
CS_TST_CAPMASK	Test to see whether a specific capability is set to CS_TRUE or CS_FALSE
CS_SET_CAPMASK	Set a specific capability to CS_TRUE
CS_CLR_CAPMASK	Set a specific capability to CS_FALSE.

When negotiating capabilities explicitly, rather than using the default settings, the following two rules apply:

- CS_CAP_REQUEST
  Applications can only turn CS_CAP_REQUEST capabilities "off" from an "on" status.

If an application tries to turn a CS_CAP_REQUEST capability "off," which is already in an "off" status, Open Server restores the *default* status and does *not* raise an error.

- CS_CAP_RESPONSE
  Applications can only turn CS_CAP_RESPONSE capabilities "on" from an "off" status.

  If an application tries to turn a CS_CAP_RESPONSE capability "on," which is already in an "on" status, Open Server restores the *default* status and does *not* raise an error.

Ad hoc retrieval of capability information

An Open Server application can call srv_capability_info from within any handler at any time to retrieve a list of capabilities in effect for that particular client connection. In a SRV_CONNECT event handler, however, the capability masks retrieved are not the final masks for the connection. Rather, they are the client's requested capabilities combined with the Open Server application's defaults. Connection capabilities are not final until the SRV_CONNECT handler has returned.

A note on pre-10.0 clients

An Open Server application can negotiate capabilities with clients running any TDS version. If a pre-10.0 client makes a connection, Open Server simulates capability negotiation. In this scenario, the Open Server application does not need to know what TDS version the client is running.

Example

The sample program *ctos.c* includes code illustrating capability negotiation.

Client command errors

A client sometimes sends an incomplete or nonsensical request to an Open Server application. Requests can be incomplete or meaningless because of faulty client code or because of a network problem. An Open Server application should handle these errors in the event handler for the client request, by sending the appropriate error messages to the client.

Sending messages with *srv_sendinfo*

An Open Server application calls srv_sendinfo to send error messages to a client. An Open Server application describes the message in a CS_SERVERMSG structure and then calls srv_sendinfo to send this description to the client.

For more information, see "CS_SERVERMSG structure" on page 60.

Sequencing long messages

An Open Server application stores the message text itself in the text field of the CS_SERVERMSG structure. text has a maximum length of CS_MAX_MSG bytes.

An Open Server application uses as many CS_SERVERMSG structures as necessary to return the full text of a message. The application returns the first CS_MAX_MSG bytes in one structure, the second CS_MAX_MSG bytes in a second structure, and so forth. This process is known as "chunking" the message.

An application calls srv_sendinfo as many times as there are "chunks". If the entire message fits in one structure, the application only needs to call srv_sendinfo once.

CS_SERVERMSG structure fields for sequenced messages

The status field in the CS_SERVERMSG structure indicates whether the structure contains a whole message or a chunk of a message.

Table 2-6 lists status values that are related to sequenced messages:

Table 2-6: Status values for sequenced messages

Symbolic value	To indicate
CS_FIRST_CHUNK	The message text is the first chunk of the message.
CS_LAST_CHUNK	The message text is the last chunk of the message. An application sets both CS_FIRST_CHUNK and CS_LAST_CHUNK on if the message text in the structure is the entire message. An application sets neither CS_FIRST_CHUNK nor CS_LAST_CHUNK on if message text in the structure is a middle chunk.

The textlen field in the CS_SERVERMSG structure always reflects the length of the current message chunk.

All other fields in the CS_SERVERMSG are repeated with each message chunk.

Extended error data

Some server messages include “extended error data” associated with them. Extended error data is simply additional information about the error.

For Adaptive Server messages, the additional information most typically indicates which column or columns provoked the error.

What is extended error data good for?

Client applications that allow users to enter or edit data often need to report errors to their users at the column level. The standard server message mechanism, however, makes column-level information available only within the text of the server message. Extended error data provides a means for applications to conveniently access column-level information.

For example, imagine a client application that allows users to enter and edit data in the titleauthor table in the pubs2 database. titleauthor uses a **key** composed of two columns, au_id and title_id. Any attempt to enter a row with an au_id and title_id that match an existing row causes a “duplicate key” message to be sent to the client application.

On receiving this message, the client application needs to identify the problem column or columns to the end user, so that the user can correct them. This information is not available in the duplicate key message, except in the message text. The information is available, however, as extended error data.

Sending extended error data to a client

An Open Server application sets the CS_HASEED bit of the status field of the CS_SERVERMSG structure if extended error data is available for the message.

An Open Server application sends extended error data as parameters to the srv_sendinfo routine. The application describes, binds, and sends the error parameters using the srv_descfmt, srv_bind, and srv_xferdata routines, respectively.

The application must describe, bind, and send the error parameters immediately after calling srv_sendinfo, before sending other results and before calling to srv_senddone. The application must invoke srv_descfmt, srv_bind and srv_xferdata with a type argument of SRV_ERRORDATA.

If an application calls srv_sendinfo with the status field of the CS_SERVERMSG structure set to CS_HASEED but fails to send error parameters, Open Server raises a fatal process error when the application calls srv_senddone.

Connection migration

Connection migration allows an Open Server application to dynamically distribute its load, provide transparent failover support, and, where there are multiple Open Server applications that perform different functions, to redirect a client to an Open Server that can fulfill the client's request.

The application programming interface (APIs) discussed below enable Open Server to start, complete, and cancel a migration request, and to react to migration messages from the client. It can also detect whether a new connection is a migrating connection and retrieves a unique identifier from the connection.

In-batch migration and idle migration

With in-batch migration, the client migrates while waiting for results from the original server. Conversely, with idle migration, the client is not waiting for any result from the original server.

In-batch migration enables Open Server to delay sending or completing results until after a connection has migrated. This is useful if Open Server cannot service the specific request or if it has no time to complete the request. With in-batch migration, Open Server can send a part of the result from the original server, and, after migration, the server the client has migrated to can send rest of the result from the SRV_MIGRATE_RESUME event handler.

Note The original server can send a complete result to the client, in which case the new server does not send any result. Likewise, the original server may not send any result to the client, in which case, the new server must send the complete result to the client.

In an in-batch migration, your application must ensure that the unsent commands and messages are part of the client context. The new server must also access the number of rows affected by the command and the transaction state of the connection. The new server sends this information to the client using srv_senddone().

Context migration

Open Server supports seamless migration of the client's connection. However, the responsibility of sharing and migrating the client's context lies with your application. You can implement context migration in different ways, such as through a shared file system or a network communication.

For an in-batch migration, the server that the client is migrating to does not know what type of event was raised in the original server. If your application needs this information, you must migrate the information as part of the client's context.

With idle migration, the client is not waiting for actual results from Open Server. Because there is no active query to migrate, idle migration is easier to implement than in-batch migration. However, idle migration still requires that your application fulfills any pending requests that may arrive before the client starts the migration.

APIs used in connection migration

This section discusses the APIs that support connection migration. For more information about using these APIs, see “Instructing clients to migrate to a different server” on page 48.

CS_REQ_MIGRATE

The CS_REQ_MIGRATE request capability indicates if a client supports the migration protocol and if the client is capable of migrating to another server when requested. You can use srv_capability_info() to retrieve the CS_REQ_MIGRATE capability information. For example:

```
CS_RETCODE ret;
CS_BOOL migratable;
ret = srv_capability_info(sp, CS_GET, CS_CAP_REQUEST,
   CS_REQ_MIGRATE, &migratable);
```

SRV_CTL_MIGRATE

SRV_CTL_MIGRATE is a srv_send_ctlinfo() control type. You can use SRV_CTL_MIGRATE to send a migration request to the client or cancel a previous migration request, provided the client supports migration and has received a session ID when it first connected to the session.

Requesting a client migration

This sample code sends a request to the client to migrate to server “target”:

```
CS_RETCODE ret;
SRV_CTLITEM *srvitems;
CS_CHAR *target;
/*
** request a migration to server 'target'
*/
srvitems = (SRV_CTLITEM *) srv_alloc(sizeof
   (SRV_CTLITEM));
srvitems[0].srv_ctlitemtype = SRV_CT_SERVERNAME;
srvitems[0].srv_ctllength = strlen(target);
srvitems[0].srv_ctlptr = target;
ret = srv_send_ctlinfo(sp, SRV_CTL_MIGRATE, 1,
   srvitems);
srv_free(srvitems);
```

Your application can still send the SRV_CTL_MIGRATE control type even if a migration has already been requested. Open Server cancels the earlier migration request and sends a new request to the client. The return values for a new migration request are:

Return value	Description
CS_SUCCEED	The migration request was sent successfully.
CS_FAIL	The migration request failed due to one of the following reasons: • The Open Server thread does not support connection migration. • An earlier migration request was sent and the client has started migrating to the new server.

Cancelling a migration

You can also use the SRV_CTL_MIGRATE control type to cancel a previous migration request. In this case, *paramcnt* must be 0 and *param* must be a NULL pointer. For example:

```
ret = srv_send_ctlinfo(sp, SRV_CTL_MIGRATE, 0, NULL);
if (ret != CS_SUCCEED)
{
...
}
```

SRV_CTL_MIGRATE can be used by any thread in an Open Server application. However, a thread cancelling the migration of a client thread's connection has different requirements than a client thread cancelling its own connection migration:

- Any Open Server thread can cancel a migration, however, the cancellation must be requested *before* the SRV_MIGRATE_STATE event handler informs the client thread that the client is ready to migrate.
- The client thread can cancel a migration even inside the SRV_MIGRATE_STATE event handler. However, the client thread cannot cancel a migration after it exits the SRV_MIGRATE_STATE event with a SRV_MIG_READY state.

The return values of a migration cancellation are:

Return value	Description
CS_SUCCEED	The migration request was cancelled successfully.
CS_FAIL	The migration cancellation failed due to one of these reasons: • There is no migration in progress. • The client has started migrating to the new server.

Note Open Server does not trigger a new migrate state event when a migration request is successfully cancelled.

SRV_MIGRATE_RESUME

When a client migrates to a new server while waiting for results, the new server invokes the SRV_MIGRATE_RESUME event after the client connection has successfully migrated. If the migration request failed or is cancelled, the event is invoked from the original server.

In the SRV_MIGRATE_RESUME event handler, your application does not have to send any actual result to the client, except for the SRV_DONE_FINAL result type that must *always* be sent. The only result that the default SRV_MIGRATE_RESUME sends to the client is SRV_DONE_FINAL.

This is an example of a SRV_MIGRATE_RESUME event handler:

```
/*
** Simple migrate_resume event handler.
*/
CS_RETCODE CS_PUBLIC
migrate_resume_handler(SRV_PROC *sp)
{
   CS_RETCODE ret;
   ret = srv_senddone(sp, SRV_DONE_FINAL,
      CS_TRAN_COMPLETED, 0);
   if (ret == CS_FAIL)
   {

      ...

   }
   return CS_SUCCEED;
}

...
```

```
/*
** Install the migrate-resume event handler
*/
srv_handle(server, SRV_MIGRATE_RESUME,
      migrate_resume_handler);
...
```

SRV_MIGRATE_STATE

SRV_MIGRATE_STATE is an event that is triggered whenever the migration state has transitioned to SRV_MIG_READY or SRV_MIG_FAILED, the transition being a result of a migration message from a client. The SRV_MIGRATE_STATE event handler is invoked in these situations:

SRV_T_MIGRATE_STATE	Situation	Possible application action
SRV_MIG_READY	The client has sent a message to the server indicating that it has detected the request and is ready to migrate. The server determines whether to continue the migration or not.	One of the following: • Make the context available for the other servers. • Cancel the migration if the application decides that migration is no longer needed. • Request another migration if a new migration target has been selected.
SRV_MIG_FAILED	The client has sent a message to the server indicating that the migration failed.	One of the following: • Access the client context and continue serving the connection. • Request another migration.

This is an example of a SRV_MIGRATE_STATE event handler:

```
/*
** Simple migrate-state event handler
*/
CS_RETCODE CS_PUBLIC
migrate_state_handler(SRV_PROC *sp)
{
   SRV_MIG_STATE migration_state;
   ret = srv_thread_props(sp, CS_GET,
      SRV_T_MIGRATE_STATE, &migration_state,
      sizeof (migration_state), NULL);

   ...
```

```
   switch(migration_state)
   {
      case SRV_MIG_READY:
           ...
      case SRV_MIG_FAILED:
           ...
   }
}

...

/*
** Install the migrate-state change event handler
*/
srv_handle(server, SRV_MIGRATE_STATE,
migrate_state_handler);
...
```

When working with the SRV_MIGRATE_STATE event handler:

- If the client thread cancels the migration from inside the SRV_MIGRATE_STATE event handler, your application must make sure that the context is consistent. For instance, you cannot expect a different server to use the context your application has created.
- If a new migration request is sent from within the SRV_MIGRATE_STATE event handler, this handler is called again when the client is ready to start with the new requested migration.

SRV_T_MIGRATE_STATE property and SRV_MIG_STATE enumerated type

SRV_T_MIGRATE_STATE indicates the migration state of the client. SRV_T_MIGRATE_STATE is a read-only property that any thread can access. The possible migration states are:

State	Value	Description
SRV_MIG_NONE	0	There is no migration in progress.
SRV_MIG_REQUESTED	1	A migration has been requested by the server.
SRV_MIG_READY	2	The client has received the request and is ready to migrate.
SRV_MIG_MIGRATING	3	The client is now migrating to the specified server.
SRV_MIG_CANCELLED	4	The migration request has been cancelled.
SRV_MIG_FAILED	5	The client failed to migrate.

SRV_MIG_STATE is an enumerated datatype that models the SRV_T_MIGRATE_STATE property. Declare SRV_MIG_STATE as:

```
typedef enum
{
   SRV_MIG_NONE,
   SRV_MIG_REQUESTED,
   SRV_MIG_READY,
   SRV_MIG_MIGRATING,
   SRV_MIG_CANCELLED,
   SRV_MIG_FAILED
} SRV_MIG_STATE;
```

This sample code shows how you can retrieve SRV_T_MIGRATE_STATE values; in case of a successful migration, the client exits and the SRV_DISCONNECT event handler is called with a SRV_MIG_MIGRATING status:

```
CS_RETCODE ret;
SRV_MIG_STATE migration_state;
ret = srv_thread_props(sp, CS_GET, SRV_T_MIGRATE_STATE,
   &migration_state, sizeof (migration_state), NULL);
if (ret != CS_SUCCEED)
{
...
}
```

SRV_T_MIGRATED

SRV_T_MIGRATED is a Boolean property that indicates whether a connection is a new connection or a migrated connection. This read-only property is set to true when the client is migrating or has migrated to the server. This sample code retrieves the value of SRV_T_MIGRATED:

```
CS_RETCODE ret;
CS_BOOL migrated;
status = srv_thread_props(sp, CS_GET, SRV_T_MIGRATED,
   &migrated, sizeof (migrated), NULL);
```

SRV_T_SESSIONID

The SRV_T_SESSIONID is a thread property that retrieves the session ID that the client sends to Open Server. You can set the SRV_T_SESSIONID property using the srv_thread_props() function, given that:

- The srv_thread_props(CS_SET, SRV_T_SESSIONID) call is made inside the SRV_CONNECT event handler and,
- The client supports connection migration or high availability.

This sample code sets the SRV_T_SESSIONID property:

```
CS_RETCODE ret;
CS_SESSIONID hasessionid;
ret = srv_thread_props(sp, CS_SET, SRV_T_SESSIONID,
   hasessionid, sizeof(hasessionid), NULL);
```

Note For HA-failover, you must program an srv_negotiate() sequence to send the session ID to the client.

Instructing clients to migrate to a different server

This section discusses the requirements for an Open Server to migrate clients to other servers. When migrating clients to a different server your application must:

1 Create a unique session ID and send it to the clients in the connection handler.

2 Initiate connection migration.

3 Handle migration events.

4 Share the context of the connection, using the connection's session ID, to other servers.

5 (Optional) Act on ongoing migrations in existing handlers.

The following sections further discuss these activities.

Requesting a client to migrate

Open Server can use srv_send_ctlinfo() to send a migration request to the client. Client migration can be requested from any Open Server thread.

Managing the connect (SRV_CONNECT) event

In the SRV_CONNECT event handler, your application must:

- Check the SRV_T_MIGRATED property and determine if the connection is a migrated connection. If it is, your application must access the context based on the session ID provided by the client. The session ID can be retrieved using the SRV_T_SESSIONID thread property.

- Check CS_REQ_MIGRATE to determine if the client supports connection migration. If the client supports connection migration, your application must send a session ID using the SRV_T_SESSIONID property to the client if the client has not yet received a session ID. By assigning the client a session ID, your application can instruct the client to migrate when the need arises.

Managing the migrate state (SRV_MIGRATE_STATE) event

The SRV_MIGRATE_STATE event handler must manage the migration state changes and execute the actions appropriate for each change:

- SRV_MIGRATE_STATE changed to SRV_MIG_READY

  A "ready" migration state indicates that the client is prepared to migrate and, for now, is not going to send any request. In the SRV_MIGRATE_STATE event handler, Open Server shares the client context with the server the client is migrating to. Afterwards, your application can return from the event handler, and Open Server can automatically instruct the client to start the migration.

- SRV_MIGRATE_STATE changed to SRV_MIG_FAILED

  If the SRV_MIGRATE_STATE event handler is triggered because the migration state changed to "failed," your application must access the context again. Your application can request another migration attempt from the SRV_MIG_STATE event handler using the srv_send_ctlinfo() function. However, the client may have sent another query before it indicates it is ready to migrate again. The application must be able to service or migrate such a request.

Sharing client context

For servers to start and continue servicing a client, the servers must have access to the client's context which is identified by the client's session ID. Typically, the client's context contains data, such as global data, that event handlers for the client can access. The amount of context required for a connection depends on the service that the Open Server application provides. The more context-free the service is, the less context needs to be shared.

Managing the migrate resume (SRV_MIGRATE_RESUME) event

Your application sends the remaining results and messages to the client inside the SRV_MIGRATE_RESUME event handler. The results and messages that Open Server sends to the client depend on your application and the migration type. However, your application must end the SRV_MIGRATE_RESUME event handler by sending the SRV_DONE_FINAL result type to the client.

Managing the disconnect (SRV_DISCONNECT) event

In the SRV_DISCONNECT event handler, your application must check SRV_T_MIGRATE_STATE to determine the client's migration state:

- A migration state of SRV_MIG_REQUESTED indicates that the SRV_DISCONNECT event has been triggered because the Open Server application terminated the connection before the client could respond to the migration request.
- A migration state of SRV_MIG_MIGRATING indicates that the SRV_DISCONNECT event has been triggered because the client application, after a successful migrating to the new server, closed the connection.
- For all other migration states, the client must make sure that connection-specific context is cleaned up because no other server will pick up this context.

Managing in-batch migration

An event handler that runs for long periods of time must occasionally inspect the migration state. Other Open Server threads can send a migration request even while an event handler process is still running. In this case, the event handler, if it is able to, must interrupt the process, and postpone the generation and sending of results until the connection has migrated to the new server.

Attention handling

When a client sends an attention message to cancel an outstanding request, the SRV_T_GOTATTENTION thread property is set to CS_TRUE and the SRV_ATTENTION event handler is called. The specific attention handling needs of a connection migration are described below:

- For the SRV_MIGRATE_STATE event handler and SRV_MIG_READY state:

If the attention message arrives in the SRV_MIGRATE_STATE event handler before the client indicates that it is ready to migrate, Open Server acknowledges the attention when the SRV_MIGRATE_STATE event handler ends. This completes the request from the client. After a successful migration, the server that the client has migrated to does not receive this attention message and, because the client is not waiting for results from Open Server, the SRV_MIGRATE_RESUME event handler is not called.

Thus, your application must check if the SRV_T_GOTATTENTION property is set to CS_TRUE before making the context available to other servers. If SRV_T_GOTATTENTION is set to CS_TRUE, you must update the context to indicate that the client has cancelled the operation.

- For the SRV_MIGRATE_RESUME event handler:

  If the client has sent the attention message after the client indicated that it is ready to migrate and the migration succeeded, the attention is sent to the server to which the client has migrated. It is therefore possible that, after a successful migration, an attention can be received by the SRV_MIGRATE_RESUME event handler even if the original server has updated the context to reflect the cancellation. Thus, your application must check if the client has sent an attention to the server before it can execute the SRV_MIGRATE_RESUME event handler.

Disconnecting Open Server

Your application can terminate a client connection even when a migration has been requested; however, a new client command that is sent just before Open Server issued the termination command may get lost. To avoid this, your application must:

- If possible, avoid terminating connections when a client is instructed to migrate.
- If there is a need to disconnect a client, Open Server must set a reasonable wait time before requesting the migration. This gives a client the time to detect the migration request before it issues another command.
- When Open Server terminates a connection, the SRV_DISCONNECT event handler is called. Inside this handler, ensure that the context is available to other servers if the migration state is still set to SRV_MIG_REQUESTED.

Accepting connections from migrated clients

Open Server can determine if a new connection is migrating or has migrated by inspecting the SRV_T_MIGRATED property in the SRV_CONNECT event handler. If SRV_T_MIGRATED is TRUE, you can retrieve the session ID from the client using the SRV_T_SESSIONID property. You can also change the session ID, but this is not required to migrate the client later.

If the client was executing a command when it migrated, the SRV_MIGRATE_RESUME event is triggered and Open Server can send results to the client to complete the command. Your application is responsible for retrieving the session information. You must also determine whether you still need to send results to the client from within the SRV_MIGRATE_RESUME event handler.

Error messages

These are the error messages that you might encounter when using the connection migration feature:

Error	Description
srv_thread_props(): Property - SRV_T_SESSIONID is not available	You try to retrieve a session ID that the client has not yet received.
srv_send_ctlinfo(SRV_CTL_MIGRATE): Connection cannot migrate	The client does not support migration.
srv_send_ctlinfo(SRV_CTL_MIGRATE): Migration can no longer be cancelled	You requested for a cancellation of a migration that has already started.
Migration failed but no SRV_MIGRATE_STATE handler was installed	The default SRV_MIGRATE_STATE handler detects a migration failure.

CS_BROWSEDESC structure

srv_tabname and srv_tabcolname use a CS_BROWSEDESC structure to return information about the underlying structure of a browse mode query.

A CS_BROWSEDESC structure is defined as follows:

```
/*
 ** CS_BROWSEDESC
 ** The Open Server browse column description
 ** structure.
```

```
*/
typedef struct _cs_browsedesc
{
    CS_INT       status;
    CS_BOOL      isbrowse;
    CS_CHAR      origname[CS_MAX_NAME];
    CS_INT       orignlen;
    CS_INT       tablenum;
    CS_CHAR      tablename[CS_OBJ_NAME];
    CS_INT       tabnlen;
} CS_BROWSEDESC;
```

where:

- status is a bitmask of the following symbols, OR'd together:

  CS_EXPRESSION indicates the column is the result of an expression – for example, "sum*2" in the query:

  ```
  select sum*2 from areas
  ```

  CS_RENAMED indicates that the column's heading is not the original name of the column. Columns will have a different heading from the column name in the database if they are the result of a query of the form:

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

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

  A column can be updated if it is neither a timestamp column nor the result of an expression and if it belongs to a browsable table. A table is browsable if it possesses a unique index and a timestamp column.

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

- origname is the original name of the column in the database.

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

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

- tablenum is the number of the table to which the column belongs. The first table in a select statement's "from" list is table number 1; the second is table number 2; and so forth.

- tablename is the name of the table to which the column belongs.

- tabnlen is the length, in bytes, of tablename.

CS_DATAFMT structure

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

- srv_bind uses a CS_DATAFMT structure to describe a source or destination program variable.
- srv_descfmt uses a CS_DATAFMT structure to describe the client data.
- cs_convert requires CS_DATAFMT structures to describe source and destination data.

Most routines use only a subset of the fields in a CS_DATAFMT. For example, srv_bind does not use the name and usertype fields, and srv_descfmt does not use the format field. For information on which fields in the CS_DATAFMT a routine uses, see that routine's reference page.

A CS_DATAFMT structure is defined as follows:

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

    /*
       ** The following field is not used in Open Server.
       ** It must be set to 1 or 0.
       */
       CS_INT     count;

     /*
       ** These fields are used to support user-defined
       ** datatypes and international datatypes:
       */
       CS_INT    usertype;            /* User-defined type.*/
       CS_LOCALE *locale;             /* Locale information. */

} CS_DATAFMT;
```

where:

- name is the name of the data, that is, the column or parameter name.

- namelen is the length, in bytes, of name. Set namelen to CS_NULLTERM to indicate a null terminated name. Set namelen to 0 if name is NULL.e
- datatype is the datatype of the data, which is one of the Open Server datatypes listed in “Types” on page 199.

Note The datatype field is used to describe the Open Server datatype of the data. usertype is only used if the data has an application-defined datatype in addition to an Open Server datatype.

For example, this Adaptive Server command creates the Adaptive Server user-defined type birthday:

```
sp_addtype birthday, datetime
```

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

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

An Open Server application that supported user-defined datatypes would return this information to the client by setting the CS_DATAFMT datatype field to CS_DATETIME_TYPE and the usertype field to the user-defined ID for the type birthday.

- format describes the destination format of character or binary data. format is a bitmask of these symbols, OR’d together. Table 2-7 summarizes the legal values for format

Table 2-7: Values for format (CS_DATAFMT)

Symbol	To indicate	Notes
CS_FMT_NULLTERM	The data should be null terminated.	For character or text data
CS_FMT_PADBLANK	The data should be padded with blanks to the full length of the destination variable.	For character or text data
CS_FMT_PADNULL	The data should be padded with NULLs to the full length of the destination variable.	For binary, image, character, or text data
CS_FMT_UNUSED	Neither padding nor null termination is applicable to the datatype.	For all datatypes

- maxlength can represent various lengths, depending on which Open Server routine is using the CS_DATAFMT. Table 2-8 describes the various lengths maxlength can represent:

Table 2-8: Meaning of maxlength (CS_DATAFMT)

Open Server routine	maxlength is
srv_bind	The length of the bind variable
srv_descfmt	The maximum possible length of the column or parameter being described
cs_convert	The length of the source data and the length of the destination buffer space

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

  Legal values for scale are from CS_MIN_SCALE to CS_MAX_SCALE. The default scale is CS_DEF_SCALE.

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

  - scale must be less than or equal to precision.
  - precision is the precision of the data. It is used only with decimal or numeric datatypes.

    Legal values for precision are from CS_MIN_PREC to CS_MAX_PREC. The default precision is CS_DEF_PREC.

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

- precision must be greater than or equal to scale.
- status is a bitmask used to indicate various types of information. Table 2-9 summarizes the types of information that status can contain:

Table 2-9: Values for status (CS_DATAFMT)

Symbolic value	To indicate
CS_CANBENULL	The column can contain NULL.
CS_DESCIN	The CS_DATAFMT structure describes a Dynamic SQL input parameter.
CS_DESCOUT	The CS_DATAFMT structure describes a Dynamic SQL output parameter.
CS_HIDDEN	The column is a "hidden" column that has been exposed.
CS_INPUTVALUE	The parameter is an input parameter value for a cursor open command or a non-return RPC parameter.
CS_KEY	The column is a key column.
CS_RETURN	The parameter is a return parameter to an RPC command.
CS_TIMESTAMP	The column is a *timestamp* column. An application uses timestamp columns when performing browse-mode updates.
CS_UPDATABLE	The column is an updatable cursor column.
CS_UPDATECOL	The parameter is the name of a column in the update clause of a cursor declare command.
CS_VERSION_KEY	The column is part of the version key for the row. Adaptive Server uses version keys for positioning.
CS_NODEFAULT	There is no default specified for the parameter.

- count is not used by Server-Library routines. It should always be set to 0 or 1.
- usertype is the user-defined datatype, if any, of data returned.
- locale is a pointer to a CS_LOCALE structure containing localization information. Set locale to NULL if localization information is not required.

CS_IODESC structure

A CS_IODESC, also called an "I/O descriptor structure," describes text or image data.

An Open Server application calls srv_text_info with a cmd argument of CS_GET when processing text or image data from a client. Only the total_textlen field of the CS_IODESC argument is filled in by this call.

If the application is sending columns of data to a client, it calls srv_text_info with a cmd argument of CS_SET. In this scenario, the CS_IODESC structure describes a text or image column being sent. A CS_IODESC is defined as follows:

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

where:

- iotype indicates the type of I/O to perform. For text and image operations, iotype always has the value CS_IODATA.
- datatype is the **datatype** of the data object. The only legal values for datatype are CS_TEXT_TYPE and CS_IMAGE_TYPE.
  locale is not currently used in Open Server. Set to NULL.
  usertype is not used in Open Server.
- total_txtlen is the total length, in bytes, of the text or image value.
- offset is reserved for future use.
- log_on_update describes whether to log the update to this text or image value.
- name is the name of the text or image column.
- namelen is the length, in bytes, of name, or CS_NULLTERM to indicate a null-terminated name.

- timestamp is the text timestamp of the column. A text timestamp marks the time of a text or image column's last modification.
- timestamplen is the length, in bytes, of timestamp.
- textptr is an array of text or image bytes for column insertion or retrieval.
- textptrlen is the length, in bytes, of textptr.

CS-Library

CS-Library is a collection of utility routines and structures useful or necessary to both Open Server and Open Client applications. In past versions, Server-Library and Client-Library provided such utility routines and structures separately, resulting in unnecessary duplication.

Common routines

CS-Library includes routines to support:

- Datatype conversion
- Arithmetic operations
- Character-set conversion
- Datetime operations
- Sort-order operations
- Localization routines

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

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

Some of these routines offer functionality provided by existing Server-Library routines. While it is not yet necessary to replace the Server-Library routines with their CS-Library counterparts, it may be in the future.

Common data structures

In addition to common routines, CS-Library provides data structures useful to both Open Client and Open Server applications. Among these data structures is a CS_CONTEXT structure, which contains information about an application programming environment, or "context."

An Open Server application programmer can tailor an application's behavior by setting global application attributes stored in this structure. "Properties" on page 139 discusses this feature in detail.

Other CS-Library structures contain information about data passed between Open Client and Open Server applications.

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

Error handling

An Open Server application should install a message **callback routine** with the cs_config routine to report CS-Library errors. A standard Open Server error handler installed with srv_props will not catch CS-Library errors, such as data conversion errors generated in a call to cs_convert.

If an Open Server application has not installed a CS-Library handler, Open Server installs a default handler when the application calls srv_version. This default handler writes CS-Library errors to the Open Server log.

For details on handling CS-Library errors and for more general information about CS-Library, see the Open Client and Open Server *Common Libraries Reference Manual*.

CS_SERVERMSG structure

A CS_SERVERMSG structure contains information about a server error message.

Open Server uses a CS_SERVERMSG structure to send error messages to a client, through the srv_sendinfo routine.

A CS_SERVERMSG structure is defined as follows:

```
/*
 ** CS_SERVERMSG
 ** The server message structure.
 */

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

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

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

where:

- msgnumber is the Open Server or application **message number** to report to the client.
- state is the state in which the message was generated. The application defines this.
- severity is the severity of the message.
- text is the text of the message.
- textlen is the length, in bytes, of text.

- svrname is the name of the server that generated the message. This value can be the name of the Open Server application running currently, or a different name.
- svrnlen is the length, in bytes, of svrname.
- proc is the name of the **stored procedure** (if any) that caused the message.
- proclen is the length, in bytes, of proc.
- line is the line number within the stored procedure (if any) that caused the message.
- status contains information on whether the message chunk is the first, last, or a middle part of the message, and whether it includes extended error data. Since status is a byte-ordered flag, you can set it to more than one value. For example:

  ```
  mrec.status = CS_FIRST_CHUNK | CS_LAST_CHUNK;
  ```

  where mrec is declared as a CS_SERVERMSG structure.

  Table 2-10 describes the legal values for status:

Table 2-10: Values for status field of CS_SERVERMSG structure

Value	Meaning
CS_HASEED	There is extended error data associated with the message.
CS_FIRST_CHUNK	The message text contained in text is the first chunk of the message. If CS_FIRST_CHUNK and CS_LAST_CHUNK are both on, then text contains the entire message. If neither CS_FIRST_CHUNK nor CS_LAST_CHUNK is on, then text contains a middle chunk of the message.
CS_LAST_CHUNK	The message text contained in text is the last chunk of the message. If CS_FIRST_CHUNK and CS_LAST_CHUNK are both on, then text contains the entire message. If neither CS_FIRST_CHUNK nor CS_LAST_CHUNK is on, then text contains a middle chunk of the message.

- sqlstate is a byte string describing the error.

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

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

For more information on sending a message in chunks, see “Client command errors” on page 38.

Cursors

Adaptive Server Enterprise implements cursors, which are supported by Server-Library and Client-Library.

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

For information on how cursors are supported by Client-Library, see the Open Client *Client-Library/C Reference Manual*.

Cursor overview

A cursor is a symbolic name that is linked with a SQL statement. Declaring a cursor establishes this link. The SQL statement can be:

- A SQL select statement
- A Transact-SQL execute statement
- A Dynamic SQL prepared statement

The SQL statement associated with a cursor is called the *body* of the cursor. When a client opens a cursor, it executes the body of the cursor, generating a result set. The Open Server application is responsible for detecting cursor requests and passing cursor results back to the client.

Advantages of cursors

Cursors allow a client application to access individual rows within a result set, rather than merely retrieve a complete set of data rows.

A single connection can have multiple cursors open at the same time. All of the cursor result sets are simultaneously available to the application, which can fetch data rows from them at will. This is in contrast to other types of result sets, which must be handled one row at a time in a sequential fashion.

Further, a client application can update underlying database tables while actively fetching rows in a cursor result set.

Open Server applications and cursors

This section contains basic information on Open Server cursor support. For specific information on how to structure a SRV_CURSOR event handler, see “How to respond to specific requests” on page 72.

How are cursor requests generated?

A client application requests a cursor by issuing a cursor command to an Open Server application.

A client application calls the Client-Library command ct_cursor to initiate a cursor command. For more information on ct_cursor, see the Open Client *Client-Library/C Reference Manual*.

A cursor request causes Open Server to generate a SRV_CURSOR event. To respond to cursor requests, an Open Server application must include a SRV_CURSOR event handler.

Types of cursor commands

Table 2-11 summarizes the types of cursor commands a client can issue:

Table 2-11: Summary of cursor commands

Type of command	What it does
Declare	Associates a cursor name with the body of the cursor.
Open	Executes the body of the cursor, generates a cursor result set.
Information	Reports the status of the cursor, or sets the cursor row fetch count.
Fetch	Fetches rows from the cursor result set.
Update or Delete	Updates or deletes the contents of the current cursor row.
Close	Makes the cursor result set unavailable. Reopening a cursor regenerates the cursor result set.
Deallocate	Renders the cursor nonexistent. A cursor that has been deallocated cannot be reopened.

A typical client application issues cursor commands in the order in which they are listed in Table 2-11, but the order can vary. For example, a client might fetch against a cursor, close the cursor, then reopen and fetch rows from it again.

How is cursor information exchanged with a client?

A SRV_CURSOR event handler uses the srv_cursor_props routine and the SRV_CURDESC structure to exchange cursor information with a client. srv_cursor_props sends current information to a client and retrieves cursor information from a client by accessing a SRV_CURDESC structure.

For more information on the srv_cursor_props routine, see srv_cursor_props on page 253.

Because a client and server can exchange information about multiple cursors during a single connection session, they need to uniquely identify each cursor. An Open Server application responds to a cursor declaration by sending back a unique cursor ID. The client and the server refer to the cursor by this ID for the cursor's lifetime.

SRV_CURDESC structure

A SRV_CURDESC structure contains information about a cursor, including:

- The cursor's unique ID
- The type of cursor command most recently issued by the client
- The status of the cursor

A SRV_CURDESC structure is defined as follows:

```
/*
 ** SRV_CURDESC
 ** The Open Server cursor description
 ** structure.
 */

 typedef struct srv_curdesc
 {
     CS_INT       curid;
     CS_INT       numupcols;
     CS_INT       fetchcnt;
     CS_INT       curstatus;
     CS_INT       curcmd;
     CS_INT       cmdoptions;
```

```
    CS_INT      fetchtype;
    CS_INT      rowoffset;
    CS_INT      curnamelen;
    CS_CHAR     curname[CS_MAX_CHAR];
    CS_INT      tabnamelen;
    CS_CHAR     tabname[CS_MAX_CHAR];
    CS_VOID     *userdata;

} SRV_CURDESC;
```

Table 2-12 describes each field in a SRV_CURDESC structure:

Table 2-12: Fields in a SRV_CURDESC structure

Field name	Description	Notes
curid	The current cursor identifier	The Open Server application must set curid when responding to a CS_CURSOR_DECLARE command from the client. Any subsequent commands from the client that pertain to the declared cursor use curid as an identifier. curid is set to 0 if there is no current cursor identifier or if the client is requesting the status of all available cursors.
numupcols	The number of columns in a cursor update clause	numupcols is set to 0 if there are no update columns. This information is available when the cursor is declared.
fetchcnt	The current row fetch count for this cursor—that is, the number of rows that will be sent to the client in response to a CS_CURSOR_FETCH command	fetchcnt is set when a CS_CURSOR_INFO command is received from the client or is sent to the client in response to such a command. fetchcnt is set to 1 if the client has not explicitly set a row fetch count. If the Open Server application cannot support the requested fetch count, it can set this field to a different value before responding.
curstatus	The status of the current cursor	Open Server sets the cursor status in response to the cursor command received from the client. See “Values for curstatus” on page 69 for a list of legal values.
curcmd	The current cursor command type	See Table 2-14 for a list of legal values.
cmdoptions	Any options associated with the cursor command	Not all commands have associated options. The value of cmdoptions depends on the cursor command. Table 2-14 describes the possible values for cmdoptions, by command.

Field name	Description	Notes
fetchtype	The type of fetch requested by a client	fetchtype is described when a CS_CURSOR_FETCH command is received from the client. The valid fetch types and their meanings are as follows: • CS_NEXT – next row • CS_PREV – previous row • CS_FIRST – first row • CS_LAST – last row • CS_ABSOLUTE – row identified in the rowoffset field • CS_RELATIVE – current row plus or minus value in the rowoffset field. Requests to an Adaptive Server will always have a fetchtype of CS_NEXT.
rowoffset	The row position for CS_ABSOLUTE or CS_RELATIVE fetches	rowoffset is undefined for all other fetch types. rowoffset is set when a CS_CURSOR_FETCH command is received from the client.
curnamelen	The length of the cursor name in curname	curnamelen is zero if curname is not valid. curnamelen returns the length of the cursor name.
curname	The name of the current cursor	
tabnamelen	The length of the table name in tabname	tabnamelen is zero if tabname is not valid. tabnamelen returns the length of the table name. tabnamelen is described when a CS_CURSOR_UPDATE or CS_CURSOR_DELETE command is received from the client.
tabname	The table name associated with a cursor update or delete command	tabname is the table name associated with a cursor update or delete command. tabname is described when a CS_CURSOR_UPDATE or CS_CURSOR_DELETE command is received from the client.
userdata	A pointer to private data space	This field allows applications to associate data with a particular cursor without using global or static variables. Open Server does not manipulate userdata; it is provided only for the convenience of Open Server application programmers.

Values for *curstatus*

The curstatus field of the SRV_CURDESC structure is a bitmask that can take any combination of these values:

Table 2-13: Values for curstatus (SRV_CURDESC)

Value	Meaning
CS_CURSTAT_DECLARED	The cursor has been declared. This status is reset after the next cursor command has been processed.
CS_CURSTAT_OPEN	The cursor has been opened.
CS_CURSTAT_ROWCNT	The cursor has specified the number of rows that should be returned for the CS_CURSOR_FETCH command.
CS_CURSTAT_RDONLY	The cursor is read-only; it cannot be updated. The Open Server application should return an error to the client if a CS_CURSOR_UPDATE or CS_CURSOR_DELETE is received for this cursor.
CS_CURSTAT_UPDATABLE	The cursor can be updated.
CS_CURSTAT_CLOSED	The cursor was closed but not deallocated. It can be opened again later. This status is also set upon declaration of a cursor. Open Server clears it when a CS_CURSOR_OPEN is received and resets it when a CS_CURSOR_CLOSE is received.
CS_CURSTAT_DEALLOC	The cursor was closed and deallocated. No other status flags should be set at this time.

Values for *curcmd*

The curcmd field of the SRV_CURDESC structure can take one of the values described in Table 2-14. The table also lists the relevant cmdoptions values.

Table 2-14: Values for curcmd (SRV_CURDESC)

Value	Meaning	Legal values for cmdoptions
CS_CURSOR_CLOSE	Cursor close command.	SRV_CUR_DEALLOC or SRV_CUR_UNUSED. SRV_CUR_DEALLOC indicates that the cursor will never be reopened. The Open Server application should delete all associated cursor resources. The cursor ID number can be reused.
CS_CURSOR_DECLARE	Cursor declare command. The application can obtain the actual text of the cursor statement through srv_langlen and srv_langcpy.	SRV_CUR_UPDATABLE, SRV_CUR_RDONLY, or SRV_CUR_DYNAMIC. SRV_CUR_DYNAMIC indicates that the client declares the cursor against a dynamically prepared SQL statement; in this case, the text of the cursor statement is actually the name of the prepared statement.
CS_CURSOR_DELETE	Cursor delete command. Performs a positional row delete through a cursor.	There are no valid options for this command. cmdoptions will always have the value SRV_CUR_UNUSED.
CS_CURSOR_FETCH	Cursor fetch command. Performs a row fetch through a cursor.	There are no valid options for this command. cmdoptions will always have the value SRV_CUR_UNUSED.

Value	Meaning	Legal values for cmdoptions
CS_CURSOR_INFO	Cursor information command. The client sends this command to the Open Server application to set the cursor row fetch count or to request cursor status information. The Open Server application sends this command to the client in response to any cursor command (including CS_CURSOR_INFO itself) to describe the current cursor.	SRV_CUR_SETROWS when the client describes the current row fetch count. The fetchcnt field contains the requested fetch count. SRV_CUR_ASKSTATUS when the client requests status information about the current cursor. This generally occurs when the client has sent an attention and wants to see which cursors are still available afterwards. The curid field contains 0. The Open Server application should send back a CS_CURSOR_INFO response for each cursor currently available. SRV_CUR_INFORMSTATUS when the Open Server application responds to a CS_CURSOR_INFO command. The curstatus field contains the cursor status.
CS_CURSOR_OPEN	Cursor open command.	SRV_CUR_HASARGS or SRV_CUR_UNUSED.
CS_CURSOR_UPDATE	Cursor update command. Performs a positional row update through a cursor. The Open Server application can obtain the actual text of the cursor update statement by calling srv_langlen and srv_langcpy.	SRV_CUR_HASARGS or SRV_CUR_UNUSED.

Handling cursor requests

An Open Server application uses a SRV_CURSOR event handler to handle cursor requests. The handler includes code to detect which of the cursor commands has been issued and to respond with the appropriate information.

The event handler first determines the current cursor and the cursor command that triggered the SRV_CURSOR event by calling srv_cursor_props with the cmd argument set to CS_GET. Open Server then fills the curcmd field of the Open Server application's SRV_CURDESC structure with the command type.

The application can then determine what other information it needs to retrieve, if any, as well as what data to send back to the client. In some cases, it may need to retrieve parameter formats and parameters; in others, it may want to ascertain the status of the current cursor and the number of rows to fetch. In some cases, it may only need to send back a CS_CURSOR_INFO command; in others, it may need to send back result data or return parameters.

How to respond to specific requests

This section describes how a SRV_CURSOR event handler should respond to specific types of cursor requests.

Prior to calling srv_cursor_props with cmd set to CS_SET, an Open Server application must always set the curid field, and any other pertinent fields, in the SRV_CURDESC structure.

Table 2-15 summarizes the valid exchange of cursor requests and responses between a client and an Open Server application. The forward arrow (→) indicates that cmd is set to CS_GET—the Open Server application retrieves information from the client. The backward arrow (←) indicates that cmd is set to CS_SET—the Open Server application sends information to the client.

Table 2-15: Valid cursor requests and responses

Client action	Open Server application response
Declares a cursor (curcmd field of SRV_CURDESC contains CS_CURSOR_DECLARE)	– >Retrieve curcmd value from SRV_CURDESC (srv_cursor_props) – >Retrieve number of cursor parameters, if any (srv_numparams) – >Retrieve format of cursor parameters, if any (srv_descfmt with type argument set to SRV_CURDATA) – >Retrieve update column information, if any (srv_descfmt with type argument set to SRV_UPCOLDATA) – >Retrieve actual text of cursor command (srv_langlen and srv_langcpy) < – Set cursor ID. Set curcmd field to CS_CURSOR_INFO and curid field to unique cursor ID (srv_cursor_props) < – Send a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)
Requests the status of the current cursor or sends a fetch count (curcmd field of SRV_CURDESC contains CS_CURSOR_INFO)	– >Retrieve curcmd and curid cmdoptions values from SRV_CURDESC structure (srv_cursor_props) < – Send number of rows to be returned per fetch, if client set cmdoptions field to SRV_CUR_SETROWS (srv_cursor_props with curcmd set to CS_CURSOR_INFO) < – Send status of all available cursors, if client set cmdoptions field to SRV_CUR_ASKSTATUS. Set curcmd field to CS_CURSOR_INFO and curid field to cursor ID (srv_cursor_props once for each active—declared, opened or closed—cursor) < – Send a DONE packet (srv_senddone with status argument set to SRV_DONE_FINAL)

Client action	Open Server application response
Opens a cursor (curcmd field of SRV_CURDESC contains CS_CURSOR_OPEN)	- > Retrieve curcmd and curid values from SRV_CURDESC structure (srv_cursor_props) – > Retrieve number of cursor parameters, if any (srv_numparams) – > Retrieve format of cursor parameters and actual parameters, if any (srv_descfmt, srv_bind, srv_xferdata with type argument set to SRV_CURDATA) < – Send cursor status. Set curid to current cursor ID and curcmd to CS_CURSOR_INFO (srv_cursor_props) < – Describe result row formats (srv_descfmt with type argument set to SRV_ROWDATA) < – Send a DONE packet (srv_senddone with status argument set to SRV_DONE_FINAL)
Fetches rows (curcmd field of SRV_CURDESC contains CS_CURSOR_FETCH)	– > Retrieve curcmd and curid values from SRV_CURDESC structure (srv_cursor_props) < – Send result rows, fetchcnt times (srv_bind, srv_xferdata with type argument set to SRV_ROWDATA) < – Send a DONE packet (srv_senddone with status argument set to SRV_DONE_FINAL)

Client action	Open Server application response
Issues cursor update command (curcmd field of SRV_CURDESC contains CS_CURSOR_UPDATE) or Issues cursor delete command (curcmd field of SRV_CURDESC contains CS_CURSOR_DELETE)	– > Retrieve curcmd and curid values from SRV_CURDESC structure (srv_cursor_props) – > Retrieve key columns for current row (srv_descfmt, srv_bind, srv_xferdata with type argument set to SRV_KEYDATA) – > Retrieve number of update values, if curcmd is CS_CURSOR_UPDATE (srv_numparams) Retrieve actual text of update statement, if curcmd is CS_CURSOR_UPDATE (srv_langlen and srv_langcpy) – > Retrieve update values, if curcmd is CS_CURSOR_UPDATE (srv_descfmt, srv_bind, srv_xferdata, with type argument set to SRV_CURDATA) < – Send a DONE packet (srv_senddone with status argument set to SRV_DONE_FINAL)
Sends a cursor close command (curcmd field of SRV_CURDESC contains CS_CURSOR_CLOSE)	– > Retrieve curcmd and curid values from SRV_CURDESC structure (srv_cursor_props) < – Send cursor status (srv_cursor_props) < – Send a DONE packet (srv_senddone with status argument set to SRV_DONE_FINAL)

Note that:

- The Open Server application's response to a cursor command always concludes with a call to srv_senddone with a status argument of "SRV_DONE_FINAL."
- Once the Open Server application issues the first srv_cursor_props command with cmd set to "SET", any further information the application sends will apply to this cursor until a srv_senddone with a status argument of SRV_DONE_FINAL is issued.
- Internally, Open Server replaces the parameter formats received when the client declares a cursor with those received when the client opens a cursor. This procedure is necessary in case the format of the parameter passed in is not exactly the same as that of the parameter declaration. For example, a parameter may be declared as a CS_INT, but the parameter being passed in when the cursor is opened may be of type CS_SMALLINT.
- srv_xferdata sends a single row of data, and should be called as many times as the number in the current cursor's row fetch count, in response to a CS_CURSOR_FETCH command.

Key data

A key is a subset of row data that uniquely identifies a row. Key data uniquely describes the **current row** in an open cursor. It is used in processing CS_CURSOR_DELETE or CS_CURSOR_UPDATE commands. If a column is a key column, the status field of the CS_DATAFMT structure that describes the column has its CS_KEY bitmask set.

Update columns

If a client has declared a cursor as being "for update," the cmdoptions field of the SRV_CURDESC structure is set to CS_FOR_UPDATE and the numupcols field is set to the number of update columns associated with the cursor.

Example

The sample *ctos.c* includes code illustrating cursor command processing.

Scrollable cursors

The scrollable cursor feature provides a way to set the current position anywhere in the result set by specifying a NEXT, PREVIOUS, FIRST, LAST, ABSOLUTE or RELATIVE clause in a FETCH statement. It implements a scrollable cursor that is read-only with either an INSENSITIVE or a SEMI_SENSITIVE property.

Non-scrollable, insensitive cursors are also supported on Open Server and are set with the CS_NOSCROLL_INSENSITIVE option.

A new capability, CS_REQ_CURINFO3, is added to Open Server to support the new scrollable cursor feature. During login, CS_REQ_CURINFO3 allows a remote client connecting to Open Server to request scrollable cursor support.

SRV_CURDESC2 structure

The SRV_CURDESC2 scrollable cursor structure in Open Server is a superset of the SRV_CURDESC2 cursor structure described in “SRV_CURDESC structure” on page 65.

In addition to fields described in Table 2-12, Table 2-16 describes additional fields in the SRV_CURDESC2 structure:

Table 2-16: Additional fields in a SRV_CURDESC2 structure

Field name	Description
currow_pos	Current row position of a cursor.
curtotalrowcount	Total number of rows in the result set; only applies to insensitive, scrollable cursors.

Values for *curstatus*

In addition to options described in Table 2-13, the following cursor declare options are available in the *curstatus* field in SRV_CURDESC2:

Table 2-17: Values for curstatus (SRV_CURDESC2)

Value	Meaning
CS_CURSTAT _SCROLLABLE	A read-only, insensitive scrollable cursor.
CS_CURSTAT _INSENSITIVE	A read-only, non-scrollable, insensitive cursor. When such a cursor is specified, CS_CURSTAT_INSENSITIVE must be enabled, and CS_CURSTAT_SCROLLABLE must be disabled. When an insensitive, scrollable cursor is specified, both CS_CURSTAT_INSENSITIVE and CS_CURSTAT_SCROLLABLE must be enabled.
CS_CURSTAT _SEMISENSITIVE	A read-only, semi-sensitive, scrollable cursor. When such a cursor is specified, CS_CURSTAT_SCROLLABLE must also be enabled.

Values for *curcmd*

In addition to values in Table 2-14, the values described in Table 2-18 are available in the curcmd field of the SRV_CURDESC2 structure. The table also lists the relevant cmdoptions values:

Table 2-18: Values for curcmd (SRV_CURDESC2)

Value	Meaning	Legal values for cmdoptions
CS_NOSCROLL_ INSENSITIVE	Non-scrollable, insensitive cursors.	There are no valid options for this command. cmdoptions will always have the value SRV_CUR_UNUSED. **Note** If you use the CTOS application, do not use the ct_scroll_fetch routine with non-scrollable cursors. Instead, use the ct_fetch routine.
CS_CURSOR_ DECLARE	Scrollable cursor command options.	SRV_CUR_SCROLL, SRV_CUR_SCROLL_ INSENS, SRV_CUR_ SCROLL_SEMISENS, SRV_CUR_NOSCROLL_ INSENS. These cmdoptions are valid only at the cursor declare cycle, where the curcmd field of the SRV_CURDESC2 structure may contain one of these options, based on the remote client issuing a ct_cursor

srv_cursor_props2 routine

The srv_cursor_props2 routine is added to Open Server to support the SRV_CURDESC2 structure.

For pre-15.0 applications, you must use the SRV_CURDESC structure and srv_cursor_props routine, if the application sets CS_VERSION_125.

For 15.0 applications that support scrollable cursors on Open Server, use the SRV_CURDESC2 structure, and set the application to CS_VERSION_150.

The arguments for srv_cursor_props2 are as follows:

```
ret = srv_cursor_props2(SRV_PROC *spp, CS_INT cmd,
SRV_CURDESC2 *cdp);
```

Data stream messages

Data stream messages overview

Data stream messages provide a way for clients and Open Server applications to exchange information.

RPCs provide similar functionality, but in the client-to-server direction only. Messages work in both directions, making them suitable for a wide variety of communications purposes. For example, Sybase uses messages to perform security handshaking at login time.

A message consists of a message ID and zero or more parameters. The client and Open Server application must be programmed to agree on the meaning of each message ID.

User-defined message IDs must be greater than or equal to CS_USER_MSGID and less than or equal to CS_USER_MAX_MSGID. Message IDs SRV_MINRESMSG through SRV_MAXRESMSG are reserved for internal Sybase use.

A client application sends a message by calling ct_command with type set to CS_MSG_CMD. This triggers a SRV_MSG event in the Open Server application.

Retrieving client data stream messages

A message data stream triggers an Open Server application's SRV_MSG event handler. This handler can retrieve the client message. To do this:

1 Call srv_msg with cmd set to CS_GET and msgidp pointing to the buffer in which Open Server should place the message ID.

   srv_msg sets the *statusp* parameter to SRV_HASPARAMS if the message has parameters.

   For more information, see srv_msg on page 311.

2 Call srv_numparams, if necessary, to retrieve the number of parameters.

3 Call srv_descfmt, srv_bind, and srv_xferdata to describe and retrieve each parameter. For more information on how to process parameters, see the "Processing parameter and row data" on page 134.

An Open Server application can only retrieve messages using its SRV_MSG event handler.

Sending data stream messages to a client

An Open Server application can send a message to a client. To perform this function, the application:

1 Calls srv_msg with cmd set to CS_SET and msgidp pointing to the buffer containing the message ID.

A *statusp value of SRV_HASPARAMS indicates that the message has parameters. A value of SRV_NOPARAMS indicates that the message has no parameters.

For more information see srv_msg on page 311.

2 Calls srv_descfmt, srv_bind, and srv_xferdata to describe and send each parameter.

An Open Server application can send messages from within any event handler except the SRV_ATTENTION, SRV_CONNECT, SRV_DISCONNECT, SRV_URGDISCONNECT, and SRV_START handlers.

Directory services

This section describes what an Open Server application needs to do to use directory services. It has these sections:

- Specifying the directory driver
- Registering an Open Server application with a directory

A directory stores information as directory entries and associates a logical name with each entry. Each directory entry contains information about some network entity such as a user, a server, or a printer. A directory service (sometimes called a naming service) manages creation, modification, and retrieval of directory entries.

See the Open Client *Client-Library/C Reference Manual* for more information, and for information about how a client uses directory services.

Specifying a directory driver

Before running an application that uses directory services, make sure that the *libtcl.cfg* file has been edited to specify the correct directory service provider. The *libtcl.cfg* file is located in the *$SYBASE/$SYBASE_OCS/config* directory or in the path specified by the context property CS_LIBTCL_CFG. The server property, SRV_DS_PROVIDER, returns the name of the driver specified in the *libtcl.cfg* file. For more information about the *libtcl.cfg* file, see the Open Client and Open Server *Configuration Guide* for each platform. See srv_props on page 334 for information on the SRV_DS_PROVIDER property.

The Open Client and Open Server *Configuration Guide* for each platform tells which directory services are supported by Open Client and Open Server for that platform.

Registering an Open Server application with a directory

An Open Server application can specify the directory provider to use and register itself with the directory at start-up.

To specify a directory service provider other than the default, use srv_props to set the SRV_S_DS_PROVIDER server property. The default value for SRV_S_DS_PROVIDER is platform specific, and is specified in the Open Client and Open Server *Configuration Guide* for your platform.

To register an Open Server application with the directory service, use srv_props to set the SRV_S_DS_REGISTER server property to CS_TRUE (the default). Setting SRV_S_DS_REGISTER to CS_FALSE prevents the registration.

Set these properties after allocating and initializing the CS_CONTEXT structure (using cs_ctx_alloc and srv_version), and before calling srv_init.

When you call srv_init, the Open Server application:

- Retrieves its listening address from the directory service.
- Instructs the directory service to update the Open Server application's directory service entry if SRV_S_DS_REGISTER is set to CS_TRUE.
- The directory service then sets its "currentStatus" attribute to "active."

Open Server automatically uses the interfaces file as a backup directory when the directory service driver initialization fails. The srv_init call may fail to successfully access the specified directory service if any of the following occur:

- The *libtcl.cfg* file is not in the expected location, or is unreadable.

An informational error is returned.

- The directory service driver is not in the expected location, or is unreadable.

  An informational error is returned.

- The directory service is not responding to requests.

  An informational error is returned.

- The server entry cannot be found in the directory service.

  An error is returned indicating that there are no listeners; the Open Server application does not use the interfaces file as a backup directory in this case.

Dynamic SQL

Dynamic SQL allows a client application to execute SQL statements containing variables whose values are determined at runtime.

A client application prepares a dynamic SQL statement by associating a SQL statement containing placeholders with an identifier and sending the statement to an Open Server application to be partially compiled and stored. The statement is then known as a *prepared statement.*

When a client application is ready to execute a prepared statement, it defines values to substitute for the SQL statement's placeholders and sends a command to execute the statement. These values become the command's input parameters.

Once the statement has executed the prescribed number of times, the client application deallocates the statement.

Advantages of dynamic SQL

Dynamic SQL permits a client application to act interactively, passing different information at different times to the Open Server application, from the user. The Open Server application can then fill in the missing pieces in the SQL query with the data the user provides.

For more information on how client applications use dynamic SQL, see the *Embedded SQL/C Programmer's Manual*.

Handling dynamic SQL requests

When a client issues a dynamic command, Open Server raises a SRV_DYNAMIC event. If an Open Server application will be returning dynamic SQL results, it must include a SRV_DYNAMIC event handler to respond to dynamic SQL requests.

The *srv_dynamic* routine

From within its SRV_DYNAMIC event handler, an Open Server application uses the srv_dynamic routine, in conjunction with other Server-Library routines, to retrieve a client's dynamic SQL command and respond to it. For more information, see srv_dynamic on page 268. Each client command type—preparation, execution, deallocation—requires a particular response from the Open Server application.

Detecting a command type

The first task within the SRV_DYNAMIC event handler is to retrieve the type of dynamic command the client issued and, in some cases, the dynamic statement's ID and text. It must store the information and refer back to it later when it responds to client requests.

Responding to client dynamic SQL commands

Table 2-19 summarizes the valid exchange of dynamic SQL requests and responses between the client and the Open Server application. The forward arrow (→) indicates that cmd is set to CS_GET—the Open Server application retrieves information from the client. The backward arrow (←) indicates that cmd is set to CS_SET—the Open Server application sends information to the client.

Table 2-19: Valid dynamic SQL requests and responses

Client action	Open Server application response
Issues a prepare request (Operation type is CS_PREPARE)	→ Retrieves the operation type. (srv_dynamic) → Retrieves the statement ID length. (srv_dynamic) → Retrieves the statement ID. (srv_dynamic) → Retrieves the statement length. (srv_dynamic) → Retrieves the statement. (srv_dynamic) ← Acknowledges the client command. (srv_dynamic) ← Sends the statement ID length. (srv_dynamic) ← Sends the statement ID. (srv_dynamic) ← Sends a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)
Requests a description of the statement's input parameters (Operation type is CS_DESCRIBE_INPUT)	→ Retrieves the operation type. (srv_dynamic) → Retrieves the statement ID length. (srv_dynamic) → Retrieves the statement ID. (srv_dynamic) ← Acknowledges the client command. (srv_dynamic) ← Sends the statement ID length. (srv_dynamic) ← Sends the statement ID. (srv_dynamic) ← Sends the format of the input parameters. (srv_descfmt and srv_xferdata with type argument set to SRV_DYNDATA. There is no need to call srv_bind, as the application sends formats but no actual data. The status field of the CS_DATAFMT structure must be OR'd with CS_DESCIN prior to calling srv_descfmt) ← Send a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)

Client action	Open Server application response
Requests a description of the statement's output parameters (Operation type is CS_DESCRIBE_OUTPUT)	→ Retrieves the operation type. (srv_dynamic) → Retrieves the statement ID length. (srv_dynamic) → Retrieves the statement ID. (srv_dynamic) ← Acknowledges the client command. (srv_dynamic) ← Sends the statement ID length. (srv_dynamic) ← Sends the statement ID. (srv_dynamic) ← Sends the result row formats. (srv_descfmt and srv_xferdata with type argument set to SRV_DYNDATA. There is no need to call srv_bind, as the application sends formats but no actual data. The status field of the CS_DATAFMT structure must be OR'd with CS_DESCOUT prior to calling srv_descfmt) ← Sends a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)

Requests a description of the statement's input parameters (Operation type is CS_DESCRIBE_INPUT)	→ Retrieves the operation type. (srv_dynamic) → Retrieves the statement ID length. (srv_dynamic) → Retrieves the statement ID. (srv_dynamic) ← Acknowledges the client command. (srv_dynamic) ← Sends the statement ID length. (srv_dynamic) ← Sends the statement ID. (srv_dynamic) ← Sends the format of the input parameters. (srv_descfmt and srv_xferdata with type argument set to SRV_DYNDATA. There is no need to call srv_bind, as the application sends formats but no actual data. The status field of the CS_DATAFMT structure must be OR'd with CS_DESCIN prior to calling srv_descfmt) ← Send a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)

Client action	Open Server application response
Client issues an execute request (Operation type is CS_EXECUTE)	→ Retrieves the operation type. (srv_dynamic) → Retrieves the statement ID length. (srv_dynamic) → Retrieves the statement ID. (srv_dynamic) → Retrieves the number of dynamic parameters. (srv_numparams) → Retrieves the input parameter values. (srv_descfmt, srv_bind, srv_xferdata with type argument set to SRV_DYNDATA) ← Acknowledges the client command. (srv_dynamic) ← Sends the statement ID length. (srv_dynamic) ← Sends the statement ID. (srv_dynamic) ← Sends result rows. (srv_descfmt, srv_bind, srv_xferdata, with type argument set to SRV_ROWDATA) ← Sends a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)
Issues an execute-immediate request (Operation type is CS_EXEC_IMMEDIATE)	→ Retrieves the operation type. (srv_dynamic) → Retrieves the statement ID length—t should be 0. (srv_dynamic) → Retrieves the statement length. (srv_dynamic) → Retrieves the statement. (srv_dynamic) ← Acknowledges the client command. (srv_dynamic) ← Sends a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)

Client action	Open Server application response
Issues a deallocation request (Operation type is CS_DEALLOC)	→ Retrieves the operation type. (srv_dynamic) → Retrieves the statement ID length. (srv_dynamic) → Retrieves the statement ID. (srv_dynamic) ← Acknowledges the client command. (srv_dynamic) ← Sends the statement ID length. (srv_dynamic) ← Sends the statement ID. (srv_dynamic) ← Sends a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)

Example

The sample *ctos.c* includes code illustrating dynamic SQL command processing.

Errors

By default, Open Server responds to errors by writing error messages to the log file. Developers can tailor an application's response by installing an error handling routine.

Typically, an error handler detects the type and severity of an error, and takes a specific action based on these values. For example, an application may send particular errors to a client through the srv_sendinfo routine, while writing others to a log file.

To install an error handler use the srv_props routine with the property argument set to SRV_S_ERRHANDLE. An application should install its error handler just after calling srv_version to catch all types of errors. When an error occurs, Open Server invokes the error-handling routine that was most recently installed.

For more information, see srv_props on page 334.

Types of errors

An Open Server application, a client application, and Open Server itself can each provoke Open Server errors. Here is a description of errors that occur in each type of category:

- *Open Server application errors* – error occurs because of a mistake in application code. For example, if an application attempted to send a row of data to a client without first describing the format of the data, Open Server raises an error.
- *Client command errors* – error occurs when a client has sent an incomplete or nonsensical request. Requests can be incomplete or meaningless because of faulty client code or because of a network problem. An Open Server application should handle these errors in the event handler for the client request, usually by sending the appropriate error messages to the client with srv_sendinfo. See “Client command errors” on page 38 for more details. The application can also set the status parameter in srv_senddone to SRV_DONE_ERROR to indicate that a client request provoked an error.
- *Open Server resource errors* – error originates with the Open Server itself. It typically occurs because of a lack of some resource, like memory or user connections.

Severity of errors

Each Open Server error is associated with a number, a severity level, and a message.

When an error occurs, the currently installed error handler function is called with the error number, error severity level, and the text of the message. If no error handler has been installed, Open Server’s log file records this information. An application can also explicitly write to the log file with a call to srv_log.

An Open Server application can set the log file’s maximum size using srv_props, with the property argument set to SRV_S_LOGSIZE.

Error numbers and severity levels are defined in the header file *oserror.h*. An application that uses the defined error values must include *oserror.h*.

Table 2-20 summarizes Open Server error severity levels:

Table 2-20: Severity of errors

Severity	Meaning	Applicable error type
SRV_INFO	An informational error. Most errors are of this severity. This level of severity indicates that an error has occurred but that it is not fatal. It is most often generated by an incorrectly invoking a Server-Library function. For example, calling srv_xferdata to send a row before describing all the columns with srv_descfmt generates a SRV_INFO error.	Open Server application error Client command error
SRV_FATAL_PROCESS	A fatal thread error. The thread that received the error has an internal error from which it cannot recover. For example, the application may have returned from an event without calling srv_senddone. An error of this severity causes Open Server to queue a SRV_DISCONNECT event for the thread, if the thread is a client thread, a SUB-PROC, or a site-handler. Open Server then kills the thread.	Open Server resource error
SRV_FATAL_SERVER	A fatal server error. Open Server has detected an internal error from which it cannot recover. This causes Open Server to queue a SRV_STOP event for the Open Server application, which causes srv_run to return CS_FAIL.	Open Server resource error

Operating system errors

When an operating system error occurs, the operating system error number is different than SRV_ENO_OS_ERR, and the operating system error text contains the description of the operating system error. For example, if srv_init cannot open the interfaces file, it may be due to an operating system permissions error.

Error numbers and corresponding message text

See the header file *oserror.h* for a complete list of error tokens. See the file *oslib.loc* for the corresponding error text.

Example

All sample programs include an Open Server error handler.

Events

This section describes the following:

- Event overview
- What is an event handler?
- Standard events
- Programmer-defined events
- Example

Event overview

An Open Server application responds to requests from clients. Some of these requests trigger a Server-Library *event*.

Not all events are provoked by client activity. The application itself queues programmer-defined events and SRV_DISCONNECT, SRV_URGDISCONNECT, and SRV_STOP events by calling the srv_event or srv_event_deferred routine. For more information on using the srv_event routine to raise events, see its reference page. Open Server can also trigger a SRV_STOP event in response to a fatal server error. Open Server raises a SRV_START event automatically, as part of the server's start-up process.

An event occurs in a specific context; it corresponds to a particular category of activity. For example, a connection attempt from a client or remote server triggers a SRV_CONNECT event, while a client's bulk copy request causes Open Server to raise a SRV_BULK event.

There are two kinds of events in Open Server: standard and programmer-defined. Standard events are defined internally in Open Server. Programmer-defined events are, as the name suggests, defined within the application. For more details on both kinds of events, see"Standard events" on page 93, and "Programmer-defined events" on page 97.

What is an event handler?

An event handler is a piece of code that executes when an event is initiated. When an event is triggered, Open Server places the event and the active thread on the run queue. The thread then executes a routine that processes the event. This routine is called an *event handler*.

Default and custom handlers

Open Server has a default event handler routine for each standard event, and one for programmer-defined events. The default handlers are placeholders for the custom event handlers that the application programmer installs with the srv_handle routine. For an application that does not use the default handlers, you must define and install each custom event-handling routine. For more information on installing handlers, see srv_handle on page 295.

Event handlers can be installed dynamically. The new event handler is called the next time the event is raised. Event handlers should always return CS_SUCCEED when successful, and CS_FAIL when they fail. Currently, the SRV_START handler is the only event handler whose return code Open Server checks. Returning CS_FAIL from a SRV_START handler causes srv_run to return CS_FAIL to the application without starting Open Server.

Coding custom handlers

It is the application programmer's responsibility to decide how to respond to an event and to code the event handler accordingly. Event handlers typically include a standard set of calls to process the event data. Any additional code is application-specific. For example, a SRV_MSG event handler should include code to retrieve the text of the message as well as any parameters. But an application can include additional code in the SRV_MSG event handler to send mail to users if a particular message is retrieved.

Standard events

Table 2-21 describes each standard Open Server event and the argument the corresponding custom event handler should take. It also describes what function the corresponding default event handler performs.

Table 2-21: Description of events

Event	Description	Argument to handler	Default event handler
SRV_ATTENTION	An attention has been received. This event usually occurs when a client calls ct_cancel to stop results processing prematurely. SRV_ATTENTION is an immediate event; Open Server services it as soon as it occurs rather than adding it to the client's event queue. A SRV_ATTENTION event executes at interrupt level.	SRV_PROC*	The default handler takes no additional action.
SRV_BULK	A client has issued a bulk copy request.	SRV_PROC*	The default handler sends the message "No bulk handler installed" to the client. Open Server discards the bulk data and returns DONE ERROR to the client.
SRV_CONNECT	A Client-Library client has called ct_connect.	SRV_PROC*	The default handler accepts the connection.
SRV_CURSOR	A client has sent a cursor request.	SRV_PROC*	The default handler sends the message "No SRV_CURSOR handler installed" to the client. Open Server returns DONE ERROR to the client.
SRV_DISCONNECT	A request to disconnect a client connection has been made. This event is triggered by a client disconnecting from a server, an Open Server fatal thread error, a SRV_STOP event, or a call to srv_event made from within the application explicitly to disconnect a client. Client-Library programs call ct_close or ct_exit to log out of the Open Server application. Remote Adaptive Server connections terminate when the remote procedure call has completed.	SRV_PROC*	The default handler takes no action.

Event	Description	Argument to handler	Default event handler
SRV_DYNAMIC	A client has sent a dynamic SQL request.	SRV_PROC*	The default handler sends the message "No SRV_DYNAMIC handler installed" to the client. Open Server returns DONE ERROR to the client.
SRV_FULLPASSTHRU	A network read for the connection has completed. (The SRV_T_FULLPASSTHRU property for the thread must have been set to CS_TRUE for this event to occur.	SRV_PROC*	There is no default event handler for this event.
SRV_LANGUAGE	A client has sent a language request, such as a SQL statement. A Client-Library client submits a language request using ct_command and ct_send. isql and other interactive query tools can also send language requests to the Open Server application.	SRV_PROC*	The default handler sends the message "No language handler installed" to the client, along with the first few characters of the language request. Open Server returns DONE ERROR to the client.
SRV_MIGRATE_STATE	This event is triggered whenever the migration state has transitioned to SRV_MIG_READY or SRV_MIG_FAILED, the transition being a result of a migration message from a client. See "Connection migration" on page 40 for more details.	SRV_PROC*	The default handler takes no action if the state is SRV_MIG_READY, and thus allows the client to continue with the migration. It logs an error if the state changes to SRV_MIG_FAILED.
SRV_MIGRATE_RESUME	When a client migrates to a new server while waiting for results, the new server invokes the SRV_MIGRATE_RESUME event after the client connection has successfully migrated. If the migration request failed or is cancelled, the event is invoked from the original server. See "Connection migration" on page 40 for more details.	SRV_PROC*	The default handler only sends a final done (SRV_DONE_FINAL) to the client to end the results.

Event	Description	Argument to handler	Default event handler
SRV_MSG	A client has sent a message.	SRV_PROC*	The default handler sends the message “No SRV_MSG handler installed” to the client. Open Server returns DONE ERROR to the client.
SRV_OPTION	A client has sent an option command.	SRV_PROC*	The default handler sends the message “No SRV_OPTION handler installed” to the client. Open Server returns DONE ERROR to the client.
SRV_RPC	A client or a remote Adaptive Server has issued a remote procedure call (RPC).	SRV_PROC*	The default handler sends the message “RPC < *rpcname* > received. No remote procedure call handler installed” to the client. Open Server returns a DONE ERROR to the client.
SRV_START	A call to srv_run triggers a SRV_START event. The Open Server application is up and running. The SRV_START event handler is a good place to initialize server resources and to spawn service threads.	SRV_SERVER*	The default handler takes no action.
SRV_STOP	A request to stop the Open Server application has been made, triggered by a call to srv_event or by an Open Server fatal server error. The Open Server application is stopped. srv_run returns CS_SUCCEED if the application requested a SRV_STOP event or CS_FAIL if a fatal server error provoked the SRV_STOP event. A custom handler for this event can perform any necessary cleanup before the Open Server application shuts down.	SRV_SERVER*	The default handler takes no action.

Event	Description	Argument to handler	Default event handler
SRV_URGDISCONNECT	This event is only triggered by an Open Server application calling srv_event. In response to this event, Open Server calls a threads SRV_DISCONNECT event handler. Open Server places the event at the top of the threads event queue, so that it is processed as the next event. An application should raise this event if it wants to terminate a thread immediately, bypassing other events in the queue. When a SRV_URGDISCONNECT event is raised, the I/O channel associated with the thread is marked dead.	SRV_PROC*	The default handler takes no action.

Programmer-defined events

An application defines programmer-defined events with srv_define_event and installs them with srv_handle. The application must call srv_event or srv_event_deferred to place the new event on the client's event queue.

The default programmer-defined event handler sends a message to the client stating that there is no handler installed. The message includes the event number and name.

Programmer-defined events can be used to provide services to other threads in the Open Server application. For example, such an event could allow threads to log transactions in a disk file. To set up this service, define the event with srv_define_event, install a handler routine that writes to the disk file, and create a service thread to which the events are queued. The service thread provides the transaction-logging code.

Example

The sample *lang.c* illustrates a simple SRV_LANGUAGE event handler.

Gateway applications

An Open Server application that acts as both a client and a server is called a *gateway* application. Gateway applications often act as intermediaries for clients and servers that cannot communicate directly.

For example, an Open Client application cannot communicate directly with an Oracle database engine, but the client application *can* communicate with an Open Server application that serves as a gateway to the Oracle database. In this case, the gateway acts as a server to the Open Client application and as a client to the Oracle database engine.

Another case is when a client cannot directly access a remote Adaptive Server because the two are running on dissimilar networks. The gateway server bridges this gap, retrieving the client data and repackaging it to send to the remote Adaptive Server. Sybase's mirror-image client and server routines simplify this process. The server and client components can even share the same data description structure; the gateway fills in a structure with information from the remote client using Server-Library calls and then extracts that same information from the structure to send along to the remote server using Client-Library or DB-Library calls.

Gateways that act as clients to a Adaptive Server or to an Open Server application use Client-Library or DB-Library routines to fill the client role that they play.

Gateways that act as servers to Open Client applications use Server-Library routines to fill the server role that they play.

Warning! Client-Library cannot be run in full asynchronous mode in an Open Server application.

The sample program *ctos.c* is an example of a "virtual Adaptive Server" gateway. The gateway demonstrates how to pass data from a remote Adaptive Server to a Sybase client.

Warning! In gateway applications, the client routines execute in the context of an Open Server process, or *thread*. If this process (or the entire Open Server application) is terminated, any client routines that are executing will yield undefined results.

Passthrough mode

In the special case of an Open Server application that connects Sybase client applications with an Adaptive Server, Client-Library and DB-Library provide a set of application protocol passthrough routines that allow the Open Server to pass Tabular Data Stream (TDS) packets between the client and server without interpreting the contents. This process works more efficiently than unpacking the TDS information as it arrives and repacking it before sending it on. The sample, *fullpass.c*, provides an example of this type of gateway. For more information, see "Passthrough mode" on page 129.

Note Pre-10.0 versions of DB-Library must not be linked into an application with Open Server version 10.0 and later, although they can be used in application programs that serve as clients to Open Server 10.0 and later.

International support

Open Server provides support for international applications by:

- Allowing an Open Server application to localize

  An Open Server application that is localized typically:

  - Generates error messages in a local language and character set
  - Uses local datetime formats
  - Uses a specific character set and **collating sequence** (also called "sort order") when converting or comparing strings

- Enabling an Open Server application to support localized clients

  A localized client uses the language, datetime formats, and character set appropriate to its locale. These may differ from the Open Server application's language, datetime formats, and character set. To support localized clients, an Open Server application must not only translate incoming data into its own language and character set but must also translate outgoing messages and data into the client's language and character set.

This topic page contains information on:

- Localizing an Open Server application

- Supporting localized clients
- Client requests related to localization
- Localization properties
- The localization sample programs

Open Client and Open Server localization is discussed thoroughly in the Open Client and Open Server *International Developer's Guide*. You must read this book to understand Server-Library's localization mechanism and how environment variables affect localization.

Platform-specific localization information can be found in the Open Client and Open Server *Configuration Guide*.

Localizing an Open Server application

An Open Server application's localization determines:

- The language and character set in which error messages are generated.

  Note The SRV_S_USESRVLANG and SRV_T_USESRVLANG properties can be used to override a server's language when generating error messages.

- The character set and collating sequence used for all data operations

.

An Open Server application can use initial localization values, custom localization values, or both.

A typical internationalized Open Server application uses the initial localization values determined by the LC_ALL and LANG environment variables, or by the "default" entry in the locales file, to localize.

Initial localization values are determined at runtime, when the Open Server application calls the CS-Library routine cs_ctx_alloc to allocate a CS_CONTEXT structure. When an application makes this call, CS-Library loads initial localization information into the new context structure.

If the initial localization values do not meet an application's needs, the application can use a CS_LOCALE structure to set custom localization values in its context structure. See "Using a CS_LOCALE structure to set custom localization values" on page 101 for more information.

Supporting localized clients

For some Open Server applications, initial localization values for localized clients are sufficient. These Open Server applications do not need to take any additional steps to support localized clients.

Other Open Server applications, however, need to provide additional support for localized clients. In particular, an Open Server application needs to take additional steps to support localized clients:

- If it will be passing CS-Library error messages back to clients

  In this case, the Open Server application needs to ensure that CS-Library generates messages in the client's language and the Open Server application's character set.

  For information on how to do this, see "Localizing CS-Library messages for clients" on page 102.

- If it is acting as a gateway

  In this case, the Open Server application needs to ensure that a connection to a remote server uses the client's language and the Open Server's character set.

  For information on how to do this, see "Creating localized connections for gateway applications" on page 103.

- If a client application asks to change its language or character set

  In this case, the Open Server application needs to change the language or character set for the client thread.

  For information on how to do this, see "Requests to change language and character set" on page 104.

Using a CS_LOCALE structure to set custom localization values

When a client connects to an Open Server application, Open Server creates a CS_LOCALE structure reflecting the client's language and character set. For example, when a french/cp850 client logs in to a us_english/iso_1/binary Open Server application, the Open Server application creates a french/cp850 CS_LOCALE structure for that connection.

The information in this structure is available to Open Server programmers, who can call cs_locale to copy the information into a newly-allocated CS_LOCALE structure.

You can install custom localization information in the application-wide context structure before calling srv_version. To do this, an application:

1 Calls cs_loc_alloc to allocate a CS_LOCALE structure.

2 Calls cs_locale with type set to CS_LC_ALL to load the CS_LOCALE with custom localization values. A type of CS_LC_ALL ensures that the CS_LOCALE is loaded with localization values that are internally consistent.

3 Calls cs_config with property set to CS_LOC_PROP to copy the custom localization values into the application's context structure.

4 Calls cs_loc_drop to deallocate the CS_LOCALE.

Localizing CS-Library messages for clients

If an Open Server application calls a CS-Library routine with its own context structure as a parameter, any error messages that CS-Library generates as the result of the call will be in the Open Server application's language and character set.

For example, if the context parameter for a cs_convert call indicates us_english/iso_1, CS-Library will generate a us_english/iso_1 message if the cs_convert call fails.

Note If a CS-Library routine takes a CS_LOCALE structure as a parameter, the localization values in this structure will override the localization values in the context parameter.

Obtaining CS-Library messages in the Open Server application's language and character set is acceptable only if the Open Server application logs the CS-Library messages or otherwise keeps them to itself.

However, if an Open Server application will be passing CS-Library error messages back to a client, it needs to ensure that CS-Library generates messages in the client's language and the Open Server application's character set.

The messages need to be in the client's language for the client to understand them.

The messages need to be in the Open Server application's character set for two reasons:

- Open Server applications commonly record all messages in the log file. It is important that all logged messages use the same character set.
- Open Server automatically performs character set translation on outgoing data, including messages. Generating messages in Open Server's character set ensures that they will be correctly translated to the client's character set.

An application can ensure that messages are generated in the correct language and character set by setting up a properly localized CS_CONTEXT structure for each client thread and then using these CS_CONTEXT structures when calling CS-Library routines on behalf of clients.

For information on how to localize a CS_CONTEXT structure, see "Localizing a CS_CONTEXT structure" on page 104.

Creating localized connections for gateway applications

If an Open Server application is acting as a gateway, it needs to ensure that a connection to a remote server uses the client's language and the Open Server's character set.

Note The Open Server's character set does not need to be the same as the remote server's character set, but it must be one that the remote server is capable of converting to its own.

Adaptive Server can convert between any two Western European character sets and can convert between any two Japanese character sets, but it cannot convert a Western European character set to a Japanese one (and vice-versa).

For example, Adaptive Server can convert between ISO 8859-1 and CP850, because both of these character sets are in the Western European language group; however, Adaptive Server cannot convert between ISO 8859-1, which is Western European, and CP 1250, which is Eastern European.

Open Server can convert between any two supported character sets, whether or not they are in the same language group. However, when converting between character sets in different language groups, non-Roman characters may be lost.

The simplest way for an application to do this is to set up a properly localized CS_CONTEXT structure for each client connection and then allocate remote connections for the client within the localized context.

See “Localizing a CS_CONTEXT structure” below for information on how to localize a CS_CONTEXT structure.

Localizing a CS_CONTEXT structure

To properly localize a CS_CONTEXT structure for a client thread, an Open Server application must:

1 Call cs_ctx_alloc to allocate a CS_CONTEXT for the client thread.

2 Call cs_loc_alloc to allocate a new CS_LOCALE structure.

3 Call srv_thread_props to copy the client thread’s existing CS_LOCALE structure. This sets the new CS_LOCALE up with the client’s language, and character set.

4 Call cs_locale with type as CS_SYB_CHARSET to replace the client’s character set with the Open Server’s character set.

5 Call cs_config with *property* as CS_LOC_PROP to copy the localization information from the CS_LOCALE into the CS_CONTEXT.

6 Call cs_loc_drop to deallocate the CS_LOCALE, if desired. An application can also reuse a CS_LOCALE structure, calling cs_locale, if necessary, to change its localization values.

Responding to client requests

Clients can:

- Request to change their language and character set
- Request localization information

Requests to change language and character set

When a client connects to an Open Server, it specifies a language and character set in the login record. Open Server uses this information to set up a CS_LOCALE and **character set conversion** routines for the client thread.

Open Server handles this automatically; an Open Server application does not need to take any steps to handle localized clients at login time.

However, after logging in, clients can change their language and character set. If a client sends a request to change its language or character set, the Open Server application must make the requested changes in the client thread's CS_LOCALE structure.

A client can request a change of language or character set in two ways:

- Using a language-based option command sent with ct_command. This type of command triggers a SRV_LANGUAGE event, so the Open Server application will process the request inside a SRV_LANGUAGE event handler.
- Using an option command sent with ct_options. This type of command triggers a SRV_OPTION event, so the Open Server application will process the request inside a SRV_OPTION event handler.

In both cases, the Open Server application responds by:

1 Setting up a CS_LOCALE structure with the new language or character set

2 Calling the srv_thread_props routine with property set to SRV_T_LOCALE to change the language or character set for the thread connection

Table 2-22 describes how to change the language or character set for a client thread:

Table 2-22: Changing the language or character set

Step	Application step	Purpose	Details
1	Call cs_loc_alloc.	Allocate a CS_LOCALE structure.	This call copies the Open Server application context's current localization information into the new CS_LOCALE structure.
2	Call srv_thread_props(GET) with *property* as SRV_T_LOCALE.	Copy the client thread's existing localization values into the new CS_LOCALE structure.	
3	Call cs_locale.	Overwrite the CS_LOCALE structure with the requested language or character set.	For more information about this process, see "Localizing a CS_CONTEXT structure" on page 104.

Step	Application step	Purpose	Details
4	Call srv_thread_props(SET) with property as SRV_T_LOCALE.	Set up the client thread with the new language or character set.	
5	Optionally, call cs_loc_drop.	Deallocate the CS_LOCALE structure.	An application can reuse the CS_LOCALE structure before deallocating it. If necessary, the application can call cs_locale to change the localization values in the structure before reusing it.

Requests for localization information

After logging in, a client can ask for:

- The name of the server's character set
- The name of the server's sort order
- The character-set definition for the client's character set
- The sort order definition for the client's sort order

Clients make these requests through the sp_serverinfo system registered procedure, using RPC commands.

In response, Open Server automatically returns the requested information by means of the sp_serverinfo system registered procedure. The Open Server application does not need to take any action at this point, and, in fact, is not aware that the request ever occurred.

For more information on these routines, see "Registered procedures" on page 162.

Localization properties

Two properties are related to localization:

- SRV_S_USESRVLANG
- SRV_T_USESRVLANG

These properties determine whether Open Server generates error messages in the Open Server application's language or a client's language.

SRV_S_USESRVLANG is a server-wide property, set through srv_props. Its value serves as the default value for SRV_T_USESRVLANG.

SRV_T_USESRVLANG is a thread property, set through srv_thread_props. When a new thread structure is allocated, SRV_T_USESRVLANG picks up a default value from SRV_S_USESRVLANG.

If SRV_T_USESRVLANG is CS_TRUE, Open Server generates error messages in the server's language.

If SRV_T_USESRVLANG is CS_FALSE, Open Server generates error messages in the client's language.

For more information on setting properties, see "Properties" on page 139.

Localization examples

The example *ctos.c* demonstrates one method of customizing a CS_LOCALE structure. The example *intlchar.c* handles character set and national language configuration and queries.

Language calls

Open Server provides functionality for processing language events in a flexible manner. A SRV_LANGUAGE event is triggered when a client application sends information through ct_command with the type argument set to CS_LANG_CMD. Whereas an RPC stream is composed of discrete elements—a name and parameters—language information arrives in a stream of undifferentiated characters. A SRV_LANGUAGE event handler must include code to parse the stream into its meaningful components. A SQL query is an example of a language stream.

This functionality is useful for applications that want to accept natural language input. For example, consider a clothing store application that lets users query a SQL database in English. A sales clerk could type in the question "How many shirts in blue?" The front-end client application could send this natural language query to an Open Server gateway application through a call to ct_command. The SRV_LANGUAGE handler parses this text, constructs this Transact-SQL query and sends it to a remote database:

```
select quantity from inventory_tab where color = "blue" and type = "shirt"
```

A SRV_LANGUAGE event handler must process language data in steps:

1 Call srv_langlen to retrieve the length of the language request buffer.

2 Allocate a local application buffer as large as the length returned by srv_langlen, plus 1 for the null-termination byte.

3 Call srv_langcpy to copy all or part of the request data into the local buffer.

4 Process the contents of the local buffer.

Login redirection and extended HA failover support

Login redirection and extended HA failover support allows a cluster of servers to perform load-balancing for all incoming client connections.

Three routines support this functionality: srv_send_ctlinfo, srv_getserverbyname, and srv_freeserveraddrs.

The srv_send_ctlinfo routine supports both login redirection and extended HA failover and srv_getserverbyname, and srv_freeserveraddrs allows an Open Server application to translate a given server name to its connection information. These routines are described in “srv_send_ctlinfo” on page 385, “srv_getserverbyname” on page 294, and “srv_freeserveraddrs” on page 281.

The following properties support the routines:

- SRV_S_HASERVER, a read-only server property that returns the HAFAILOVER value from the interfaces file, which corresponds to the server name as set by srv_init.
- SRV_T_REDIRECT, a read-only thread property that returns the setting of the TDS_HA_LOG_REDIRECT bit in the login record.
- SRV_T_HA, a thread property that returns the setting of HA-related information from the login record as a CS_INT bitmask. Information provided includes session (SRV_HA_LOGIN), failover (SRV_HA_LOGIN_FAILOVER), and resume (SRV_HA_LOGIN_RESUME) bits.
- CS_SESSIONID, a type definition that holds the Session ID.
- SRV_T_SESSIONID, returns the Session ID that the client sends to Open Server in the login record.

You can also use SRV_T_SESSIONID to send a Session ID to the client in the SRV_CONNECT handler. For more information see “Instructing clients to migrate to a different server” on page 48.

- SRV_NEG_SESSIONID, a type of negotiated login information in the parlance of srv_negotiate that supports the sending of client Session ID information.

Messages

There are three types of messages in Open Server:

- Data stream messages – clients and servers can use data stream messages to exchange information. See “Data stream messages” on page 80.
- Thread messages – threads can use thread messages to exchange information. See “Multithread programming” on page 109.
- Error messages – Open Server reports error conditions by means of error messages. See “Errors” on page 89.

Multithread programming

Open Server employs a multithreaded architecture. A multithreaded server application acts as a collection of *threads*, each executing routines to accomplish its specific task.

What is a thread?

A thread can be thought of as a particular path of execution through the Open Server application code. Each client uses a thread to manage its connection and call the event handlers and procedures that fulfill its requests. The Open Server runtime system has several threads that manage server activities such as delivering messages, handling server-to-server communications, and scheduling tasks. An application can spawn server threads for other application-specific activities.

As a multithreaded system, an Open Server application must schedule the variety of activities the threads perform, negotiate the threads' access to shared resources, and provide a means by which the threads communicate with each other. For more information, see "Scheduling" on page 113, and "Tools and techniques" on page 115.

Thread types

Open Server employs four kinds of threads: *preemptive*, *event-driven*, *service*, and *site-handler*.

Preemptive threads

Open Server versions 12.5 and higher include preemptive threading on all platforms. There are several issues to be aware of before building applications with these threaded libraries.

Thread-safe functions

To ensure that your application is reentrant, make sure that:

- It uses the reentrant versions of C library functions, where provided
- It uses non-reentrant C (or other) library functions safely
- It protects global variables and shared structures with mutex (or other) locks
- None of its functions return a pointer to a static buffer
- It compiles with the correct processor flags and linker directives

---

Note A C library function that is reentrant on one UNIX system is not necessarily reentrant on other UNIX systems. Consult a porting guide for your platform to determine if the C function is reentrant.

Thread-safe code and preemptive mode

More than one Open Server thread can be running at the same time, and one can be preempted in favor of another. This has the benefit of increased concurrency, especially in SMP systems. However, it does require code to be thread-safe. This applies to Open Server code, the user's event handlers and callback functions.

SRV_S_PREEMPT behavior

When SRV_S_PREEMPT is set to CS_TRUE, multiple Open Server threads execute concurrently and are preempted in favor of each other by the operating system. These threads become unbound.

If SRV_S_PREEMPT is set to CS_FALSE, one Open Server thread cannot be preempted by another Open Server thread, and two Open Server threads cannot run at the same time.

Also, whether SRV_S_PREEMPT is set to CS_TRUE or CS_FALSE, when used in conjunction with threaded libraries, some functions of SRV_S_CURTHREAD become disabled. This is because threaded libraries use signals handled by a signal-handled thread, regardless of the SRV_S_PREEMPT setting.

A single mutex is enabled whenever an Open Server thread resumes executing. The mutex is released when an Open Server thread is ready with a specific task and after the SRV_C_SUSPEND callback is executed. There is only one server-wide mutex for this.

The callback functions SRV_C_RESUME and SRV_C_SUSPEND are never invoked when the operating system resumes such a thread. These functions are invoked only when a specific Open Server thread stops or resumes execution; for instance, when a language request arrives for a user Open Server thread, and before it goes to sleep after running the language event handler.

Implementation specifics

For most UNIX platforms, threads are based on POSIX threads and are unbound. On HP and Linux, threads are bound. On Windows, threads are Win32 threads.

See the vendor documentation for your platform for more information about using threads on that platform.

Event-driven threads

Threads that control client connections are event-driven. A request for action triggers a server event. See “Events” on page 92 for details on events.

When a client event occurs, Open Server places the event in the thread’s event queue. The next time the thread executes, it reads the next event request from the event queue. Open Server calls the event handler associated with this event. When the handler returns, the thread attempts to read the next event in the queue. If there is no event, the thread “sleeps.”

For example, when a client application attempts to log in to the server, Open Server creates a thread to handle the connection and puts the SRV_CONNECT event in the threads queue. When the thread runs, it executes the routine installed to handle the SRV_CONNECT event. The default handler simply accepts the connection. You could install a custom SRV_CONNECT handler that checks the **login name** and password, and, if both are valid, allows the user to log in.

Event-driven threads exist primarily to handle client requests, but they can also be used with programmer-defined events to execute service routines within the server.

Service threads

You can create Open Server threads that run independently of any client connection. Such threads are called service threads because the routines they execute usually perform services for event-driven client threads. Unlike a client thread, a service thread is not activated by events. Instead, you supply a routine for the thread to execute when you create it. The server puts it in the run queue immediately. A service thread disappears once the routine it was created to execute returns.

An application can use service threads to accomplish a variety of tasks in an Open Server application. In fact, the Open Server runtime system is composed of service threads running server management routines. Service threads cannot be used to perform client I/O—that is, to read in client commands and return results.

Open Server schedules event code to run when an event is triggered. By contrast, an application must explicitly schedule service thread code using the srv_wakeup, srv_sleep, srv_yield routines, and it must schedule message queues when not running in preemptive mode.

Site-handler threads

Open Server creates a site-handler thread when an Adaptive Server connects to an Open Server application.

Open Server creates a SUB-PROC when the Open Server application receives a server-to-server RPC. A SUB-PROC disappears when the server-to-server RPC completes. A site-handler thread disappears when the Adaptive Server closes its connection to the Open Server application.

An Open Server application only accesses a site-handler thread inside a SRV_CONNECT or SRV_DISCONNECT event handler. Site-handler threads are otherwise purely internal.

Scheduling

Open Server provides concurrency by periodically suspending the running thread and resuming another. This *context switch* can occur frequently and quickly so that, from the point of view of an Open Server client, threads run continuously.

The *scheduler* is the runtime system thread that performs context switches. A thread has an execution context that includes its stack and its machine register environment. The scheduler saves the execution context of the running thread, selects the thread to resume, restores its context, and runs it. Although the scheduler works invisibly, to write Open Server code you should understand:

- How the scheduler is called (the *scheduling method*)
- How the scheduler selects a thread to resume

Scheduling methods

The scheduling method determines when control is transferred from one running thread to another. An Open Server application uses one of two scheduling methods: *non-preemptive* or *preemptive*. Non-preemptive is the default method and the *only* method available on most platforms.

Non-preemptive scheduling

With non-preemptive scheduling, context switches are predictable. They can occur only in these situations:

- A thread calls a Server-Library or Client-Library routine that performs network I/O.

When a thread reads from or writes to a network connection, the runtime system suspends execution of the thread waiting for the read or write to complete. Network I/O is relatively slow, and the server can use the time more efficiently by allowing other threads to run while the I/O completes.

- A thread sleeps while waiting for execution to resume.

  For example, a thread should wait for another thread to finish updating a data object in shared memory before accessing the object. A thread sleeps when the application calls:

  - srv_sleep
  - One of the Server-Library routines where the thread sleeps while waiting for a requested resource, such as srv_getmsgq(SRV_M_WAIT) or srv_lockmutex

- A thread calls srv_yield to intentionally suspend itself and allow other threads to run. The thread remains executable and resumes operation later at the statement after the srv_yield call. If you write a time-consuming routine that does not sleep or perform network I/O, you should call srv_yield occasionally to prevent the routine from monopolizing the server.

Preemptive scheduling

With preemptive scheduling, a context switch can occur when any of the above events occurs, or when the system interrupts the running thread. Preemptive scheduling depends upon the thread management facilities of the operating system, so system-initiated context switches are not predictable. Operating systems often employ sophisticated algorithms to ensure optimal time distribution among threads.

You can choose preemptive scheduling using the srv_props routine with property set to SRV_S_PREEMPT. Preemptive scheduling is not available on every platform. Call srv_capability to determine whether it is available on your application's platform.

Selecting a thread to resume

Open Server maintains a set of run queues—lists of threads that are suspended but not sleeping. Each queue contains threads with the same execution priority. The scheduler restores the thread that has remained the longest on the highest priority queue. Threads normally run at the same priority level, so this selection method usually distributes execution time on a first-in, first-out basis.

You can adjust the priority of a thread so that the scheduler runs it before other threads in the run queue, or only when there are no other threads to run. For example, a thread that reads real-time data could have a higher priority so that it runs whenever there is data to process. Be careful when adjusting priorities. As long as a thread has a higher priority than any other and is able to run, the scheduler continues to run it. If the priority stays high and the thread never sleeps, threads with lower priorities will never run. See srv_setpri on page 411 for information on adjusting a thread's priority.

When Open Server establishes a new thread, the scheduler must perform some work before the thread can fully share CPU time with other threads. During this start-up period, the scheduler effectively performs a series of internal srv_yield calls to allow existing threads to run. As a result, established, executable threads may appear to "hog" CPU and delay start-up of the new thread. Once the thread is established and executable, it shares CPU time according to its priority.

Execution priority is only an issue in Open Server applications that run in non-preemptive mode.

Tools and techniques

Writing programs in a multithreaded environment requires constant attention to the interaction between threads. There are programming tools and methods that are especially useful in this environment. Open Server provides mutual exclusion semaphores (mutexes) to control access to shared resources, and message queues to allow threads to coordinate and communicate with one another.

Mutexes

A mutual exclusion semaphore, or **mutex**, is a logical object that Open Server allows one thread, at most, to lock. It is useful for protecting shared resources and for building more sophisticated tools.

To understand how a mutex can be used, consider this problem:

The standard input and output is the same for every thread in an Open Server application running on a UNIX platform. If threads regularly write to the standard output, the application code must avoid mixing the output of several threads on the standard output.

One way to prevent threads from mixing their output is to associate a mutex with the *stdout* device and require a thread to lock the mutex before writing to *stdout*. Since only one thread can lock the mutex at a time, only one thread can write on *stdout* at a time. Other threads have to wait until they are able to lock the mutex.

See the srv_createmutex, srv_lockmutex, srv_unlockmutex and srv_deletemutex reference pages for programming details.

Message queues

Message queues enable threads to communicate with each other. Message queues are often used to send data to spawned service threads that perform services for other threads. For example, you could create a message queue into which all threads put data destined for the log file. A spawned thread could read the messages from the queue and write them, in the order received, to the log file.

The message in a message queue is a 4-byte value, usually a pointer that addresses data somewhere in memory shared by the sending and receiving thread. The thread that puts a message into a queue and the threads that read the message must agree on the message format.

If the message references data elsewhere, you must make sure that the thread that reads the message finishes with the data before the thread that sent the message updates or releases the data area. To prevent the sending routine from overwriting or freeing the message before the message is received, the routine that writes messages, srv_putmsgq, has an option that causes the sending thread to sleep until the message is read from the queue.

See the srv_createmsgq, srv_putmsgq, srv_getmsgq, and srv_deletemsgq reference pages for programming details.

Protecting critical sections

To prevent Open Server from suspending a thread, you can temporarily raise the priority of the thread by calling srv_setpri. Server threads all start at the same priority level, which is represented by the SRV_C_DEFAULTPRI constant defined in *ospublic.h*. Thread priorities range from SRV_C_LOWPRIORITY to SRV_C_MAXPRIORITY, with SRV_C_DEFAULTPRI in the middle.

Open Server always resumes the executable thread that has the highest priority. If more than one executable thread has the same priority, Open Server resumes the one that became executable first. If you raise the priority of a thread above that of any other thread, Open Server continues to execute the thread until it is no longer executable or its priority is lowered, preventing other threads from executing.

While raising the priority of a thread is an effective way to guarantee that no other thread can interfere during a critical section, it can have a detrimental effect on concurrency. Raising the priority permits a single thread to take over the server. Even the threads that make up the Open Server runtime system are prevented from running if you raise the priority of a thread above SRV_C_DEFAULTPRI. To minimize the effects, delay raising the priority until absolutely necessary, and lower it again as soon as possible. Do not put unnecessary code inside the critical section.

Callback routines

The srv_callback routine allows you to install a callback handler for a thread. Open Server calls your routine whenever the state of the thread changes to the state you specify. For example, you can install a SRV_C_SUSPEND callback handler that executes whenever the thread is suspended.

Note The ability to install and execute callback handlers is platform-dependent. Use srv_capability to find out if a callback handler can be installed for a particular state transition on your current platform.

Table 2-23 summarizes the state transitions for which srv_callback can install callback handlers:

Table 2-23: State transitions

State transition	Meaning
SRV_C_EXIT	The thread has finished executing the routine it was spawned to execute, or it is associated with a disconnected client. The handler executes in the context of the exiting thread.
SRV_C_PROCEXEC	Open Server calls this callback when a registered procedure is about to execute. The handler executes in the context of the thread that requested the registered procedure. As a result, the SRV_C_PROCEXEC callback handler executes whenever a client attempts any registered procedure operation. You can install a callback handler that restricts clients' abilities to create, delete, or execute registered procedures.
SRV_C_RESUME	The thread is resuming. The handler executes in the context of the scheduler thread and uses the scheduler's stack.
SRV_C_SUSPEND	The thread is suspending. The handler executes in the context of the thread that is suspending and uses its stack.
SRV_C_TIMESLICE	A thread has executed for a period of time (time slice) determined by the SRV_TIMESLICE, SRV_VIRTCLKRATE, and SRV_VIRTTIMER configuration parameters. You can use this handler to signal a long-running thread to call srv_yield so that other threads can run.

Programming considerations

Although Open Server threads are threads of execution that have their own stack and register environments, they share the resources of the operating system process that is executing the Open Server runtime system.

Here are some multithread programming considerations:

- Shared resources, such as global data, file handles, and devices, must be protected.

  While you are updating a shared global data item, do not call a routine that could suspend the thread unless you have taken steps to prevent other threads from accessing the data. Otherwise, another thread could be working with inconsistent data.

Watch for program logic that behaves as though it has sole access to a resource. An example is a routine that performs part of a calculation using a value from a global variable, then suspends, allowing other threads to alter the global variable. This can cause serial consistency problems. The calculation may be incorrect before it is even complete.

- Avoid static variables in routines that more than one thread can execute.

  If a routine alters a static variable, and multiple threads can call the routine, you must ensure that multiple instances of the routine do not conflict. There is a greater probability of inconsistent data if the routine returns a pointer to a static variable, since the contents of the variable can be altered while a thread is suspended. It is safer to use automatic variables, because each thread has a stack of its own. The application should provide memory and copy the result there. When you must use static variables, protect them with the techniques discussed above.

- SRV_ATTENTION events can be executed at interrupt level. If SRV_ATTENTION handlers manipulate application structures that are also changed or tested in noninterrupt level code, such as other event handlers or service threads, the results of the change or test are unpredictable. Use attention-level wakeups and sleeps to coordinate between interrupt-level SRV_ATTENTION handlers and non interrupt-level code.

Example

The sample, *multthrd.c*, illustrates various aspects of multithreaded programming.

Negotiated behavior

An Open Server application negotiates with a client to determine the application's behavior in a number of areas. Some negotiation takes place when the client logs in. Other negotiations can occur on an ad hoc basis during the lifetime of the Open Server runtime system.

Login negotiations

Several issues are negotiated at login time. Some are negotiated transparently by Open Server and require no action on the part of the Open Server application. Others are handled explicitly with application calls. Login negotiations always take place inside a SRV_CONNECT event handler.

Transparent negotiation

Issues resolved that are transparent to the application include the following:

- The character set in which character data appears. When a client logs in, it provides, among other information, the name of the character set appropriate to its locale. If the server's character set differs from the client's, Open Server converts the data to the client's character set.
- The national language in which Open Server error messages appear.
- Byte ordering, which is platform-dependent.
- The TDS protocol level.
- Floating point representation, which is platform-dependent.

The server's default national language and character set are established during initialization of the server.

A client can renegotiate the character set and national language at a later time. See "Ad hoc negotiations" on page 122 for more information.

Explicit negotiation

The application itself negotiates with clients to resolve these issues:

- The kinds of requests the client can make and the kinds of responses the Open Server application can return, if the application declines the defaults.
- The security level at which the client and server communicate.

A client sends capabilities information after sending a login record. A client and the Open Server application must agree upon the set of possible requests and responses that can be sent on their particular connection. These capabilities must be established before any further requests or responses are sent. See "Capabilities" on page 24 for details on capabilities.

Negotiating a secure connection

An Open Server application may want to establish a *secure connection* with a client. A secure connection is one which is established after a rigorous authentication of the client's identity and verification of its password.

Note Applications can use external security systems offered by security service providers, rather than including their own security code. "Security services" on page 170 explains how to configure an Open Server application to make use of third-party security service providers.

An application may perform this security check using one, some, or all of the following methods:

- Send the client a *challenge*, which challenges the client to respond with the matching response.
- Send the client an *encryption key*, to which the client should respond with an encrypted password, which the application may then decrypt and verify.
- Send the client a request for *security labels*, which the client sends to establish the level of security for the connection.
- Initiate an application-defined login handshake.
- Initiate a transparent security handshake. This requires a security entry in the *libtcl.cfg* file, and that drivers for the required security services are installed. See "Changes to the interfaces file" on page 183, and "Security services" on page 170, for more information.
- Exchange the security session negotiation data between the remote server and the gateway client using a security session callback. See "Full passthrough gateway with direct security session" on page 191, and the Open Client *Client-Library/C Reference Manual*, for more information on security session callbacks.

An application negotiates a secure login using the srv_negotiate routine inside the SRV_CONNECT event handler.

Ad hoc negotiations

An application may negotiate or renegotiate several issues with a client at any point during the time the server is up and running. Ad hoc negotiations take place inside a SRV_LANGUAGE event handler or a SRV_OPTION event handler. A client may:

- Renegotiate the character set and national language through either a Transact-SQL language command or an option command.
- Determine aspects of query processing behavior through a Transact-SQL language command or an option command. Clients can request that options be set or cleared, as well as requesting the current status of a particular option.

For a discussion of the SRV_OPTION event and a list of options, see “Options” on page 122.

“International support” on page 99 covers negotiation of national language and character set in detail.

For more information on identifying and authenticating users in a secure database system, see the Adaptive Server Enterprise *Reference Manual* and “Security services” on page 170.

Example

The sample *ctos.c* includes code illustrating a negotiated login.

Options

Adaptive Server permits clients to determine how to handle query processing. It provides a variety of configurable options that govern aspects of query processing behavior. For more information on Adaptive Server query-processing options, see the set command in the Adaptive Server Enterprise *Reference Manual*.

An Open Server application can respond to client requests about query processing options.

A client application can set, clear, and request the current value of Adaptive Server query-processing options in one of two ways:

- Through a Transact-SQL language command
- By issuing an option command

If an application expects a client to issue language commands to make option requests that the application needs to process, it must include code to parse such requests in its SRV_LANGUAGE event handler.

Client option commands trigger a SRV_OPTION event. An application responds to such requests from within its SRV_OPTION event handler, using the srv_options command.

Inside the SRV_OPTION event handler

A client can request that an option be set or cleared, or that its current value be returned. Any of these commands triggers a SRV_OPTION event. Using the SRV_OPTION event handler, the application should:

1 Call srv_options with the cmd argument set to CS_GET. The type of command the client issued (SRV_SETOPTION, SRV_CLEAROPTION, or SRV_GETOPTION) will be returned in optcmdp. The option itself will be returned in optionp. *bufp will contain all legal values associated with the option.

For example, if the client has requested that Adaptive Server not report the number of rows affected by the query, optcmdp will contain SRV_SETOPTION, *optionp will contain CS_OPT_NOCOUNT, and *bufp will contain CS_TRUE.

2 If optcmdp is either SRV_SETOPTION or SRV_CLEAROPTION, the application should clear or set the option accordingly in a standalone Open Server application. If the application is a gateway, it should send the appropriate client calls to manipulate the remote server's option.

3 If optcmdp is SRV_GETOPTION, the application should call srv_options with cmd set to CS_SET, optcmd set to SRV_SENDOPTION, optionp set to the option the client seeks the value of, and bufp set to the current value.

Option descriptions and default values

Table 2-24 describes the options a client may set, retrieve, or clear, and each option's default value.

Table 2-24: Symbolic constants for server options

Symbolic constant	What the option does	Default value
CS_OPT_ANSINULL	If this option is set to CS_TRUE, Adaptive Server enforces the ANSI behavior that "=NULL" and "is NULL" are not equivalent. In standard Transact SQL, "=NULL" and "is NULL" are treated as equivalent. This option affects "<> NULL" and "is not NULL" behavior in a similar fashion.	CS_FALSE
CS_OPT_ANSIPERM	If this option is set to CS_TRUE, Adaptive Server will be ANSI compliant in its permission checks on update and delete statements.	CS_FALSE
CS_OPT_ARITHABORT	If this option is set to CS_TRUE, Adaptive Server aborts a query when an arithmetic exception occurs during its execution.	CS_FALSE
CS_OPT_ARITHIGNORE	If this option is set, Adaptive Server substitutes NULL for selected or updated values when an arithmetic exception occurs during query execution. Adaptive Server does not return a warning message. If neither CS_OPT_ARITHABORT nor CS_OPT_ARITHIGNORE is set, Adaptive Server substitutes NULL and prints a warning message after the query has been executed.	CS_FALSE
CS_OPT_AUTHOFF	Turns the specified authorization level off for the current server session. When a user logs in, all authorizations granted to that user are automatically turned on.	Not applicable
CS_OPT_AUTHON	Turns the specified authorization level on for the current server session. When a user logs in, all authorizations granted to that user are automatically turned on.	Not applicable
CS_OPT_CHAINXACTS	If this option is set to CS_TRUE, Adaptive Server uses chained transaction behavior. Chained transaction behavior means that each server command is considered to be a distinct transaction. Unchained transaction behavior requires an explicit commit transaction statement to define a transaction.	CS_FALSE, meaning unchained transaction behavior
CS_OPT_CURCLOSEONXACT	If this option is set to CS_TRUE, all cursors opened within a transaction space are closed when the transaction completes.	CS_FALSE
CS_OPT_CURREAD	Sets a security label specifying the current read level.	NULL
CS_OPT_CURWRITE	Sets a security label specifying the current write level.	NULL

Symbolic constant	What the option does	Default value
CS_OPT_DATEFIRST	This option sets the day considered to be the "first" day of the week.	For us_english, the default is CS_OPT_SUNDAY.
CS_OPT_DATEFORMAT	This option sets the order of the date parts month/day/year for entering datetime or smalldatetime data.	For us_english, the default is CS_OPT_FMTMDY.
CS_OPT_FIPSFLAG	If this option is set to CS_TRUE, Adaptive Server flags any nonstandard SQL commands that are sent.	CS_FALSE
CS_OPT_FORCEPLAN	If this option is set to CS_TRUE, Adaptive Server joins tables in the order in which the tables are listed in the "from" clause of the query.	CS_FALSE
CS_OPT_FORMATONLY	If this option is set to CS_TRUE, Adaptive Server sends back a description of the data rather than the data itself, in response to a select query.	CS_FALSE
CS_OPT_GETDATA	If this option is set to CS_TRUE, then Adaptive Server returns information on every insert, delete, or update command. Adaptive Server returns this information in the form of a message result set and parameters that an application can use to construct the name of the temporary table that will contain the rows that will be inserted or deleted. An update consists of insertions and deletions.	CS_FALSE
CS_OPT_IDENTITYOFF	Disables inserts into a table's identity column. For more information, see the set command in your Adaptive Server documentation.	Not applicable
CS_OPT_IDENTITYON	Enables inserts into a table's identity column. For more information, see the set command in your Adaptive Server documentation.	Not applicable
CS_OPT_ISOLATION	This option is used to specify a transaction isolation level. Legal levels are CS_OPT_LEVEL1 and CS_OPT_LEVEL3. Setting CS_OPT_ISOLATION to CS_OPT_LEVEL3 causes all pages of tables specified in a select query inside a transaction to be locked for the duration of the transaction.	CS_OPT_LEVEL1
CS_OPT_NOCOUNT	This option causes Adaptive Server to stop sending back information about the number of rows affected by each SQL statement.	CS_FALSE

Symbolic constant	What the option does	Default value
CS_OPT_NOEXEC	If this option is set to CS_TRUE, Adaptive Server processes queries through the compile step but does not execute them. This option is used with CS_OPT_SHOWPLAN.	CS_FALSE
CS_OPT_PARSEONLY	If this option is set, the server checks the syntax of queries, returning error messages as necessary, but does not execute the queries.	CS_FALSE
CS_OPT_QUOTED_IDENT	If this option is set to CS_TRUE, Adaptive Server treats all strings enclosed in double quotes as identifiers.	CS_FALSE
CS_OPT_RESTREES	If this option is set, Adaptive Server checks the syntax of queries but does not execute them, returning parse resolution trees (in the form of image columns in a regular row result set) and error messages as needed, to the client.	CS_FALSE
CS_OPT_ROWCOUNT	If this option is set, Adaptive Server returns only a maximum specified number of regular rows for select statements. This option does not limit the number of compute rows returned. CS_OPT_ROWCOUNT works somewhat differently from most options. It is always set on, never off. Setting CS_OPT_ROWCOUNT to 0 sets it back to the default, which is to return all the rows generated by a select statement. Therefore, the way to turn CS_OPT_ROWCOUNT off is to set it on with a count of 0.	0, meaning all rows are returned
CS_OPT_SHOWPLAN	If this option is set to CS_TRUE, Adaptive Server will generate a description of its processing plan after compilation and continue executing the query.	CS_FALSE
CS_OPT_STATS_IO	This option determines whether Adaptive Server internal I/O statistics are returned to the client after each query.	CS_FALSE
CS_OPT_STATS_TIME	This option determines whether Adaptive Server parsing, compilation, and execution time statistics are returned to the client after each query.	CS_FALSE
CS_OPT_STR_RTRUNC	If this option is set to CS_TRUE, Adaptive Server will be ANSI-compliant with regard to right truncation of character data.	CS_FALSE

Symbolic constant	What the option does	Default value
CS_OPT_TEXTSIZE	This option changes the value of the Adaptive Server global variable *@@textsize*, which limits the size of text or image values that Adaptive Server returns. When setting this option, you supply a parameter which is the length, in bytes, of the longest text or image value that Adaptive Server should return.	32,768 bytes
CS_OPT_TRUNCIGNORE	If this option is set to CS_TRUE, Adaptive Server ignores truncation errors, which is standard ANSI behavior. If this option is set to CS_FALSE, Adaptive Server raises an error when conversion results in truncation.	CS_FALSE

srv_options on page 323 lists the legal values and datatype for each option.

Example

The sample, *ctos.c*, includes code for processing client option commands.

Partial update

Open Client and Open Server supports the partial update of text and image columns. A partial update allows you to specify the part of the text or image field that you want to replace, delete, or insert at, and update that part only instead of modifying the entire field. For more information about text and image data handling, see the Open Client *Client-Library /C Reference Manual*.

Note Currently, Adaptive Server does not support partial update of text or image columns.

Open Server set-up

This section discusses how Open Server must be set up to support partial updates.

sp_mda

sp_mda is a stored procedure that retrieves metadata from the server. To support partial updates, your Open Server application must define an sp_mda stored procedure and specify the updatetext syntax that an Open Client application must use.

An Open Client application must invoke sp_mda using these parameters and values:

Parameter	Value	Description
clienttype	5	5 indicates that the client is Client-Library.
mdaversion	1	
clientversion	0	clientversion is an optional parameter that indicates the client version. The default is 0.

If the server supports partial updates, sp_mda returns:

Parameter	Value
mdinfo	"UPDATETEXT"
querytype	2
query	*updatetext_syntax* Example: `updatetext ? ? ? {NULL | ?} {NULL | ?}` where "?" indicates the updatetext parameters.

For more information about the sp_mda stored procedure, see the Mainframe Connect™ DB2 UDB Options for IBM CICS and IMS *Installation and Administration Guide*. For a sample implementation of sp_mda, see *$SYBASE/$SYBASE_OCS/sample/srvlibrary/updtext.c*.

SRV_T_BULKTYPE

To correctly retrieve the partially updated data sent by the client, the Open Server application must set SRV_T_BULKTYPE to SRV_TEXTLOAD, SRV_UNITEXTLOAD, or SRV_IMAGELOAD. For more information about SRV_T_BULKTYPE, see "SRV_T_BULKTYPE" on page 156.

Handlers

The SRV_LANGUAGE and SRV_BULK handlers have to be installed in Open Server. Open Server uses SRV_LANGUAGE to receive the updatetext statement from the Client-Library. SRV_BULK, on the other hand, receives the data sent through ct_send_data().

For more information about SRV_LANGUAGE and SRV_BULK, see the Open Client and Open Server *Common Libraries Reference Manual*.

Passthrough mode

An Open Server application that is acting as a gateway between an Open Client application and an Adaptive Server can pass TDS packets between client and server without examining their contents. An Open Server that handles TDS packets in this way operates in passthrough mode.

Because the Open Server gateway application does not have to unpack the TDS information as it arrives from the client, and repacks information before sending it to the Adaptive Server, passthrough mode is very efficient.

For Open Client Server 12.5.1 and earlier, passthrough mode ensures that the negotiated packet size is correct by limiting the packet size requested by the client to the maximum size supported by the Open Server.

When a remote server supporting server-specified packetsize sets a packetsize larger than that configured in Open Server, the larger packetsize is used, regardless of the configured SRV_S_NETBUFSIZE.

There are two types of passthrough modes:

- Regular passthrough mode
- Event handler passthrough mode

Both types of passthrough modes use the passthrough routines srv_recvpassthru, ct_sendpassthru, ct_recvpassthru, and srv_sendpassthru. The differences are as follows:

- In regular passthrough mode, the Open Server application recognizes events and triggers event handlers. These event handlers are coded to call the passthrough routines.

  For more information on regular passthrough mode, see “Regular passthrough mode” on page 130.

- In event handler passthrough mode, the Open Server application does not recognize most types of events on the connection. Instead, the full passthrough event handler is triggered whenever a network read for the connection completes. The full passthrough event handler is coded to call the passthrough routines.

  For more information on event handler passthrough mode, see "Event handler passthrough mode" on page 132.

DB-Library also provides routines to support passthrough mode. See the Open Client *DB-Library/C Reference Manual* for details.

Regular passthrough mode

Initially, Sybase supported only this type of passthrough mode.

In regular passthrough mode, Open Server recognizes events (SRV_LANGUAGE, SRV_RPC, and so on) and triggers the appropriate event handlers. Individual event handlers must be coded to call passthrough routines.

Negotiating the TDS protocol level in passthrough mode

When Sybase clients and servers connect, they first agree upon the TDS protocol level to use, usually the latest version of the protocol that both programs recognize. See "Negotiated behavior" on page 119 for more information on initial protocol negotiation.

When an Open Server gateway application operates in passthrough mode, the TDS packets are created and interpreted by the remote Sybase client and Adaptive Server—not by the gateway. Therefore, TDS negotiation occurs between the two remote programs. The gateway must facilitate this negotiation by relaying responses between the two parties. The TDS negotiation process must occur inside a SRV_CONNECT event handler and involves the following steps:

1 Set one of these properties:

  - SRV_T_PASSTHRU, to indicate that the thread will use regular passthrough mode
  - SRV_T_FULLPASSTHRU, to indicate that the thread will use event handler passthrough mode

  You must set one of these properties for srv_getloginfo and ct_setloginfo to negotiate client/server capabilities correctly for passthrough mode.

2 srv_getloginfo – allocate a CS_LOGINFO structure and fill it with login information from the client thread.

3 ct_setloginfo – prepare a CS_LOGINFO structure with the login information retrieved in step 2.

4 If the client application is using network-based authentication, perform these steps to transfer the client's security principal name. These steps are required because the security principal name is not part of the CS_LOGINFO structure.

   - Call srv_thread_props(..CS_GET, SRV_T_USER) to retrieve the client's security principal name.
   - Call ct_con_props(..CS_SET, CS_USERNAME) to set the principal name for the connection to the target server.

5 Log in to the remote server by calling ct_connect.

6 ct_getloginfo – transfer login response information from a CS_CONNECTION structure to the newly allocated CS_LOGINFO structure.

7 srv_setloginfo – send the remote server's response, retrieved in step 6, to the client, then release the CS_LOGINFO structure.

Using regular passthrough mode

Regular TDS passthrough takes place inside any event handler except SRV_ATTENTION, SRV_CONNECT, SRV_DISCONNECT, SRV_START, or SRV_STOP.

Client requests arrive in a stream of one or more TDS packets. The handler repeatedly calls srv_recvpassthru as long as the info argument remains set to SRV_I_PASSTHRU_MORE. As each packet is received, the handler calls ct_sendpassthru to pass the packet on to the remote Adaptive Server or Open Server. The remote server receives exactly the same TDS stream it would receive from a directly connected client.

Warning! The latest version of TDS introduces multiple commands in a single batch. Only the first command triggers an event handler. Open Server will not call event handlers for the remaining commands.

A Client-Library routine, ct_recvpassthru, receives the TDS packets as they arrive at the connection. The srv_sendpassthru Server-Library routine sends the packet on to the client. The ct_recvpassthru routine retrieves another TDS packet as long as it returns CS_PASSTHRU_MORE.

Example

The sample, *fullpass.c*, illustrates a passthrough mode gateway.

Event handler passthrough mode

In this type of passthrough mode, Open Server does not recognize most types of events. Instead, Open Server invokes the full-passthrough event handler each time a network read for the connection completes.

Event handler passthrough mode is designed to enable client/server connections using per-packet security services (such as encryption) to use passthrough mode.

Regular passthrough mode requires that Open Server interpret packets to identify particular events. When packets are encrypted, this is not possible.

To use event handler passthrough mode for a thread:

- Code a full-passthrough event handler and install it. For more information, see “Coding and installing a full passthrough event handler” on page 132.
- Enable event handler passthrough mode for a thread by setting SRV_T_FULLPASSTHRU to CS_TRUE in the Open Server connection handler. For more information, see “Enabling event handler passthrough mode for a thread” on page 133.
- Call routines to negotiate the TDS protocol level between the client and the target server. For more information, see “Negotiating the TDS protocol level” on page 133.

Coding and installing a full passthrough event handler

The prototype for a full-passthrough event handler is:

```
CS_RETCODE CS_PUBLIC func (SRV_PROC *sproc);
```

A full-passthrough event handler calls these routines to receive and send packets:

- srv_recvpassthru
- ct_sendpassthru
- ct_recvpassthru
- srv_sendpassthru

You will not be able to forward attention events while performing a srv_recvpassthru/ct_sendpassthru loop. You must add logic to the event-handler code and attn-handler code so that an attentionevent is not forwarded until after the full command has been forwarded to the remote server.

A full-passthrough event handler should return CS_SUCCEED to report normal completion. A return value other than CS_SUCCEED kills the current Open Server thread.

To install a full-passthrough event handler, call srv_handle with srv_handle's *event* parameter as SRV_FULLPASSTHRU and the *handler* parameter as the address of the handler routine.

Enabling event handler passthrough mode for a thread

To enable event handler passthrough mode for a thread, set the SRV_T_FULLPASSTHRU thread property to CS_TRUE in the Open Server connection handler.

Once event handler passthrough mode is enabled, Open Server invokes the full-passthrough handler each time a network read from the connection completes.

No events of type SRV_LANGUAGE, SRV_RPC, SRV_BULK, SRV_CURSOR, SRV_MSG, SRV_OPTION, or SRV_DYNAMIC are raised for the thread.

SRV_ATTENTION events, however, are raised. The Open Server application must install a SRV_ATTENTION handler to correctly handle cancel requests.

Negotiating the TDS protocol level

Gateway applications using event handler passthrough mode facilitate the negotiation of the TDS protocol level between client application and target server in exactly the same way as applications using regular passthrough mode.

Inside the application's connection handler, after setting SRV_FULLPASSTHRU to CS_TRUE, call the srv_getloginfo, ct_setloginfo, ct_getloginfo, and srv_setloginfo routines.

For more information on calling these routines, see “Negotiating the TDS protocol level in passthrough mode” on page 130.

Processing parameter and row data

A note on terminology

The term *parameter data* refers to parameters retrieved from or returned to a client. Some can be input parameters, while others can be output or *return* parameters. Return parameters are processed in two steps: they are partially processed when the Open Server application reads them into program variables. The processing is completed when they are sent back to the client.

The Open Server data processing model

In Open Server, three routines work together to retrieve parameter data and formats from a client and to send row data, and return parameters and their formats to a client. These routines are srv_descfmt, srv_bind, and srv_xferdata.

An application uses these routines to process any client command that provides parameters or requests results. RPC commands, language commands, cursor commands, dynamic SQL commands, message commands, and negotiated login commands fall into this category.

Each of the three routines takes a type argument, which indicates the type of data being described, bound, or transferred. For example, type would be set to SRV_CURDATA when describing the format of cursor command input parameters, whereas type would be set to SRV_ROWDATA when processing result rows. For a list of legal type values, see each routine’s reference page in Chapter 3.

All three routines take a cmd argument as well, which indicates the direction of data flow. A value of CS_GET instructs the Open Server application to retrieve information from the client, while CS_SET instructs the application to return results to a client.

An application can use these routines to:

- Retrieve input and return parameter information inside a SRV_RPC, SRV_CURSOR, SRV_DYNAMIC, SRV_MSG, or SRV_CONNECT event handler.
- Send back result row information inside a SRV_RPC, SRV_CURSOR, SRV_DYNAMIC, SRV_LANGUAGE, or SRV_MSG event handler.
- Send back return parameter information inside a SRV_LANGUAGE or SRV_RPC handler.

Retrieving parameters

To process parameters, an application must:

1 Call srv_numparams to determine how many parameters, if any, the command includes.

2 Call srv_descfmt to obtain a description of each parameter. Among other things, the description will indicate if the parameter is a return parameter. If it is, the retrieval process stops here. If the parameter is an input parameter, the application must continue with steps 3 and 4.

3 Call srv_bind to provide program variables in which to store the parameter data coming over the network from the client.

4 Call srv_xferdata to transfer the client data into the application program variables specified in step 3.

Return parameters contain no valid data when retrieved from a client. The application fills valid data in when it returns the return parameters to the client. Open Server transparently converts the return parameter format from the program variable format to the client format.

Note that from within a SRV_LANGUAGE handler, an application can "construct" return parameters out of an undifferentiated language stream without having first retrieved actual parameters. See "Returning parameters in a language data stream" on page 138, for further explanation.

srv_descfmt and srv_bind are called once for each parameter, while srv_xferdata is called once for the entire parameter stream. An application must not call srv_xferdata until all parameters have been described and bound.

An application must invoke the three routines with their cmd arguments set to CS_GET, as the application retrieves information from the client.

Returning rows

The processing of row data requires three basic steps:

1 Describe each column in the row by calling srv_descfmt.

2 Indicate where the application has stored the row data and identify its format by calling srv_bind.

3 Transfer the data from the application program variables specified in step 2 to the client by calling srv_xferdata.

The srv_descfmt routine must be called once for each column in a row; however srv_xferdata and srv_bind routines are called as many times as there are result rows. An application must not call srv_xferdata until all columns have been described and bound.

An application must invoke the three routines with their cmd arguments set to CS_SET as the application returns results to the client.

Returning return parameters

The processing of return parameters requires two basic steps:

1 Indicate where the application has stored the return parameter data and identify its format by calling srv_bind.

2 Transfer the return parameter data from the application program variables specified in step 2 to the client by calling srv_xferdata.

An application must invoke the two routines with their cmd arguments set to CS_SET as the application returns results to the client.

If return parameters have been “constructed” out of a text stream, they need to be described, in addition to being bound and transferred. See “Returning parameters in a language data stream” on page 138, for further explanation.

A closer look at describing, binding, and transferring

This section provides more detail on the describe, bind, and transfer processes.

Describing

The srv_descfmt routine gives an Open Server application the information it needs to send back data to the client in the format the client expects. Conceptually, it conveys information about how the client viewed (CS_GET) or will view the data (CS_SET).The srv_descfmt routine retrieves or sets a variety of parameter and row column characteristics.

These characteristics include, among other information:

- The parameter or column name
- The parameter or column name length
- The parameter or column number, where the first parameter or column in a stream is numbered 1
- The parameter or column datatype
- Whether the parameter or column can be set to null
- Whether a parameter is a return parameter

The clfmtp argument to srv_descfmt points to a CS_DATAFMT structure containing this information. For details, see “CS_DATAFMT structure” on page 54.

Binding

To examine data it receives from clients, an Open Server application must store the data in local program variables. When an application calls srv_bind, it associates parameter or column data with a local application program variable and describes the format of that variable.

A call to srv_bind with cmd set to CS_GET instructs Open Server where to put the data coming from the client. A call to srv_bind with cmd set to CS_SET instructs Open Server where to find the data it is sending back to the client.

The osfmtp argument to srv_bind points to a CS_DATAFMT structure containing format information about the local program variables.

Transferring

The srv_xferdata routine moves data in and out of the local program variables named in a srv_bind call. When cmd is set to CS_GET, a call to srv_xferdata moves input parameter data from the client into the variables. When cmd is set to CS_SET, the routine pulls column and return parameter data out of the local program variables and sends it to the client.

Note Although srv_senddone currently flushes formats and column information to the network, it will not in future versions. Applications should always use srv_xferdata to flush information to the network.

For more information on srv_bind, srv_descfmt, and srv_xferdata, see their respective reference pages.

Automatic conversion

When an application retrieves data, Open Server converts the data to the local format if the format in which the client sent the data in differs from the format of the application's local program variables. If the same situation arises when an application sends data back to a client, Open Server converts the data to the client format.

Returning parameters in a language data stream

There is no notion of parameters in a language data stream. An Open Server application equipped to parse a text stream, however, can "construct" return parameters from the incoming stream. It can then load the parameters with data and send them back using the describe/bind/transfer procedures.

For example, a client can send a Transact-SQL stored procedure query that includes return parameters. An Open Server application expecting this query can parse for the string "output = @var" (where *var* is the placeholder for the return parameter) and send back format information and data for *var*.

An application can call srv_descfmt with cmd set to CS_SET and type set to SRV_RPCDATA from within a language event handler only.

Example

The sample, *ctos.c*, processes parameter and column data using the describe/bind/transfer series of calls.

Properties

Properties define aspects of an Open Server application's behavior. Open Server properties fall into three categories:

- Context properties
- Server properties
- Thread properties

Context and server properties pertain to the Open Server application as a whole. They govern server-wide behavior and hold true for all client-server connections.

Thread properties pertain to client and service threads. Most are only able to be retrieved, not set. An application can override certain server-wide attributes on a per-connection basis by setting certain thread properties.

A programmer can tailor an Open Server application's functionality by setting properties. In addition, an application can retrieve certain properties when it needs information.

You use cs_config, srv_props, and srv_thread_props, to set and retrieve context, server, and thread properties, respectively.

See "Context properties" on page 140, "Server properties" on page 141 and "Thread properties" on page 148 for more information on each type of property.

See the Open Client and Open Server *Common Libraries Reference Manual* section on cs_config, and the srv_props and srv_thread_props reference pages in this manual, for more information on setting and retrieving properties.

Context properties

Context properties are stored in a CS-Library CS_CONTEXT structure. An application sets or retrieves context properties using the CS-Library routine cs_config. See the Open Client and Open Server *Common Libraries Reference Manual* for information on this routine.

There are three kinds of context properties:

- Context properties specific to CS-Library

  cs_config sets and retrieves the values of CS-Library-specific context properties. With the exception of CS_LOC_PROP, properties set through cs_config affect only CS-Library. CS-Library-specific context properties are listed on the manual page for cs_config in the Open Client and Open Server *Common Libraries Reference Manual.*

- Context properties specific to Client-Library

  ct_config sets and retrieves the values of Client-Library-specific context properties. Properties set through ct_config affect only Client-Library. See the Open Client *Client-Library/C Reference Manual* for more information.

- Context properties specific to Server-Library

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

The context properties that an Open Server application can set include:

- The routine Open Server calls when it detects a CS-Library error.
- Localization information, including the Open Server's national language, character set, and sort order.
- The location of a pointer to application data space. This property allows applications to associate control information with Open Server's context. Open Server does not use this pointer; it is provided for the convenience of Open Server application programmers.

These context properties can be both set and retrieved through the cs_config routine. For more information on context properties and their associated routines and structures, see the Open Client and Open Server *Common Libraries Reference Manual*.

Server properties

Server properties are stored in a CS_CONTEXT structure. An application sets or retrieves server properties using the Server-Library routine srv_props.

Server properties determine many aspects of an Open Server application's behavior, including its memory-allocation routines, and the maximum number of physical network connections it can establish.

For server properties to take effect, an application must set them prior to initialization. Open Server raises an error if a server property is set following initialization.

An application's initialization code must include these steps:

1 Allocate a CS_CONTEXT structure, through a call to cs_ctx_alloc.

2 Call srv_version to set the Open Server version number. srv_version takes a pointer to a CS_CONTEXT structure.

3 Call srv_props to set property defaults.

4 Call srv_init to initialize the server.

5 Start the server running with a call to srv_run.

Some properties can be set and retrieved, while others are set-only or retrieve-only. srv_props on page 334 provides this information.

Table 2-25: Server properties

Property name	Definition	Notes
SRV_S_ALLOCFUNC	The address of the routine Open Server will use to allocate memory.	
SRV_S_APICHK	A Boolean indicating whether to enable (CS_TRUE) or disable (CS_FALSE) the validation of Server-Library arguments and state checking.	Many Server-Library routines internally call CS-Library routines. For this reason, application programmers who want thorough argument and state checking should set the cs_config property CS_NOAPICHK to CS_FALSE.

Property name	Definition	Notes
SRV_S_ATTNREASON	The reason an Open Server application's attention handler was called.	Returns SRV_ATTENTION if a client attention triggered the SRV_ATTENTION event, and SRV_DISCONNECT if a client disconnect triggered the event.
SRV_S_CERT_AUTH	CS_CHAR Specify the path to the file containing trusted CA certificates.	The maximum allowable length for this property is SRV_MAXCHAR bytes.
SRV_S_CURTHREAD	The address of the active thread's internal control structure.	Some SRV_S_CURTHREAD functionality becomes disabled when SRV_S_PREEMPT is used in conjunction with threaded libraries.
SRV_S_DEFQUEUESIZE	Deferred event queue size.	
SRV_S_DISCONNECT	Set this property to CS_TRUE to call an application's SRV_ATTENTION event handler when a client disconnects.	The SRV_ATTENTION event handler can be called at interrupt level, if the client disconnect is detected at interrupt time.
SRV_S_DSPROVIDER	The directory service provider name. The default value is platform specific. See the Open Client and Open Server *Configuration Guide* for your platform.	The maximum allowable length for this property is SRV_MAXCHAR bytes.
SRV_S_DSREGISTER	Set to CS_TRUE to indicate that Server-Library should register itself with a directory at start-up. Set to CS_FALSE to prevent registration.	
SRV_S_ERRHANDLE	The address of the Open Server error handler.	
SRV_S_FREEFUNC	The address of the routine Open Server uses to free memory.	
SRV_S_IFILE	The name of the interfaces file available for use by Open Server.	The maximum allowable length for this property is SRV_MAXCHAR bytes.

Property name	Definition	Notes
SRV_S_LOGFILE	The name of the log file Open Server writes to.	The SRV_S_LOGFILE property can be set after calling srv_init. After srv_init is called, setting the SRV_S_LOGFILE property with *bufp*set to an empty string ("") and *buflen* set to 0 will close the log file. The maximum allowable length for this property is SRV_MAXCHAR bytes.
SRV_S_LOGSIZE	The maximum size of the log file. If the log exceeds this size, Open Server will move the current contents of the log file to another file with the name *currentfilename*_old and will truncate the current log to 0 bytes.	
SRV_S_MSGPOOL	The number of messages available to an Open Server application at runtime.	Open Server applications use messages through srv_putmsgq. A message remains in use until it is received through srv_getmsgq. The value of an application's SRV_S_MSGPOOL configuration parameter should be based on its use of these two routines.
SRV_S_NETBUFSIZE	The maximum size of the network I/O buffer to be used by client connections. Unless explicitly set, SRV_S_NETBUFSIZE is the default maximum value of 8192 bytes.	For Open Client Server 12.5.1 and earlier, the size of the network buffer is determined at login time. If a smaller size is requested, Open Server does not resize the memory buffer; it leaves part of it unused. For this reason, do not make the value larger than required or unused memory will be allocated.

Property name	Definition	Notes
SRV_S_NETTRACEFILE	Net-Library tracing written to this file.	The maximum allowable length for this property is SRV_MAXCHAR bytes.
SRV_S_NUMCONNECTIONS	The maximum number of physical network connections the Open Sever application will accept.	A server-to-server connection is only one physical connection, regardless of how many subchannels are used. Outgoing Client-Library connections, for example in a passthrough Open Server application, are limited by the CS_MAX_CONNECT property. CS_MAX_CONNECT can be set using ct_config().
SRV_S_NUMMSGQUEUES	The number of message queues available to the Open Server application.	
SRV_S_NUMMUTEXES	The number of mutual exclusion semaphores available to the Open Server application.	
SRV_S_NUMREMBUF	The window size used on server-to-server connections. It indicates the maximum number of packets that can be outstanding on a logical subchannel before an acknowledgment is required.	
SRV_S_NUMREMSITES	The maximum number of remote server site handlers that can be active at a given time.	
SRV_S_NUMTHREADS	The maximum number of threads available to an Open Server application.	
SRV_S_NUMUSEREVENTS	The number of user events an application can define.	

Property name	Definition	Notes
SRV_S_PREEMPT	A Boolean. If CS_TRUE, Open Server will use preemptive scheduling. If CS_FALSE, Open Server uses non-preemptive scheduling.	Preemptive scheduling is not available on all platforms. Use srv_capability to determine whether it is available. When SRV_S_PREEMPT is used in conjunction with threaded libraries, some functionality of SRV_S_CURTHREAD become disabled.
SRV_S_REALLOCFUNC	The address of the routine Open Server uses to reallocate memory.	
SRV_S_REQUEST_CAP	The default client requests that the Open Server application accepts.	See "Capabilities" on page 24.
SRV_S_RESPONSE_CAP	The default responses to the client that the Open Server application supports.	See "Capabilities" on page 24.
SRV_S_RETPARAMS	Return parameters are sent if an error occurs during execution	This server property can be used to limit the behavior to specific threads by using the default (false).
SRV_S_SEC_KEYTAB	The keytab file name (including the path name) for use with the DCE security driver.	You can specify a principal other than the currently logged-in user who is running the application. The property SRV_S_SEC_PRINCIPAL sets the principal name. The DCE utility dcecp allows you to create a keytab file. The keytab file is an ordinary UNIX file, so you need to set permissions on the file to restrict access. The file must be readable by the user who starts the Open Server application. See "Security services" on page 170 for more information. The maximum allowable length for this property is SRV_MAXCHAR bytes.

Property name	Definition	Notes
SRV_S_SEC_PRINCIPAL	The principal name to use when acquiring credentials for the Open Server application. The value of this property defaults to the Open Server application's network name, which can be specified through srv_init.	The maximum allowable length for this property is SRV_MAXCHAR bytes. See "Security services" on page 170 for more information about this property.
SRV_S_SERVERNAME	The name of the Open Server application.	This is the name the Open Server application is known by when it is up and running. It is also the name used to look up its listen address in the interfaces file. The maximum allowable length for this property is SRV_MAXCHAR bytes.
SRV_S_SSL_CIPHER	Comma-separated list of CipherSuite names.	The maximum allowable length for this property is SRV_MAXCHAR bytes.
SRV_S_SSL_LOCAL_ID	A structure containing a file name and a password used to decrypt the information in the file.	The maximum allowable length for this property is SRV_MAXCHAR bytes.
SRV_S_SSL_REQUEST_ CLIENT_CERT	Requires that the client provide a certificate to log in to an Open Server application.	
SRV_S_SSL_VERSION	Must be one of a list of defined values.	The defined values are: • CS_SSLVER_20 • CS_SSLVER_30 • CS_SSLVER_TLS1 Adaptive Server only accepts connections using the default, CS_SSLVER_TLS1.
SRV_S_STACKSIZE	The size of the stack allocated for each thread.	
SRV_S_TDSVERSION	The Tabular Data Stream protocol version that Open Server uses to negotiate all client connections.	See "SRV_S_TDSVERSION" on page 147 for a list of values.

Property name	Definition	Notes
SRV_S_TIMESLICE	The number of clock ticks an active thread consumes before the time slice callback routine is called.	See the srv_callback reference page for information on time slice callbacks.
SRV_S_TRACEFLAG	The type of tracing desired.	See "SRV_S_TRACEFLAG" on page 148 for a list of flags.
SRV_S_TRUNCATELOG	A Boolean. If CS_TRUE, Open Server truncates the log file during start-up.	The SRV_S_TRUNCATELOG property can be set after calling srv_init.
SRV_S_USERVLANG	A Boolean. If CS_TRUE, the Open Server application's native language is used for error messages. If CS_FALSE, the client's national language is used for error messages.	
SRV_S_VERSION	A character string that contains the name, version date, and copyright information of the Open Server Server-Library in use.	
SRV_S_VIRTCLKRATE	The clock rate, in microseconds, per tick.	
SRV_S_VIRTIMER	A Boolean. If CS_TRUE, the virtual timer is enabled. If CS_FALSE, the virtual timer is disabled.	

SRV_S_TDSVERSION

During the client login process, Open Server negotiates with the client application to agree on a TDS version. The SRV_S_TDSVERSION property value determines Open Server's starting point. The client agrees to communicate at or below this starting point. Later on in the login process, the Open Server application can renegotiate the TDS version for a particular connection, using the SRV_T_TDSVERSION thread property. See "Thread properties" on page 148 for details.

Table 2-26 describes the legal values for this property:

Table 2-26: Values for SRV_S_TDSVERSION

SRV_S_TDSVERSION value	Meaning
SRV_TDSNONE	Unknown version of TDS
SRV_TDS_4.0	Negotiation starts at TDS 4.0
SRV_TDS_4_0_2	Negotiation starts at TDS 4.0.2
SRV_TDS_4_2	Negotiation starts at TDS 4.2
SRV_TDS_4_6	Negotiation starts at TDS 4.6
SRV_TDS_4_9_5	Negotiation starts at TDS 4.9.5
SRV_TDS_5_0	Negotiation starts at TDS 5.0

SRV_S_TRACEFLAG

The SRV_S_TRACEFLAG property is a bitmap. Its flags, which can be OR'd together, are described in Table 2-27:

Table 2-27: Values for SRV_S_TRACEFLAG

Flag	Meaning
SRV_TR_ATTN	Open Server displays information indicating whether the Open Server application has received or acknowledged an attention.
SRV_TR_DEFQUEUE	Open Server traces event queue activity.
SRV_TR_EVENT	Open Server displays information about the events it has triggered.
SRV_TR_MSGQ	Open Server traces **message queue** activity.
SRV_TR_NETDRIVER	Open Server traces TCL Net-Lib driver requests.
SRV_TR_NETREQ	Open Server traces TCL requests.
SRV_TR_NETWAKE	Open Server traces TCL wakeup requests.
SRV_TR_TDSDATA	Open Server displays TDS packet contents in hexadecimal and ASCII format. This is the actual TDS traffic between a client and the Open Server application.
SRV_TR_TDSHDR	Open Server displays the TDS protocol packet header information, such as packet type and length.

Thread properties

A thread is a piece of code that executes to accomplish a specific task or set of tasks. There are several types of Open Server threads. Thread properties define aspects of a thread's behavior and set limits on its resources.

For more details on Open Server threads, see “Multithread programming” on page 109.

Only a few thread properties can be set, but all are retrievable. An application calls srv_thread_props to retrieve and set a thread property value. Properties that can be set are noted as such in the srv_thread_props reference page. An application can retrieve and set thread properties at any point after initialization.

Open Server assigns defaults for each thread property that can be set when it creates threads at initialization time. See srv_thread_props on page 435 for a list of defaults.

Table 2-28: Thread properties

Property name	Definition	Notes
SRV_T_APPLNAME	The client application’s name.	
SRV_T_BYTEORDER	The client’s requested byte-ordering scheme. SRV_LITTLE_ENDIAN indicates that the least significant byte is the high byte. SRV_BIG_ENDIAN indicates that the least significant byte is the low byte.	
SRV_T_BULKTYPE	The type of bulk transfer being sent by the client.	See “SRV_T_BULKTYPE” on page 156 for a list of legal values.
SRV_T_CHARTYPE	The type of character data representation.	See “SRV_T_CHARTYPE” on page 157 for a list of legal values.
SRV_T_CIPHER_SUITE	CS_CHAR* The CipherSuite that is used to encrypt and decrypt data exchanged during the SSL-based session. The CipherSuite is negotiated during the connection handshake.	
SRV_T_CLIB	The name of the library product used by the client to connect to the Open Server application.	
SRV_T_CLIBVERS	The version of the library product used by the client to connect to the Open Server application.	
SRV_T_CLIENTLOGOUT	A Boolean. Indicates whether the client completed an orderly or aborted logout, where CS_TRUE indicates an orderly logout.	This property can only be retrieved from inside the SRV_DISCONNECT event handler.

Property name	Definition	Notes
SRV_T_CONVERTSHORT	A Boolean. Indicates whether to automatically convert 4-byte datetime, 4-byte floating point, and 4-byte money datatypes to their 8-byte counterparts.	
SRV_T_DUMPLOAD	A Boolean. Indicates whether to disallow the use of dump/load and bulk insert for this client connection.	
SRV_T_ENDPOINT	The file descriptor or file handle of the connected client. For subchannels, the site handler end point value is returned. SRV_T_ENDPOINT is equivalent to the CS_ENDPOINT value in Client-Library.	Valid for client threads, site handlers and subchannels. Not valid for service threads. See "SRV_T_ENDPOINT" on page 157 for an example of using SRV_T_ENDPOINT.
SRV_T_EVENT	The Open Server event the thread is currently in.	See "SRV_T_EVENT" on page 158 for a list of legal values.
SRV_T_EVENTDATA	A generic data address associated with a particular event raised by the Open Server application.	Data address set using srv_event.
SRV_T_FULLPASSTHRU	A Boolean. When set to CS_TRUE, the SRV_FULLPASSTHRU event handler is activated for the thread.	Can only be set inside the Open Server application's connect handler. The value of the SRV_T_EVENT property is SRV_FULLPASSTHRU when retrieved inside the full-passthrough event handler.
SRV_T_FLTTYPE	The type of floating point representation used by the client.	See "SRV_T_FLTTYPE" on page 158 for a list of legal values.
SRV_T_GOTATTENTION	A Boolean. Indicates whether the client thread has received an attention.	
SRV_T_HOSTNAME	The name of the host machine from which the client connection originated.	
SRV_T_HOSTPROCID	The process ID of the client program.	This is the operating system process ID received in the client login record.

Property name	Definition	Notes
SRV_T_IODEAD	A Boolean. Indicates whether a thread's I/O channel is valid.	CS_TRUE means a thread cannot successfully perform I/O, CS_FALSE means it can. Open Server always returns CS_FALSE for service threads.
SRV_T_LOCALE	A pointer to a CS_LOCALE structure allocated by the Open Server application.	Use this property to retrieve or set localization information.
SRV_T_LOGINTYPE	The type of login record received.	See "SRV_T_LOGINTYPE" on page 159 for a list of legal values.
SRV_T_MIGRATED	A Boolean. Indicates whether a connection is a new or a migrated connection. This read-only property is set to true when the client is migrating or has migrated to the server.	See "SRV_T_MIGRATED" on page 159 for more details.
SRV_T_MIGRATE_STATE	Indicates the migration state of the client. It is a read-only property that any thread can access.	See "SRV_T_MIGRATE_STATE " on page 159 for more details.
SRV_T_MACHINE	The host name of the machine the client thread is running on.	

Property name	Definition	Notes
SRV_T_NEGLOGIN	The type of negotiated login, if any, the client has requested.	This property is a bitmask that can take any of five values: • SRV_CHALLENGE signals the client's intent to negotiate through a challenge/response exchange. • SRV_ENCRYPT signals the client's intent to pass a symmetrically encrypted password. • SRV_SECLABEL indicates that the client will send security labels. • SRV_APPDEFINED indicates that an application-defined login handshake is in use. • SRV_EXTENDED_ENCRYPT signals the client's intent to pass an assymetrically encrypted password.
SRV_T_NOTIFYCHARSET	A Boolean. Indicates whether the client should be notified when the character set in use has changed.	
SRV_T_NOTIFYDB	A Boolean. Indicates whether the client should be notified of the outcome of a use db Transact-SQL command.	
SRV_T_NOTIFYLANG	A Boolean. Indicates whether the client should be notified when the national language in use has changed.	
SRV_T_NOTIFYPND	The number of pending notifications to be delivered to the client.	This property is retrieve-only.
SRV_T_NUMRMTPWDS	The number of remote passwords.	
SRV_T_PACKETSIZE	The negotiated packet size used to communicate with the client.	The packet size is negotiated transparently at login time.

Property name	Definition	Notes
SRV_T_PASSTHRU	A Boolean. Indicates whether the client thread is operating in passthrough mode.	With version 11.1, this property can be set inside the application's connect handler. When set to CS_TRUE, the srv_getloginfo and ct_setloginfo routines negotiate the client connection's capabilities independently of the Open Server's capabilities. Since a full-passthrough gateway does not recognize different command and result types, this is the desired behavior.
SRV_T_PRIORITY	The priority level at which Open Server should schedule the thread.	This property is retrieve-only. To set a thread's priority, call srv_setpri.
SRV_T_PWD	The password string the client sent in the login record.	For remote server connections, this property returns the remote server password.
SRV_T_RETPARAMS	Return parameters are sent if an error occurs during execution.	If the SRV_S_RETPARAMS is set the RPC return behavior applies to all threads.
SRV_T_RMTCERTIFICATE	CS_SSLCERT * A pointer that describes the client certificate.	
SRV_T_RMTPWDS	An array of SRV_RMTPWDs.	See "SRV_T_RMTPWDS" on page 160 for the structure's definition.
SRV_T_RMTSERVER	The local server name for client connections. The remote server name for server-to-server connections.	
SRV_T_ROWSENT	The number of rows returned to the client in this event.	
SRV_T_SEC_CHANBIND	A Boolean indicating whether channel binding is being used on the client/server connection associated with this thread.	
SRV_T_SEC_CONFIDENTIALITY	A Boolean indicating whether data confidentiality is being used on the client/server connection associated with this thread.	This is usually implemented using data encryption.

Property name	Definition	Notes
SRV_T_SEC_CREDTIMEOUT	The number of seconds remaining for which the credentials remain valid on the client/server connection associated with this thread.	Possible values are: • CS_NO_LIMIT – never expires • CS_UNEXPIRED – unexpired • 0 – expired • A positive number – the number of seconds remaining
SRV_T_SEC_DATAORIGIN	A Boolean indicating whether data origination service is being used on the client/server connection associated with this thread.	
SRV_T_SEC_DELEGATION	A Boolean indicating whether delegation is enabled by the client.	All work done in this thread should use the client's authorization level. Use the SRV_T_USER property to access the principal name. Use the SRV_T_SEC_DELEGCRED property to obtain the delegated credentials to use in initiating a security session with another security peer.
SRV_T_SEC_DELEGCRED	The delegated credentials (if any) of the client in the current security session.	The SRV_T_SEC_DELEGATION property indicates whether delegation is enabled by the client. If it is enabled, the Open Server application may obtain the delegated credentials using the SRV_T_SEC_DELEGCRED property.
SRV_T_SEC_DETECTREPLAY	A Boolean indicating whether detection of message replay is being used on the client/server connection associated with this thread.	
SRV_T_SEC_DETECTSEQ	A Boolean indicating whether detection of out-of-sequence messages is being used on the client/server connection associated with this thread.	

Property name	Definition	Notes
SRV_T_SEC_INTEGRITY	A Boolean indicating whether integrity service is being used on the client/server connection associated with this thread.	This is usually implemented using a cryptographic signature.
SRV_T_SEC_MECHANISM	The local name of the security mechanism being used on the client/server connection associated with this thread.	
SRV_T_SEC_MUTUALAUTH	A Boolean indicating whether mutual authentication was performed on the client/server connection associated with this thread.	
SRV_T_SEC_NETWORKAUTH	A Boolean indicating whether network authentication was performed on the client/server connection associated with this thread.	
SRV_T_SEC_SESSTIMEOUT	The number of seconds remaining for which the security session remains valid on the client/server connection associated with this thread.	Possible values are: • CS_NO_LIMIT – never expires • CS_UNEXPIRED – unexpired • 0 – expired • A positive number – the number of seconds remaining
SRV_T_SESSIONID	Retrieves the session ID that the client sends to Open Server. Also, sets the session ID to be sent to the client in the SRV_CONNECT handler.	See “SRV_T_SESSIONID” on page 161 for more details.
SRV_T_SSL_VERSION	The SSL/TLS protocol version that was negotiated during the connection handshake.	
SRV_T_SPID	The thread’s process identifier.	This is the unique ID assigned to this thread. Thread IDs are reused once a thread has exited.
SRV_T_STACKLEFT	The size of unused stack available to the thread.	

Property name	Definition	Notes
SRV_T_TDSVERSION	The version of TDS the client thread is using.	Setting this thread in the SRV_CONNECT event handler allows an Open Server application to negotiate the TDS version to some value other than Open Server's default for the thread. See "SRV_T_TDSVERSION" on page 161 for a list of legal values.
SRV_T_TYPE	The thread type.	See "SRV_T_TYPE" on page 162 for a list of legal types.
SRV_T_USER	The user name the client thread logged on with.	
SRV_T_USERDATA	A generic data address used for application-specific purposes.	Can be set.
SRV_T_USESRVLANG	A Boolean. Set to CS_TRUE if error messages should be in the server's national language, CS_FALSE if in the client's.	Set this to override the server-wide SRV_S_USESRVLANG property for a thread.
SRV_T_USTATE	A string describing the current state of the thread.	Can be set.

SRV_T_BULKTYPE

Client applications can transfer three types of bulk data to Open Server applications: bulk copy data, text data, and image data. The SRV_T_BULKTYPE property is used to set or retrieve the type of bulk data transfer being initiated by a client.

Table 2-29 describes the legal values for the SRV_T_BULKTYPE thread property:

Table 2-29: Values for SRV_T_BULKTYPE

Value	Meaning
SRV_BULKLOAD	The client is preparing to transfer bulk copy data.
SRV_TEXTLOAD	The client is preparing to transfer text data.
SRV_IMAGELOAD	The client is preparing to transfer image data.
SRV_UNITEXTLOAD	The client is preparing to transfer unitext data.

Open Server cannot determine automatically the type of bulk data stream a client sends. The Open Server application must obtain this information and give it to Open Server in advance of the actual SRV_BULK event, using the srv_thread_props routine. The application then retrieves the data inside the SRV_BULK event handler once the actual bulk request has been made.

For more information on bulk copy, see the Open Client and Open Server *Common Libraries Reference Manual*. For more information on text and image processing, see "Text and image" on page 196.

SRV_T_CHARTYPE

A client application expects character data to be represented in a particular way. An Open Server application can retrieve the client's expected character data representation by calling srv_thread_props with property set to SRV_T_CHARTYPE and cmd set to CS_GET. The client will return the following values in *bufp:

Table 2-30: Character data representations

Value	Meaning
SRV_CHAR_ASCII	ASCII character format
SRV_CHAR_EBCDIC	EBCDIC character format
SRV_CHAR_UNKNOWN	Unknown character format

SRV_T_ENDPOINT

This example shows how to use SRV_T_ENDPOINT:

```
CS_INT ep;
/*
 ** Get the end point
 */

if(srv_thread_props(spp, CS_GET, SRV_T_ENDPOINT, (CS_VOID *)&ep,
     CS_SIZEOF(ep), (CS_INT *)NULL) == CS_FAIL)
{
    return(CS_FAIL);
```

```
}
```

SRV_T_EVENT

A thread executes a particular event handler at any one time. A thread can be said to be inside an event when executing the event handler associated with that event. An Open Server application can retrieve the event that a thread is in by calling srv_thread_props with property set to SRV_T_EVENT and cmd set to CS_GET. This procedure is useful if an application uses the same event handler code for multiple events.

Possible events include:

- SRV_ATTENTION
- SRV_BULK
- SRV_CONNECT
- SRV_CURSOR
- SRV_DISCONNECT
- SRV_DYNAMIC
- SRV_FULLPASSTHRU
- SRV_LANGUAGE
- SRV_MSG
- SRV_OPTION
- SRV_RPC
- SRV_START
- SRV_STOP
- User-defined events

For more information on events, see “Events” on page 92.

SRV_T_FLTTYPE

A client application expects floating point data to be represented in a particular way. An Open Server application can retrieve the client’s floating point representation by calling srv_thread_props with property set to SRV_T_FLTTYPE and cmd set to CS_GET. The client returns one of the following values in the address space to which bufp points.

- SRV_FLT_IEEE – IEEE floating point format.
- SRV_FLT_ND5000 – ND5000 floating point format.
- SRV_FLT_ VAX – VAX ‘D’ floating point format.
- SRV_FLT_UNKNOWN – unknown floating point format.

SRV_T_LOGINTYPE

An Open Server application can receive any of several types of thread login records during the login process. The SRV_T_LOGINTYPE property indicates the login type. The application can call srv_thread_props with property set to SRV_T_LOGINTYPE and cmd set to CS_GET to retrieve the login type, which is returned in the buffer to which bufp points. Table 2-31 describes each login type:

Table 2-31: Thread login types

Value	Login type
SRV_SITEHANDLER	A site handler login request from a remote server.
SRV_SUBCHANNEL	A site handler subchannel login from a remote server.
SRV_CLIENT	A login request from a client application.

SRV_T_MIGRATED

A Boolean property that indicates whether a connection is a new connection or a migrated connection. This read-only property is set to true when the client is migrating or has migrated to the server. This sample code retrieves the value of SRV_T_MIGRATED:

```
CS_RETCODE ret;
CS_BOOL migrated;
status = srv_thread_props(sp, CS_GET, SRV_T_MIGRATED,
   &migrated, sizeof (migrated), NULL);
```

See “Connection migration” on page 40 and “SRV_T_MIGRATED” on page 159 for more details.

SRV_T_MIGRATE_STATE

SRV_T_MIGRATE_STATE indicates the migration state of the client. It is a read-only property that any thread can access. The possible migration states are:

State	Value	Description
SRV_MIG_NONE	0	There is no migration in progress.
SRV_MIG_REQUESTED	1	A migration has been requested by the server.
SRV_MIG_READY	2	The client has received the request and is ready to migrate.
SRV_MIG_MIGRATING	3	The client is now migrating to the specified server.
SRV_MIG_CANCELLED	4	The migration request has been cancelled.
SRV_MIG_FAILED	5	The client failed to migrate.

SRV_MIG_STATE is an enumerated datatype that has been added to model the SRV_T_MIGRATE_STATE property. SRV_MIG_STATE is declared as:

```
typedef enum
{
   SRV_MIG_NONE,
   SRV_MIG_REQUESTED,
   SRV_MIG_READY,
   SRV_MIG_MIGRATING,
   SRV_MIG_CANCELLED,
   SRV_MIG_FAILED
} SRV_MIG_STATE;
```

This sample code shows how you can retrieve SRV_T_MIGRATE_STATE values; in case of a successful migration, the client exits and the SRV_DISCONNECT event handler is called with a SRV_MIG_MIGRATING status:

```
CS_RETCODE ret;
SRV_MIG_STATE migration_state;
ret = srv_thread_props(sp, CS_GET, SRV_T_MIGRATE_STATE,
   &migration_state, sizeof (migration_state), NULL);
if (ret != CS_SUCCEED)
{
...
}
```

See “Connection migration” on page 40 and “SRV_T_MIGRATE_STATE” on page 159 for more details.

SRV_T_RMTPWDS

An application uses the SRV_T_RMTPWDS property to obtain name/password pairs for a remote server. The pairs are stored in a SRV_T_RMTPWD structure which is defined as follows:

```
typedef struct srv_rmtpwd
```

```
{
    CS_INT servnamelen;
    CS_BYTEservname[CS_MAX_NAME];
    CS_INTpwdlen;
    CS_BYTEpwd[CS_MAX_NAME];
} SRV_RMTPWD;
```

SRV_T_SESSIONID

The SRV_T_SESSIONID is a thread property that retrieves the session ID that the client sends to Open Server. An Open Server application can also set the SRV_T_SESSIONID property using the srv_thread_props() function, given that:

- The srv_thread_props(CS_SET, SRV_T_SESSIONID) call is made inside the SRV_CONNECT event handler and,
- The client supports connection migration or high availability.

This sample code sets the SRV_T_SESSIONID property:

```
CS_RETCODE ret;
CS_SESSIONID hasessionid;
ret = srv_thread_props(sp, CS_SET, SRV_T_SESSIONID,
   hasessionid, sizeof(hasessionid), NULL);
```

Note In version 15.0 ESD#14 and earlier, for HA-failover, you must program an srv_negotiate() sequence to send the session ID to the client.

SRV_T_TDSVERSION

During the client login process, Open Server negotiates with the client application to agree on a TDS version for all threads. The SRV_S_TDSVERSION property value determines Open Server's starting point. The client agrees to communicate at or below this starting point. See "Thread properties" on page 148 for details on the SRV_S_TDSVERSION property. Later on in the login process, the Open Server application can renegotiate the TDS version for a particular thread, using the SRV_T_TDSVERSION property.

Table 2-32 describes the legal values for this property:

Table 2-32: Values for SRV_T_TDSVERSION

SRV_T_TDSVERSION value	Meaning
SRV_TDSNONE	Unknown version of TDS
SRV_TDS_4.0	Negotiation starts at TDS 4.0
SRV_TDS_4_0_2	Negotiation starts at TDS 4.0.2
SRV_TDS_4_2	Negotiation starts at TDS 4.2
SRV_TDS_4_6	Negotiation starts at TDS 4.6
SRV_TDS_4_9_5	Negotiation starts at TDS 4.9.5
SRV_TDS_5_0	Negotiation starts at TDS 5.0

SRV_T_TYPE

There are several types of Open Server threads. The SRV_T_TYPE thread property indicates the type of thread. An application can retrieve the thread's type by calling srv_thread_props with property set to SRV_T_TYPE and cmd set to CS_GET.

Table 2-33 identifies the legal thread types:

Table 2-33: Thread types

Value	Thread type
SRV_TCLIENT	A client thread
SRV_TSITE	A site handler thread
SRV_TSUBPROC	A remote server connection over a site handler thread
SRV_TSERVICE	A service thread

See "Multithread programming" on page 109 for more information about thread types.

Registered procedures

A registered procedure is a piece of code identified by a name. When an application registers a procedure, it maps the procedure name to a routine, so that when Open Server detects this procedure name in an incoming RPC data stream, it can call a specific routine immediately without raising a SRV_RPC event.

When an Open Server receives an RPC, Open Server looks up the procedure name in the list of registered procedures. If the name is registered, the runtime system executes any existing routine associated with the registered procedure. If the procedure name is not found in the list of registered procedures, Open Server calls the SRV_RPC event handler.

Standard remote procedure calls

An Open Server application processes a conventional RPC from within the application's SRV_RPC event handler. The handler code must parse the RPC data stream and retrieve the RPC name, the number of parameters, the parameter formats, and the parameter values in the process. The handler can then take actions based on these values. A SRV_RPC event handler must be coded for all possible RPCs the application programmer anticipates will come over the network.

Advantages of registered procedures

Registered procedures simplify RPC handling in an Open Server application for these reasons:

- Registered procedures consolidate code in one place. They are executable objects that an Open Server application can call from other event handlers in addition to the SRV_RPC event handler.
- Registered procedures can be created at any time when the server is running, through Server-Library calls or external Client-Library or DB-Library calls. The SRV_RPC event handler, by contrast, must be coded in advance of starting up the server.
- Registered procedures provide automatic datatype checking and require no parsing on the part of the Open Server application code.
- Clients can request notification when a registered procedure executes. The "notification" consists of:
    - The name of the registered procedure
    - The parameter values associated with this execution of the registered procedure
- The notification request can be issued internally with Server-Library calls or externally with Client-Library or DB-Library calls.

- Clients can request a list of registered procedures or a list of the procedures for which they have requested notifications.

Notification procedures

Without any programmer-supplied code, an Open Server application allows Client-Library or DB-Library clients to create registered procedures, execute them, and receive notification when they execute.

Registered procedures are not required to have an executable routine in the Open Server application. In fact, registered procedures created by DB-Library or Client-Library calls *cannot* call a routine in Open Server. A registered procedure that has no executable routine associated with it is called a "notification procedure" because its sole purpose is to notify clients watching for it to execute.

Client applications communicate with each other through any Open Server application by using notification procedures.

Although you do not need to write any code to enable this feature, you may want to install a callback handler to disable or regulate the use of registered procedures. See "Using callback handlers with registered procedures" on page 167, for details.

Creating registered procedures

Open Server applications can create both standard registered procedures and notification procedures. Client-Library and DB-Library applications can create notification procedures. For information on how to create registered procedures using Client- Library routines, see the Open Client *Client-Library/C Reference Manual*.

The mechanics of registered procedures

This section provides information on how to create and execute registered procedures from within an Open Server application.

Registering procedures

Registering a procedure through Open Server calls requires these steps:

1 Call srv_regdefine to define the procedure name and map the name to the function to be called when the procedure is executed.

2 Call srv_regparam to describe the parameter or parameters for the procedure being defined.

3 Call srv_regcreate to complete the registration of a procedure.

4 Call srv_regdrop to unregister a procedure.

Executing registered procedures

Open Server executes registered procedures in response to a client or remote Adaptive Server RPC, if the RPC has been registered. However, an Open Server application can also explicitly execute a registered procedure, instead of executing it in response to an RPC. For example, an application can synchronize the activity of multiple clients by executing a particular notification procedure at a particular point in the application.

Explicitly executing a registered procedure also requires several steps. They are as follows:

1 Call srv_reginit to begin executing a registered procedure. This routine specifies the name of the registered procedure to be executed. The Open Server application also uses this routine to determine whether one or all of the client threads on the notification list will be notified.

2 Call srv_regparam to supply the parameter data for the execution.

3 Call srv_regexec to actually execute the registered procedure.

Maintaining lists

An Open Server application maintains lists of all registered procedures and which clients to notify when a particular registered procedure executes. This notification happens automatically. The following routines pertain to list maintenance:

- srv_reglist – returns a list of all the procedures registered in the Open Server application.
- srv_regwatchlist – returns a list of all registered procedures for which the named client thread indicates notification requests are pending.
- srv_regwatch – adds a thread to the notification list for a registered procedure.

- srv_regnowatch – removes a client from the notification list for a specified registered procedure.
- srv_reglistfree – frees a SRV_PROCLIST structure previously allocated by srv_reglist or srv_regwatchlist.

System registered procedures

Every Open Server application contains built-in registered procedures, called *system registered procedures*. The runtime system creates them when the server starts up. The system registered procedures are described in Chapter 4, “System Registered Procedures” Some of these procedures are useful for administering an Open Server application interactively. For example, you can use sp_who and sp_ps to list active server processes and sp_terminate to destroy a process.

Client applications can execute system registered procedures to perform the following operations:

- Get a list of registered procedures
- Execute a registered procedure
- Request notification of a registered procedure’s execution
- Get a list of notification requests

Most system registered procedures map to an equivalent Open Server routine. An Open Server application and a client can request the same kind of information through distinct routines.

Table 2-34 matches each system registered procedure to the corresponding Server-Library routine, if applicable:

Table 2-34: System registered procedures and corresponding Server-Library routines

System registered procedure	Server-Library routine
sp_ps	N/A
sp_regcreate	srv_regcreate/srv_regdefine
sp_regdrop	srv_regdrop
sp_reglist	srv_reglist
sp_regnowatch	srv_regnowatch
sp_regwatch	srv_regwatch
sp_regwatchlist	srv_regwatchlist
sp_serverinfo	N/A
sp_terminate	srv_termproc
sp_who	N/A

Using callback handlers with registered procedures

As noted in Table 2-34, several of the built-in registered procedures parallel Server-Library and DB-Library routines that create, delete, and execute registered procedures. These procedures make it possible to implement a security system for registered procedures by installing a callback handler that executes whenever a registered procedure is about to execute. When a client application executes a system registered procedure or one of the parallel Client-Library or DB-Library routines, the callback handler executes. If it returns SRV_S_INHIBIT, the registered procedure does not execute.

For example, to prevent clients other than “sa” from executing a procedure named “reinitialize”, the registered procedure callback handler could contain the following code:

```
/*
 **   Stop users other than "sa" from executing the "reinitialize"
 **   registered procedure.
 **
 ** Parameters:
 **    spp - Handle to the current client connection.
 **
 ** Returns:
 **    CS_TRUE  Allow the user to execute
 **    CS_FALSE Disallow execution.
 */
 CS_BOOL    rpc_permission(spp)
 SRVPROC    *spp;
```

```
{
      CS_INT ulen;       /* User name length  */
      CS_INT rlen;       /* RPC name length   */
      CS_CHAR *rname;    /* Pointer to the RPC name */
      CS_CHAR user[256]; /* Buffer for the user name */

      /*
      ** Get the name of the rpc command
      */
      if ((rname = srv_rpcname(spp, &rlen)) == (CS_CHAR *)NULL)
      {
      return (CS_FALSE);
      }

      /*
      ** Get the user name.
      */
      if (srv_thread_props(spp, CS_GET, SRV_T_USER,
      (CS_VOID *)user,CS_SIZEOF(user), &ulen) == CS_FAIL)
      {
      return (CS_FALSE);
      }

      /*
      ** If either the user name or the rpc name is NULL,
      ** indicate an error.
      */
      if (rlen <= 0 || ulen <= 0)
      {
            error ("API error");
            return (CS_FALSE);
      }

      /* Null terminate the user name buffer */
      user[ulen] == '\0';

      /*
      ** Compare the RPC name and User name for permission.
      */
      if ((strcmp(rname, "reinitialize") == 0) &&
       (strcmp(user, "sa") == 0))
      {
      return (CS_TRUE);
      }

      return (CS_FALSE);
```

```
}
```

Example

The sample *regproc.c* illustrates an Open Server application's use of registered procedures.

Remote procedure calls

A remote procedure call, or *RPC*, is a mechanism by which a client application communicates with an Open Server application. Typically, the client issues the RPC to obtain information from the Open Server application. An RPC consists of a name and often, but not always, parameters. For example, a department store application could return a customer's name and address in response to an RPC called get_cust. This RPC could take one parameter, a customer ID number.

When a client sends an RPC, Open Server checks to see whether the RPC is *registered*. A *registered procedure* is a special kind of RPC that Open Server recognizes and executes directly without calling an application's SRV_RPC event handler. For more information on registered procedures, see "Registered procedures" on page 162.

If the RPC is not registered, Open Server triggers a SRV_RPC event. From within the SRV_RPC event handler, the application can retrieve the RPC's name, and parameters if any, and respond appropriately. The event handler is coded to verify the names of all possible RPCs the client could send and the number of parameters each uses. The handler includes code for responding to each RPC and returns the error information to the client if it does not recognize the RPC.

From within its SRV_RPC event handler, the application should perform the following steps:

1 Call srv_rpcname to retrieve the RPC name. (An application can also choose to retrieve the RPC number, owner, and associated database, using srv_rpcnumber, srv_rpcowner, and srv_rpcdb, respectively.) If no RPC by that name exists, or the number, owner, or database information are invalid, the application returns error information through srv_sendinfo.

2 Verify that the appropriate number of parameters were sent by calling srv_numparams. If any of the parameter information is invalid, return error information through srv_sendinfo.

3 Process the parameters by calling srv_descfmt, srv_bind, and srv_xferdata. For details, see "Processing parameter and row data" on page 134.

4 Return any data the client expects by calling srv_descfmt, srv_bind, and srv_xferdata. For details, see "Processing parameter and row data" on page 134.

RPC parameters are passed either by name or by position. If the RPC is invoked with some parameters passed by name and some parameters passed positionally, an error will result.

An application could register all its procedures and use the SRV_RPC event handler to trap errors. Open Server would only call the SRV_RPC event handler if the client sent an unregistered and therefore invalid RPC. The SRV_RPC event handler, then, would use srv_sendinfo to inform the client that it had issued an invalid RPC.

Example

The sample, *regproc.c*, illustrates remote procedure calls.

Security services

Security services allow Open Server applications to use third-party distributed security to authenticate users and protect data as it is transmitted between clients and servers.

Check your Open Client and Open Server *Configuration Guide* for the distributed security service providers that are available on your platform.

The security services available from a particular provider are referred to as a *security mechanism*. An Open Server application can support multiple security mechanisms, depending on availability. Open Server applications select security mechanisms on a per client-server dialog basis (based on client connection requests).

You can use Open Server's security services to:

- Access *credentials* that are established on a system.

  Credentials are the data that is transferred between peers (clients and servers) to establish the identity of a peer.

- Communicate the requested security mechanism during dialog establishment.

- Establish a security session with a remote client or server.

  The security services are negotiated during security session establishment. Security sessions map directly to client dialogs.

- Communicate opaque *tokens* over a dialog to allow a security mechanism to communicate with its peer component. These tokens are sent during session establishment, and, if required, can be used for per-packet security services.

  A token is a bit string generated by the security mechanism for security information exchange between peers. A token may be cryptographically protected.

- Bind channel identification information to a security session.

- Digitally sign tokens to assure the origin of tokens.

Security service properties

Network security services can be split into three broad categories:

- Login authentication services
- Per-packet security services
- Secure Sockets Layer (SSL) encryption

Login authentication services

The fundamental security service is *login authentication*, or confirming that users are who they say they are. Login authentication involves user names and passwords. Users identify themselves by their user name, then supply their password as proof of their identity.

In Sybase applications, each connection between a client and a server has one user name associated with it. If the application uses a security mechanism, then Sybase uses the mechanism to authenticate this user name when the connection is established. The advantage of this service is that the user name/password pairs can be managed in a central repository, and not in the system catalogs of individual servers.

When an application requests to connect to a server using network-based authentication, Client-Library queries the connection's security mechanism to confirm that the given user name represents an authenticated user. This means that users do not have to supply a password to connect to the server. Instead, users authenticate themselves to the network security system before the connection attempt is made. When connecting, Client-Library obtains a *credential token* from the security mechanism and sends it to the server in lieu of a password. The server then passes the token to the security mechanism again to confirm that the user name has been authenticated.

The following properties are related to login authentication:

Table 2-35: Properties that control login authentication

Property	Description
CS_USERNAME	Specifies the user name to connect with.
CS_SEC_ NETWORKAUTH	Enables network-based user authentication.
CS_SEC_ CREDTIMEOUT	Tells whether the user's credentials have expired.
CS_SEC_ SESSTIMEOUT	Tells whether the session between the client and the server has expired.
CS_SEC_ MUTUALAUTH	Set by client applications to request that the server authenticate itself to the client.
CS_SEC_ DELEGATION	Set by client applications to permit a gateway server to connect to a remote server by using the client's delegated credential token.
CS_SEC_ CREDENTIALS	Used by gateway applications to forward a delegated credential token from the gateway's client to a remote server.

Network-authentication is supported by all security mechanisms. Credential and session timeouts are supported by some but not all security mechanisms. See the Open Client and Open Server *Configuration Guide* for information on which services are supported by which security mechanisms.

See the Open Client *Client-Library/C Reference Manual* for more information about these security services, and about use of security services in client applications.

Per-packet security services

In some environments, distributed applications have to deal with the fact that the network is not physically secure. For example, unauthorized parties can listen to a dialog by attaching analyzers to a physical line or capturing wireless transmissions.

In these environments, use applications protection and authentication of transmitted data to assure a secure dialog.

The following properties control the use of the various per-packet services:

Table 2-36: Data authentication properties

Property	Description
CS_SEC_ CONFIDENTIALITY	Enables data confidentiality service. Data confidentiality encrypts all transmitted data and assures that strangers cannot understand in-transit data.
CS_SEC_ INTEGRITY	Enables data integrity service. Data integrity service assures that attempts to tamper with in-transit data are detected.
CS_SEC_ DATAORIGIN	Enables data origin stamping. Data origin stamping assures that received data was really sent by the client or the server.
CS_SEC_ DETECTREPLAY	Enables replay detection service. Replay detection assures that attempts by strangers to replay captured transmissions are detected.
CS_SEC_ DETECTSEQ	Enables sequence verification service. Sequence verification detects transmissions that arrive in a different order than they were sent.
CS_SEC_ CHANBIND	Enables channel binding service. Channel binding stamps each transmission with an encrypted description of the client's and server's addresses.

Note Applications that use the services described in this section incur a per-packet overhead on all communication between the client and the server. Data authentication services should not be used unless application security is more important than application performance.

All per-packet services will perform one or both of the operations below for each TDS packet to be sent over a connection:

- Encryption of the packet's contents

- Computation of a digital signature that encodes the packet contents as well as other needed information.

If an application selects multiple per-packet services, each operation is performed only once per packet. For example, if the application selects the data confidentiality, sequence verification, data integrity, and channel binding services, then each packet is encrypted and accompanied by a digital signature that encodes the packet contents, packet sequence information, and a network channel identifier.

See the Open Client *Client-Library/C Reference Manual* for more information about these security services, and about use of security services in client applications.

SSL overview

SSL is an industry standard for sending wire- or socket-level encrypted data over client-to-server and server-to-server connections. Before the SSL connection is established, the server and the client exchange a series of I/O round trips to negotiate and agree upon a secure encrypted session. This is called the SSL handshake.

SSL handshake

When a client application requests a connection, the SSL-enabled server presents its certificate to prove its identity before data is transmitted. Essentially, the SSL handshake consists of the following steps:

- The client sends a connection request to the server. The request includes the SSL (or Transport Layer Security, TLS) options that the client supports.
- The server returns its certificate and a list of supported CipherSuites, which includes SSL/TLS support options, the algorithms used for key exchange, and digital signatures.
- A secure, encrypted session is established when both client and server have agreed upon a CipherSuite.

For more specific information about the SSL handshake and the SSL/TLS protocol, see the Internet Engineering Task Force Web site at http://www.ietf.org.

SSL in Open Client and Open Server

SSL provides several levels of security.

- When establishing a connection to an SSL-enabled server, the server authenticates itself—proves that it is the server you intended to contact—and an encrypted SSL session begins before any data is transmitted.
- Once the SSL session is established, user name and password are transmitted over a secure, encrypted connection.
- A comparison of the server certificate's digital signature can determine if any information received from the server was modified in transit.

SSL filter

When establishing a connection to an SSL-enabled Adaptive Server, the SSL security mechanism is specified as a filter on the master and query lines in the interfaces file (*sql.ini* on Windows). SSL is used as an Open Client and Open Server protocol layer that sits on top of the TCP/IP connection.

The SSL filter is different from other security mechanisms, such as DCE and Kerberos, which are defined with SECHMECH (security mechanism) lines in the interfaces file (*sql.ini* on Windows). The master and query lines determine the security protocols that are enforced for the connection.

For example, a typical interfaces file on a UNIX machine using SSL looks like this:

```
[SERVER]

query tcp ether hostname, port ssl
master tcp ether hostname, port ssl
```

A typical *sql.ini* file on Windows using SSL looks like this:

```
[SERVER]

query=TCP,hostname, port, ssl
master=TCP,hostname, port, ssl
```

where *hostname* is the name of the server to which the client is connecting and *port* is the port number of the host machine. All connection attempts to a master or query entry in the interfaces file with an SSL filter must support the SSL protocol. A server can be configured to accept SSL connections and have other connections that accept plain text (unencrypted data), or use other security mechanisms.

For example, an Adaptive Server interfaces file on UNIX that supports both SSL-based connections and plain-text connections looks like:

```
SYBSRV1
master tcp ether hostname 2748 ssl
```

```
query tcp ether hostname 2748 ssl
master tcp ether hostname 2749
```

In this examples, the SSL security service is specified on port number 2748. On SYBSRV1, Adaptive Server listens for clear text on port number 2749, which is without any security mechanism or security filter.

Validating the server by its certificate

Any Open Client and Open Server connection to an SSL-enabled server requires that the server have a certificate file, which consists of the server's certificate and an encrypted private key. The certificate must also be digitally signed by a CA.

Open Client applications establish a socket connection to Adaptive Server similarly to the way that existing client connections are established. Before any user data is transmitted, an SSL handshake occurs on the socket when the network transport-level connect call completes on the client side and the accept call completes on the server side.

To make a successful connection to an SSL-enabled server:

- The SSL-enabled server must present its certificate when the client application makes a connection request.
- The client application must recognize the CA that signed the certificate. A list of all "trusted" CAs is in the trusted roots file. See "The trusted roots file" on page 178.
- For connections to SSL-enabled servers, the default behavior is to compare the common name in the server's certificate with the server name in the interfaces file. In Shared Disk Cluster (SDC) environment, a client may specify the SSL certificate common name independent of the server name or the SDC instance name. For information about common name validation in an SDC environment see "Common name validation in an SDC environment" on page 177.

When establishing a connection to an SSL-enabled Adaptive Server, Adaptive Server loads its own encoded certificates file at start-up from:

UNIX – *$SYBASE/$SYBASE_ASE/certificates/servername.crt*

Windows – *%SYBASE%\%SYBASE_ASE%\certificates\servername.crt*

where *servername* is the name of the Adaptive Server as specified on the command line when starting the server with the `-S` flag or from the server's environment variable $DSLISTEN.

Other types of servers may store their certificate in a different location. See the vendor-supplied documentation for the location of your server's certificate.

Common name validation in an SDC environment

The default behavior for SSL validation in Open Client and Open Server is to compare the common name in the server's certificate with the server name specified by ct_connect(). In a Shared Disk Cluster (SDC) environment, a client may specify the SSL certificate common name independent of the server name or the SDC instance name. A client may connect to an SDC by its cluster name—which represents multiple server instances—or to a specific server instance.

Because the client can use the transport address to specify the common name used in the certificate validation, the ASE SSL certificate common name can be different from the server or cluster name. The transport address can be specified in one of the directory services like the *interfaces* file, LDAP or NT registry, or through the connection property CS_SERVERADDR.

Syntax for UNIX

This is the syntax of the server entries for the SSL-enabled ASE and cluster for UNIX:

```
CLUSTERSSL
query tcp ether hostname1 5000 ssl="CN=name1"
query tcp ether hostname2 5000 ssl="CN=name2"
query tcp ether hostname3 5000 ssl="CN=name3"
query tcp ether hostname4 5000 ssl="CN=name4"

ASESSL1
master tcp ether hostname1 5000 ssl="CN=name1"
query tcp ether hostname1 5000 ssl="CN=name1"

ASESSL2
master tcp ether hostname2 5000 ssl="CN=name2"
query tcp ether hostname2 5000 ssl="CN=name2"

ASESSL3
master tcp ether hostname3 5000 ssl="CN=name3"
query tcp ether hostname3 5000 ssl="CN=name3"

ASESSL4
master tcp ether hostname1 5000 ssl="CN=name4"
query tcp ether hostname1 5000 ssl="CN=name4"
```

Syntax for Windows

This is the syntax of the server entries for the SSL-enabled ASE and cluster for Windows:

```
[CLUSTERSSL]
query=tcp,hostname1,5000, ssl="CN=name1"
query=tcp,hostname2,5000, ssl="CN=name2"
query=tcp,hostname3,5000, ssl="CN=name3"
query=tcp,hostname4,5000, ssl="CN=name4"

[ASESSL1]
master=tcp,hostname1,5000, ssl="CN=name1"
query=tcp,hostname1,5000, ssl="CN=name1"

[ASESSL2]
master=tcp,hostname2,5000, ssl="CN=name2"
query=tcp,hostname2,5000, ssl="CN=name2"

[ASESSL3]
master=tcp,hostname3,5000, ssl="CN=name3"
query=tcp,hostname3,5000, ssl"CN=name3"

[ASESSL4]
master=tcp,hostname4,5000, ssl="CN=name4"
query=tcp,hostname4,5000, ssl="CN=name4"
```

The trusted roots file

The list of known and trusted CAs is maintained in the trusted roots file. The trusted roots file is similar in format to a certificate file, except that it contains certificates for CAs known to the entity (client applications, servers, network resources, and so on). The System Security Officer adds and deletes CAs using a standard ASCII-text editor.

The trusted roots file for Open Client and Open Server is located in:

UNIX – *$SYBASE/$SYBASE_OCS/config/trusted.txt*

Windows – *%SYBASE%\%SYBASE_OCS%\ini\trusted.txt*

Currently, the recognized CAs are Thawte, Entrust, Baltimore, VeriSign and RSA.

By default, Adaptive Server stores its own trusted roots file in:

UNIX – *$SYBASE/$SYBASE_ASE/certificates/servername.txt*

Windows – *%SYBASE%\%SYBASE_ASE%\certificates\servername.txt*

Both Open Client and Open Server allow you to specify an alternate location for the trusted roots file:

- Open Client:

```
ct_con_props (connection, CS_SET, CS_PROP_SSL_CA,
"$SYBASE/config/trusted.txt", CS_NULLTERM, NULL);
```

where *$SYBASE* is the installation directory. CS_PROP_SSL_CA can be set at the context level using ct_config(), or at the connection level using ct_con_props().

- Open Server:

```
srv_props (context, CS_SET, SRV_S_CERT_AUTH,
"$SYBASE/config/trusted.txt", CS_NULLTERM, NULL);
```

where *$SYBASE* is the installation directory.

bcp and isql utilities also allow you to specify an alternative location for the trusted roots file.The parameter -x is included in the syntax, allowing you to specify an alternative location for the *trusted.txt* file.

For a description of SSL and public-key cryptography, see the Open Client *Client-Library Reference Manual*.

How do security services work with Open Server?

To initiate security services the client sends an *object identifier*, which maps to a security mechanism, to the server when establishing a dialog. The server maps the object identifier to its own local name for the security mechanism. If the server does not support the requested security mechanism or does not support security sessions at all, the dialog request fails and Open Server returns an error.

Use of object identifiers allows local names for a security mechanism to be different on clients and servers. System administrators and application programmers can then develop their own separate local naming conventions for security mechanisms. See “Object identifiers” on page 182 for more information about object identifiers.

Server-Library allows you to specify the *principal name* to be used when acquiring credentials. This principal name is the name by which the Open Server application is known to the security service provider. You can use the SRV_S_SEC_PRINCIPAL server property with the srv_props function to set the application’s principal name.

If not set, the principal name defaults to the Open Server application’s network name, which is generally specified through srv_init.

Open Server uses credentials when establishing security sessions with clients.

The login name of the client is obtained from the security session; whatever is specified in the login record is ignored.

See the Open Client *Client-Library/C Reference Manual* for information on the client's role in using security services.

Steps involved in a Client/Server dialog using security services

Open Server performs the following steps when a client initiates a dialog using security services:

1 Establishes a transport connection with the client.

2 Receives the client's login record and any opaque security tokens and responds with any necessary opaque tokens to the client.

3 Establishes a security session when the security message handshake succeeds.

When an Open Server application receives information from a client, it performs these steps:

1 Processes any security messages—for instance a cryptographic signature—associated with the response received from the client. (A cryptographic signature ensures the integrity of the message).

2 Based on the security services supported on the security session, calls the appropriate routines—for example, to verify the signature.

3 Processes the TDS data stream as normal.

Open Server sends a response to the client in the following steps:

1 Checks for credential or security session expiration. If an expiration is detected, Open Server performs error processing.

2 Based on the security services supported on this dialog, calls the appropriate routines—for example, to generate a cryptographic signature for the response.

3 Generates the required TDS to identify any per-packet security services.

A security session is terminated when the associated client dialog terminates. Termination may occur because of a normal client logout or error conditions.

Using security mechanisms with Open Server applications

This section describes the changes you need to make to use third-party security with an Open Server application. These changes include adding:

- An entry for each security-mechanism-to-driver mapping in the *libtcl.cfg* file.
- An entry mapping the local name of each security mechanism to a globally unique object identifier, in the global object identification file, *objectid.dat*.
- An entry in the interfaces file for each server using a third-party security mechanism, specifying all of the security mechanisms supported by a server.

Security drivers

Sybase provides *security drivers* that allow Client-Library and Server-Library applications to take advantage of an installed network security system. Client-Library and Server-Library provide a generic interface for implementing secure applications; each Sybase security driver maps this generic interface to the security provider's interface.

Security drivers are dynamically loadable, and support one or more security mechanisms.

The drivers for each of the currently supported security providers are:

- *libsybsdce*

  For DCE Security Services.

- *libsybsmssp*

  For Microsoft NT SSPI.

libtcl.cfg configuration file

The *libtcl.cfg* configuration file maps the local name of the security mechanism to the security driver required to support that mechanism. The *libtcl.cfg* file is located in the *$SYBASE/$SYBASE_OCS/config* directory or in the path specified by the CS_LIBTCL_CFG context property. See the Open Client and Open Server *Programmer's Supplement* for your platform for its exact location.

There must be an entry for each security driver in the *libtcl.cfg* file. Each driver may support one or more security mechanism. If a driver supports more than one security mechanism, it requires an entry for each security mechanism in the *libtcl.cfg* file.

The format of the file is as follows:

```
[SECURITY]
 local-name-of-security-mechanism = path-to-the-driver init-string
```

where:

- *path-to-the-driver* – is the fully qualified pathname to the object file.
- *init-string* – is an argument list which varies according to each driver, of the general form: *token = value, token = value, ...*

For example, on a UNIX platform:

```
[SECURITY]
 csfkrb5=libsybskrb.so secbase=@MYREALM libgss=/krb5/lib/libgss.so
```

The first entry in the *libtcl.cfg* file is the default security mechanism. Open Server uses the default security mechanism when an application requests security services, but it does not set a security mechanism.

See the Open Client and Open Server *Configuration Guide* for your platform, for more information on adding entries to *libtcl.cfg*.

Object identifiers

Each security mechanism has an object identifier associated with it. The globally unique object identifier maps to the local name for a security mechanism in the global object identification file, *Objectid.dat*. This provides a consistent and flexible way to communicate security mechanism names between clients and servers. The *Objectid.dat* file is located in the *$SYBASE/config* directory.

The format for the global identification file is:

```
[Object Class]
      Object_Identifier Object_Name_List
```

For a security mechanism the entry is as follows:

Object Class – is “secmech.”

Object_Identifier – is a sequence of non-negative integer values separated by dots. The object identifier is based on a naming tree defined by the international standards bodies CCITT and ISO. An example of an object identifier from the *sybase* root for the DCE security driver would be 897.4.6.1.

Object_Name_List – is a comma-delimited list of local security mechanism names.

For example:

```
[secmech]
        1.3.6.1.4.1.897.4.6.3 = NTLM
```

Changes to the interfaces file

The format of the interfaces file has been expanded to allow specification of the security mechanisms supported by a server. The format is:

```
SERVERNAME
      query tcp sun-ether joyce 2901
      master tcp sun-ether joyce 2901
      secmech mechanism1, mechanism2,..., mechanismN
```

The *secmech* identifier lists all of the security mechanisms supported by a server, and applies under the following conditions:

- This line is optional and is only used if the server is not using a Sybase-specific security mechanism.
- If there is no secmech entry for a server in the interfaces file, the server supports all the security mechanisms specified in the *libtcl.cfg* secmech entries.
- If there is a secmech entry for a server in the interfaces file, but no security mechanisms are specified, then the server does not support any security mechanisms.

mechanism1, *mechanism2*,...*mechanismN* are the object identifiers of the security mechanisms supported by the server. You can specify multiple security mechanisms using a comma (,) separator. See “Object identifiers” on page 182 for more information on object identifiers.

Changes to the interfaces file: the SSL filter

The SSL filter is different from other security mechanisms, such as DCE and Kerberos, which are defined with SECMECH (security mechanism) lines in the interfaces file (*sql.ini* on Windows). The master and query lines determine the security protocols that are enforced for the connection.

For example, a typical interfaces file on a UNIX machine using SSL looks like this:

```
[SERVER]
query tcp ether hostname port ssl
master tcp ether hostname port ssl
```

A typical *sql.ini* file on Windows using SSL looks like this:

```
[SERVER]

query=TCP,hostname,port, ssl
master=TCP,hostname,port, ssl
```

where *hostname* is the name of the server to which the client is connecting and *port* is the port number of the host machine. All connection attempts to a master or query entry in the interfaces file with an SSL filter must support the SSL protocol. A server can be configured to accept SSL connections and have other connections that accept plain text (unencrypted data), or use other security mechanisms.

Determining which security services are active

To determine which security services are active on a client-server dialog, use srv_thread_props to retrieve the value of the following thread properties:

- SRV_T_SEC_CHANBIND
- SRV_T_SEC_CONFIDENTIALITY
- SRV_T_SEC_DATAORIGIN
- SRV_T_SEC_DELEGATION
- SRV_T_SEC_DETECTREPLAY
- SRV_T_SEC_DETECTSEQ
- SRV_T_SEC_INTEGRITY
- SRV_T_SEC_MUTUALAUTH
- SRV_T_SEC_NETWORKAUTH

See Table 2-28 on page 149 for descriptions of these thread properties.

Scenarios for using security services with Open Server applications

This section describes how you might use security services with various Open Server application configurations. It discusses the following situations:

- Simple Open Server application using a security session.
- Gateway Open Server application with separate security sessions.
- Gateway Open Server application with separate security sessions using delegation.
- Full passthrough gateway Open Server application with direct security session.

Simple application using a security session

In the simplest configuration, the client establishes a dialog using authentication services provided by the security mechanism. Open Server performs the login negotiation before the connection event handler is called. After the connection handler issues a srv_senddone(SRV_DONE_FINAL), Open Server sends a login acknowledgment with status “success” to the client.

You are not required to install a connection handler for this configuration; the default connection handler is sufficient. If you do install a connection handler, the must at least send a srv_senddone(SRV_DONE_FINAL), as shown in this example:

```
CS_RETCODE CS_PUBLIC connect_handler(spp)
 SRV_PROC *spp;
 {

   .......

   /*
    ** You do not need to test this srv_senddone's return value
    ** since Open Server will kill this thread if this call fails.
    */
    (CS_VOID)srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                    (CS_INT)0);
    return(CS_SUCCEED);
 }
```

Gateway application with separate security sessions

In the scenario shown in the code below, the Open Server application acts as a gateway between the client and another server. The network identity used to establish the security session between the client and the gateway application may be different from that used to establish the security session between the gateway and the remote server.

The gateway application completes the login security negotiation with its client, pending the final login acknowledgment, before calling the connection handler. The connection handler needs to initiate a security-session-based login to the remote server using Client-Library calls before sending a srv_senddone(SRV_DONE_FINAL) to the client to complete the login. An example connection handler follows:

```
CS_RETCODE CS_PUBLIC connect_handler(spp)
 SRV_PROC *spp;
 {
    CS_CONNECTION   *conn;    /* the connection handle */
    CS_BOOL         trueval = CS_TRUE;
    CS_INT          outlen;

  ......

  allocate and set user data in spp...

  ......

  /* Allocate a connection handle */
    if (ct_con_alloc(Context, &(userdata->conn)) == CS_FAIL)
    {
       clean up and report error...
      return(CS_FAIL);

   }

   ......

   conn = userdata->conn;
    /*
    ** Initiate security session based login with the remote
    ** server. The user name used here may be the same as the
    ** client user name or different
    */
    if (ct_con_props(conn, CS_SET, CS_USERNAME,
        (CS_VOID*)Username, STRLEN(Username), (CS_INT*)NULL)
        == CS_FAIL)
    {
       handle failure...
   }
```

```
   /*
    ** Set the desired security mechanism(s) or use the default
    ** security mechanism.
    */
    if (ct_con_props(conn, CS_SET, CS_SEC_MECHANISM,
          (CS_VOID*)Mechanismname, STRLEN(Mechanismname),
          (CS_INT*)NULL) == CS_FAIL)
    {
       handle failure...
   }

   /* Set the security service-network authentication */
    if (ct_con_props(conn, CS_SET, CS_SEC_NETWORKAUTH,
          (CS_VOID*)&trueval, CS_SIZEOF(CS_BOOL), (CS_INT*)NULL)
          == CS_FAIL)
    {
       handle failure...
   }

   set other security services if required
   get and set the user's application name, response capabilities
   set the locale and other login properties
   /* Attempt a connection to the remote server */
    if (ct_connect(conn, Servername, CS_NULLTERM) == CS_FAIL)
    {
       cleanup...
       return(CS_FAIL);
    }

   get and set the REQUEST capabilities
   get and set the RESPONSE capabilities
   ......
   /*
    ** You do not need to test this srv_senddone's return value
    ** since Open Server will kill this thread if this call fails.
    */
    (CS_VOID)srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                  (CS_INT)0);
    return(CS_SUCCEED);
}
```

Gateway with separate security sessions using delegation

The Open Server application can also act as a gateway between the client and another server, but the gateway application uses the delegated client credentials when establishing the security session with the remote server. A client can delegate only its own credentials.

The client needs to request the CS_SEC_DELEGATION service so that the Open Server application can obtain the delegated credentials once the security session is established.

As in "Simple application using a security session" on page 185, the security session between the client and the gateway Open Server application is established, except for the final login acknowledgment.

In the connection handler, the gateway application:

1 Retrieves the delegated credentials using srv_thread_props(CS_GET, SRV_T_SEC_DELEGCRED).

2 Using ct_con_props(CS_SET, CS_SEC_CREDENTIALS), sets the delegated credentials in the Client-Library connection structure for use in connecting to the remote server.

3 Attempts to connect to the remote server using ct_connect.

4 Sends a srv_senddone(SRV_DONE_FINAL), to acknowledge the client's login.

An example connection handler follows:

```
CS_RETCODE CS_PUBLIC connect_handler(spp)
 SRV_PROC *spp;
 {
    CS_CONNECTION *conn;    /* Connection handle */
    CS_VOID       *creds;     /* security credentials */
    CS_BOOL       trueval = CS_TRUE;
    CS_BOOL       boolval;
    CS_CHAR       mechanismname[MAX_NAMESIZE];
    CS_CHAR       username[MAX_NAMESIZE];
    CS_INT        outlen;
   ......
   allocate and set user data in spp
   ......
   /* Allocate a connection handle for the connection attempt. */
    if (ct_con_alloc(Context, &(userdata->conn)) == CS_FAIL)
    {
       return(CS_FAIL);
    }
   ......
   conn = userdata->conn;
   /*
    ** Initiate security session based login to the target server
    */
   /* Retrieve the client user name */
    if (srv_thread_props(spp, CS_GET, SRV_T_USER,
```

```
     (CS_VOID *)username, MAX_NAMESIZE, &outlen) == CS_FAIL)
 {
    handle failure...
}
/*
 ** Set the client's security principal name to connect to the
 ** target server
 */
 if (ct_con_props(conn, CS_SET, CS_USERNAME,
    (CS_VOID *)username, outlen, (CS_INT *)NULL) == CS_FAIL)
 {
    handle failure...
}
/* Retrieve and set the security mechanism */
 if (srv_thread_props(spp, CS_GET, SRV_T_SEC_MECHANISM,
     (CS_VOID *)mechanismname, MAX_NAMESIZE, &outlen)
     == CS_FAIL)
 {
    handle failure...
}
if (ct_con_props(conn, CS_SET, CS_SEC_MECHANISM,
    (CS_VOID *)mechanismname, outlen, (CS_INT *)NULL)
     == CS_FAIL)
 {
    handle failure...
}
/*
 ** Set security service-network authentication. Alternatively
 ** retrieve services from the current thread and set it.
 */
 if (ct_con_props(conn, CS_SET, CS_SEC_NETWORKAUTH,
    (CS_VOID *)&trueval, CS_SIZEOF(CS_BOOL), (CS_INT *)NULL)
    == CS_FAIL)
 {
    handle failure...
}
set other security services if needed...
/* Ensure that the client enabled security delegation */
 if (srv_thread_props(spp, CS_GET, SRV_T_SEC_DELEGATION,
    (CS_VOID *)&boolval, CS_SIZEOF(CS_BOOL), (CS_INT *)NULL)
    == CS_FAIL)
 {
     handle failure...
 }
if (boolval != CS_TRUE)
 {
```

```
      /* delegation not handled on this dialog */
     handle failure...
  }
  /* Retrieve the delegated credentials */
   if (srv_thread_props(spp, CS_GET, SRV_T_SEC_DELEGCRED,
      (CS_VOID *)&creds, CS_SIZEOF(CS_VOID*), (CS_INT *)NULL)
      == CS_FAIL)
   {
       handle failure...
   }
  /*
   ** Set the delegated credentials to authenticate to the target
   ** server.
   */
   if (ct_con_props(conn, CS_SET, CS_SEC_CREDENTIALS,
      (CS_VOID *)&creds, CS_SIZEOF(CS_VOID *), (CS_INT *)NULL)
      == CS_FAIL)
   {
      handle failure...
  }
  get and set the user's application name and response
   capabilities...
  set the locale and other properties...
  /* Attempt a connection to the remote server */
   if (ct_connect(conn, Servername, CS_NULLTERM) == CS_FAIL)
   {
      handle failure...
  }
  Get and set the REQUEST capabilities...
  Get and set the RESPONSE capabilities...
  ......
  /*
   ** You do not need to test this srv_senddone's return value
   ** since Open Server will kill this thread if this call fails.
   */
   (CS_VOID)srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                (CS_INT)0);
   return(CS_SUCCEED);
}
```

Full passthrough gateway with direct security session

A client can establish a security session using the remote server only. No per-packet security services are performed at any intermediaries between the client and the remote server. If the client requests confidentiality, then the gateway cannot retrieve TDS tokens from the message packets. This arrangement saves overhead since no per-packet services are performed within the gateway, such as those used to decrypt received packets, and to re-encrypt them before transmission.

There may be multiple gateway intermediaries forming a chain of "forwarding servers." In this case, each of these forwarding servers must support the same security mechanism.

To set up a direct security session, take the following steps in the connection handler of the Open Server gateway application:

1 Use srv_getloginfo to obtain login information from the client thread.

2 Use ct_setloginfo to set this information in the connection structure to be used for connecting to the remote server.

3 Install a security session callback, using the following command:

```
ct_callback(conn, CS_SET, CS_SECSESSION_CB, secsession_cb)
```

When the connection to the remote server is made, the callback acts as an intermediary for the handshaking required between the remote server and the gateway's client.

See "Security session callbacks" on page 193 for information on what the callback should contain.

See the Open Client *Client-Library/C Reference Manual* for further information on callbacks.

4 Call ct_connect to connect to the remote server. This call initiates negotiations between the client and remote server to establish a security session. If ct_connect returns CS_SUCCEED, then a security session has been successfully established.

5 Use srv_senddone(SRV_DONE_FINAL) to signal to the client that the login is complete.

Example connection handler

```
CS_RETCODE CS_PUBLIC connect_handler(spp)
 SRV_PROC *spp;
 {
```

```
 CS_CONNECTION *conn;    /* connection handle */
 CS_VOID       *creds;   /* security credentials */
 CS_LOGINFO    *loginfo; /* login information */
 CS_BOOL       boolval;
.....
allocate and set user data in spp
/* Allocate a connection handle for the connection attempt. */
 if (ct_con_alloc(Context, &(userdata->conn)) == CS_FAIL)
 {
    handle failure...
}
.....
conn = userdata->conn;
/*
 ** Save the pointer to thread control structure in the
 ** connection handle
 */
 if (ct_con_props(conn, CS_SET, CS_USERDATA, &spp,
    CS_SIZEOF(spp), (CS_INT *)NULL) == CS_FAIL)
 {
    handle failure...
}
/* Verify that security based login is requested */
 if (srv_thread_props(spp, CS_GET, SRV_T_SEC_NETWORKAUTH,
    (CS_VOID *)&boolval, CS_SIZEOF(CS_BOOL), (CS_INT *)NULL)
    == CS_FAIL)
 {
    handle failure...
}
 if (boolval != CS_TRUE)
  {
    handle the client request that does not use security
    session based login
   .....
   return(CS_SUCCEED);
  }
/* Get and set the login information */
  if (srv_getloginfo(spp, &loginfo) == CS_FAIL)
  {
    handle failure...
}
 if (ct_setloginfo(conn, loginfo) == CS_FAIL)
  {
    handle failure...
}
 /* Install a security session callback for this connection */
```

```
    if (ct_callback((CS_CONTEXT *)NULL, conn, CS_SET,
       CS_SECSESSION_CB, (CS_VOID *)secsession_cb) == CS_FAIL)
    {
       handle failure...
   }
   /* Attempt a connection to the remote server */
    if (ct_connect(conn, Servername, CS_NULLTERM) == CS_FAIL)
    {
       handle failure...
   }
   /* Get and set the login information */
    if (ct_getloginfo(conn, &loginfo) == CS_FAIL)
    {
       handle failure...
   }
   if (srv_setloginfo(spp, loginfo) == CS_FAIL)
    {
       handle failure...
   }
   ......
   /*
    ** You do not need to test this srv_senddone's return value
    ** since Open Server will kill this thread if this call fails.
    */
    (CS_VOID)srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                  (CS_INT)0);
    return(CS_SUCCEED);
}
```

Security session callbacks

The security session callback routine exchanges security tokens between the target server (or the next intermediary of the gateway) and the gateway's client applications to establish a direct security session between the client and the remote server. This callback procedure is similar to a challenge-response callback, except that it uses different parameters.

When the gateway calls ct_connect, the remote server issues one or more messages that contain security session information. For each security message, Client-Library invokes the callback with the message parameters sent by the remote server.

The callback routine must perform the following functions:

1 Retrieve the parameters from the remote server's message.

2 Send the parameters to the client, using:

- srv_negotiate(..., CS_SET, SRV_NEG_SECSESSION)
- srv_descfmt(..., CS_SET, SRV_NEGDATA, ...)
- srv_bind(..., CS_SET, ...)
- srv_xferdata(..., CS_SET, ...)

3 Send a srv_senddone(SRV_DONE_FINAL) to the client.

4 Wait for a response from the client, using srv_negotiate(CS_GET, SRV_NEG_SECSESSION).

5 When the client responds, the callback routine copies the corresponding session data from the client to output buffers and sends it to the remote server, using the following functions:

- srv_descfmt(CS_GET)
- srv_bind(CS_GET)
- srv_xferdata(CS_GET)

6 If the remote server sends another security message, the process repeats.

See the Open Client *Client-Library/C Reference Manual* for information on defining security session callbacks.

Example Client-Library security session callback routine

```
CS_RETCODE CS_PUBLIC secsession_cb(conn, innumparams, infmt,
    inbuf, outnumparams, outfmt, outbuf, outlen)
 CS_CONNECTION *conn;
 CS_INT        innumparams;
 CS_DATAFMT    *infmt;
 CS_BYTE       **inbuf;
 CS_INT        *outnumparams;
 CS_DATAFMT    *outfmt;
 CS_BYTE       **outbuf;
 CS_INT        *outlen;
 {
   SRV_PROC *spp; /* The SRVPROC structure associated with the
                   ** client connection */
    CS_INT i;

   /* Get the previously saved spp for the client */
    if (ct_con_props(conn, CS_GET, CS_USERDATA, &spp,
       CS_SIZEOF(spp), (CS_INT *)NULL) != CS_SUCCEED)
    {
       return(CS_FAIL);
```

```
   }
  /*
   ** Use srv_negotiate to tell the client to expect a security
   ** token
   */
   if (srv_negotiate(spp, CS_SET, SRV_NEG_SECSESSION)
      != CS_SUCCEED)
   {
      return(CS_FAIL);
   }
/* Describe and send the security token */
  for (i = 0; i < innumparams; i++)
  {
     if (srv_descfmt(spp, CS_SET, SRV_NEGDATA, i + 1, &infmt[i]
        != CS_SUCCEED)
     {
        return(CS_FAIL);
     }
    if (srv_bind(spp, CS_SET, SRV_NEGDATA, i + 1, &infmt[i],
        inbuf[i], &(infmt[i]->maxlength), (CS_SMALLINT *)NULL)
        != CS_SUCCEED)
     {
        return(CS_FAIL);
     }
  }
  if (srv_xferdata(spp, CS_SET, SRV_NEGDATA) != CS_SUCCEED)
   {
      return(CS_FAIL);
   }
  /* Complete this portion of the exchange */
   if (srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED, 0)
      != CS_SUCCEED)
   {
      return(CS_FAIL);
   }
  /* Wait until the client responds */
   if (srv_negotiate(spp, CS_GET, SRV_NEG_SECSESSION)
      != CS_SUCCEED)
   {
      return(CS_FAIL);
   }
  /* Get the number of parameters in the client's response */
   if (srv_numparams(spp, outnumparams) != CS_SUCCEED)
```

```
    {
       return(CS_FAIL);
    }

   /* Read in the client's response */
    for (i = 0; i < (*outnumparams); i++)
    {
       srv_bzero(&outfmt[i], sizeof(CS_DATAFMT));

      if (srv_descfmt(spp, CS_GET, SRV_NEGDATA, i + 1, &outfmt[i]
          != CS_SUCCEED)
       {
          return(CS_FAIL);
       }

      if (srv_bind(spp, CS_GET, SRV_NEGDATA, i + 1, &outfmt[i],
          outbuf[i], &outlen[i], (CS_SMALLINT *)NULL)
          != CS_SUCCEED)
       {
          return(CS_FAIL);
       }

   }

   if (srv_xferdata(spp, CS_GET, SRV_NEGDATA) != CS_SUCCEED)
    {
       return(CS_FAIL);
    }

   /* Return success */
    return(CS_SUCCEED);
}
```

Text and image

The text and image Adaptive Server datatypes hold large text or image values. The text datatype will hold up to 2,147,483,647 bytes of printable characters. The image datatype will hold up to 2,147,483,647 bytes of binary data.

Because they can be so large, text and image values are not actually stored in database tables. Instead, a pointer to the text or image value is stored in the table. This pointer is called a *text pointer*.

To ensure that competing client applications do not overwrite one another's modifications to the database, a timestamp is associated with each text or image column. This timestamp is called a *text timestamp*.

Processing text and image data

Clients send text and image data as an undifferentiated data stream, known as a *writetext* stream. Because it is not differentiated into parameters, an Open Server application cannot rely on the routines it normally uses in processing incoming parameter data: srv_descfmt srv_bind, and srv_xferdata. Instead, it must use a special set of text and image routines.

An Open Server application can send text or image data back to a client in one of two ways, depending on how many columns the return row contains. If the return row contains just one column, and that column contains text or image data, it can be treated as an undifferentiated data stream, and its processing deviates from the norm. If, however, the row contains other columns in addition to a text or image column, the text or image data is processed using the describe/bind/transfer method. Note that both methods have some steps in common.

See “Processing parameter and row data” on page 134 for details on the describe/bind/transfer triad of calls.

Retrieving data from a client

A writetext stream triggers a SRV_BULK event. Because text and image data retrieved from a client are considered bulk data, an Open Server application processes incoming text and image data from inside its bulk handler. For more information on types of bulk data see the Open Client and Open Server *Common Libraries Reference Manual*.

An application processes incoming text or image data in two steps:

1 The srv_text_info routine retrieves a description of the text or image data and places the information in a CS_IODESC structure. This call returns a variety of information, the most important of which is the total length of the data. Based on the length, the application can decide whether to retrieve the data all at once or in sections, as well as how large a buffer to allocate to store the data. srv_text_info is called with the cmd argument set to CS_GET.

2 The srv_get_text routine actually brings the data over from the client in the specified section size and stores it in the specified buffer.

Note that a call to srv_text_info must always precede a call to srv_get_text. The srv_get_text routine must be called until all text has been read from the client.

Returning data to a client

An application can return text or image data inside of any event handler that can return row results. An application processes outgoing text or image data in several different steps, depending on how many columns are in the data row. If there is just one column, and it is a text or image column, the application takes the following steps:

1 It describes the format in which the client will receive the text or image column, using srv_descfmt.

2 It calls srv_text_info with cmd set to CS_SET to provide the total text length.

3 It calls srv_send_text to send the data to the client in chunks.

If there are other columns in addition to the text and image column or columns, the application must take the following steps:

1 It describes the format in which the client will receive the data using srv_descfmt, which is called once for each column.

2 It describes the format and location of the local program variables in which the Open Server application stores the information, using srv_bind, which must be called once for each column.

3 It provides text pointer and timestamp information by calling srv_text_info, which must be called once for each text or image column, with cmd set to CS_SET.

4 Transfer the data using srv_xferdata, which must be called as many times as there are rows.

See “Processing parameter and row data” on page 134 for details on partial update of text and image columns.

Example

The sample, *ctos.c*, includes code to process text and image data.

Types

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

Table 2-37 lists the Open Server type definitions, together with their corresponding type constants and Adaptive Server datatypes. More detailed information on each datatype follows the chart.

2.0 Open Server datatypes are included in this version for the sake of backward compatibility. 2.0 Server-Library routines must use 2.0 datatypes in this version. Table 2-37 summarizes the Open Server datatypes that all routines must use in future Open Server versions.

Table 2-37: Datatype summary

Type	Open Client and Open Server type constant	Description	Corresponding Open Client and Open Server type definition	Corresponding Adaptive Server datatype
Binary types	CS_BINARY_TYPE	Binary type	CS_BINARY	binary, varbinary
	CS_LONGBINARY_TYPE	Long binary type	CS_LONGBINARY	NONE
	CS_VARBINARY_TYPE	Variable-length binary type	CS_VARBINARY	NONE
Bit types	CS_BIT_TYPE	Bit type	CS_BIT	boolean
Character types	CS_CHAR_TYPE	Character type	CS_CHAR	char, varchar
	CS_LONGCHAR_TYPE	Long character type	CS_LONGCHAR	NONE
	CS_VARCHAR_TYPE	Variable-length character type	CS_VARCHAR	NONE
	CS_UNICHAR_TYPE	Variable-length or fixed-length character type	CS_UNICHAR	unichar, univarchar
XML type	CS_XML_TYPE	Variable-length character type	CS_XML	xml

Type	Open Client and Open Server type constant	Description	Corresponding Open Client and Open Server type definition	Corresponding Adaptive Server datatype
Datetime types	CS_DATE_TYPE	4-byte date datatype	CS_DATE	date
	CS_TIME_TYPE	4-byte time datatype	CS_TIME	time
	CS_DATETIME_TYPE	8-byte datetime type	CS_DATETIME	datetime
	CS_DATETIME4_TYPE	4-byte datetime type	CS_DATETIME4	smalldatetime
Numeric types	CS_TINYINT_TYPE	1-byte integer type	CS_TINYINT	tinyint
	CS_SMALLINT_TYPE	2-byte integer type	CS_SMALLINT	smallint
	CS_INT_TYPE	4-byte integer type	CS_INT	int
	CS_BIGINT_TYPE	8-byte integer type	CS_BIGINT	bigint
	CS_USMALLINT_TYPE	Unsigned 2-byte integer type	CS_USMALLINT	usmallint
	CS_UINT_TYPE	Unsigned 4-byte integer type	CS_UINT	uint
	CS_UBIGINT_TYPE	Unsigned 8-byte integer type	CS_UBIGINT	ubigint
	CS_DECIMAL_TYPE	Decimal type	CS_DECIMAL	decimal
	CS_NUMERIC_TYPE	Numeric type	CS_NUMERIC	numeric
	CS_FLOAT_TYPE	8-byte float type	CS_FLOAT	float
	CS_REAL_TYPE	4-byte float type	CS_REAL	real
Money types	CS_MONEY_TYPE	8-byte money type	CS_MONEY	money
	CS_MONEY4_TYPE	4-byte money type	CS_MONEY4	smallmoney

Type	Open Client and Open Server type constant	Description	Corresponding Open Client and Open Server type definition	Corresponding Adaptive Server datatype
Text and image types	CS_TEXT_TYPE	Text type	CS_TEXT	text
	CS_UNITEXT_TYPE	Unsigned variable-length character type	CS_UNITEXT	unitext
	CS_IMAGE_TYPE	Image type	CS_IMAGE	image

Routines that manipulate datatypes

CS-Library provides several routines that are useful for manipulating datatypes. They include:

- cs_calc, which performs arithmetic operations on decimal, float, money, numeric, and real datatypes.
- cs_cmp, which compares datetime, decimal, float, money, numeric, and real datatypes.
- cs_convert, which converts a data value from one datatype to another.
- cs_dt_crack, which converts a machine readable datetime value into a user-accessible format.
- cs_dt_info, which retrieves datetime information for a national language.

These routines are documented in the Open Client and Open Server *Common Libraries Reference Manual*.

Open Server datatypes

Binary types

Open Server has three binary types, CS_BINARY, CS_LONGBINARY, and CS_VARBINARY.

- CS_BINARY corresponds to the Adaptive Server datatypes binary and varbinary. That is, Server-Library interprets both the server binary and varbinary types as CS_BINARY. For example, srv_descfmt returns CS_BINARY_TYPE when retrieving a description of a binary parameter from a client.

  CS_BINARY is defined as:

  ```
  typedef unsigned char     CS_BINARY;
  ```

- CS_LONGBINARY does not correspond to any Adaptive Server datatype, but some Open Server applications may support CS_LONGBINARY. An application can use the CS_DATA_LBIN capability to determine whether a Client-Library connection supports CS_LONGBINARY.

  A CS_LONGBINARY value has a maximum length of 2,147,483,647 bytes. CS_LONGBINARY is defined as:

  ```
  typedef unsigned char       CS_LONGBINARY;
  ```

- CS_VARBINARY does not correspond to any Adaptive Server datatype. For this reason, Open Server routines do not return CS_VARBINARY_TYPE. If a datatype is described as CS_VARBINARY_TYPE, Open Server automatically converts it to a nullable CS_BINARY_TYPE before sending it to a client. CS_VARBINARY_TYPE can only be used when binding program variables. CS_VARBINARY enables programmers to write non-C programming language veneers for Open Server. Typical server applications will not use CS_VARBINARY.

  CS_VARBINARY is defined as follows:

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

  where:

  - *len* is the length of the binary array.
  - *array* is the array itself.

Bit type

Open Server supports a single bit type, CS_BIT. This datatype holds server bit (or Boolean) values of 0 or 1. When converting other types to bit, all non-zero values are converted to 1:

```
typedef unsigned char      CS_BIT;
```

Character types

Open Server has four character types, CS_CHAR, CS_LONGCHAR, CS_VARCHAR, and CS_UNICHAR:

- CS_CHAR corresponds to the Adaptive Server datatypes char and varchar. That is, Server-Library interprets both the server char and varchar datatypes as CS_CHAR. For example, srv_descfmt returns CS_CHAR_TYPE when retrieving the description of a character parameter from a client.

  CS_CHAR is defined as follows:

  ```
  typedef char               CS_CHAR;
  ```

- CS_LONGCHAR does not correspond to any Adaptive Server datatype, but some Client-Library applications may support CS_LONGCHAR. An application can use the CS_DATA_LCHAR capability to determine whether a Client-Library connection supports CS_LONGCHAR.

  A CS_LONGCHAR value supports a maximum length of 2,147,483,647 bytes. CS_LONGCHAR is defined as follows:

  ```
  typedef unsigned char      CS_LONGCHAR;
  ```

- CS_VARCHAR does not correspond to any Adaptive Server datatype. For this reason, Open Server routines do not return CS_VARCHAR_TYPE. If a datatype is described as CS_VARCHAR_TYPE, Open Server automatically converts it to a nullable CS_CHAR_TYPE before sending it to a client. CS_VARCHAR_TYPE can only be used when binding program variables. CS_VARCHAR enables programmers to write non-C programming language veneers for Open Server. Typical server applications will not use CS_VARCHAR.

  CS_VARCHAR is defined as follows:

  ```
  typedef struct _cs_varchar
   {
       CS_SMALLINT          len;
       CS_BYTE              str[CS_MAX_CHAR];
  ```

```
} CS_VARCHAR;
```

where:

- *len* is the length of the string.
- *str* is the string itself. Note that *str* is not a null-terminated string.

- CS_UNICHAR corresponds to the Adaptive Server unichar fixed-width and univarchar variable-width datatypes. CS_UNICHAR is a shared, C-programming datatype that can be used anywhere the CS_CHAR datatype is used. The CS_UNICHAR datatype stores character data in the 2-byte Unicode UTF-16 format.

  CS_UNICHAR is defined as follows:

  ```
  typedef unsigned char       CS_UNICHAR;
  ```

XML type

CS_XML corresponds directly to Adaptive Server xml variable-length datatype. CS_XML can be used anywhere CS_TEXT and CS_IMAGE are used to represent XML documents and contents.

CS_XML is defined as follows:

```
typedef unsigned char       CS_XML
```

Datetime types

Open Server supports two datetime types, CS_DATETIME and CS_DATETIME4. These datatypes are intended to hold 8-byte and 4-byte datetime values, respectively.

In addition, Open Server supports CS_DATE and CS_TIME datatypes. These datatypes behave like CS_DATETIME and CS_DATETIME4, but rather than store data in a single datetime value, they store data in separate 4-byte fixed-width date or time values.

An Open Server application can use the CS-Library routine cs_dt_crack to extract date parts (year, month, day, and so on) from a datetime structure.

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

  ```
  typedef struct _cs_datetime
  ```

```
{
    CS_INT              dtdays;
    CS_INT              dttime;
} CS_DATETIME;
```

where:

- *dtdays* is the number of days since 1/1/1900.
- *dttime* is the number of 300ths of a second since midnight.

- CS_DATETIME4 corresponds to the Adaptive Server smalldatetime datatype. The range of legal CS_DATETIME4 values is from January 1, 1900, to June 6, 2079, with a precision of 1 minute:

```
typedef struct _cs_datetime4
{
    unsigned short        days;
    unsigned short        minutes;
} CS_DATETIME4;
```

where:

- *days* is the number of days since 1/1/1900.
- *minutes* is the number of minutes since midnight.

- CS_DATE corresponds to the Adaptive Server date datatype. The range of legal CS_DATE values is from January 1, 1753 to December 31, 9999.

```
typedef struct _cs_date
{
    CS_INT          days;
} CS_DATE;
```

where *days* is the number of days since 1/1/1900

- CS_TIME corresponds to the Adaptive Server time datatype. The range of legal CS_TIME values is with a precision of 1/300th of a second (3.33 milliseconds):

```
typedef struct _cs_time
{
    CS_INT          time;
} CS_TIME;
```

where *time* is the number of 300ths of a second since midnight.

Integer types

Open Server supports seven integer types: CS_TINYINT, CS_SMALLINT, CS_INT, CS_BIGINT, CS_USMALLINT, CS_UINT, and CS_UBIGINT.

On most platforms, CS_TINYINT is a 1-byte integer; CS_SMALLINT is a 2-byte integer, CS_INT is a 4-byte integer, CS_BIGINT is an 8-byte integer, CS_USMALLINT is an unsigned 2-byte integer, CS_UINT is an unsigned 4-byte integer and CS_UBIGINT is an unsigned 8-byte integer:

```
typedef unsigned char        CS_TINYINT;
typedef short                CS_SMALLINT;
typedef int                  CS_INT;
typedef long long            CS_BIGINT;
typedef unsigned char        CS_USMALLINT;
typedef unsigned int         CS_UINT;
typedef unsigned long long   CS_UBIGINT;
```

Real, float, numeric, and decimal types

- CS_REAL corresponds to the Adaptive Server datatype real. It is implemented as a platform-dependent C-language float type:

  ```
  typedef float                CS_REAL;
  ```

  Note When converting 6-digit precision bigint or ubigint datatypes to *real* datatypes, note the following maximum and minimum values:

  - -9223370000000000000.0 < bigint < 9223370000000000000.0
  - 0 < ubigint < 18446700000000000000.0

  Values outside of these ranges cause overflow errors.

- CS_FLOAT corresponds to the Adaptive Server datatype float. It is implemented as a platform-dependent, C-language double type:

  ```
  typedef double               CS_FLOAT;
  ```

  Note When converting 15-digit precision bigint or ubigint datatypes to *float* datatypes, note the following maximum and minimum values:

  - -9223372036854770000.0 < bigint < 9223372036854770000.0
  - 0 < ubigint < 18446744073709500000.0

  Values outside of these ranges cause overflow errors.

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

  The Adaptive Server datatypes numeric and decimal are equivalent; and CS_DECIMAL is defined as CS_NUMERIC:

```
typedef struct  _cs_numeric
 {
     CS_BYTE                  precision;
     CS_BYTE                  scale;
     CS_BYTE                  array[CS_MAX_NUMLEN];
 } CS_NUMERIC;

 typedef CS_NUMERIC           CS_DECIMAL;
```

  where:

  - *precision* is the precision of the numeric value. Legal values for *precision* are from CS_MIN_PREC to CS_MAX_PREC. The default precision is CS_DEF_PREC. CS_MIN_PREC, CS_MAX_PREC, and CS_DEF_PREC define the minimum, maximum, and default precision values, respectively.
  - *scale* is the scale of the numeric value. Legal values for *scale* are from CS_MIN_SCALE to CS_MAX_SCALE. The default scale is CS_DEF_SCALE. CS_MIN_SCALE, CS_MAX_SCALE, and CS_DEF_SCALE defines the minimum, maximum, and default scale values, respectively.
  - *scale* must be less than or equal to *precision*.

  CS_DECIMAL types use the same default values for *precision* and *scale* as CS_NUMERIC types.

Money types

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

- CS_MONEY corresponds to the Adaptive Server money datatype. The range of legal CS_MONEY values is between +/- $922,337,203,685,477.5807:

```
typedef struct _cs_money
 {
     CS_INT               mnyhigh;
```

```
    CS_UINT             mnylow;
} CS_MONEY;
```

- CS_MONEY4 corresponds to the Adaptive Server smallmoney datatype. The range of legal CS_MONEY4 values is between -$214,748.3648 and +$214,748.3647:

```
typedef struct _cs_money4
 {
    CS_INT              mny4;
 } CS_MONEY4;
```

Security types

Open Server supports Secure Adaptive Server boundary and sensitivity datatypes by defining the type constants CS_BOUNDARY_TYPE and CS_SENSITIVITY_TYPE.

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

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

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

Text and image types

Open Server supports text datatypes, CS_TEXT and CS_UNITEXT, as well as an image datatype, CS_IMAGE.

- CS_TEXT corresponds to the server datatype text, which describes a variable-length column containing up to 2,147,483,647 bytes of printable character data. CS_TEXT is defined as an unsigned character:

```
typedef unsigned char         CS_TEXT;
```

- CS_UNITEXT corresponds to the Adaptive Server unitext variable-length datatype. CS_UNITEXT exhibits identical syntax and semantics to CS_TEXT, except that CS_UNITEXT encodes character data in the 2-byte Unicode UTF-16 format. CS_UNITEXT can be used anywhere CS_TEXT is used. The maximum length of the CS_UNITEXT string parameter is half of the maximum length of CS_TEXT.

  CS_UNITEXT is defined as follows:

  ```
  typedef unsigned short       CS_UNITEXT;
  ```

- CS_IMAGE corresponds to the server datatype image, which describes a variable-length column containing up to 2,147,483,647 bytes of binary data. CS_IMAGE is defined as an unsigned character:

  ```
  typedef unsigned char         CS_IMAGE;
  ```

CHAPTER 3

Routines

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

Routine	Description	Page
srv_deletemutex	Delete a mutex created by srv_createmutex.	263
srv_descfmt	Describe or retrieve the description of a column or a parameter going to, or coming from, a client.	265
srv_dynamic	Read or respond to a client dynamic SQL command.	268
srv_envchange	Notify the client of an environment change.	273
srv_event	Add an event request to a thread's request-handling queue.	275
srv_event_deferred	Add an event request to the event queue of a thread as the result of an asynchronous event.	278
srv_free	Free previously allocated memory.	280
srv_freeserveraddrs	Frees memory allocated by srv_getserverbyname.	281
srv_get_text	Read a text or image datastream from a client, in chunks.	282
srv_getloginfo	Obtain login information from a client thread to prepare a passthrough connection with a remote server.	284
srv_getmsgq	Get the next message from a message queue.	286
srv_getobjid	Look up the object ID for a message queue or mutex with a specified name.	289
srv_getobjname	Get the name of a message queue or mutex with a specified name.	292
srv_getserverbyname	Returns the connection information for *server_name*, allocating memory as needed.	294
srv_handle	Install an event handler into an Open Server application.	295
srv_init	Initialize an Open Server application.	298
srv_langcpy	Copy a client's language request into an application buffer.	300
srv_langlen	Return the length of the language request buffer.	302
srv_lockmutex	Lock a mutex.	304

Routine	Description	Page
srv_log	Write a message to the Open Server application log file.	307
srv_mask	Initialize, check, set, or clear bits in a SRV_MASK_ARRAY structure.	309
srv_msg	Send or receive a message datastream.	311
srv_negotiate	Send and receive negotiated login information to or from a client.	314
srv_numparams	Return the number of parameters contained in the current client command.	321
srv_options	Send or receive option information to or from a client.	323
srv_orderby	Return an order-by list to a client.	329
srv_poll (UNIX only)	Check for I/O events on a set of open streams file descriptors.	331
srv_props	Define and retrieve Open Server properties.	334
srv_putmsgq	Put a message into a message queue.	340
srv_realloc	Reallocate memory.	342
srv_recvpassthru	Receive a protocol packet from a client.	344
srv_regcreate	Complete the registration of a registered procedure.	346
srv_regdefine	Initiate the process of registering a procedure.	348
srv_regdrop	Unregister a procedure.	352
srv_regexec	Execute a registered procedure.	354
srv_reginit	Begin executing a registered procedure.	356
srv_reglist	Obtain a list of all of the procedures registered in the Open Server.	358
srv_reglistfree	Free a previously allocated SRV_PROCLIST structure.	360
srv_regnowatch	Remove a client thread from the notification list for a registered procedure.	361
srv_regparam	Describe a parameter for a registered procedure being defined; or supply data for the execution of a registered procedure.	363
srv_regwatch	Add a client thread to the notification list for a specified procedure.	366
srv_regwatchlist	Return a list of all registered procedures for which a client thread has notification requests pending.	369

Routine	Description	Page
srv_rpcdb	Return the database component of the current remote procedure call's designation.	371
srv_rpcname	Return the name component of the current remote procedure call's designation.	372
srv_rpcnumber	Return the number component of the current remote procedure's designation.	375
srv_rpcoptions	Return the runtime options for the current remote procedure call.	376
srv_rpcowner	Return the owner component of the current remote procedure call's designation.	378
srv_run	Start an Open Server.	380
srv_s_ssl_local_id	Used to specify the path to the local ID (certificates) file.	381
srv_select (UNIX only)	Check to see if a file descriptor is &ready for a specified I/O operation.	381
srv_send_ctlinfo	Sends control messages to Client-Library.	385
srv_send_data	Transfers rows containing multiple columns to clients.	386
srv_send_text	Send a text or image datastream to a client, in chunks.	390
srv_senddone	Send a results completion message or flush some results to a client.	393
srv_sendinfo	Send error or informational messages to the client.	398
srv_sendpassthru	Send a protocol packet to a client.	401
srv_sendstatus	Send a status value to a client.	404
srv_setcolutype	Define the user datatype to be associated with a column.	405
srv_setcontrol	Describe user control or format information for columns.	407
srv_setloginfo	Return protocol format information from a remote server to a client.	409
srv_setpri	Modify the scheduling priority of a thread.	411
srv_signal (UNIX only)	Install a UNIX signal handler for the SIGIO or SIGURG signals, using the same interface as signal.	413
srv_sleep	Suspend the currently executing thread.	416
srv_spawn	Allocate a service thread.	419

srv_alloc

Description

Allocate memory.

Syntax

```
CS_VOID *srv_alloc(size)
CS_INT       size;
```

Parameters

size

The number of bytes to allocate.

Return value

Table 3-1: Return values (srv_alloc)

Returns	To indicate
A pointer to the newly allocated space	The location of the new space.
A null CS_VOID pointer	Open Server could not allocate *size* bytes.

Examples

```
#include    <ospublic.h>
/*
 ** Local Prototype
 */
 CS_RETCODE     ex_srv_alloc PROTOTYPE((
 CS_BYTE        **bpp,
 CS_INT         size
 ));
/*
 ** EX_SRV_ALLOC
 **
 **    Example routine to allocate the specified amount of memory
 **    using srv_alloc.
 **
 ** Arguments:
 **    bpp     Return pointer to allocated memory here.
 **    size    Amount of memory to allocate.
 **
 ** Returns:
 **
 **    CS_SUCCEED            Memory was allocated successfully.
 **    CS_FAIL               An error was detected.
 */
 CS_RETCODE            ex_srv_alloc(bpp, size)
 CS_BYTE              **bpp;
 CS_INT               size;
 {
     /* Initialization. */
     *bpp = (CS_BYTE *)NULL;

   /*
     ** Allocate size number of bytes.
     */
     if((*bpp = (CS_BYTE *)srv_alloc(size)) == (CS_BYTE *)NULL)
     {
         return(CS_FAIL);
     }
     return(CS_SUCCEED);
```

```
}

   /*
** Allocate size number of bytes.
*/
if((*bpp = (CS_BYTE *)srv_alloc(size)) == (CS_BYTE *)NULL)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

Usage

- srv_alloc allocates memory dynamically. It returns a pointer to *size* bytes if that many bytes are available.
- Any memory allocated using srv_alloc should be freed by calling srv_free.
- Use srv_alloc wherever the standard C memory allocation routines would be used.
- Currently, srv_alloc calls the C routine, malloc. An Open Server application, however, can install its own memory management routines using the srv_props routine. The parameter-passing conventions of the user-installed routines must be the same as those of malloc. If the application is not configured to use user-installed routines, Open Server will call malloc.

See also srv_free, srv_props, srv_realloc

srv_alt_bind

Description Describe and bind the source data for a compute row column.

Syntax

```
CS_RETCODE srv_alt_bind(spp, altid, item, osfmtp,
        varaddr, varlenp, indp)

SRV_PROC        *spp;
CS_INT          altid;
CS_INT          item;
CS_DATAFMT      *osfmtp;
CS_BYTE         *varaddrp;
CS_INT          *varlenp;
CS_SMALLINT     *indp;
```

Parameters

spp

A pointer to an internal thread control structure.

altid
The unique identifier for the compute row in which this compute column is contained. The *altid* is defined using srv_alt_header.

item
The column's column number in the compute row. Compute row column numbers start at 1.

osfmtp
A pointer to a CS_DATAFMT structure. This structure describes the format of the compute row column data that the application program variable contains.

varaddrp
A pointer to the program variable to which the outgoing data is bound.

varlenp
A pointer to the program variable containing **varaddrp*'s length.

indp
A pointer to the buffer containing the null value indicator. The following table summarizes the values **indp* can contain:

Table 3-2: Values for indp (srv_alt_bind)

Value	Indicates
CS_NULLDATA	Column data is null.
CS_GOODDATA	Column data is not null.

If *indp* is NULL, the column data is assumed to be valid; that is, not null.

Return value

Table 3-3: Return values (srv_alt_bind)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include     <ospublic.h>
/*
 ** Local prototype
 */
 CS_RETCODE             ex_srv_alt_bind PROTOTYPE((
 SRV_PROC               *spp,
 CS_INT                 altid,
 CS_VOID                *sump
```

```
));
/*
** EX_SRV_ALT_BIND
**
**    Example routine to describe and bind the source data for
**    a compute row column. This example binds a value which
**    is the sum of the first column of row data.
**
** Arguments:
**    spp    - A pointer to an internal thread control structure.
**             The thread must be an active client thread that
**             can handle row data.
**
**    altid    -     The id for this compute row.
**
**    sump    -    A pointer to the variable which will contain
**                 the sum of the first column of row data.
**
** Returns:
**     CS_SUCCEED - Compute row column was successfully bound.
**     CS_FAIL    - An error was detected.
**
*/
CS_RETCODE          ex_srv_alt_bind(spp, altid, sump)
SRV_PROC            *spp;
CS_INT              altid;
CS_VOID             *sump;
{
   CS_DATAFMT compute_colfmt;
     /*
     **Format for this compute column.
     */
     CS_INT         namelen;
     /*
     **Length of compute column name
     */

   CS_INT        compute_colnum;
    /*
    **    The column number for this compute column.
    */

  CS_SMALLINT     indicator;
    /*
    **    Null indicator.
    */
    CS_INT            sumlen;
```

```
    /*
    **    Length of the compute value
    */
    CS_RETCODE        result;
    /*
    **Return value from srv_alt_bind.
    */

  /*
    ** Initialize the compute column's data format. This compute
    ** column represents a sum of the first column of data.
    */
    namelen = 3;
    srv_bmove("sum", compute_colfmt.name, namelen);

  compute_colfmt.namelen = namelen;
    compute_colfmt.datatype = CS_INT_TYPE;
    compute_colfmt.format = CS_FMT_UNUSED;
    compute_colfmt.maxlength = sizeof(CS_INT);
    compute_colfmt.scale = 0;

  compute_colfmt.precision = CS_DEF_PREC;
    compute_colfmt.status = 0;
    compute_colfmt.count = 0;
    compute_colfmt.usertype = 0;
    compute_colfmt.locale = (CS_LOCALE *)NULL;

  /*
    ** Perform the bind
    */
    compute_colnum = 1;
    indicator = CS_GOODDATA;
    sumlen = sizeof(CS_INT);

  result = srv_alt_bind(spp, altid, compute_colnum,
            &compute_colfmt, sump, &sumlen, &indicator);
   return (result);
}
```

Usage

- Only applications that mimic Adaptive Server's feature of returning compute row information will need to call srv_alt_bind. srv_alt_bind is most useful to applications acting as a gateway to an Adaptive Server.
- srv_alt_bind describes the format of the application program variable in which a compute row column's data is stored. An application must call it once for each column in a compute row.

- The srv_alt_bind routine reads from (CS_GET) or sets (CS_SET) the CS_DATAFMT fields listed in the table below. All other fields are undefined for srv_alt_bind. (Note that "osfmtp" is a pointer to the structure.

Table 3-4: CS_DATAFMT fields used (srv_alt_bind)

Field	CS_SET	CS_GET
osfmtp→datatype	Datatype of application program variable	Datatype of application program variable
osfmtp→maxlength	Unused	Maximum length of program variable
osfmtp→count	0 or 1	0 or 1

- If the format described by *osfmtp* differs from the client format set with srv_alt_descfmt (*clfmtp*), Open Server automatically converts the data to the client format.
- A compute result set contains only one row. However, an application can return multiple result sets, each with a distinct *altid*.
- To process compute row data, an Open Server application must:
  a Call srv_alt_header to define a compute row identifier.
  b Call srv_alt_descfmt for each column to describe the format the column data is in when the client receives it.
  c Call srv_alt_bind for each column to bind the data to a local program variable.
  d Call srv_alt_xferdata to send the row to the client, once each column in the compute row has been described and its data bound to a program variable.
- The contents of the buffers to which *varaddrp, lenp,* and *indp* point need not be valid until srv_xferdata is called.

See also srv_alt_descfmt, srv_alt_header, srv_alt_xferdata, "CS_DATAFMT structure" on page 54

srv_alt_descfmt

Description Describe the aggregate operator of a compute row column and the format of the column data returned to the client.

Syntax

```
CS_RETCODE srv_alt_descfmt(spp, altid, optype,
        operand, item, clfmtp)

SRV_PROC        *spp;
CS_INT          altid;
CS_INT          optype;
CS_TINYINT      operand;
CS_INT          item;
CS_DATAFMT      *clfmtp;
```

Parameters

spp

A pointer to an internal thread control structure.

altid

The unique identifier for the compute row in which this compute column is contained. The *altid* is defined using srv_alt_header.

item

The column's column number in the compute row. Compute row column numbers start at 1.

optype

The aggregate operator type of the compute row column. The following table lists the legal operator types:

Table 3-5: Values for optype (srv_alt_descfmt)

Operator type	Function
CS_OP_COUNT	Count aggregate operator
CS_OP_SUM	Sum aggregate operator
CS_OP_AVG	Average aggregate operator
CS_OP_MIN	Minimum aggregate operator
CS_OP_MAX	Maximum aggregate operator

operand

The select-list column the aggregate is operating on.

clfmtp

A pointer to the CS_DATAFMT structure. This structure describes the format the column data is in when the client receives it.

Return value

Table 3-6: Return values (srv_alt_descfmt)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include       <ospublic.h>
 /*
 ** Local Prototype
 */
 CS_RETCODE      ex_srv_alt_descfmt PROTOTYPE((
 SRV_PROC        *sproc,
 CS_INT          altid,
 CS_DATAFMT      clfmtp[]
 ));
/*
 ** EX_SRV_ALT_DESCFMT
 **   An example routine to describe the aggregate operator of 2
 **   compute row columns and the format of each of the two column
 **   data returned to the client. We will do the sum on the first
 **   column and average on the second column.
 **
 ** Arguments:
 **   sproc     A pointer to an internal thread control structure.
 **   altid     The id for the compute row in which this compute
 **             column is contained. The altid is obtained by
 **             calling srv_alt_header.
 **   clfmtp    A pointer to the array of structures describing
 **             the format of the compute row column
 **             data when the client receives it.
 **
 ** Returns:
 **   CS_SUCCEED     If the aggregate operator and the datatype of
 **                  the compute row columns were successfully
 **                  described.
 **   CS_FAIL        An error was detected.
 */
 CS_RETCODE     ex_srv_alt_descfmt(sproc, altid, clfmtp)
 SRV_PROC       *sproc;
 CS_INT         altid;
 CS_DATAFMT     clfmtp[];
 {
      /*
      ** Describe the aggregate operator of the first compute row
      ** column and the format of the column data.
```

```
        */
        if ( srv_alt_descfmt(sproc, altid, (CS_INT)1, CS_OP_SUM,
             (CS_TINYINT)1, &clfmtp[0]) == CS_FAIL )
        {
             return(CS_FAIL);
        }

       /*
        ** Now do the same for the second column if (srv_alt_descfmt
        ** (sproc, altid, (CS_INT)2, CS_OP_AVG, (CS_TINYINT)2,
        ** &clfmtp[1]) == CS_FAIL )
        {
             return(CS_FAIL);
        }
        */

        return(CS_SUCCEED);
    }
```

Usage

- Only applications that mimic Adaptive Server's feature of returning compute row information will need to call srv_alt_descfmt. srv_alt_descfmt is most useful to applications acting as a gateway to an Adaptive Server.
- srv_alt_descfmt describes a compute row column that the application will send to the client. The application calls it once for each column in the compute row.
- The srv_alt_descfmt routine reads from (CS_GET) or sets (CS_SET) the CS_DATAFMT fields listed in the table below. All other fields are undefined for srv_alt_descfmt. (Note that "clfmtp" is a pointer to the structure.

Table 3-7: CS_DATAFMT structure fields used (srv_alt_descfmt)

Field	CS_SET	CS_GET
clfmtp→namelen	Length of name	Length of name
clfmtp→status	Parameter/column status	Parameter status
clfmtp→name	Parameter/column name	Parameter name
clfmtp→datatype	Remote datatype set here	Remote datatype retrieved from here
clfmtp→maxlength	Maximum length of remote datatype set here	Maximum length of remote datatype retrieved from here
clfmtp→format	Remote datatype format	Remote datatype formats

- If the format described by *clfmtp* differs from the application program variable format subsequently described with srv_alt_bind (*osfmtp*), Open Server automatically converts the data to the *clfmtp* format description.

- To process compute row data, an Open Server application must:
  a Call srv_alt_header to define a compute row identifier.
  b Call srv_alt_descfmt for each column to describe the format the column data is in when the client receives it.
  c Call srv_alt_bind for each column to bind the data to a local program variable.
  d Call srv_alt_xferdata to send the row to the client, once each column in the compute row has been described and its data bound to a program variable.

See also srv_alt_bind, srv_alt_header, srv_alt_xferdata, "CS_DATAFMT structure" on page 54

srv_alt_header

Description Describe a compute row's row identifier and bylist.

Syntax

```
CS_RETCODE srv_alt_header(spp, altid, numbylist,
        bylistarrayp)

SRV_PROC        *spp;
CS_INT          altid;
CS_INT          numbylist;
CS_SMALLINT     *bylistarrayp;
```

Parameters

spp
A pointer to an internal thread control structure.

altid
A unique identifier for this compute row.

numbylist
The number of columns in the bylist of a compute row.

bylistarrayp
A pointer to an array of column numbers that make up the bylist for a compute row. There are as many elements as specified in *numbylist*. If *numbylist* is 0, *bylistarrayp* is ignored.

Return value

Table 3-8: Return values (srv_alt_header)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include            <ospublic.h>
/*
 ** Local Prototype
 */
 CS_RETCODE           ex_srv_alt_header PROTOTYPE((
 SRV_PROC             *spp
 ));

/*
 ** EX_SRV_ALT_HEADER
 **
 **     Example routine to illustrate the use of srv_alt_header
 **     to describe a compute row's row identifier and bylist.
 **
 ** Arguments:
 **     spp - A pointer to an internal thread control structure.
 **
 ** Returns:
 **
 **   CS_SUCCEED     A compute row was successfully described.
 **   CS_FAIL        An error was detected.
 */
 CS_RETCODE     ex_srv_alt_header(spp)
 SRV_PROC     *spp;

{
      CS_INT          altid;
      CS_SMALLINT     bylist[2];

    /*
      ** Let us describe a fictitious compute row with altid =1,
      ** and bylist = [2,4].
      */
      altid = (CS_INT)1;
      bylist[0] = (CS_SMALLINT)2;
      bylist[1] = (CS_SMALLINT)4;

     if (srv_alt_header(spp, altid,
           sizeof(bylist)/sizeof(CS_SMALLINT),
                bylist) == CS_FAIL)
           return (CS_FAIL);
```

```
        return (CS_SUCCEED);
    }
```

Usage

- Only applications that mimic Adaptive Server's feature of returning compute row information will need to call srv_alt_header. srv_alt_header is most useful to applications acting as a gateway to an Adaptive Server.
- srv_alt_header assigns a unique identifier to each compute row and describes the bylist associated with each compute row. It must be called once for each compute row.
- In the Adaptive Server, compute rows result from the compute clause of a Transact- SQL select statement. If a Transact -SQL select statement contains multiple compute clauses, separate compute rows are generated by each clause. Open Server can return rows of compute data, mimicking an Adaptive Server's response to a Transact- SQL compute clause.
- A Transact -SQL select statement's compute clause can contain the **keyword** by, followed by a list of columns. This list, known as the "bylist," divides the results into subgroups, based on changing values in the specified columns. The compute clause's aggregate operators are applied to each subgroup, generating a compute row for each subgroup.
- The array in **bylistarrayp* stores the number associated with each column in the bylist. That number is determined by the column's position in the select statement. For example, if a column were the third item in the select statement, it would be listed as the number 3 in the array.
- To process compute row data, an Open Server application must:
  a Call srv_alt_header to define a compute row identifier.
  b Call srv_alt_descfmt for each column to describe the format the column data is in when the client receives it.
  c Call srv_alt_bind for each column to bind the data to a local program variable.
  d Call srv_alt_xferdata to send the row to the client, once each column in the compute row has been described and its data bound to a program variable.

See also

srv_alt_bind, srv_alt_descfmt, srv_alt_xferdata

srv_alt_xferdata

Description

Send a compute row to a client.

Syntax

```
CS_RETCODE srv_alt_xferdata(spp, altid)
SRV_PROC    *spp;
CS_INT      altid;
```

Parameters

spp

A pointer to an internal thread control structure.

altid

The unique identifier for the compute row being sent to the client. The *altid* is defined using srv_alt_header.

Return value

Table 3-9: Return values (srv_alt_xferdata)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include          <ospublic.h>
/*
 ** Local Prototype.
 */

CS_RETCODE      ex_srv_alt_xferdata PROTOTYPE((
 SRV_PROC        *spp,
 CS_INT          altid
 ));

/*
 ** EX_SRV_ALTXFERDATA
 **
**     Example routine to send a compute row the the client using
 **     srv_altxferdata.

**
** Arguments:
**     spp     A pointer to an internal thread control structure.
**  altid      The compute row identifier (defined using
 **             srv_alt_header).
 **
 ** Returns:
 **
 **    CS_SUCCEED    The row was sent to the client.
**    CS_FAIL       An error was detected.
```

```
 */
 CS_RETCODE          ex_srv_alt_xferdata(spp, altid)
SRV_PROC        *spp;
CS_INT          altid;
{
     /*
     ** Send the compute row to the client.
     */
     if (srv_alt_xferdata(spp, altid) != CS_SUCCEED)
     {
          return (CS_FAIL);
     }
     return (CS_SUCCEED);
}
```

Usage

- Only applications that mimic Adaptive Server's feature of returning compute row information will need to call srv_alt_xferdata. It is most useful to applications acting as a gateway to an Adaptive Server.
- srv_alt_xferdata sends a compute row to the client. It is called once for each *altid*.
- To process compute row data, an Open Server application must:
  a Call srv_alt_header to define a compute row identifier.
  b Call srv_alt_descfmt for each column to describe the format the column data is in when the client receives it.
  c Call srv_alt_bind for each column to bind the data to a local program variable.
  d Call srv_alt_xferdata to send the row to the client, once each column in the compute row has been described and its data bound to a program variable.
- All compute rows must be sent to the client before sending the completion status with srv_senddone.

See also srv_alt_bind, srv_alt_header, srv_alt_descfmt

srv_bind

Description Describe and bind a program variable for a column or parameter.

Syntax CS_RETCODE srv_bind(spp, cmd, type, item, osfmtp,

```
                varaddrp, varlenp, indp)
SRV_PROC        *spp;
CS_INT          cmd;
CS_INT          type;
CS_INT          item;
CS_DATAFMT      *osfmtp;
CS_BYTE         *varaddrp;
CS_INT          *varlenp;
CS_SMALLINT     *indp;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

cmd indicates whether the program variable stores data going out to a client or coming in from a client. The following table describes the legal values for *cmd*:

Table 3-10: Values for cmd (srv_bind)

Value	Description
CS_SET	Data in the **varaddrp* is sent to a client when srv_xferdata is called.
CS_GET	**varaddrp* is initialized with data from a client after a call to srv_xferdata.

type

The type of data stored into or read from the program variable. Table 3-11 describes the legal values for *type*:

Table 3-11: Values for type (srv_bind)

Type	Valid cmd	Description of data
SRV_RPCDATA	CS_SET or CS_GET	RPC or stored procedure parameter
SRV_ROWDATA	CS_SET only	Result row column
SRV_CURDATA	CS_GET only	Cursor parameter
SRV_KEYDATA	CS_GET only	Cursor key column
SRV_ERRORDATA	CS_SET only	Error message parameter
SRV_DYNAMICDATA	CS_SET or CS_GET	Dynamic SQL parameter
SRV_NEGDATA	CS_SET or CS_GET	Negotiated login parameter
SRV_MSGDATA	CS_SET or CS_GET	Message parameter
SRV_LANGDATA	CS_GET only	Language parameter

item

The column or parameter number. Column and parameter numbers start at 1.

osfmtp

A pointer to a CS_DATAFMT structure. This structure describes the format of the data stored in **varaddrp*.

varaddrp

A pointer to the program variable to which the column or parameter data is bound.

varlenp

A pointer to the length of *varaddrp*. Its precise meaning and characteristics differ depending on the value of *cmd*. Table 3-12 summarizes the legal values for *varlenp*:

Table 3-12: Values for varlenp (srv_bind)

If cmd is	Then varlenp
CS_SET (data going out to client)	• Cannot be NULL • Points to the actual length of the data in **varaddrp* • Need not be valid until srv_xferdata is called
CS_GET (data coming in from client)	• Can be NULL (indicating that the Open Server application al&ready knows the length of the data) • Is a pointer to the program variable in which Open Server places the actual length of the data • Is filled in after a call to srv_xferdata

When retrieving data, **varlenp* is empty until the application calls srv_xferdata. Open Server then fills the buffer with the length of the newly received value. When sending data, an application fills in **varlenp* points before calling srv_xferdata to send the data.

indp

A pointer to a buffer containing a null value indicator. Table 3-13 lists the legal values for **indp*:

Table 3-13: Values for indp (srv_bind)

Value	Indicates
CS_NULLDATA	Column or parameter data is null.
CS_GOODDATA	Column or parameter data is not null.

If *indp* is NULL, the column data is assumed to be valid; that is, not null.

Return value

Table 3-14: Return values (srv_bind)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE      ex_srv_bind PROTOTYPE((
SRV_PROC                *spp,
CS_INT                  *nump,
CS_BYTE                 *namep,
CS_INT                  *lenp
));
/*
 ** EX_SRV_BIND
**
**     Example routine using srv_bind to describe and bind two
 **     program.
**     variables to receive client RPC parameters. For this
 **     example, the
**     RPC is passed an employee number, and last name. A third
 **     program.
**     variable will be bound to receive the length of the
 **     employee's name.
**     This routine is called prior to srv_xferdata, which will
 **     actually transfer the data into the program variables.
**
** Arguments:
**     spp     A pointer to an internal thread control structure.
**     nump    A Pointer to the integer to receive the employee
 **             number.
**     namep   A Pointer to the memory area to receive the
 **             employee name.
**    lenp     A Pointer to the integer to receive the length of
 **             the employee's name. (On input, points to the
 **             maximum length of the memory area available.)
**
** Returns:
**    CS_SUCCEED    Program variables were successfully bound.
**    CS_FAIL       An error was detected.
 */
CS_RETCODE     ex_srv_bind(spp, nump, namep, lenp)
```

```
SRV_PROC        *spp;
CS_INT          *nump;
CS_BYTE         *namep;
CS_INT          *lenp;
{
     CS_INT              param_no;
     CS_DATAFMT          varfmt;
     srv_bzero((CS_VOID *)&varfmt, (CS_INT)sizeof(varfmt));
     /*
     ** First, bind the integer to receive the employee number,
     ** param 1. Here, we know the length of the data, so no
     ** length pointer is required.
     */
     param_no = 1;
     varfmt.datatype = CS_INT_TYPE;
     varfmt.maxlength = (CS_INT)sizeof(CS_INT);
     if (srv_bind(spp, (CS_INT)CS_GET, (CS_INT)SRV_RPCDATA,
          param_no, &varfmt, (CS_BYTE *)nump, (CS_INT *)NULL,
          (CS_SMALLINT *)NULL) != CS_SUCCEED)
     {
          return(CS_FAIL);
     }
     /*
      ** Then, bind the character memory to receive the
      ** employee name, param 2.
      */
     param_no = 2;
     varfmt.datatype = CS_CHAR_TYPE;
     varfmt.maxlength = *lenp;
     if (srv_bind(spp, (CS_INT)CS_GET, (CS_INT)SRV_RPCDATA,
           param_no,
          &varfmt, namep, lenp, (CS_SMALLINT *)NULL) !=
           CS_SUCCEED)
     {
          return(CS_FAIL);
     }
     return(CS_SUCCEED);
}
```

Usage

- srv_bind describes the format of a row column or parameter and associates it with an application program variable.
- srv_bind must be called once for each column in a results row or parameter in a parameter stream.

- Applications that want to change local program variable addresses (*varaddrp*, *varlenp*, or *indp*) between sending rows must call srv_bind followed by srv_xferdata each time such a change occurs.
- A Server-Library application sends data to a client in two stages:

  First, it calls srv_bind with *cmd* equal to CS_SET. The parameters *varaddrp*, *varlenp*, and *indp* contain a pointer to the data being found, a pointer to its length, and a pointer to an indicator variable. At this time, Server-Library records the addresses passed in these pointer parameters.

  These values must remain valid until the application calls srv_xferdata, which is when Server-Library reads the values from those memory locations. For example, different buffers must be used when multiple data items are passed in separate calls to srv_bind.
- Error data parameters must be described (srv_descfmt), bound (srv_bind) and sent to the client (srv_xferdata) immediately after a call to srv_sendinfo and before calling srv_senddone. The type argument of the srv_descfmt, srv_bind, and srv_xferdata routines is set to SRV_ERRORDATA.
- Message data parameters must be described (srv_descfmt), bound (srv_bind), and transferred (srv_xferdata) following a call to the srv_msg routine. The type argument of the srv_descfmt, srv_bind, and srv_xferdata routines is set to SRV_MSGDATA.
- The srv_bind routine reads from (CS_GET) or sets (CS_SET) the CS_DATAFMT fields listed in the table below. All other fields are undefined for srv_bind. (Note that "osfmtp" is a pointer to the structure.

Table 3-15: CS_DATAFMT fields used (srv_bind)

Field	In CS_SET operations, it is:	In CS_GET operations, it is:
osfmtp→datatype	Datatype of application program variable	Datatype of application program variable
osfmtp→maxlength	Actual length of program variable	Maximum length of program variable
osfmtp→count	0 or 1	0 or 1
osfmtp→status	CS_CANBENULL must be set if you are sending null values.	Unused

To send a null value in a column, the *status* value of that column's CS_DATAFMT structure must have the CS_CANBENULL bit set. Refer to Table 2-9 on page 57 for possible values of *status* in the CS_DATAFMT structure.

- If the format described by *osfmtp* differs from the format of the data received from the client (*cmd* set to CS_GET), Open Server automatically converts the data to *osfmtp*. If it differs from the format in which the data is sent to the client (*cmd* set to CS_SET), Open Server automatically converts it to the client format (*clfmtp*).

See also

srv_cursor_props, srv_descfmt, srv_msg, srv_sendinfo, srv_xferdata, "CS_DATAFMT structure" on page 54, "Processing parameter and row data" on page 134.

srv_bmove

Description

Copy bytes from one memory location to another.

Syntax

CS_VOID srv_bmove(sourcep, destp, count)

```
CS_VOID    *sourcep;
CS_VOID    *destp;
CS_INT     count;
```

Parameters

sourcep

A non-null pointer to the source of the data to be copied.

destp

A non-null pointer to the destination for the data to be copied.

count

The number of bytes to copy from *sourcep* to *destp*.

Return value

None.

Examples

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_VOID          ex_srv_bmove PROTOTYPE((
CS_VOID          *src,
CS_VOID          *dest,
CS_INT           count
));

/*
** EX_SRV_BMOVE
**
```

```
**     Example routine to copy data from one area of memory to
        another.
 **
** Arguments:
**     src     - The address of the source data.
**     dest    - The address of the destination buffer.
**     count   - The number of bytes to copy.
**
** Returns:
**          Nothing.
 */
CS_VOID          ex_srv_bmove(src, dest, count)
CS_VOID          *src;
CS_VOID          *dest;
CS_INT           count;
{
     /*
     ** Call the Open Server routine that will do the
      ** actual copy.
     */
     srv_bmove(src, dest, count);

     /*
     ** All done.
     */
     return;
}
```

Usage

- srv_bmove copies *count* bytes from the memory location **sourcep* to the memory location **destp*.
- Both *sourcep* and *destp* must be valid non-null pointers or a memory fault will occur.
- Only *count* bytes are moved and no null terminator is added.

See also srv_bzero

srv_bzero

Description Set the contents of a memory location to zero.

Syntax CS_VOID srv_bzero(locationp, count)

```
CS_VOID    *locationp;
CS_INT     count;
```

Parameters

locationp
A non-null pointer to the address of the buffer to be zeroed.

count
The number of bytes at *locationp* to set to 0x00.

Return value

None.

Examples

```
#include          <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE           ex_srv_bzero PROTOTYPE((
 CS_VOID              *locationp,
 CS_INT               cnt
 ))
/*
 ** EX_SRV_BZERO
**     Example routine to set the contents of a section of memory
 **     to zero using srv_bzero
**
** Arguments:
**
**     memp          Pointer to section of memory.
**     count         Number of bytes to set to zero.
**
** Returns
**     CS_SUCCEED     Arguments were valid and srv_bzero called.
**     CS_FAIL        An error was detected.
**
 */
CS_RETCODE      ex_srv_bzero(memp, count)
CS_VOID         *memp;
CS_INT          count;
{
     /* Check arguments. */
     if(memp == (CS_VOID *)NULL)
     {
          return(CS_FAIL);
     }
     if(count < 0)
     {
          return(CS_FAIL);
```

```
        }

        /*
        ** Set the section of memory to the value 0x00.
        */
        (CS_VOID)srv_bzero(memp,count);
        return(CS_SUCCEED);
    }
```

Usage

- srv_bzero sets *count* bytes to the value 0x00 at memory location *locationp*.
- *locationp* must be a valid non-null pointer or a memory fault will occur.

See also

srv_bmove

srv_callback

Description

Install a state transition handler for a thread.

Syntax

```
CS_RETCODE srv_callback(spp, callback_type, funcp)

SRV_PROC        *spp;
CS_INT          callback_type;
CS_RETCODE      (*funcp)();
```

Parameters

spp

A pointer to an internal thread control structure.

callback_type

An integer that indicates the state transition for which the callback is being installed. Table 3-16 summarizes the legal values for *callback_type*:

Table 3-16: Values for callback_type (srv_callback)

Value	Description
SRV_C_EXIT	The thread has returned from the entry point specified in srv_spawn or is associated with a disconnected client. The handler is executed in the context of the exiting thread.
SRV_C_PROCEXEC	A registered procedure has been invoked and is about to execute. The handler executes in the context of the thread that requested the registered procedure.
SRV_C_RESUME	The thread is resuming. The handler executes in the scheduler thread's context and uses its stack.
SRV_C_SUSPEND	The thread is suspending. The handler executes in the context of the thread that is suspending and uses its stack.
SRV_C_TIMESLICE	The callback routine you install for this state transition is called when a thread has executed for a period of time (time slice) determined by the SRV_S_TIMESLICE, SRV_S_VIRTCLKRATE, and SRV_S_VIRTTIMER server properties. See srv_props on page 334 and "Properties" on page 139 for more information about these parameters.

funcp

A pointer to the function to call when the specified state transition occurs.

A callback function takes a thread pointer argument.

Return value

Table 3-17: Return values (srv_callback))

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include          <stdio.h>
#include          <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE        suspend_handler PROTOTYPE((
SRV_PROC          *srvproc
));
CS_RETCODE ex_srv_callback PROTOTYPE((
SRV_PROC          *srvproc
));

CS_RETCODE     suspend_handler(srvproc)
SRV_PROC       *srvproc;
```

```
{
     printf("Wake me when it's over...\n");
     return(CS_SUCCEED);
}

/*
** EX_SRV_CALLBACK
**
**     Example routine to install a state transition handler.
**
** Arguments:
**   srvpro  - A pointer to an internal thread control structure.
**
** Returns:
**
**     CS_SUCCEED
**     CS_FAIL
*/
CS_RETCODE          ex_srv_callback(srvproc)
SRV_PROC            *srvproc;
{
     return(srv_callback(srvproc, SRV_C_SUSPEND,
                 suspend_handler));
}
```

Usage

- Use srv_callback to specify a routine to execute when a thread passes from one state to another.
- An application calls the callback routine with a pointer to the thread that is changing states.
- Table 3-18 summarizes the value each type of callback routine should return:

Table 3-18: Valid returns for callback routines (srv_callback)

Type of callback routine	Return value	Description of return value
SRV_C_EXIT	Ignored by Open Server, but should be set to SRV_CONTINUE for the sake of future compatibility.	
SRV_C_PROCEXEC	SRV_S_INHIBIT	Cancel execution of the registered procedure.
	SRV_S_CONTINUE	Continue execution of the registered procedure.
SRV_C_RESUME	Ignored by Open Server, but should be set to SRV_CONTINUE for the sake of future compatibility.	
SRV_C_SUSPEND	Ignored by Open Server, but should be set to SRV_CONTINUE for the sake of future compatibility.	
SRV_C_TIMESLICE	SRV_CONTINUE	Continue execution uninterrupted.
	SRV_TERMINATE	Terminate the thread.
	SRV_DEBUG	Add the thread to the debug queue for subsequent examination with a debugger.

- Some callback types are not available on some platforms. You can call srv_capability to find out if a handler can be installed for a callback type on the current platform.
- To remove a callback routine installed by a previous call to srv_callback, install a null function in its place. For example, to de-install a previously SRV_C_TIMESLICE handler, issue the following command:

```
srv_callback(spp, SRV_C_TIMESLICE, NULL);
```

- Set the *funcp* argument to NULL if your application will use the callback handler for notifications only. See “Registered procedures” on page 162 for more details.

See also srv_capability, srv_props, srv_termproc

srv_capability

Description

Determine whether Open Server supports a platform-dependent service.

Syntax

```
CS_BOOL srv_capability(capability)
CS_INT          capability;
```

Parameters

capability

A constant that represents the Open Server services to test. Table 3-19 describes the legal values for *capability*:

Table 3-19: Values for capability (srv_capability)

Value	Description
SRV_C_DEBUG	srv_dbg_stack and srv_dbg_switch are supported.
SRV_C_EXIT	A callback routine can be invoked when a thread terminates.
SRV_C_RESUME	A callback routine can be invoked when a thread resumes execution.
SRV_C_PREEMPT	Preemptive scheduling is supported.
SRV_C_SELECT	srv_select is supported.
SRV_C_SUSPEND	A callback routine can be invoked when a thread is suspended.
SRV_C_TIMESLICE	A callback routine can be invoked when a thread exceeds the maximum number of clock ticks.
SRV_POLL	srv_poll is supported.

Return value

Table 3-20: Return values (srv_capability)

Returns	To indicate
CS_TRUE	Open Server supports the service.
CS_FALSE	Open Server does not support the service.

Examples

```
#include          <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE ex_srv_capability PROTOTYPE((void));
/*
 ** EX_SRV_CAPABILITY
**
**     Example routine to determine whether srv_poll is supported
 **     on this platform.
 **
** Arguments:
**     None.
```

```
**
** Returns:
**
**      CS_SUCCEED   srv_poll is supported on this platform.
**      CS_FAIL      srv_poll is not supported on this platform.
 */
CS_RETCODE           ex_srv_capability()
{
     CS_BOOL supported;
     /*
      ** Check to see whether srv_poll is supported on this
      ** platform.
      */
      supported = srv_capability(SRV_C_POLL);
     /*
      ** If "supported" is CS_TRUE, we return CS_SUCCEED, if it is
      ** CS_FALSE we return CS_FAIL.
      */
      return(supported ? CS_SUCCEED : CS_FAIL);
}
```

Usage

- srv_capability allows you to write a portable Open Server application program and still use services that are not available on all platforms.
- Open Server has two types of capabilities: platform capabilities and protocol capabilities. The srv_capability routine pertains to platform capabilities. The srv_capability_info routine pertains to protocol capabilities. See the srv_capability_info reference page, for details.

See also

srv_callback, srv_capability, srv_dbg_stack, srv_dbg_switch, srv_poll (UNIX only), srv_select (UNIX only), srv_capability_info

srv_capability_info

Description

Define or retrieve capability information on a client connection.

Syntax

```
CS_RETCODE srv_capability_info(spp, cmd, type,
        capability, valp)

SRV_PROC    *spp;
CS_INT      cmd;
CS_INT      type;
CS_INT      capability;
CS_VOID     *valp;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

Indicates whether the Open Server application is defining or retrieving the capability information. Table 3-21 describes the legal values for *cmd*:

Table 3-21: Values for cmd (srv_capability_info)

Value	Meaning
CS_SET	The Open Server application is defining capability information.
CS_GET	The Open Server application is retrieving capability information from the client.

type

The capability group type. Table 3-22 summarizes the two legal types:

Table 3-22: Values for type (srv_capability_info)

Value	Meaning
CS_CAP_REQUEST	The possible commands a client may want to send.
CS_CAP_RESPONSE	The possible responses a client may want an Open Server application to withhold.

capability

Specifies the capability item of interest. To set or get the bitmap for all capability items in a *type* category, set *capability* to CS_ALL_CAPS. See "Capabilities" on page 24 for a list of all request and response capabilities.

valp

A pointer to a program variable. When sending information to a client (CS_SET), the application sets the capability value in this variable. When retrieving information from a client, (CS_GET), Open Server places the capability value in this variable. *valp* should be a CS_BOOL pointer when the application is defining or retrieving individual capability items, and a CS_CAP_TYPE pointer when the application is defining or retrieving the full bitmap for all capability items (that is, *capability* is CS_ALL_CAPS).

Return value

Table 3-23: Return values (srv_capability_info)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <ospublic.h>
CS_RETCODE     ex_srv_capability_info PROTOTYPE((
```

```
 SRV_PROC       *spp
));
/*
** EX_SRV_CAPABILITY_INFO
**
**     Example routine to retrieve and define capability
 **     information on a client connection.
**
**     This routine must called in the context of the connect
 **     handler, so that it is legal to negotiate capabilities.
**
** Arguments:
**     spp  A pointer to an internal thread control structure.
**
** Returns:
**      CS_SUCCEED - Successfully retrieved and bound capability
 **                  information.
**      CS_FAIL - An error was detected.
**
*/
CS_RETCODE              ex_srv_capability_info(spp)
SRV_PROC                *spp;

{
     CS_RETCODE       retval;  /* Return value from Open */
                               /* Server API calls. */

     CS_CAP_TYPE      capabilities;  /* Our bit mask. */

     CS_BOOL     value; /* Set to CS_TRUE or CS_FALSE */
                        /* for individual capabilities. */

     /*
     ** In this example, we don't want to support text or image,
      ** so we'll see first if the client has requested this.
      ** We'll do this by getting the entire bit mask.
     */
     retval = srv_capability_info(spp, CS_GET, CS_CAP_REQUEST,
                          CS_ALL_CAPS, (CS_VOID *)&capabilities);

     if (retval == CS_FAIL)

     {
          return (CS_FAIL);
     }
```

```
        /*
        ** Turn off text and image.
        **
        ** The other way to do this is to just clear the
         ** CS_DATA_TEXT and CS_DATA_IMAGE bits in the capabilities
         ** bit mask, and then call srv_capability_info() with
         ** CS_ALL_CAPS for the "type" parameter and the altered
         ** bit mask as the value.
        */
        if (CS_TST_CAPMASK(&capabilities, CS_DATA_TEXT) == CS_TRUE)
        {
             value = CS_FALSE;
             retval = srv_capability_info(spp, CS_SET,
                   CS_CAP_REQUEST, CS_DATA_TEXT, (CS_VOID *)&value);
             if (retval == CS_FAIL)
             {
                  return (CS_FAIL);
             }
        }

        if (CS_TST_CAPMASK(&capabilities, CS_DATA_IMAGE) == CS_TRUE)
        {
             value = CS_FALSE;
             retval = srv_capability_info(spp, CS_SET,
                        CS_CAP_REQUEST, CS_DATA_IMAGE, (CS_VOID*)
                        &value);
             if (retval == CS_FAIL)
             {
                  return (CS_FAIL);
             }
        }

        return (CS_SUCCEED);
   }
```

Usage

- An Open Server application and a client must agree on what requests the client can issue and what responses the Open Server application will return. A client/server connection's capabilities determine the types of client requests and server responses permitted for that connection.
- Open Server assigns a default set of capabilities for all connections. An Open Server application that does not want the default set of capabilities to apply to a given connection can call srv_capability_info to negotiate explicitly a different set of capabilities.

- See “Capabilities” on page 24 for a list of the default set of requests and response capabilities.

Note Response capabilities indicate the kinds of responses the client does *not* want to receive.

- Open Server has two types of capabilities: platform capabilities and protocol capabilities. The srv_capability routine pertains to platform capabilities. The srv_capability_info routine pertains to protocol capabilities. For more information on srv_capability, see srv_capability.

See also

srv_capability, srv_props, “Capabilities” on page 24, “Properties” on page 139

srv_createmsgq

Description

Create a message queue.

Syntax

```
CS_RETCODE srv_createmsgq(msgqnamep, msgq_namelen,
        msgqidp)
CS_CHAR     *msgqnamep;
CS_INT      msgqname_len;
SRV_OBJID   *msgqidp;
```

Parameters

msgqnamep

A pointer to the name of the queue to create. It is an error to attempt to create a queue that al&ready exists.

msgqname_len

The length of the name in **msgqnamep*. If the name is null terminated, an application can set *msgqname_len* to CS_NULLTERM. A message queue can be up to SRV_MAXNAME characters long.

msgqidp

Open Server returns the ID of the newly created message queue in **msgqidp*.

Return value

Table 3-24: Return values (srv_createmsgq)

Returns:	To indicate:
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE      ex_srv_createmsgq PROTOTYPE((
     SRV_OBJID     *msgqp,
     CS_CHAR       *msgqnm
));

/*
** EX_SRV_CREATEMSGQ
**
**     Example routine to create an Open Server message queue
 **     using srv_createmsgq.
**
** Arguments:
**     msgqp   Return pointer to the created message queue
 **             identifier.
**     msgqn   Null terminated name for the created queue.
**
** Returns:
**  CS_SUCCEED  Message queue with given name successfully
                created.
**  CS_FAIL     An error was detected.
*/
CS_RETCODE            ex_srv_createmsgq(msgqp, msgqnm)
SRV_OBJID             *msgqp;
CS_CHAR               *msgqnm;
{
     /* Check parameters. */
     if ((CS_INT)strlen(msgqnm) > SRV_MAXNAME)
     {
          return(CS_FAIL);
     }

     /* Create the message queue. */
     if (srv_createmsgq(msgqnm, (CS_INT)CS_NULLTERM, msgqp) !=
           CS_SUCCEED)
```

```
        {
             return(CS_FAIL);
        }
        return(CS_SUCCEED);
    }
```

Usage

- When creating a message queue, an application must assign it a name. Once a message queue has been created, an application can reference it either by name or by ID.
- Given the ID of a message queue, use srv_getobjname to look up the name.
- SRV_OBJID is defined as a CS_INT.
- The SRV_S_NUMMSGQUEUES server property determines the number of message queues available to an Open Server application. Refer to "Server properties" on page 141 for more information.
- The SRV_S_MSGPOOL server property determines the number of messages available to an Open Server application at runtime. Refer to "Server properties" on page 141 for more information.

See also srv_deletemsgq, srv_getmsgq, srv_getobjname, srv_putmsgq

srv_createmutex

Description Create a mutual exclusion semaphore.

Syntax

```
CS_RETCODE srv_createmutex(mutex_namep, mutex_namelen,
        mutex_idp)

CS_CHAR     *mutex_namep;
CS_INT      mutex_namelen;
SRV_OBJID   *mutex_idp;
```

Parameters

mutex_namep
A pointer to the name of the mutex to create.

mutex_namelen
The length of the name in **mutex_namep*. If the string is null terminated, an application can set *mutex_namelen* to CS_NULLTERM.

mutex_idp
Open Server returns the ID of the new mutex in the **mutex_idp*.

Return value

Table 3-25: Return values (srv_createmutex)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include     <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE               ex_srv_createmutex PROTOTYPE((
CS_CHAR           *name,
CS_INT            namelen,
SRV_OBJID         *idp
));

/*
** EX_SRV_CREATEMUTEX
**
**     Example routine to create an Open Server mutex.
**
** Arguments:
**
**     name      The name of the mutex to create.
**     namelen   The length of name.
**     idp       The address of a SRV_OBJID, which will be set
 **               to the unique identifier for the created mutex.
**
** Returns:
**          CS_SUCCEED      The mutex was created successfuly.
**          CS_FAIL         An error was detected.
*/
CS_RETCODE           ex_srv_createmutex(name, namelen, idp)
CS_CHAR              *name;
CS_INT               namelen;
SRV_OBJID            *idp;
{
     /*
     ** Call the Open Server routine that will create
      ** the mutex.
     */
     if( srv_createmutex(name, namelen, idp) == CS_FAIL )
     {
          /*
          ** An error was al&ready raised.
```

```
            */
            return CS_FAIL;
        }

        /*
        ** All done.
        */
        return CS_SUCCEED;
    }
```

Usage

- When creating a mutex, an application must assign it a name. Once a mutex has been created, the application can reference it either by name or by ID.
- If you have the ID of a mutex, you can use srv_getobjname to look up the name.
- Creating a mutex does not grant a lock to its creator. Use srv_lockmutex to lock it once a mutex has been created.
- SRV_OBJID is defined as a CS_INT.

See also srv_deletemutex, srv_getobjname, srv_lockmutex, srv_unlockmutex

srv_createproc

Description Create a non-client, event-driven thread.

Syntax

```
SRV_PROC    *srv_createproc(ssp)
SRV_SERVER  *ssp;
```

Parameters

ssp
A pointer to the Open Server state information control structure.

Return value If successful, srv_createproc returns a pointer to the new thread control structure. If unsuccessful, srv_createproc returns a NULL thread pointer, and Open Server raises an error.

Table 3-26: Return values (srv_createproc)

Returns	To indicate
A pointer to the new thread control structure	Open Server created the thread.
A null thread pointer	Open Server could not create the thread. Open Server raises an error.

Examples

```
#include          <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE               ex_srv_creatp     PROTOTYPE((
SRV_SERVER               *ssp,
SRV_PROC                 *newsp
));
/*
** EX_SRV_CREATP
**     Example routine to create a non-client, event driven
 **     thread.
**
** Arguments:
**
**     ssp     A pointer to the Open Server state information
 **             control structure.
**    newsp    A pointer that will be returned by srv_createproc
 **             and point to the new thread control structure.
**
** Returns
**
**     CS_SUCCEED            Thread was created.
**     CS_FAIL               An error was detected.
**
 */
CS_RETCODE             ex_srv_creatp(ssp, newsp)
SRV_SERVER             *ssp;
SRV_PROC               *newsp;
{
     /* Check arguments. */
     if(ssp == (SRV_SERVER *)0)
          return(CS_FAIL);

     /*
     ** Create the new thread
     */
```

```
        newsp = srv_createproc(ssp);
        if(newsp == (SRV_PROC *)NULL)
             return(CS_FAIL);
        return(CS_SUCCEED);
    }
```

Usage

- srv_createproc creates a thread that is driven by programmer-defined events raised by srv_event or srv_event_deferred.
- Non-client threads receive only programmer-defined events. They never receive client-generated events.
- Use srv_termproc to terminate a thread created with srv_createproc.
- Non-client threads have no client I/O. Calling srv_thread_props with the property argument set to (SRV_T_IODEAD) always returns CS_FALSE for a non-client thread.

See also srv_event,srv_event_deferred, srv_spawn, srv_termproc, srv_thread_props

srv_cursor_props

Description Retrieve or set information about the current cursor.

Syntax CS_RETCODE srv_cursor_props(spp, cmd, cdp)

```
SRV_PROC        *spp;
CS_INT          cmd;
SRV_CURDESC     *cdp;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

Indicates whether srv_cursor_props sends cursor information to the client or retrieves cursor information from the client. The following table describes the legal values for *cmd*:

Table 3-27: Values for cmd (srv_cursor_props)

Value	Description
CS_SET	srv_cursor_props sends information about the current cursor to the client.
CS_GET	srv_cursor_props retrieves information about the current cursor command from the client.

cdp

A pointer to a SRV_CURDESC structure. When the application is setting cursor information, the SRV_CURDESC structure describes the current cursor. When the application is retrieving information, Open Server updates the SRV_CURDESC structure with information about the current cursor. Various fields are set or filled in at various times, depending on the current cursor command. For an explanation of each field in *cdp* and how and when they are filled in, see "SRV_CURDESC structure" on page 65.

Return value

Table 3-28: Return values (srv_cursor_props)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include          <ospublic.h>
/*
 ** Local Prototype.
 */
extern CS_RETCODE     ex_srv_cursor_props PROTOTYPE((
CS_VOID               *spp
));
/*
 ** EX_SRV_CURSOR_PROPS
**
**     Example routine to retrieve information on the current
 **     cursor.
** Arguments:
**  spp  Apointer to an internal control structure.
**
** Returns:
**
**  CS_SUCCEED     Cursor information was retrieved successfully.
**  CS_FAIL        An error was detected.
*/
CS_RETCODE         ex_srv_cursor_props(spp)
SRV_PROC           *spp;
```

```
{
        SRV_CURDESC   curdesc;

        if(srv_cursor_props(spp, CS_GET, &curdesc) == CS_FAIL)
        {
             return(CS_FAIL);
        }
        return(CS_SUCCEED);
}
```

Usage

- An Open Server application uses srv_cursor_props to exchange active cursor information with the client.
- The client always initiates this exchange by issuing a cursor command. The client, therefore, specifies the current cursor.
- An application can only call srv_cursor_props from inside a SRV_CURSOR event handler.
- Open Server generates a SRV_CURSOR event in response to each cursor command received from a client. An application's SRV_CURSOR event handler can then call srv_cursor_props with *cmd* set to CS_GET to determine the current cursor and the type of cursor command received. It can then decide how to respond. For a description of valid cursor command types and legal responses, see "Cursors" on page 63.
- Each cursor command provokes a distinct response from an Open Server application. The application pulls information from the SRV_CURDESC structure (the requested fetch count, for example), makes decisions based on that data, and then sets information in the structure and sends it back to the client using srv_cursor_props. An application can also read in parameters, or send back result rows and parameters, depending on the circumstances.
- The SRV_CURSOR event handler must acknowledge all cursor commands except fetch, update, and delete by sending back a cursor information command. The handler sets the *curcmd* field in the SRV_CURDESC structure to CS_CURSOR_INFO and then calls srv_cursor_props with cmd set to CS_SET. This is the very first piece of information the handler sends back.
- In response to a CURSOR_DECLARE command, an Open Server application chooses a cursor ID to uniquely identify the current cursor. The application then sends the cursor ID back to the client by calling srv_cursor_props with *cmd* set to CS_SET. The client and Open Server application subsequently refer to the current cursor by its ID rather than its name.

See also srv_bind, srv_descfmt, srv_numparams, srv_xferdata, “Cursors” on page 63

srv_dbg_stack

Description Display the call stack of a thread.

Syntax

```
CS_RETCODE srv_dbg_stack(spp, depth, funcp)

SRV_PROC        *spp;
CS_INT          depth;
CS_RETCODE      (*funcp)();
```

Parameters

spp

A pointer to an internal thread control structure.

depth

The maximum number of call stack levels to display. If *depth* is -1, all levels are displayed.

funcp

A pointer to a function that you provide to process each line of the call stack display. Your function is called with a pointer to a null terminated string and an integer that is the length of the string. The string contains the program counter and the routine’s parameters formatted in hexadecimal. If your function returns CS_FAIL, the stack trace is terminated. If it returns anything else, the stack trace continues until all of the routines on the call stack are processed or until *depth* stack frames are processed. If *funcp* is NULL, Open Server writes the call stack contents to *stderr*.

The following is a typical implementation for a function:

```
CS_RETCODE callstack_display(linebuf, length)
 CS_CHAR  *linebuf;
 CS_INT   length;
 {
      /*
      ** Output each line of the stack trace to stderr.
      */
      fprintf(stderr,"%s\n", linebuf);
      return(CS_SUCCEED);
 }
```

Return value

Table 3-29: Return values (srv_dbg_stack)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <ospublic.h>
/*
** Local prototype.
*/
CS_RETCODE      ex_srv_dbg_stack PROTOTYPE((
SRV_PROC        *spp
));

/*
** EX_SRV_DBG_STACK
**
**     Example routine to display the call stack of a thread.
**
** Arguments:
**     spp  -  A pointer to an internal thread control structure.
**
** Returns:
**     CS_SUCCEED    Call stack successfully displayed.
**     CS_FAIL       An error was detected.
**
*/
CS_RETCODE           ex_srv_dbg_stack(spp)
SRV_PROC       *spp;
{
     CS_RETCODE retval;

     retval = srv_dbg_stack(spp, -1, (CS_RETCODE(*)())NULL);

     return (retval);
}
```

Usage

- srv_dbg_stack is not available on all platforms. Use srv_capability to determine if it is available on the current platform.
- srv_dbg_stack allows you to examine the call stack of a thread during debugging or when handling execution errors. It can be called from a debugger or from the running application.
- A typical use for srv_dbg_stack is to record the stack frame in the error log when a serious error occurs.

- Each routine on the call stack is formatted into a string consisting of the program counter, in hexadecimal, followed by each parameter, also in hexadecimal. You will need a load map of the executable to translate the program counter to a function name.
- If called to display the stack of the currently running thread, srv_dbg_stack and the routines it calls appear at the top of the stack.

See also

srv_capability, srv_dbg_switch

srv_dbg_switch

Description

Temporarily restore another thread context for debugging.

Syntax

```
CS_RETCODE srv_dbg_switch(spid)
CS_INT     spid;
```

Parameters

spid

The server process ID (spid) of the thread whose context should be temporarily restored.

Return value

Table 3-30: Return values (srv_dbg_switch)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Usage

- srv_dbg_switch is not available on all platforms. Use srv_capability to determine whether a platform supports srv_dbg_switch.
- Once a thread context is switched, continuing execution of the application restores the original thread context and the application continues to run normally.
- The thread whose context has been restored is not runnable. It can only be examined.
- On UNIX systems, do not call srv_dbg_switch from within system service routines. If you do, a SIGTRAP signal is raised and the program terminates.
- The *spid* can be obtained by calling srv_thread_props with the property argument set to SRV_T_SPID. It is an error to attempt to restore the context for the currently running thread.

See also srv_capability, srv_dbg_stack

srv_define_event

Description Define a user event.

Syntax int srv_define_event(ssp, type, namep, namelen)

```
SRV_SERVER    *ssp;
CS_INT         type;
CS_CHAR       *namep;
CS_INT         namelen;
```

Parameters

ssp
A pointer to the Open Server control structure.

type
The type of event. Currently, programmer-defined events must be of type SRV_EQUEUED.

namep
A pointer to the name of the event.

namelen
The length, in bytes, of string in **namep*. If the string is null terminated, *namelen* can be CS_NULLTERM.

Return value

Table 3-31: Return values (srv_define_event)

Returns	To indicate
A non-zero integer	The unique id for the vent.
0	Open Server cannot define the event. Open Server raises an error.

Examples

```
#include          <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE      ex_srv_define_event PROTOTYPE((
CS_CHAR        *namep,
CS_INT         namelen,
CS_INT         *event_no
));
/*
```

```
 ** EX_SRV_DEFINE_EVENT
**
**   Example routine to illustrate the use of srv_define_event to
 **   define an user event.
**
** Arguments:
**     namep   A pointer to the name of event.
**     namelen The length, in bytes, of string in *namep.
**     event_no  A CS_INT pointer that is initialized with
 **               the unique number for the event.
** Returns:
**
**  CS_SUCCEED   If the event was defined successfully.
**  CS_FAIL      An error was detected.
 */
CS_RETCODE     ex_srv_define_event(namep, namelen, event_no)
CS_CONTEXT *cp;
 CS_VOID        *bufp;
 CS_CHAR        *namep;
 CS_INT         namelen;
 CS_INT         *event_no;
 CS_INT         result;
{
     SRV_PROC *srvproc_ptr; /* A pointer to an internal thread
                             ** control structure */

     result = srv_props(cp, CS_GET, SRV_S_CURTHREAD,
            bufp, sizeof(CS_INT));
     if (result == CS_FAIL)
      {
          return (CS_FAIL);
      }
     /* Now define the event. */
     if ((*event_no = srv_define_event(srvproc_ptr, SRV_EQUEUED,
           namep, namelen)) == (CS_INT)0)
          return (CS_FAIL);
     return (CS_SUCCEED);
}
```

Usage

- Programmer-defined events are triggered by calling srv_event rather than by client actions. The Open Server programmer provides a handler routine that executes when the event is triggered.
- Event handlers for programmer-defined events are installed in the usual way, with srv_handle.
- Handlers for programmer-defined events receive a pointer to the thread control structure for the thread that received the event.

- Events cannot be defined unless the Open Server application has been configured to allow programmer-defined events. For details, see the srv_props reference page.

See also srv_event, srv_event_deferred, srv_handle, srv_props, “Events” on page 92

srv_deletemsgq

Description Delete a message queue.

Syntax

```
CS_RETCODE srv_deletemsgq(msgqnamep, msgqname_len,
        msgqid)
CS_CHAR     *msgqnamep;
CS_INT      msgqname_len;
SRV_OBJID   msgqid;
```

Parameters

msgqnamep
A pointer to the name of the message queue to delete. It is an error to attempt to delete a message queue that does not exist.

msgqname_len
The length of the name pointed to by msgqname. If the name is null terminated, *msgqname_len* can be set to CS_NULLTERM.

msgqid
A SRV_OBJID that specifies the identifier of message queue to delete.

Return value

Table 3-32: Return values (srv_deletemsgq)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include          <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE      ex_srv_deletemsgq PROTOTYPE((
CS_CHAR         *msgqname,
CS_INT          msgqname_len,
SRV_OBJID msgqid
```

```
));
/*
 ** EX_SRV_DELETEMSGQ
**
**     Example routine using srv_deletemsgq to delete an Open
 **     Server message queue previously create by srv_createmsgq.
** This routine can be passed a value message queue name, or
 ** NULL,in which case the message queue identifier will be used.
**
** Arguments:
**  msgqname        The name of the message queue to delete. If
 **                  NULL, the msgqid is used.
 **  msgqname_len    The length of the name to which msgqname
 **                  points.
 **  msgqid          A SRV_OBJID that specifies the identifier of
 **                  the message queue to delete.
 **
** Returns:
**
**    CS_SUCCEED    The message queue was successfully deleted.
**    CS_FAIL       An error was detected.
 */
CS_RETCODE      ex_srv_deletemsgq(msgqname, msgqname_len, msgqid)
CS_CHAR         *msgqname;
CS_INT          msgqname_len;
SRV_OBJID       msgqid;
{
     /*
      ** Delete a message queue.
      */
     if (srv_deletemsgq(msgqname, msgqname_len, msgqid) !=
           CS_SUCCEED)
     {
          return(CS_FAIL);
     }
     return(CS_SUCCEED);
}
```

Usage

- Message queues can be deleted by either name or ID. If *msgqname* is not NULL, the message queue name is used; otherwise, the message queue ID is used.
- Unread messages in the queue are flushed before the queue is deleted. Threads waiting in srv_putmsgq wake up. srv_putmsgq returns CS_FAIL.

- When a message queue is deleted, threads waiting for messages from the queue wake up with a CS_FAIL return value from srv_getmsgq, and srv_getmsgq's *infop* argument is set to SRV_I_DELETED.
- The SRV_S_NUMMSGQUEUES server property determines the number of message queues available to an Open Server application. Refer to "Server properties" on page 141 for more information.
- The SRV_S_MSGPOOL server property determines the number of messages available to an Open Server application at runtime. Refer to "Server properties" on page 141 for more information.

See also

srv_createmsgq, srv_getmsgq, srv_getobjname, srv_putmsgq

srv_deletemutex

Description

Delete a mutex created by srv_createmutex.

Syntax

```
CS_RETCODE srv_deletemutex(mutex_namep, mutex_namelen,
        mutex_id)
CS_CHAR     *mutex_namep;
CS_INT      mutex_namelen;
SRV_OBJID   mutex_id;
```

Parameters

mutex_namep

A pointer to the name associated with the mutex when it was created.

mutex_namelen

The length, in bytes, of the *mutex_namep*. If the string is null terminated, *mutex_namelen* can be set to CS_NULLTERM.

mutex_id

The unique identifier returned by srv_createmutex.

Return value

Table 3-33: Return values (srv_deletemutex)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <ospublic.h>
/*
 ** Local Prototype.
```

```
 */
CS_RETCODE    ex_srv_deletemutex PROTOTYPE((
CS_CHAR         *mtxnm,
SRV_OBJID       mtxid
));
/*
 ** EX_SRV_DELETEMUTEX
**     Example routine using srv_deletemutex to delete an
 **     Open Server mutex previously created by srv_createmutex.
 **     This routine can be passed a mutex name, or NULL,
 **     in which case the mutex identifier will be used.
** Arguments:
**     mtxnm     Null terminated mutex name, or NULL to use mutex
 **               id.
 **    mtxid      Mutex identifier (valid only if mtxnm is NULL).
** Returns:
**   CS_SUCCEED     mutex was successfully queued for deletion.
 **   CS_FAIL        An error was detected.
 */
CS_RETCODE              ex_srv_deletemutex(mtxnm, mtxid)
CS_CHAR                 *mtxnm;
SRV_OBJID               mtxid;
{
     /* Delete the mutex. */
      if (srv_deletemutex(mtxnm, (CS_INT)CS_NULLTERM, mtxid) !=
           CS_SUCCEED)
     {
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

Usage

- The mutex to delete can be referenced by its name or ID. If *mutex_namep* is not NULL, the name is used; otherwise, the ID is used.
- A mutex is not deleted until other threads waiting to lock the mutex have had their requests satisfied and have released their locks.
- An example of the use of mutexes appears on the srv_createmutex reference page.

See also

srv_createmutex, srv_getobjid, srv_getobjname, srv_lockmutex

srv_descfmt

Description

Describe or retrieve the description of a column or parameter going to or coming from a client.

Syntax

```
CS_RETCODE srv_descfmt(spp, cmd, type, item,
        clfmtp)
SRV_PROC      *spp;
CS_INT        cmd;
CS_INT        type;
CS_INT        item;
CS_DATAFMT    *clfmtp;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

Indicates whether srv_descfmt describes data being sent to the client or retrieves a description of data received from the client. Table 3-34 describes the legal values for *cmd*:

Table 3-34: Values for cmd (srv_descfmt)

Value	Description
CS_SET	srv_descfmt describes the format the data will be in when the client receives it.
CS_GET	srv_descfmt retrieves the format the data was in when the client sent it.

type

If *cmd* is CS_SET, the type of data being described. If *cmd* is CS_GET, the type of data being retrieved. Table 3-35 describes the valid types and their appropriate context:

Table 3-35: Values for type (srv_descfmt)

Type	Permissible settings for cmd	Description
SRV_RPCDATA	CS_SET or CS_GET	RPC or stored procedure parameters
SRV_ROWDATA	CS_SET only	Row data
SRV_CURDATA	CS_GET only	Cursor parameters
SRV_UPCOLDATA	CS_GET only	Cursor update columns
SRV_KEYDATA	CS_GET only	Cursor key data
SRV_ERRORDATA	CS_SET only	Extended error data
SRV_DYNDATA	CS_SET or CS_GET	Dynamic SQL data
SRV_NEGDATA	CS_SET or CS_GET	Negotiated login data
SRV_MSGDATA	CS_SET or CS_GET	MSG parameters
SRV_LANGDATA	CS_GET only	Language parameters

item

The parameter or column number. Parameter and column numbers start at 1.

clfmtp

A pointer to a CS_DATAFMT structure containing a description of the data.

Return value

Table 3-36: Return values (srv_descfmt)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE      ex_srv_descfmt PROTOTYPE((
SRV_PROC        *spp,
CS_INT          item,
CS_DATAFMT      *dp
));

/*
** EX_SRV_DESCFMT
**
**     Example routine used to get an RPC parameter description.
```

```
**
** Arguments:
**
**      spp          A pointer to an internal thread control
                      structure.
**      item         The parameter number we're looking for.
**      dp           The address of a CS_DATAFMT to be filled with
 **                   the parameter's description.
**
** Returns:
**          CS_SUCCEED if the description was obtained, or
 **          CS_FAIL if an error was detected.
*/
CS_RETCODE      ex_srv_descfmt(sp, item, dp)
SRV_PROC        *sp;
CS_INT          item;
CS_DATAFMT      *dp;
{
     /*
     ** Call srv_descfmt to get the RPC parameter description.
     */
     if( srv_descfmt(sp, CS_GET, SRV_RPCDATA, item, dp) ==
           CS_FAIL )
     {
          /*
          ** An error was al&ready raised.
          */
          return CS_FAIL;
     }

     /*
     ** All done.
     */
     return CS_SUCCEED;
}
```

Usage

- srv_descfmt describes the format of a variety of kinds of columns and parameter. See "CS_DATAFMT structure" on page 54 for details.
- When sending rows or parameters to the client (CS_SET), you must call srv_descfmt to describe how the data will look to the client. When receiving parameters from the client (CS_GET), call srv_descfmt to retrieve a description of the format the data was in when the client sent it. A gateway application may want to save this client format information to pass it on to the remote server.

- The srv_descfmt routine reads from (CS_GET) or sets (CS_SET) the CS_DATAFMT fields listed in the table below. All other fields are undefined for srv_descfmt. (Note that "clfmtp" is a pointer to the structure.

Table 3-37: CS_DATAFMT fields used (srv_descfmt)

Field	CS_SET	CS_GET
clfmtp→ namelen	Length of name	Length of name
clfmtp→ status	Parameter/column status	Parameter status
clfmtp→ name	Parameter/column name	Parameter name
clfmtp→datatype	Remote datatype set here	Remote datatype retrieved from here
clfmtp→maxlength	Maximum length of remote datatype set here	Maximum length of remote datatype retrieved from here
clfmtp→format	Remote datatype format	Remote datatype formats

- If the format described in the CS_DATAFMT structure (*clfmtp*) differs from the format described in the subsequent call to srv_bind (*osfmtp*), Open Server automatically converts to the client format (*clfmtp*) when *cmd* is CS_SET or the application format (*osfmtp*) when *cmd* is CS_GET.
- Once each column or parameter in the datastream has been described and bound, call srv_xferdata to send the data in the program variable to the client or update the program variable with data from the client.
- SRV_NEGDATA parameters can be sent or received as part of a negotiated login operation, after srv_negotiate has returned successfully.
- Key column numbers correspond to their number in the row.

See also

srv_bind, srv_cursor_props,srv_dynamic, srv_msg, srv_negotiate, srv_numparams, srv_sendinfo, srv_xferdata, "CS_DATAFMT structure" on page 54

srv_dynamic

Description

Read or respond to a client dynamic SQL command.

Syntax

```
CS_RETCODE srv_dynamic(spp, cmd, item, bufp,
                       buflen, outlenp)

SRV_PROC      *spp;
CS_INT         cmd;
CS_INT         item;
```

```
CS_VOID         *bufp
CS_INT          buflen;
CS_INT          *outlenp
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

Indicates whether a dynamic command is being read from or sent to a client. Table 3-38 describes the legal values for *cmd*:

Table 3-38: Values for cmd (srv_dynamic)

Value	Description
CS_SET	srv_dynamic is sending a response to a dynamic command back to a client.
CS_GET	srv_dynamic is reading a dynamic command from a client.

item

Indicates what kind of information is being sent or retrieved. Table 3-39 describes the legal values for *item*:

Table 3-39: Values for item (srv_dynamic)

Value	Meaning
SRV_DYN_TYPE	The type of dynamic operation being performed.
SRV_DYN_IDLEN	The length of the dynamic statement ID.
SRV_DYN_ID	The dynamic statement ID.
SRV_DYN_STMTLEN	The length of the dynamic statement.
SRV_DYN_STMT	The dynamic statement that is being prepared or executed.

bufp

A pointer to the buffer in which the *item* value is returned (CS_GET) or set (CS_SET).

buflen

The length, in bytes, of the **bufp* buffer. Table 3-40 summarizes the required buffer sizes:

Table 3-40: Required buffer sizes (srv_dynamic)

Value	Required format (size)
SRV_DYN_TYPE	sizeof(CS_INT).
SRV_DYN_IDLEN	sizeof(CS_INT).
SRV_DYN_ID	Varies. Determine length by first calling srv_dynamic with item set to CS_DYN_IDLEN and then allocate buffer size accordingly.
SRV_DYN_STMT LEN	sizeof(CS_INT).
SRV_DYN_STMT	Varies. Determine length by first calling srv_dynamic with item set to CS_DYN_STMTLEN and then allocate buffer size accordingly.

outlenp

A pointer to an integer variable which is set to the actual length of data copied into **bufp* when retrieving data from the client (*cmd* is CS_GET). This argument is not required if *cmd* is CS_SET.

Return value

Table 3-41: Return values (srv_dynamic)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include        <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE               ex_srv_dynamic PROTOTYPE((
CS_VOID     *spp,
CS_INT      *optypep
));
/*
 ** EX_SRV_DYNAMIC
**
**    Example routine to retrieve dynamic operation type from a
 **    client.
**
** Arguments:
**    spp      Thread control structure.
**    optypep  Dynamic operation type.
**
** Returns:
**
```

```
**    CS_SUCCEED   Dynamic information was retrieved
 **                 successfully.
**    CS_FAIL      An error was detected.
 */
CS_RETCODE      ex_srv_dynamic(spp, optypep)
SRV_PROC   *spp;
CS_INT     *optypep;
{
CS_INT     outlen;

    if(srv_dynamic(spp, CS_GET, SRV_DYN_TYPE, optypep,
         sizeof(*optypep), &outlen) == CS_FAIL)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

Usage

- The srv_dynamic routine allows an Open Server application to read a dynamic SQL command or send a response to such a command.
- Valid operation types (SRV_DYN_TYPE) include:

  CS_PREPARE – prepare a statement (CS_GET only).

  CS_DESCRIBE_INPUT – request input parameter formats for the current prepared statement (CS_GET only).

  CS_DESCRIBE_OUTPUT – request column formats for the current prepared statement (CS_GET only).

  CS_EXECUTE – execute a prepared statement (CS_GET only).

  CS_EXEC_IMMEDIATE – execute an unprepared statement, which has no parameters and does not return results (CS_GET only).

  CS_DEALLOC – deallocate a prepared statement (CS_GET only).

  CS_ACK – acknowledge a dynamic SQL command from client (CS_SET only).
- Each dynamic command received from a client triggers a SRV_DYNAMIC event. An Open Server application can then call srv_dynamic, in response to each client dynamic command, to retrieve and store the operation type, statement ID and statement, and then acknowledge the client communication, by issuing a srv_dynamic call with *type* set to CS_ACK.

- It is an error to call srv_dynamic in any event handler other than a SRV_DYNAMIC handler.
- CS_ACK is the only dynamic operation type that can be set (*cmd* set to CS_SET).
- CS_PREPARE, CS_DESCRIBE_INPUT, CS_DESCRIBE_OUTPUT, CS_EXECUTE, CS_EXEC_IMMEDIATE and CS_DEALLOC are the only dynamic operation types that can be retrieved (*cmd* set to CS_GET).
- Sending a full dynamic SQL response to a client requires passing the ID length, the ID, and the operation type. This requires three distinct calls to srv_dynamic. It is an error, for example, to set just the statement ID and then call srv_senddone. The only exception is if the operation type is CS_EXEC_IMMEDIATE, for which there is no associated statement ID.
- Parameter data formats and output column formats can be sent to a client, in response to a CS_PREPARE dynamic command, using srv_descfmt and srv_xferdata with a type argument of SRV_DYNDATA. Note that srv_bind is not necessary here, as the application is simply sending formats.
- An Open Server application retrieves and store the parameter data sent by a client following the CS_EXECUTE dynamic command using srv_descfmt, srv_bind, and srv_xferdata, with a type argument of SRV_DYNDATA. The application determines the number of parameters using srv_numparams.
- The application sends dynamic SQL result rows to the client, in response to a CS_EXECUTE dynamic SQL command, using srv_descfmt, srv_bind, and srv_xferdata with a type argument of SRV_ROWDATA.
- A dynamic SQL command of CS_EXEC_IMMEDIATE indicates that the client wishes to execute a statement without parameters and receive only a DONE as a result. The statement is contained in the CS_EXEC_IMMEDIATE command stream and is accessible through SRV_DYN_STMT. The statement has not been previously prepared—the statement ID length (SRV_DYN_IDLEN) will be 0—and will cease to exist once the SRV_DYNAMIC event handler has exited.

See also

srv_bind, srv_descfmt, srv_numparams, srv_xferdata, “Dynamic SQL” on page 83

srv_envchange

Description

Notify the client of an environment change.

Syntax

```
CS_RETCODE srv_envchange(spp, type, oldvalp
        oldvallen, newvalp, newvallen)

SRV_PROC    *spp;
CS_INT       type;
CS_CHAR     *oldvalp;
CS_INT       oldvallen
CS_CHAR     *newvalp;
CS_INT       newvallen
```

Parameters

spp

A pointer to an internal thread control structure.

type

The environment being changed. Currently, the only legal values are SRV_ENVDATABASE and SRV_ENVLANG, the name of the current database and the current national language, respectively.

oldvalp

A pointer to the character string containing the old value. It can be NULL. Its length in bytes is stored in *oldvallen*.

oldvallen

The length, in bytes, of the string in **oldvalp*. It can be CS_NULLTERM, which indicates that the string in **oldvalp* is null terminated. It can also be CS_UNUSED, indicating that the string in **oldvalp* is NULL.

newvalp

A pointer to the character string containing the new value of the environment variable. It can be null. Its length in bytes is stored in *newvallen*.

newvallen

The length, in bytes, of the string in **newvalp*. It can be CS_NULLTERM, which indicates that the string in *newvalp* is null terminated. It can also be CS_UNUSED, indicating that the string in **newvalp* is NULL.

Return value

Table 3-42: Return values (srv_envchange)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include   <ospublic.h>
```

```
/*
 ** Local Prototype.
 */
CS_RETCODE  ex_srv_envchange PROTOTYPE((
SRV_PROC    *spp
));
/*
 ** EX_SRV_ENVCHANGE
**
**    Example routine to notify the client of an environment
 **    change.
**
** Arguments:
**    spp    A pointer to an internal thread control structure.
**
** Returns:
**     CS_SUCCEED    Succesfully notified client of environment
 **                    change.
**     CS_FAIL       An error was detected.
**
*/
CS_RETCODE   ex_srv_envchange(spp)
SRV_PROC     *spp;
{
    CS_RETCODE   retval;
    /*
     ** Notify the client that we've changed the database
     ** from "master" to "pubs2".
     */
    retval = srv_envchange(spp, SRV_ENVDATABASE, "master",
             CS_NULLTERM, "pubs2", CS_NULLTERM);
    return (retval);
}
```

Usage

- There are various environment variables which can be set. Open Server handles some automatically, while others must be handled by an Open Server application. Currently, an application can only inform a client of a change to the current database or national language.

- Open Server calls an Open Server application's error handler any time one of the values changes. An Open Server application can change it through srv_envchange, or Open Server can change it using internal code, or both. The error number passed to the error handler is the Adaptive Server message number sent back to a client when one of these values changes. This allows a client application to match the same message number to a changing value, whether the client is connected to an Open Server or an Adaptive Server. Table 3-43 lists the message number and *oserror.h* #define that correspond to each changing value.

Table 3-43: Environment variables (srv_envchange)

Changing value	Message number	#define in oserror.h
Current Database	5701	SQLSRV_ENVDB
National Language	5703	SQLSRV_ENVLANG

srv_event

Description

Add an event request to a thread's request-handling queue.

Syntax

```
CS_INT srv_event(spp, event, datap)

SRV_PROC    *spp;
CS_INT       event;
CS_VOID     *datap;
```

Parameters

spp

A pointer to an internal thread control structure.

event

The token for the event to add to the client's event queue. See "Events" on page 92 for a list of defined events.

datap

A pointer (CS_VOID) to data supplied by the Open Server programmer. An application can retrieve the data by calling srv_thread_props with property set to SRV_T_EVENTDATA, from within the event handler.

Return value

Table 3-44: Return values (srv_event)

Returns	To indicate
The token for the requested event.	Open Server added the new event.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE   ex_srv_event PROTOTYPE((
SRV_PROC     *spp,
CS_INT       event,
CS_VOID     *datap
));
/*
 ** EX_SRV_EVENT
**
**    Example routine to queue an event request to an Open Server
 **    thread's request-handling queue.
**
**    Note that if the event is an user-defined one, it
 **    must have been defined earlier using srv_define_event.
**
** Arguments:
**    spp        A pointer to a control structure for an Open
 **               Server thread.
**    event      The token for the event to be added to the queue.
**    datap      Data pointer.
**
** Returns:
**
**    CS_SUCCEED    The event was queued successfully
**    CS_FAIL       An error was detected.
*/
CS_RETCODE  ex_srv_event(spp, event, datap)
SRV_PROC    *spp;
CS_INT      event;
CS_VOID     *datap;
{
    if (srv_event(spp, event, datap) == CS_FAIL)
        return (CS_FAIL);
    else
        return (CS_SUCCEED);
}
```

Usage

- Add an event request to the event queue of a particular client thread. Event requests are usually added to a event request queue automatically, for example, by Client-Library calls from the client application. However, Open Server programmers can specifically add requests with srv_event.

  The following events can be added to an event queue by srv_event:

  - SRV_DISCONNECT
  - SRV_URGDISCONNECT
  - SRV_STOP
  - Programmer-defined events
- srv_handle tells Open Server which event handler to call when an event occurs. If no handler is defined for a particular event, the default Open Server event handler is called.
- The SRV_URGDISCONNECT event causes an Open Server application's SRV_DISCONNECT event handler to be called.
- The SRV_URGDISCONNECT event is queued as an urgent event. This allows an application to place a disconnect event at the top of a thread's event queue, skipping any currently queued events. This is useful to implement immediate termination of an Open Server thread.
- If the event is programmer-defined, it must first be defined with srv_define_event before it can be triggered.
- srv_event adds any event except SRV_STOP or SRV_START to a thread's event queue. In the case of a SRV_STOP or SRV_START event, *spp* points to the internal thread control structure for the thread requesting the event.
- An Open Server application cannot call any routine that does I/O from inside a user-defined event.

Warning! In interrupt-level code, use srv_event_deferred instead of srv_event.

See also

srv_define_event, srv_handle, srv_event_deferred, srv_thread_props, "Events" on page 92

srv_event_deferred

Description

Add an event request to the event queue of a thread as the result of an asynchronous event.

Syntax

CS_INT srv_event_deferred(spp, event, datap)

```
SRV_PROC    *spp;
CS_INT       event;
CS_VOID     *datap;
```

Parameters

spp

A pointer to an internal thread control structure.

event

The event to add to the thread's event queue.

datap

A pointer (CS_VOID) to data supplied by the Open Server programmer. An application can retrieve the data by calling srv_thread_props with property set to SRV_T_EVENTDATA from within the event handler.

Return value

The requested event. If there was an error, -1 is returned.

Table 3-45: Return values (srv_event_deferred)

Returns	To indicate
The token for the requested event.	Open Server added the new event.
-1	The routine failed.

Examples

```
#include   <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE   ex_srv_event_deferred PROTOTYPE((
SRV_PROC     *spp,
CS_INT       event,
CS_VOID      *datap
));
/*
 ** EX_SRV_EVENT_DEFERRED
**    Example routine to queue up a deferred event using
 **    srv_event_deferred. A deferred event request will
 **    typically be made from within interrupt-level code.
** Arguments:
**    spp       A pointer to the internal thread control
 **               structure.
**    event     The event to add to the thread's queue.
```

```
**    datap     A pointer to data to attach to the event.
** Returns:
**    CS_SUCCEED    The event was sucecssfully queued.
**    CS_FAIL       An error was detected.
 */
CS_RETCODE          ex_srv_event_deferred(spp, event, datap)
SRV_PROC            *spp;
CS_INT              event;
CS_VOID             *datap;
{
      /*
      ** Add a deferred event to the event queue.
      */
     if (srv_event_deferred(spp, event, datap) == -1)
     {
            return(CS_FAIL);
      }
     return(CS_SUCCEED);
}
```

Usage

- srv_event_deferred adds an event request to the event queue of a thread from interrupt-level code, such as signal delivery on UNIX. The event request is deferred until critical functions internal to Open Server have been completed, if any such functions were being performed when srv_event_deferred was called.
- Some Open Server applications must be able to raise events from interrupt-level code. For example, if you want to raise an event within the attention handler or you are using the alarm signal in the Open Server application code, you must use srv_event_deferred instead of srv_event. srv_event_deferred ensures that critical functions, such as updating linked lists or performing internal housekeeping, are completed before the event request is acted on.

  Warning! In interrupt-level code, use srv_event_deferred instead of srv_event.

- Open Server usually adds event requests to a thread's event request queue automatically. However, you can specifically add requests with srv_event_deferred.
- The following events can be added to an event queue by srv_event_deferred:
  - SRV_DISCONNECT

- SRV_URGDISCONNECT
- SRV_STOP
- Programmer-defined events

- srv_handle tells the Open Server which event handler to call when an event occurs. If no handler is defined for a particular event, the default event Open Server handler is called.
- If the event is programmer-defined, it must be defined with srv_define_event before it can be triggered.
- srv_event adds any event except SRV_STOP or SRV_START to a thread's event queue. In the case of a SRV_STOP or SRV_START event, *spp* points to the internal thread control structure for the thread requesting the event.
- An Open Server application cannot call any routine that does I/O from inside a user-defined event.

See also

srv_define_event, srv_event, srv_handle, srv_thread_props, "Events" on page 92

srv_free

Description

Free previously allocated memory.

Syntax

```
CS_RETCODE srv_free(mp)
CS_VOID        *mp;
```

Parameters

mp

A pointer to the memory to be freed.

Return value

Table 3-46: Return values (srv_free)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE ex_srv_free PROTOTYPE((
```

```
CS_BYTE   *p
));
/*
 ** EX_SRV_FREE
**
**   Example routine to free memory allocated through srv_alloc.
**
** Arguments:
**        p - The address of the memory block to be freed.
**
** Returns:
**
**   CS_SUCCEED      Memory was freed successfully.
**   CS_FAIL         An error was detected.
*/
CS_RETCODE     ex_srv_free(p)
CS_BYTE        *p;
{
    /*
    ** Free the memory block.
    */
    if( srv_free(p) == CS_FAIL )
    {
        return CS_FAIL;
    }
    return CS_SUCCEED;
}
```

Usage

- Use srv_free only to free memory allocated by srv_alloc, srv_init, or srv_realloc.
- Currently, srv_free calls the C routine, free. An Open Server application, however, can install its own memory management routines using the srv_props routine. The parameter-passing conventions of the user-installed routine must be the same as those of free. If the application is not configured to use the user-installed routines, it will use free.

See also srv_alloc, srv_props, srv_realloc, srv_init

srv_freeserveraddrs

Description Frees memory allocated by srv_getserverbyname.

Syntax CS_RETCODE **srv_freeserveraddrs**(void **resultptr*)

Parameters

resultptr
A pointer to memory returned by srv_getserverbyname.

Return value

Table 3-47: Return values (srv_freeserveraddrs)

Returns	To indicate
CS_SUCCEED	The call to srv_freeserveraddrs ran successfully.
CS_FAIL	*resultptr* is NULL or deallocation failed.

See also

srv_getserverbyname, srv_send_ctlinfo

srv_get_text

Description

Read a text or image datastream from a client, in chunks.

Syntax

```
CS_RETCODE srv_get_text(spp, bp, buflen, outlenp)

SRV_PROC    *spp;
CS_BYTE      *bp;
CS_INT        buflen;
CS_INT       *outlenp;
```

Parameters

spp
A pointer to an internal thread control structure.

bp
A pointer to a buffer where the data from the client is placed.

buflen
The size of the **bp* pointer. This indicates how many bytes are transferred in each chunk.

outlenp
The number of the bytes read into the **bp* buffer is returned here.

Return value

Table 3-48: Return values (srv_get_text)

Returns	To indicate
CS_SUCCEED	The call to srv_get_text ran successfully.
CS_FAIL	The routine failed.
CS_END_DATA	Open Server read in the entire text or image data stream.

Examples

```
#include        <ospublic.h>
#include        <stdio.h>
/*
```

```
 ** Local Prototype
 */
CS_RETCODE    ex_srv_get_text        PROTOTYPE((
SRV_PROC      *spp,
CS_INT        *outlenp,
CS_BYTE       *bbuf
));
/*
 ** EX_SRV_GET_TEXT
 **
**    Example routine to read chunks of text or image datastream
 **    from a client into a buffer and then write it to a disk
 **    file.
**
** Arguments:
**
**    spp        Pointer to thread control structure.
**    outlenp    Number of bytes read and written.
**    bbuf       Pointer to very large buffer for text.
**
** Returns
**
**    CS_SUCCEED    The data was successfully read.
**    CS_FAIL       An error was detected.
 **
 */
#define    BUFSIZE     256
#define    FPUTS(a,b)      fputs(a,b)
CS_RETCODE  ex_srv_get_text(spp,outlenp,bbuf)
SRV_PROC    *spp;
CS_INT      *outlenp;
CS_BYTE     *bbuf;
{
   CS_INT       llen;    /* Local length. */
    CS_INT       lout;    /* Local read count. */
    CS_RETCODE   lret;    /* Local return code. */
    CS_BYTE      *lbufp;    /* Local pointer into bbuf. */
   /* Check arguments. */
   if(bbuf == (CS_VOID *)0)
      return(CS_FAIL);
   if(spp == (SRV_PROC *)0)
      return(CS_FAIL);
   llen = BUFSIZE;
   lbufp = bbuf;
   /*
    ** Loop around getting data and copy it to bbuf.
```

```
        */
       while(lret != CS_END_DATA)
       {
            (CS_VOID)srv_bzero(lbufp,BUFSIZE);
            lout = 0;
            lret = srv_get_text(spp, lbufp, llen, &lout);
            if(lret == CS_FAIL)
                break;
            *outlenp += lout;
            lbufp += lout;
        }
        if(lret == CS_END_DATA)
            return(CS_SUCCEED);
        else
            return(lret);
    }
```

Usage

- srv_get_text is used to read bulk data from a client. The bulk data can be of type text or image.
- srv_get_text must be called until all of the bulk data has been read from a client. It returns CS_END_DATA when the whole data stream has been read in.
- srv_get_text can only be called from inside the SRV_BULK event handler.
- A column read with srv_get_text must be of type text or image.
- An Open Server application must call srv_text_info prior to the first call to srv_get_text for the data stream. The application then calls srv_get_text to retrieve a chunk. srv_get_text is called as many times as are necessary to read in the whole column.
- Open Server treats text and image data streams except that it converts only text data before sending it to the Open Server application. The only conversion by Open Server performs is character set translation.

See also

srv_bind, srv_descfmt, srv_send_text, srv_text_info, srv_thread_props, srv_xferdata, “International support” on page 99, “Text and image” on page 196

srv_getloginfo

Description

Obtain login information from a client thread to prepare a passthrough connection with a remote server.

Syntax

CS_RETCODE srv_getloginfo(spp, loginfo)

SRV_PROC *spp;
CS_LOGINFO **loginfo;

Parameters

spp

A pointer to an internal thread control structure.

loginfo

A pointer to a CS_LOGINFO pointer that will be set to the address of a newly allocated CS_LOGINFO structure.

Return value

Table 3-49: Return values (srv_getloginfo)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE   ex_srv_getloginfo PROTOTYPE((
CS_VOID             *spp,
CS_VOID             **loginfopp
));
/*
 ** EX_SRV_GETLOGINFO
**
**    Example routine to retrieve the client's login structure.
**
** Arguments:
**   spp        Thread control structure.
**   loginfopp  A pointer to client's login record returned here.
**
** Returns:
**
**    CS_SUCCEED    Login structure was retrieved successfully.
**    CS_FAIL       An error was detected.
*/
CS_RETCODE ex_srv_getloginfo(spp, loginfopp)
SRV_PROC   *spp;
CS_LOGINFO **loginfopp;
{
    /* Initialization. */
    *loginfopp = (CS_LOGINFO *)NULL;
    if(srv_getloginfo(spp, loginfopp) == CS_FAIL)
```

```
        {
            return(CS_FAIL);
        }
        return(CS_SUCCEED);
    }
```

Usage

- Use srv_getloginfo in gateway applications that use passthrough mode. In passthrough mode, a gateway application passes packets between clients and remote Sybase servers without interpreting the protocol.
- When a client connects directly to a server, the two programs negotiate the protocol format they will use to send and receive data. When you use protocol passthrough in a gateway application, the Open Server forwards protocol packets between a client and a remote server. Therefore, the client and the remote server must agree on the protocol version.
- srv_getloginfo is the first of four calls, two of them CS-Library calls, that allow a client and remote server to negotiate a protocol format. The calls, which can only be made in a SRV_CONNECT event handler, are:
  a. srv_getloginfo – allocate a CS_LOGINFO structure and fill it with protocol information from the client thread.
  b. ct_setloginfo – prepare a CS_LOGINFO structure with the protocol information retrieved in step 1, then log in to the remote server with ct_connect.
  c. ct_getloginfo – transfer protocol login response information from a CS_CONNECTION structure to the newly allocated CS_LOGINFO structure.
  d. srv_setloginfo – send the remote server's response, retrieved in step 3, to the client, then release the CS_LOGINFO structure.

See also srv_recvpassthru, srv_sendpassthru, srv_setloginfo

srv_getmsgq

Description Get the next message from a message queue.

Syntax CS_RETCODE srv_getmsgq(msgqid, msgp, getflags, infop)

```
SRV_OBJID    msgqid;
CS_VOID      **msgp;
CS_INT       getflags;
CS_INT       *infop;
```

Parameters

msgqid

The identifier for the message queue from which to get a message. To reference the message queue by name, call srv_getobjid with the name to yield the message queue ID.

msgp

A pointer to a pointer variable that srv_getmsgq sets to the message's address.

getflags

The values for *getflags* can be OR'd together. Table 3-50 lists the legal values for *getflags*, and their significance:

Table 3-50: Values for getflags (srv_getmsgq)

Value	Significance
SRV_M_WAIT	If no message is available, srv_getmsgq sleeps until a message is delivered.
SRV_M_NOWAIT	srv_getmsgq returns immediately whether a message is available or not.
SRV_M_READ_ONLY	The default behavior of srv_getmsgq is to remove the message from the message list and to wake up any thread that is waiting for the message to be read. If SRV_M_READ_ONLY is set, a message pointer is returned, but the message is not removed from the list and the thread waiting for the message to be read does not wake up. This option can be used to peek at the head of the message queue to see if the message is intended for the thread.

infop

A pointer to a CS_INT. Table 3-51 describes the possible values returned in **infop* if srv_getmsgq returns CS_FAIL:

Table 3-51: Values for infop (srv_getmsgq)

Value	Meaning
SRV_I_WOULDWAIT	The SRV_M_NOWAIT flag was set in the getflags field and there are no messages to be read.
SRV_I_DELETED	While waiting for a message, the message queue was deleted.
SRV_I_INTERRUPTED	The SRV_M_WAIT flag was set in the getflags field and this call was interrupted before the message arrived.
SRV_I_UNKNOWN	Some other error occurred. Look in the log file for a message.

Return value

Table 3-52: Return values (srv_getmsgq)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <ospublic.h>
/*
 ** Local prototype
 */
CS_VOID    ex_srv_getmsgq PROTOTYPE((
SRV_OBJID  msgqid,
CS_INT     *infop
));
/*
 ** EX_SRV_GETMSGQ
**
**    Example routine to get messages from a message queue.
**
** Arguments:
**    msgqid-    The id of the message queue from which to get
 **               the message.
**
**    infop-     Will hold information about why this routine
 **               failed. Comes directly from srv_getmsg.
** Returns:
**    Nothing. If this routine returns, it is because srv_getmsgq
 **    failed. Check infop to see why it failed.
*/
CS_VOID    ex_srv_getmsgq(msgqid, infop)
SRV_OBJID  msgqid;
CS_INT     *infop;
{
```

```
        CS_CHAR   *message;   /* This message is a string. */
       /*
        ** Loop processing messages. Go to sleep if no messages are
        ** available.
        */
       while (srv_getmsgq(msgqid, (CS_VOID *)&message, SRV_M_WAIT,
                infop)== CS_SUCCEED)
       {
           /* Process message.*/
       }
       /* infop will contain the reason why it failed. */
       return ;
   }
```

Usage

- srv_getmsgq puts the address of the next message from the message queue *msgqid* in **msgp*.
- If the thread that sent the message specified that it would sleep until the message is read, it wakes up.

See also srv_createmsgq, srv_deletemsgq, srv_getobjid, srv_putmsgq

srv_getobjid

Description Look up the object ID for a message queue or mutex with a specified name.

Syntax

```
CS_RETCODE srv_getobjid(obj_type, obj_namep,
                obj_namelen, obj_idp, infop)
CS_INT          obj_type;
CS_CHAR        *obj_namep;
CS_INT          obj_namelen;
SRV_OBJID      *obj_idp;
CS_INT         *infop;
```

Parameters

obj_type
Indicates whether the object is a mutex (SRV_C_MUTEX) or a message queue SRV_C_MQUEUE).

obj_namep
A pointer to a CS_CHAR buffer that contains the name of the object.

obj_namelen
The length of the string in **obj_namep*. If the string is null terminated, *obj_namelen* can be CS_NULLTERM.

obj_idp
A pointer to a SRV_OBJID structure that will receive the identifier for the object, if found.

infop
A pointer to a CS_INT. Table 3-53 describes the possible values returned in **infop* if srv_getobjid returns CS_FAIL:

Table 3-53: Values for infop (srv_getobjid)

Value	Meaning
SRV_I_NOEXIST	The object does not exist.
SRV_I_UNKNOWN	Some other error occurred, for example, a null object name.

Return value

Table 3-54: Return values (srv_getobjid)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_INT    ex_srv_getobjid PROTOTYPE((
CS_INT    obj_type,
CS_CHAR   *obj_name,
SRV_OBJID *obj_idp
));
/*
 ** EX_SRV_GETOBJID
**   An example routine to retrieve the object id for a specified
 **   message queue or mutex name.
** Arguments:
**   obj_type  SRV_C_MUTEX if requesting a mutex object id, and
 **             SRV_C_MQUEUE if requesting a message queue object
                id.
**   obj_name  A null terminated string which specifies the name
 **             of the message queue or the mutex.
**   obj_idp   A pointer to a SRV_OBJID structure that will store
 **             the identifier for the object.
** Returns:
**   CS_SUCCEED    If the object id was retrieved
                    successfully.
**   SRV_I_NOEXIST   If the object does not exist.
** CS_FAIL    If the object was not retrieved due to an error
```

```
 */
CS_INT          ex_srv_getobjid(obj_type, obj_name, obj_idp)
CS_INT          obj_type;
CS_CHAR         *obj_name;
SRV_OBJID       *obj_idp;
{
     CS_INT   info;  /* The reason for failure. */
     CS_INT   status;  /* The return status. */
    /* Validate the obj_type. */
     if ( (obj_type != SRV_C_MUTEX) && (obj_type !=
             SRV_C_MQUEUE) )
     {
         return(CS_FAIL);
     }
    /* Make sure that the object name is not null. */
     if ( obj_name == (CS_CHAR *)NULL )
    {
         return(CS_FAIL);
     }
    /* Ensure that the pointer to the SRV_OBJID is not null */
     if ( obj_idp == (SRV_OBJID *)NULL )
    {
         return(CS_FAIL);
     }
    /* Get the object id. */
     status = (CS_INT)srv_getobjid( obj_type, obj_name,
              CS_NULLTERM, obj_idp, &info);
    /* Check the status. */
     if ( (status == CS_FAIL) && (info == SRV_I_NOEXIST) )
    {
         status = SRV_I_NOEXIST;
     }
    return(status);

}
```

Usage

Open Server maintains a table that maps the unique object identifiers of message queues and mutexes to their names. Given the name, srv_getobjid finds the identifier.

See also

srv_createmsgq, srv_createmutex, srv_deletemsgq, srv_deletemutex, srv_getmsgq, srv_getobjname, srv_lockmutex,srv_putmsgq, srv_unlockmutex

srv_getobjname

Description

Get the name of a message queue or mutex with a specified identifier.

Syntax

```
CS_RETCODE srv_getobjname(obj_type, obj_id, obj_namep,
        obj_namelenp, infop)

CS_INT          obj_type;
SRV_OBJID       obj_id;
CS_CHAR         *obj_namep;
CS_INT          *obj_namelenp;
CS_INT          *infop;
```

Parameters

obj_type
Indicates whether the object is a mutex (SRV_C_MUTEX) or a message queue (SRV_C_MQUEUE).

obj_id
The unique identifier of the object.

obj_namep
A pointer to a CS_CHAR buffer into which the name of the object is copied. The buffer must be large enough to accommodate the object name and, if *obj_namelenp* is NULL, a null character. The maximum length for an object name is SRV_MAXNAME characters, not including the null termination byte.

obj_namelenp
A pointer to a CS_INT that receives the length of the object. If *obj_namelenp* is NULL, the name that is found is copied into **obj_namep* and terminated with a null character. Otherwise, the length of the name in **obj_namep* is placed in **obj_namelenp*.

infop
A pointer to a CS_INT that is set to SRV_I_NOEXIST if the object with ID *obj_id* does not exist.

Return value

Table 3-55: Return values (srv_getobjname)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
#include  <stdio.h>
/*
 ** Local Prototype
 */
```

```
CS_RETCODE  ex_srv_getobjname PROTOTYPE((
CS_INT      obj_type,
SRV_OBJID   obj_id
));
/*
 ** EX_SRV_GETOBJNAME
**    Example routine to illustrate the use of srv_getobjname to
 **    get the name of mutex or message queue with id = obj_id
 **    where obj_id was earlier returned by srv_createmutex or
 **    srv_createmsgq.
** Arguments:
**    obj_type - Type of object; SRV_C_MUTEX or SRV_C_MQUEUE.
**    obj_id   - The unique identifier of the object.
** Returns:
**    CS_SUCCEED   Memory was allocated successfully.
**    CS_FAIL      Memory allocation failure occured.
 */
CS_RETCODE   ex_srv_getobjname(obj_type, obj_id)
CS_INT       obj_type;
SRV_OBJID    obj_id;
{
    CS_CHAR      obj_name[SRV_MAXNAME+1];
    CS_INT       obj_namelen;
    CS_INT       info;
    CS_RETCODE   ret;
    /* Get object name. */
     ret = srv_getobjname(obj_type, obj_id, obj_name,
     &obj_namelen, &info);
    /* Print information depending on retcode */
     switch(ret)
    {
       case CS_FAIL:
         if (info == SRV_I_NOEXIST)
         {
             fprintf(stderr, "%s object with id: %d does not
                     exist\n", (obj_type == SRV_C_MUTEX) ?
                     "Mutex" :"Message Queue", (CS_INT)obj_id);
         }
         else
             fprintf (stderr, "srv_getobjname failed\n");
        break;
     case CS_SUCCEED:
         fprintf (stderr, "%s name: %s for id: %d\n",
            (obj_type == SRV_C_MUTEX) ? "Mutex" : "Message Queue",
             obj_name, (CS_INT)obj_id);
         break;
```

```
        default:
            fprintf (stderr, "Unknown return code from
                     srv_getobjname\n");
           ret = CS_FAIL;
            break;
       }
       return (ret);
   }
```

Usage

- Open Server maintains a table that maps the unique identifiers of message queues and mutexes to their names. Given the identifier, srv_getobjname finds the name.
- In some applications, it may make more sense to reference message queues or mutexes by name. srv_getobjid can be used to look up the identifier that is used by the mutex and message queue services.

See also

srv_createmsgq,srv_createmutex, srv_deletemsgq, srv_deletemutex, srv_getmsgq, srv_getobjid,srv_lockmutex, srv_putmsgq, srv_unlockmutex

srv_getserverbyname

Description

Returns the connection information for *server_name*, allocating memory as needed. Memory allocated by srv_getserverbyname can be freed by calling srv_freeserveraddrs.

Syntax

CS_RETCODE **srv_getserverbyname**(CS_CHAR **server_name*, CS_INT *namelen*, CS_INT *querytype*, CS_INT *result_type*, void **resultptr*, CS_INT **result_cnt*)

Parameters

server_name
Name of the server to be looked up.

namelen
Length of *server_name*. Can be specified as CS_NULLTERM.

querytype
Selects master (CS_ACCESS_CLIENT_MASTER) or query (CS_ACCESS_CLIENT_QUERY) entries for *server_name*.

result_type
Indicates the data format of connection information. *result_type* can be specified as SRV_C_GETADDRS or SRV_C_GETSTRS.

resultptr
A pointer allocated by srv_getserverbyname to hold the results of a query. resultptr is the address of a pointer which will receive the address of the query results.

result_cnt
A pointer to CS_INT that contains the number of addresses returned for *server_name*.

Usage

result_type can be specified as SRV_C_GETADDRS, where the information will be returned as an array of CS_TRANADDR structures. Alternatively, you can specify *result_type* as SRV_C_GETSTRS, which returns an array of pointers to character strings in the *network-protocol protocol-address filter-information* format. For example, where *network-protocol* is "tcp", *protocol-address* is "myhost 4000", and *filter-information* is "ssl", you will receive a result of "tcp myhost 4000 ssl".

See also

srv_freeserveraddrs, srv_send_ctlinfo

srv_handle

Description

Install an event handler in an Open Server application.

Syntax

```
SRV_EVENTHANDLE_FUNC (*srv_handle(ssp, event,
                handler))()

SRV_SERVER                  *ssp;
CS_INT                      event;
SRV_EVENTHANDLE_FUNC   handler;
```

Parameters

ssp
A pointer to the Open Server control structure. This parameter is optional. It is present only to provide backward compatibility.

event

The event that *handler* will handle. Here is a list of all the regular Open Server events:

- SRV_ATTENTION
- SRV_BULK
- SRV_CONNECT
- SRV_CURSOR
- SRV_DISCONNECT/SRV_URGDISCONNECT
- SRV_DYNAMIC
- SRV_FULLPASSTHRU
- SRV_LANGUAGE
- SRV_MSG
- SRV_OPTION
- SRV_RPC
- SRV_START
- SRV_STOP

Programmer-defined events – A programmer-defined event is defined using srv_define_event.

For a description of each event, see “Events” on page 92.

handler

A pointer to the function to call when an *event* request occurs. Passing NULL as the handler installs the default event handler.

Return value

Table 3-56: Return values (srv_handle)

Returns	To indicate
A pointer to the event handling function	The location of the function.
A null pointer	The routine failed.

Examples

```
#include   <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE  ex_srv_handle PROTOTYPE((
SRV_EVENTHANDLE_FUNC       funcp
```

```
));
/*
 ** EX_SRV_HANDLE
**   Install a SRV_START handler.
** Arguments:
**    funcp  Handler to install.
** Returns:
**    CS_SUCCEED   Start handler was installed successfully.
 **    CS_FAIL      An error was detected.
 */
CS_RETCODE               ex_srv_handle(funcp)
SRV_EVENTHANDLE_FUNC     funcp;
{
    if(srv_handle((SRV_SERVER *)NULL, SRV_START, funcp) ==
         CS_FAIL)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

Usage

- srv_handle tells Open Server to call a particular function when it receives a request to handle a particular event.
- Open Server calls *handler* with one argument.

  The event handlers for the following events take a pointer to an Open Server control structure as an argument:

  - SRV_START
  - SRV_STOP

  The event handlers for the following events take a pointer to a thread control structure as an argument:

  - SRV_ATTENTION
  - SRV_BULK
  - SRV_CONNECT
  - SRV_CURSOR
  - SRV_DISCONNECT/SRV_URGDISCONNECT
  - SRV_DYNAMIC
  - SRV_FULLPASSTHRU
  - SRV_LANGUAGE

- SRV_MSG
- SRV_OPTION
- SRV_RPC

Any programmer-defined event

- Each Open Server event has a default handler with a known name. Installing an event handler with srv_handle replaces the default handler.
- Event handlers can be installed dynamically. The new event handler is called the next time the event is raised.
- Event handlers must return CS_SUCCEED.

See also

srv_define_event, srv_event, srv_event_deferred, "Events" on page 92

srv_init

Description

Initialize an Open Server application.

Syntax

```
SRV_SERVER *srv_init(scp, servernamep, namelen)

SRV_CONFIG      *scp;
CS_CHAR         *servernamep;
CS_INT          namelen;
```

Parameters

scp

The configuration structure that holds the values of all the Open Server configuration options. This argument is optional. It is included for backward compatibility.

servernamep

A pointer to the Open Server application name. The name you supply is looked up in the interfaces file to get the necessary network information. If you use (CS_CHAR *) NULL as the Open Server name, the value of DSLISTEN will be the server's name. If DSLISTEN has not been explicitly set, the name defaults to the string "SYBASE".

namelen

The length, in bytes, of the string in **servernamep*. If the string is (CS_CHAR *) NULL, *namelen* is ignored. If the string is null terminated, *namelen* can be CS_NULLTERM.

Return value

Table 3-57: Return values (srv_init)

Returns	To indicate
SRV_SERVER pointer	The routine ran successfully.
(SRV_SERVER *) NULL	The routine failed.

Examples

```
#include <ospublic.h>
/*
 ** Local prototype.
 */
SRV_SERVER    *ex_srv_init PROTOTYPE((
SRV_CONFIG    *scp
));
/*
 ** EX_SRV_INIT
**
**    Example routine to initialize an Open Server application.
**
** Arguments:
**    scp   -   A pointer to the configuration structure.
 **
** Returns:
**    On success, a pointer to a newly allocated SRV_SERVER
       structure.
**    On failure, NULL.
 **
 */
SRV_SERVER   *ex_srv_init(scp)
SRV_CONFIG   *scp;
{
    SRV_SERVER   *server;
    CS_CHAR      *servername = "EX_SERVER";
    server = srv_init(scp, servername, CS_NULLTERM);
    return (server);
}
```

Usage

- A server must be initialized before it is started with srv_run.
- srv_init initializes an Open Server application. The initialization process consists primarily of allocating the necessary data structures for the server, initializing the server state, and starting up the network listener.
- Most configuration options must be set before srv_init is called if values other than the defaults are desired. See the srv_props reference page, for a list of configurable options.

- srv_version must be called prior to srv_init to set up library version information and default internationalization values.
- Open Server releases the SRV_SERVER structure when a SRV_STOP event occurs. An Open Server application should not release it.
- For information on designating an interfaces file, see the srv_props reference page. For more information on the interfaces file itself, see the Open Client and Open Server *Programmer's Supplement* for your platform.

See also

srv_props, srv_run, srv_version

srv_langcpy

Description

Copy a client's language request into an application buffer.

Syntax

```
CS_INT srv_langcpy(spp, start, nbytes, bp)

SRV_PROC    *spp;
CS_INT       start;
CS_INT       nbytes;
CS_BYTE     *bp;
```

Parameters

spp

A pointer to an internal thread control structure.

start

The point at which to start copying characters from the request buffer. The first character in the request buffer is the 0'th character.

nbytes

The number of characters to copy. If *nbytes* is -1, srv_langcpy copies as many bytes as possible. It is legal to copy 0 bytes. If there are not *nbytes* characters available to copy, srv_langcpy copies as many as are in the request buffer.

bp

A CS_CHAR pointer to the programmer-supplied buffer into which to copy the bytes.

Return value

Table 3-58: Return values (srv_langcpy)

Returns	To indicate
An integer	The number of bytes copied.
-1	There is no current language request from this client.

Examples

```
#include  <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE     ex_srv_langcpy PROTOTYPE((
SRV_PROC       *spp,
CS_CHAR        *buf,
CS_INT         size,
CS_INT         *outlen
));

/*
** EX_SRV_LANGCPY
**
**    Example routine to illustrate the use of srv_langcpy to
 **    copy language commands sent by a client.
**
** Arguments:
**    spp      A pointer to internal thread control structure.
**    buf      A CS_CHAR pointer to buffer for language commands.
**    size     The size of the buffer; A CS_INT.
**    outlen   A pointer to CS_INT; the actual length of
                language query copied to buf is returned here. -1
 **             is returned in case of failure.
**
** Returns:
**
**    CS_SUCCEED   Language request was copied successfully.
**    CS_FAIL      An error was detected.
*/
CS_RETCODE   ex_srv_langcpy(spp, buf, size, outlen)
SRV_PROC     *spp;
CS_CHAR      *buf;
CS_INT       size;
CS_INT       *outlen;
{
   CS_INT        act_len; /* actual length of language request */
```

```
        /* Initialization.*/
        *outlen = (CS_INT)-1;

        /* Get the length of language request.*/
        if ((act_len = srv_langlen(spp)) == -1)
            return (CS_FAIL);

        /* Check to see whether we got a buffer of adequate size. */
        if (size < (act_len +1))
            return (CS_FAIL);

        /* Copy language commands.*/
        if (srv_langcpy(spp, (CS_INT)0, act_len, buf) <= 0)
            return (CS_FAIL);

        /* Set the actual length copied. */
        *outlen = act_len;

        return (CS_SUCCEED);
    }
```

Usage

- When a language request is received from the client, srv_langcpy can be used to copy a portion of the request buffer to a Open Server program variable. The copy placed in the destination buffer is null terminated.
- srv_langcpy is also used to process language strings in cursor declare or update statements.

  Warning! srv_langcpy assumes that the destination buffer is large enough to handle nbytes + 1 bytes.

- To set the total length of the language request buffer call srv_langlen.
- The request buffer can contain any string of characters, including Transact-SQL statements. It's up to the Open Server application to process the string.

See also srv_langlen

srv_langlen

Description Return the length of the language request buffer.

Syntax CS_INT srv_langlen(spp)

```
SRV_PROC    *spp;
```

Parameters

spp
A pointer to an internal thread control structure.

Return value

Table 3-59: Return values (srv_langlen)

Returns	To indicate
An integer	The length in bytes of the language request buffer.
-1	There is no current language request from this client.

Examples

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE ex_srv_langlen PROTOTYPE((
SRV_PROC  *spp,
CS_INT    *len
));

/*
 ** EX_SRV_LANGLEN
**  Example routine to return the length of the language request
 **  buffer using srv_langlen.
**
** Arguments:
**  spp    A pointer to the internal thread control structure.
**  len    Return pointer for the length of the language string.
 **         If there is no language command -1 is returned.
**
** Returns:
**
**   CS_SUCCEED   Language length was retrieved successfully.
**   CS_FAIL      An error was detected.
 */
CS_RETCODE   ex_srv_langlen(spp, len)
SRV_PROC     *spp;
CS_INT       *len;
{
    /* Retrieve the language length.    */
    if ((*len = srv_langlen(spp)) < 0)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

Usage

- When a language request has been received from a client, srv_langlen returns the length of the request buffer.
- srv_langlen is also used to process language strings in cursor declare or update statements.
- All or part of the request buffer can be accessed with srv_langcpy.
- The request buffer can contain any string, including Transact-SQL statements. It is up to the Open Server application to process the string.

See also

srv_langcpy

srv_lockmutex

Description

Lock a mutex.

Syntax

```
CS_RETCODE srv_lockmutex(mutex_id, waitflag, infop)

SRV_OBJID       mutex_id;
CS_INT          waitflag;
CS_INT          *infop;
```

Parameters

mutex_id

The unique mutex identifier that was returned by the call to srv_createmutex. Given the name of the mutex, the *mutex_id* can be obtained by calling srv_getobjid.

waitflag

Specifies whether the thread requesting the mutex lock should wait or just return if the mutex cannot be granted immediately. The value in **indp* indicates whether the lock was granted. The two valid values for *waitflag* are SRV_M_WAIT, which indicates that the thread should wait if the lock cannot be granted immediately, and SRV_M_NOWAIT, which indicates that the thread should return without waiting if the lock cannot be granted.

infop

A pointer to a CS_INT that is set to one of the following values:

SRV_I_SYNC – The lock was granted synchronously—the thread requesting the lock was not suspended to wait for the lock. srv_lockmutex returned CS_SUCCEED.

SRV_I_GRANTED – The lock was granted after the requesting thread was suspended to wait for another thread to release a lock on the mutex. srv_lockmutex returned CS_SUCCEED.

SRV_I_INTERRUPTED – The thread received an attention while waiting for the lock. The lock was not granted, and srv_lockmutex returned CS_FAIL.

SRV_I_WOULDWAIT – The *waitflag* parameter was set to SRV_M_NOWAIT and the thread would have had to wait for the lock. The lock was not granted, and srv_lockmutex returned CS_FAIL.

SRV_I_UNKNOWN – Some other error occurred, for example, the mutex does not exist. srv_lockmutex returned CS_FAIL.

Return value

Table 3-60: Return values (srv_lockmutex)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
/*
 ** Local Prototype
 */
CS_RETCODE    ex_srv_lockmutex PROTOTYPE((
SRV_OBJID     mid
));
/*
 ** EX_SRV_LOCKMUTEX
 **
 **  Example routine to illustrate the use of srv_lockmutex.
 **
 ** Arguments:
 **      mid - The id of the mutex to lock.
 **
 ** Returns:
 **
 **  CS_SUCCEED   Mutex successfully locked.
```

```
 **  CS_FAIL      An error was detected.
 */
CS_RETCODE   ex_srv_lockmutex(mid)
SRV_OBJID    mid;      /* The mutex id. */
{
    CS_INT   info;    /* Information output variable. */

    /*
     ** Request the mutex lock - sleep until we get it.
     */
    if( srv_lockmutex(mid, SRV_M_WAIT, &info) == CS_FAIL )
    {
        /*
         ** An error was al&ready raised.
         */
        return CS_FAIL;
    }

    /*
     ** All done.
     */
    return CS_SUCCEED;
}
```

Usage

- Mutexes are associated with data objects and program resources that must be protected from simultaneous access by multiple threads.
- Mutex locks are granted to threads on a first-come, first-served basis.
- The lock is granted only if no other thread has al&ready obtained a lock on the mutex.
- srv_lockmutex cannot be used in a SRV_START or SRV_ATTENTION handler.
- A thread can lock a mutex more than once, but must call srv_unlockmutex once for each call to srv_lockmutex before another thread can lock the mutex.
- If the mutex was waiting for is deleted, srv_lockmutex returns CS_FAIL.

See also

srv_createmutex, srv_deletemutex, srv_getobjid, srv_unlockmutex

srv_log

Description

Write a message to the Open Server log file.

Syntax

CS_RETCODE srv_log(ssp, datestamp, msgp, msglen)

```
SRV_SERVER    *ssp;
CS_BOOL        datestamp;
CS_CHAR        *msgp;
CS_INT         msglen;
```

Parameters

ssp

The handle to the Open Server. This argument is optional. It is only present for backward compatibility.

datestamp

If *datestamp* is CS_TRUE, the current date and time is added to the beginning of the log message. If *datestamp* is CS_FALSE, the log message is not timestamped.

msgp

A pointer to the actual text of the message.

msglen

The length in bytes of *msg*. If the string in **msgp* is null terminated, *msglen* can be CS_NULLTERM.

Return value

Table 3-61: Return values (srv_log)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
#include  <string.h>
/*
 ** Local Prototype.
 */
CS_RETCODE   ex_srv_log PROTOTYPE((
SRV_SERVER   *ssp,
CS_CHAR      *msg_txt
));
/*
 ** EX_SRV_LOG
**
**    Example routine to log a message.
**
** Arguments:
```

```
**
**  ssp         A pointer to the Open Server state information
 **              control structure.
**    msg_txt   Text of message to log.
** Returns
**
**  CS_SUCCEED      Thread was created.
**  CS_FAIL         An error was detected.
**
 */
CS_RETCODE  ex_srv_log(ssp, msg_txt)
SRV_SERVER  *ssp;
CS_CHAR     *msg_txt;
{
    CS_RETCODE        lret;
    CS_INT            msg_len;
    /* Check arguments.                    */
    if(ssp == (SRV_SERVER *)0)
        return(CS_FAIL);
    if(msg_txt == (CS_CHAR *)NULL)
        return(CS_FAIL);
    msg_len=strlen(msg_txt);
    /*
    ** Log the message -  We use CS_TRUE as the second argument
     **                    to force the date and time to be
     **                    added to the beginning of the logged
     **                    message. If you do not want a
     **                    datestamp then use CS_FALSE.
    */
    lret = srv_log(ssp,CS_TRUE,msg_txt,msg_len);
    return(lret);
}
```

Usage

- srv_log writes messages to the Open Server log file. The default name of the log file is *srv.log*. The name can be set with srv_props.
- Messages are always appended to the log file.
- The name of the log file can be accessed with the srv_props routine.
- The newline character is not added to the text in **msgp*.
- The log file is truncated based on the SRV_TRUNCATELOG property set through srv_props.
- If the message length exceeds SRV_MAXMSG, Open Server truncates the message. This holds true whether or not the message is null terminated.
- If srv_init has not completed, the message goes to the boot window.

See also srv_props

srv_mask

Description Initialize, set, clear or check bits in a SRV_MASK_ARRAY structure.

Syntax

```
CS_RETCODE srv_mask(cmd, maskp, bit, infop)

CS_INT          cmd;
SRV_MASK_ARRAY  *maskp;
CS_INT          bit;
CS_BOOL         *infop;
```

Parameters

cmd

The action being performed. Table 3-62 summarizes the legal values for *cmd*:

Table 3-62: Legal values for cmd (srv_mask)

Value	Action
CS_SET	Set the bit in the SRV_MASK_ARRAY in **maskp*.
CS_GET	Find out whether the bit is currently set in the SRV_MASK_ARRAY in **maskp*. If bit is set, **infop* is set to CS_TRUE. Otherwise, it is set to CS_FALSE.
CS_CLEAR	Clear the bit in the SRV_MASK_ARRAY in **maskp*.
CS_ZERO	Initialize the SRV_MASK_ARRAY in **maskp* so that all the bits are off. When *cmd* is set to CS_ZERO, *bit* and *infop* are ignored.

maskp

A pointer to a SRV_MASK_ARRAY structure.

bit

The bit being initialized, set, cleared, or checked in the SRV_MASK_ARRAY. This must be an integer between 0 and SRV_MAXMASK_LENGTH. SRV_MAXMASK_LENGTH is defined in *ospublic.h*.

infop

A pointer to a variable that will indicate whether or not *bit* is set. This parameter is ignored when *cmd* is CS_SET, CS_CLEAR, or CS_ZERO.

Return value

Table 3-63: Return values (srv_mask)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_mask PROTOTYPE((
SRV_MASK_ARRAY  *maskptr,
CS_INT          bit
));

/*
** EX_SRV_MASK
**
**    Example routine to manipulate bits in a SRV_MASK_ARRAY
 **    structure.
**
** Arguments:
**    maskptr    A pointer to a mask array.
**    bit        The bit to examine.
**
** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE       ex_srv_mask(maskptr, bit)
SRV_MASK_ARRAY  *maskptr;
CS_INT  bit;
{
    CS_BOOL    info = CS_TRUE;

    if (srv_mask(CS_GET, maskptr, bit, &info) == CS_FAIL)
    {
         return(CS_FAIL);
    }
    else
    {
        /* Has the bit been set? */
        if (info == CS_FALSE)
```

```
                return(CS_FAIL);
            else
                return(CS_SUCCEED);
        }
    }
```

Usage

srv_mask is used to access and modify a SRV_MASK_ARRAY structure.

srv_msg

Description

Send or receive a message datastream.

Syntax

CS_RETCODE srv_msg(spp, cmd, msgidp, status)

```
SRV_PROC    *spp;
CS_INT       cmd;
CS_INT       *msgidp;
CS_INT      *statusp;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

Indicates whether the application is calling srv_msg to send or retrieve a message. Table 3-64 describes the legal values for cmd:

Table 3-64: Values for cmd (srv_msg)

Value	Description
CS_SET	srv_msg is setting the values for *status* and *msgid* prior to sending the message to the client.
CS_GET	srv_msg is retrieving the *status* and *msgid* values for the message being received.

msgidp

A pointer to the message ID of the current message. If the Open Server application is sending a message (CS_SET), it must provide the message ID here. If the application is reading a message (CS_GET), the message ID of the received message is returned here. Values of SRV_MINRESMSG through SRV_MAXRESMSG are reserved for internal Sybase usage. Since the message ID is subsequently sent as a smallint (2 bytes) through TDS, the available range you can use for your own messages is SRV_MAXRESMSG to 65535, if you define message ID as an unsigned CS_SMALLINT.

statusp

A pointer to the status of the current message. If the Open Server application is receiving a message (CS_GET), Open Server will update **statusp* with the message status. If the application is sending a message (CS_SET), **statusp* must contain the status of the message to be sent. Table 3-65 describes the legal values for **statusp*:

Table 3-65: Values for statusp (srv_msg)

Value	Description
SRV_HASPARAMS	The message has parameters.
SRV_NOPARAMS	The message has no parameters.

Return value

Table 3-66: Return values (srv_msg)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
/*
 ** Local prototype.
 */
CS_RETCODE    ex_srv_msg PROTOTYPE((
SRV_PROC      *spp
));

/*
** EX_SRV_MSG
 **
 **    Example routine to receive and send a message datastream.
 **
 ** Arguments:
 **  spp  A pointer to an internal thread control structure.
 **
 ** Returns:
 **  CS_SUCCEED if we were successful in both receiving and
 **  sending a message stream.
 **
 **  CS_FAIL if an error was detected.
 **
 */
CS_RETCODE    ex_srv_msg(spp)
SRV_CONFIG    *scp;
{
```

```
        CS_RETCODE      result;
        CS_INT          msgid;
        CS_INT          status;

        /*
         ** We will first get a message and process any parameters.
         */

        result = srv_msg(spp, CS_GET, &msgid, &status);

        if (result == CS_FAIL)
        {
            return (CS_FAIL);
        }

        if (status == SRV_HASPARAMS)
        {
            /*
             ** Process parameters here using srv_bind and
             ** srv_xferdata.
             */
        }

        /*
        ** Now, an example of sending a message.
        */
        msgid = 32768;
        status = SRV_NOPARAMS;

        result = srv_msg(spp, CS_SET, &msgid, &status);

        if (result == CS_FAIL)
        {
            return (CS_FAIL);
        }
        /*
        ** If the message has parameters, send it across using    ** srv_xferdata
         */
        if (status == SRV_HASPARAMS)
         {
            result = srv_xferdata(spp, CS_SET, SRV_MSGDATA);
        }
        return(result);
    }
```

Usage

- srv_msg is used to send or receive a TDS message data stream.

- Each message data stream received from a client raises a SRV_MSG event. A separate event is raised for each message received.
- If a message has parameters, **statusp* will contain the value CS_HASPARAMS. The application can retrieve and store the parameters using srv_descfmt, srv_bind, and srv_xferdata with *type* set to SRV_MSGDATA.
- An application can determine the number of parameters for a message by calling srv_numparams.
- The srv_msg routine is used to send the status and ID. The actual parameters of the message, if any, are sent using srv_descfmt, srv_bind, and srv_xferdata with a *type* argument of SRV_MSGDATA.
- An application can send or receive multiple message data streams.
- srv_xferdata is only needed to retrieve or send message parameters. When using it for these cases, srv_xferdata must be called once for each message being sent or received. If you use srv_xferdata when no parameters exist, Open Server returns an error.
- srv_msg can only be called in a SRV_MSG event handler when *cmd* is CS_GET. It can be called in any event handler when *cmd* is CS_SET.

See also

srv_bind, srv_descfmt, srv_numparams, srv_xferdata, "Data stream messages" on page 80

srv_negotiate

Description

Send to and receive from a client, negotiated login information.

Syntax

```
CS_RETCODE srv_negotiate(spp, cmd, type)

SRV_PROC    *spp;
CS_INT       cmd;
CS_INT       type;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

Indicates whether the application is calling srv_negotiate to send or retrieve negotiated login information. Table 3-67 describes the legal values for *cmd*:

Table 3-67: Values for cmd (srv_negotiate)

Value	Description
CS_SET	The negotiated login information defined by *type* is to be sent to the client.
CS_GET	The negotiated login information defined by *type* is to be read from the client.

type

The type of negotiated login information to be sent to or read from a client. Table 3-68 describes the legal values for type:

Table 3-68: Values for type (srv_negotiate)

Value	Description
SRV_NEG_CHALLENGE	The negotiated login information is a challenge byte stream sent to the client (CS_SET) or a challenge response byte stream read from the client (CS_GET).
SRV_NEG_ENCRYPT	The negotiated login information consists of an encryption key sent to the client. The client will then use this to encrypt its local and remote passwords. This type is only valid when *cmd* is CS_SET.
SRV_NEG_EXTENDED_ ENCRYPT	The negotiated login information and public key used to encrypt the password.These information are used by the client. This type is only valid when *cmd* is CS_SET.
SRV_NEG_EXTENDED_ LOCPWD	The public key encrypted password sent by the client in response to a SRV_NEG_EXTENDED_ENCRYPT challenge. This type is only valid when *cmd* is CS_GET.
SRV_NEG_EXTENDED_ REMPWD	The negotiated login information is a variable number of pairs of remote server names and corresponding public key encrypted password sent by the client in response to a SRV_NEG_EXTENDED_ENCRYPT challenge. This type is only valid when *cmd* is CS_GET.
SRV_NEG_LOCPWD	The encrypted local password sent by the client in response to a SRV_NEG_ENCRYPT challenge. This type is only valid when *cmd* is CS_GET.
SRV_NEG_REMPWD	The negotiated login information is a variable number of remote server name and encrypted remote password pairs sent by the client in response to a SRV_NEG_ENCRYPT challenge. This type is only valid when *cmd* is CS_GET.
SRV_NEG_SECLABEL	The negotiated login information is a request for security labels sent to the client, or a set of security labels sent by the client to the server.
SRV_NEG_SECSESSION	The negotiated login information is used by a full passthrough gateway application to establish a direct security session between a gateway client and a remote server. This is similar to challenge-response security negotiation. Refer to “Security services” on page 170 for more information and for an example security session callback.

Value	Description
An integer value between CS_USER_MSGID and CS_USER_MAX_MSGID, inclusive.	The negotiated login information is part of an application-defined handshake, identified by the *type* argument itself.

Return value

Table 3-69: Return values (srv_negotiate)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE    ex_srv_negotiate PROTOTYPE((
SRV_PROC      *sproc
));

/*
** EX_SRV_NEGOTIATE
**  An example routine to retrieve negotiated login information
 **  by using srv_negotiate.
**
** Arguments:
**  sproc  A pointer to an internal thread control structure.
**
** Returns:
**  CS_SUCCEED   The login information was retrieved.
**  CS_FAIL      An error was detected.
*/
CS_RETCODE   ex_srv_negotiate(sproc)
SRV_PROC     *sproc;
{
    /*
    ** Check to make sure that the thread control structure is
     ** not NULL.
    */
    if ( sproc == (SRV_PROC *)NULL )
    {
        return(CS_FAIL);
    }
```

```
        /* Now get the login information. */
        if ( srv_negotiate(sproc, CS_GET, SRV_NEG_CHALLENGE) ==         CS_FAIL )
        {
            return(CS_FAIL);
        }

        return(CS_SUCCEED);
    }
```

Usage

- srv_negotiate is used to send negotiated login information to, and receive negotiated login responses from, a client.
- through srv_negotiate, Open Server applications can implement a secure login process inside their SRV_CONNECT event handler. In a secure computing environment, an application may want to perform more rigorous authentication at connect time to verify that clients are who they claim to be, by issuing negotiated login challenges and encrypted passwords.
- An Open Server application can choose to send a challenge or encrypted password to the client while in the SRV_CONNECT event handler, to authenticate the login attempt.
- Once an application has sent a negotiated login challenge or encrypted password, it must read the client's response before the connection process can continue.
- An Open Server application can go through as many challenge or response iterations as are necessary to authenticate the login attempt. However, the application must read in the response to each challenge before sending another challenge.
- Once a negotiated login challenge has been sent to a client, the application must read the response before the connection process can continue.
- An Open Server application must punctuate any type of challenge with a call to srv_senddone. If the application issues a **batch** of several challenges before it reads a response, it must call srv_senddone with a *status* argument of SRV_DONE_MORE after each challenge but the last one in the batch. After the last challenge in the batch, the application must call srv_senddone with a *status* argument of SRV_DONE_FINAL.

- For application-defined handshakes, an Open Server application can set the *type* argument to a value between CS_USER_MSGID and CS_USER_MAX_MSGID to set the handshake type (CS_SET) or specify the type of reply the client should be sending in response (CS_GET). If the Open Server application receives an unexpected value, Open Server raises an error.
- When a client responds to a challenge or encrypted password, srv_negotiate suspends the thread's execution until the client's response has arrived. Applications should bear this in mind when coding a secure SRV_CONNECT event handler.
- Negotiated login challenges and responses carry data values through parameters, which are sent and received through srv_bind, srv_descfmt, and srv_xferdata.These three routines take a *type* argument of SRV_NEGDATA to define or access negotiated login data.
- Table 3-70 lists the parameter or parameters that accompany each type of challenge sent to a client:

Table 3-70: Required challenge parameters (srv_negotiate)

Negotiated login type	Parameters required
SRV_NEG_CHALLENGE	One parameter – Challenge-data value. Datatype is CS_BINARY_TYPE with the CS_DATAFMT *status* field set to CS_CANBENULL.
SRV_NEG_ENCRYPT	One parameter – Encryption key data value. Datatype is CS_BINARY_TYPE with the CS_DATAFMT *status* field set to CS_CANBENULL.
SRV_NEG_SECLABEL	No parameters.
SRV_NEG_SECSESSION	The security session callback specifies the number of parameters and their data formats. Refer to "Security session callbacks" on page 193 and to the Open Client *Client-Library/C Reference Manual*.
An integer value between CS_USER_MSGID and CS_USER_MAX_MSGID, inclusive.	One parameter – Application-defined login handshake data value.

- Table 3-71 lists the parameter that should be read from a client for each type of negotiated login challenge:

Table 3-71: Expected challenge parameters (srv_negotiate)

Negotiated login type	Parameters present
SRV_NEG_CHALLENGE	One parameter – Challenge response data.
SRV_NEG_LOCPWD	One parameter – Encrypted local password.
SRV_NEG_REMPWD	A variable number of server-name/password pairs.
SRV_NEG_SECLABEL	Four parameters: Param 1: Maximum read level label. Param 2: Maximum write level label. Param 3: Minimum write level label. Param 4: Current write level label.
SRV_NEG_SECSESSION	The security session callback specifies the number of parameters and their data formats. Refer to "Security session callbacks" on page 193 and to the Open Client *Client-Library/C Reference Manual*.
An integer value between CS_USER_MSGID and CS_USER_MAX_MSGID, inclusive.	One parameter – Application-defined login handshake data value.

- Note that a response to a password encryption challenge, SRV_NEG_ENCRYPT, can consist of two sets of parameters. The SRV_NEG_LOCPWD response carries a parameter indicating the client's encrypted password. The client can also send a SRV_NEG_REMPWD response, which carries parameters indicating the client's encrypted remote server password and the remote server name, respectively. The SRV_NEG_LOCPWD response to a SRV_NEG_ENCRYPT challenge will always be present. If no remote server passwords were sent by the client, a request to receive a SRV_NEG_REMPWD response will fail.
- Applications that use Open Client and Open Server to implement gateway functionality must use Open Client's negotiated login callback mechanism to route negotiated login challenges and responses between clients and the remote server. In this type of application, the Open Client negotiated login callback must contain the Server-Library routine calls necessary to forward a challenge to the client, and receive the response, which Open Client then returns to the remote server.

If the gateway application intends to establish a direct security session between clients and a remote server, then an Open Client security session callback is required. This callback must contain the Server-Library calls necessary to forward the opaque security tokens to the client, and receive the response, which the Open Client then returns to the remote server. Refer to "Security session callbacks" on page 193 and to the Open Client *Client-Library/C Reference Manual*, for more information.

See also srv_senddone, srv_thread_props

srv_numparams

Description Return the number of parameters contained in the current client command.

Syntax

```
CS_RETCODE srv_numparams(spp, numparamsp)

SRV_PROC    *spp;
CS_INT      *numparamsp;
```

Parameters

spp

A pointer to an internal thread control structure.

numparamsp

A pointer to the number of arguments in the current client command or cursor data stream is returned in **numparamsp*.

Return value

Table 3-72: Return values (srv_numparams)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE    ex_srv_numparams PROTOTYPE((
SRV_PROC      *spp,
CS_INT        *countp
));

/*
** EX_SRV_NUMPARAMS
```

```
**
**    Example routine to illustrate the use of srv_numparams to
 **    get the number parameters contained in the current client
 **    command.
**
** Arguments:
**    spp    A pointer to an internal thread control structure.
**    countp A pointer to the buffer in which the number of
 **           parameters in the client command is returned.
**
** Returns:
**
**  CS_SUCCEED    The number of parameters was successfully
 **                returned.
**  CS_FAIL       An error was detected.
*/
CS_RETCODE        ex_srv_numparams(spp, countp)
SRV_PROC          *spp;
CS_INT            *countp;
{
     if (srv_numparams(spp, countp) == CS_FAIL)
        return (CS_FAIL);

    return(CS_SUCCEED);
}
```

Usage

- srv_numparams returns the number of parameters in the current MSG, RPC, DYNAMIC or cursor data stream, or the number of parameters in a client's response to a srv_negotiate(CS_GET) call. This number includes any default parameters filled in by Open Server at runtime.
- srv_numparams can only be called from handlers for specific events. Table 3-73 lists those events and their parameters:

Table 3-73: Events and parameters (srv_numparams)

Event	Parameters
SRV_CURSOR	Cursor parameters.
SRV_RPC	RPC parameters.
SRV_DYNAMIC	Dynamic SQL parameters.
SRV_MSG	MSG parameters.
SRV_LANGUAGE	Language parameters. srv_numparams requires a TDS level of 5.0 or above to check for and retrieve parameter data in a language handler. You may need to add code to your application to check the TDS level on the connection, and skip srv_numparams if the TDS version is less than SRV_TDS_5_0. You can use the SRV_S_TDSVERSION property of the srv_props routine to get the TDS protocol version on the connection (see Table 2-25 on page 141).
After a srv_negotiate (CS_GET) call.	Parameters in the client's response. For example, in the sample program, *ctos.c*.

See also

srv_bind, srv_cursor_props, srv_descfmt, srv_dynamic, srv_msg, srv_xferdata, "Processing parameter and row data" on page 134

srv_options

Description

Send option information to a client or receive option information from a client.

Syntax

```
CS_RETCODE srv_options(spp, cmd, optcmdp, optionp,
    bufp, bufsize, outlenp)

SRV_PROC    *spp;
CS_INT      cmd;
CS_INT      *optcmdp;
CS_INT      *optionp;
CS_CHAR     *bufp;
CS_INT      bufsize;
CS_INT      *outlenp;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

Indicates whether the application is calling srv_options to send or receive option information. Table 3-74 describes the legal values for *cmd*:

Table 3-74: Values for cmd (srv_options)

Value	Description
CS_SET	The Open Server application is sending an option command to a client.
CS_GET	The Open Server application is receiving an option command from a client.

optcmdp

A pointer either to the program variable that will contain a client's option command (CS_GET) or to the program variable that contains the Open Server application's option command (CS_SET). Table 3-75 summarizes the legal values for **optcmdp*:

Table 3-75: Values for optcmdp (srv_options)

Value	Description	Cmd
SRV_SETOPTION	The client is requesting that the option be set. The value associated with *optionp* is returned in **bufp*. Open Server will set *bufsize* to the size, in bytes, of the data returned. If **bufp* is not large enough to hold the data, the function will return CS_FAIL, the actual size of the option value, in bytes, is returned in **outlenp*, and the values of *optionp* and *bufp* will remain undefined.	CS_GET
SRV_CLEAROPTION	The client is requesting that *optionp* be set to its default value. The *bufp* and *optionp* values will remain undefined.	CS_GET
SRV_GETOPTION	A client is requesting information on the current value in **optionp*. The *bufp* and *optionp* values will remain undefined.	CS_GET
SRV_SENDOPTION	The application is sending the current option value to the client in response to a SRV_GETOPTION command. *bufp* points to the argument associated with the option, and *bufsize* holds the size, in bytes, of the data in **bufp*.	CS_SET

optionp

A pointer either to the client's requested option (CS_GET) or to the option with which the Open Server application is responding (CS_SET).

bufp
A pointer to a buffer that will contain either the value associated with the option (CS_GET) or the value of the option to be sent to the requestor (CS_SET). The **optionp* contains the option in question and **bufp* contains its value (on a CS_SET). For a complete list of options and their legal values, see below.

bufsize
The length of the **bufp* buffer. When sending an option that takes a character string option value, if the value in *bufp* is null terminated, pass *bufsize* as CS_NULLTERM.

outlenp
A pointer to a program variable which is set to the size, in bytes, of the option value returned in **bufp*. This parameter is only used when *cmd* is set to CS_GET, and is optional.

Return value

Table 3-76: Return values (srv_options)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE    ex_srv_options PROTOTYPE((
SRV_PROC      *spp,
CS_INT        *rowcount
));

/*
** EX_SRV_OPTIONS
**
**    Example routine to recieve option information for the
 **    maximum number of regular rows to return (CS_OPT_ROWCOUNT)
 **    from a client.
**
** Arguments:
**   spp       A pointer to an internal thread control structure.
**   rowcount  Return pointer for the number of rows to return.
**
** Returns:
```

```
**
**    CS_SUCCEED   Successfully retrieved option.
**    CS_FAIL      An error was detected.
*/
CS_RETCODE       ex_srv_options(spp, rowcount)
SRV_PROC         *spp;
CS_INT           *rowcount;
{
    CS_INT       optcmdp;    /* The client's option command. */
    CS_INT       optionp;    /* The client's option request. */

    /* Initialization. */
    optcmdp = SRV_GETOPTION;
    optionp = CS_OPT_ROWCOUNT;

    /*
    ** Get the maximum number of rows to return.
    */
    if (srv_options(spp, CS_GET, &optcmdp, &optionp, (CS_VOID
         *)rowcount, CS_SIZEOF(CS_INT), (CS_INT *)NULL) !=
         CS_SUCCEED)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEED);
}
```

Usage

- srv_options allows an Open Server application to read option information from a client or send option information to a client.
- Table 3-77 summarizes the valid options, their legal values, and the datatype of the optionp parameter:

Table 3-77: Description of options (srv_options)

Option	Legal value	bufp points to
CS_OPT_ANSINULL	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_ANSIPERM	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_ARITHABORT	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_ARITHIGNORE	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_AUTHOFF	CS_OPT_SA, CS_OPT_SSO, CS_OPT_OPER	A character string
CS_OPT_AUTHON	CS_OPT_SA, CS_OPT_SSO, CS_OPT_OPER	A character string
CS_OPT_CHAINXACTS	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_CURCLOSEONXACT	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_CURREAD	Read label (string)	A character string
CS_OPT_CURWRITE	Write label (string)	A character string
CS_OPT_DATEFIRST	CS_OPT_SUNDAY CS_OPT_MONDAY CS_OPT_TUESDAY CS_OPT_WEDNESDAY CS_OPT_THURSDAY CS_OPT_FRIDAY CS_OPT_SATURDAY	A symbolic value representing the day to use as the first day of the week
CS_OPT_DATEFORMAT	CS_OPT_FMTMDY CS_OPT_FMTDMY CS_OPT_FMTYMD CS_OPT_FMTYDM CS_OPT_FMTMYD CS_OPT_FMTDYM	A symbolic value representing the order of year, month and day to be used in datetime values
CS_OPT_FIPSFLAG	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_FORCEPLAN	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_FORMATONLY	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_GETDATA	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_IDENTITYOFF	A string value representing a table name	A character string
CS_OPT_IDENTITYON	A string value representing a table name	A character string
CS_OPT_ISOLATION	CS_OPT_LEVEL1 CS_OPT_LEVEL3	A symbolic value representing the isolation level

Option	Legal value	bufp points to
CS_OPT_NOCOUNT	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_NOEXEC	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_PARSEONLY	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_QUOTED_IDENT	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_RESTREES	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_ROWCOUNT	The maximum number of regular rows to return	A CS_INT 0 means all rows are returned
CS_OPT_SHOWPLAN	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_STATS_IO	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_STATS_TIME	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_STR_RTRUNC	CS_TRUE, CS_FALSE	A CS_BOOL
CS_OPT_TEXTSIZE	The length, in bytes, of the longest text or image value the server should return	A CS_INT
CS_OPT_TRUNCIGNORE	CS_TRUE, CS_FALSE	A CS_BOOL

"Options" on page 122 describes each option and lists its default value.

- Open Server raises a SRV_OPTION event for each option command received from a client. Inside its SRV_OPTION event handler, the application can then call srv_options with *cmd* set to CS_GET to retrieve the option information. When srv_options returns, *optcmdp*, *optionp*, and **bufp* will contain all of the option information received from the client. It is an error to call srv_options in any event handler other than a SRV_OPTION event handler.
- In response to SRV_SETOPTION and SRV_CLEAROPTION, the application must call srv_senddone with an argument of SRV_DONE_FINAL. If option processing is unsuccessful, the application must call srv_senddone with an argument of SRV_DONE_FINAL | SRV_DONE_ERROR.
- The application must respond to every SRV_GETOPTION command it receives with a call to srv_options, with *optcmdp* set to SRV_SENDOPTION and *bufp* pointing to the current value of the option.
- It is the application's responsibility to ensure that the **bufp* buffer is large enough to receive arguments sent by a client with a SRV_SETOPTION command. If the buffer is not large enough, srv_options will return CS_FAIL and *outlenp* will be set to the required size.

- Open Server has no notion of what particular options mean. It is the Open Server application's responsibility to save the client's option commands and perform any actions that they require. If there is no SRV_OPTION event handler installed, option commands received from clients will be rejected with an error.

See also srv_senddone, "Options" on page 122

srv_orderby

Description Return an order-by list to a client.

Syntax

```
CS_RETCODE srv_orderby(spp, numcols, collistp)

SRV_PROC    *spp;
CS_INT       numcols;
CS_INT      *collistp;
```

Parameters

spp
A pointer to an internal thread control structure.

numcols
The number of columns in the order-by list. Because the columns are passed as an array of CS_INTs, *numcols* is really the number of elements in the *collistp* array.

collistp
A pointer to the array of column numbers. The size of this array is *numcols*.

Return value

Table 3-78: Return values (srv_orderby)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>
/*
** Local Prototype
*/
CS_RETCODE   ex_srv_orderby PROTOTYPE((
SRV_PROC     *spp
));

/*
```

```
** EX_SRV_ORDERBY
**
**    Example routine using srv_orderby to define and return to a
 **    client application the order-by list for a simple SQL
 **    command.
**    This example uses the SQL command:
**
**        "select a,b,c,d from my_tab
 **         order by c,a"
**
** Arguments:
**    spp    A pointer to the internal thread control structure.
**
** Returns:
**    CS_SUCCEED      Order-by list was successfully defined.
**    CS_FAIL         An error was detected..
*/
CS_RETCODE            ex_srv_orderby(spp)
SRV_PROC              *spp;
{
    /* There are two columns specified in the order-by clause. */
    CS_INT        collist[2];
    CS_INT        numcols;

    /* Initialization. */
    numcols = 2;

    /*
    ** Initialize the collist array in the order the
     ** columns occur in the order-by clause.
    **
    ** "c" is the 1st column specified in the order-by,
     ** and is the 3rd column specified in the select-list.
    */
    collist[0] = (CS_INT)3;
    /*
    ** "a" is the 2nd column specified in the order-by,
     ** and is the 1st column specified in the select-list.
    */
    collist[1] = (CS_INT)1;
    /*
    ** Define the order-by list.
    */
    if (srv_orderby(spp, numcols, collist) != CS_SUCCEED)
    {
        return(CS_FAIL);
```

```
        }
        return(CS_SUCCEED);
    }
```

Usage

- srv_orderby is necessary only if you want to mimic Adaptive Server's feature of returning order-by information.
- srv_orderby allows an Open Server application to return information about sort order to a client. In the SQL command:

  ```
  select a, b, c, d
   order by c, a
  ```

  The sort order is column c followed by column a. The application returns this information to the client by listing column 3 followed by column 1 in the column number array.
- The first column in a select list is column 1.
- srv_orderby must be called after a call to srv_descfmt and before a call to srv_bind.

srv_poll (UNIX only)

Description

Check for I/O events on file descriptors for a set of open streams.

Syntax

CS_INT srv_poll(fdsp, nfds, waitflag)

```
SRV_POLLFD      *fdsp;
CS_INT          nfds;
CS_INT          waitflag;
```

Parameters

fdsp

A pointer to an array of SRV_POLLFD structures with one element for each open file descriptor of interest. The SRV_POLLFD structure has the following members:

```
CS_INT   srv_fd;          /* File descriptor. */
 CS_INT   srv_events;      /* Relevant events. */
 CS_INT   srv_revents;     /* Returned events. */
```

nfds

The number of elements in the **fdsp* array.

waitflag

A CS_INT value that indicates whether the thread should be suspended until a file descriptor is available for the desired operation. If set to SRV_M_WAIT, the thread is suspended and will wake when any file descriptor represented in the **fdsp* array is available for the specified operation. If the flag is set to SRV_M_NOWAIT, srv_poll will perform a single check and return to the caller. A return status greater than zero indicates that a file descriptor was available for the desired operation.

Return value

Table 3-79: Return values (srv_poll)

Returns	To indicate
An integer	The number of &ready file descriptors.
-1	The routine failed.
0	No file descriptors are &ready.

Examples

```
#include   <ospublic.h>
/*
** Local Prototype
*/
CS_RETCODE      ex_srv_pollPROTOTYPE((
struct pollfd   *fdp,
CS_INT          nfds
));

/*
** EX_SRV_POLL
**
**   This routine demonstrates how to use srv_poll to poll
**   application-specific file descriptors.
**
** Arguments:
**         fdp - The address of the file descriptor array.
**         nfds - The number of file descriptors to poll.
**
** Returns
**
**    CS_SUCCEED    If the data address is returned.
**    CS_FAIL       If the call to srv_poll failed.
**
*/
CS_RETCODE      ex_srv_poll(fdp, nfds)
struct pollfd   *fdp;
CS_INT          nfds;
{
```

```
        /*
        ** Initialization.
        */
        lp = (CS_VOID *)NULL;

        /*
        ** Calls srv_poll to check if any of these file
        ** descriptors are active; ask to sleep until at
        ** least one of them is.
        */
        if( srv_poll(fdp, nfds, SRV_M_WAIT) == (CS_INT)-1 )
        {
            return CS_FAIL;
        }

        /*
        ** All done.
        */
        return CS_SUCCEED;
    }
```

Usage

- An application can use srv_poll to poll the file descriptor or to suspend a thread until there is I/O to be performed.
- Table 3-80 summarizes legal values for srv_events and srv_revents:

Table 3-80: Values for srv_events and revents (srv_poll)

Value	Description
SRV_POLLIN	Normal read event.
SRV_POLLPRI	Priority event received.
SRV_POLLOUT	File descriptor is writable.
SRV_POLLERR	Error occurred on file descriptor.
SRV_POLLHUP	A hang up occurred on the file descriptor. This value is valid in returned events only.
SRV_POLLNVAL	Invalid file descriptor specified in SRV_POLLFD.

- srv_poll is available on all UNIX platforms.

Note If an application uses srv_poll on a UNIX platform that supports the native poll(2) system call, the application must include <*sys/poll.h*> before *ospublic.h*.

See also

srv_capability, srv_select (UNIX only)

srv_props

Description

Define and retrieve Open Server properties.

Syntax

```
CS_RETCODE srv_props(cp, cmd, property, bufp, buflen,
        outlenp)

CS_CONTEXT      *cp;
CS_INT          cmd;
CS_INT          property;
CS_VOID          *bufp;
CS_INT          buflen;
CS_INT          *outlenp;
```

Parameters

scp

A pointer to a CS_CONTEXT structure previously allocated using cs_ctx_alloc.

cmd

The action to take. Table 3-81 summarizes the legal values for *cmd*:

Table 3-81: Values for cmd (srv_props)

Value	Meaning
CS_SET	The Open Server application is setting the property. In this case, *bufp* should contain the value the property is to be set to, and *buflen* should specify the size, in bytes, of that value.
CS_GET	The Open Server application is retrieving the property. In this case, *bufp* should point to the buffer where the property value is placed, and *buflen* should be the size, in bytes, of the buffer.
CS_CLEAR	The Open Server application is resetting the property to its default value. In this case, *bufp*, *buflen*, and *outlenp* are ignored.

property

The property being set, retrieved or cleared. See below for a list of this argument's legal values.

bufp

A pointer to the Open Server application data buffer where property value information is placed (CS_SET) or property value information is retrieved (CS_GET).

buflen

The length, in bytes, of the buffer.

outlenp

A pointer to a CS_INT variable, which Open Server will set to the size, in bytes, of the property value retrieved. This argument is only used when *cmd* is CS_GET, and is optional.

Return value

Table 3-82: Return values (srv_props)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include<ospublic.h>
/*
** Local prototype
*/
CS_RETCODE ex_srv_set_propPROTOTYPE((
CS_CONTEXT *cp,
CS_INT     property,
CS_VOID    *bufp,
CS_INT     buflen
));
/*
** EX_SRV_SET_PROP
**
**    Example routine to set a property using srv_props.
**
** Arguments:
**
**    *cp       Pointer to a CS_CONTEXT structure previously
**              allocated by cs_ctx_alloc.
**    property  The property being set.
**    *bufp     Pointer to the value the property is to be
**              set to.
**    buflen    The length of the value.
**
** Returns
**
**   CS_SUCCEED   Arguments were valid and srv_props was called.
**   CS_FAIL      An error was detected.
**
*/
CS_RETCODE      ex_srv_set_prop(cp, property, bufp, buflen)
CS_CONTEXT      *cp;
CS_INT          property;
CS_VOID         *bufp;
CS_INT          buflen;
{
      /* Check arguments. */
      if(cp == (CS_CONTEXT *)NULL)
      {
```

```
            return(CS_FAIL);
        }
        if(buflen < 1)
            return(CS_FAIL);
    return(srv_props(cp,(CS_INT)CS_SET,property,bufp,buflen,
      (CS_INT *)0));
    }
```

Usage

- srv_props is called to define and retrieve server-wide configuration parameters and properties.
- srv_version must be called before srv_props can be called.
- All properties to be set by srv_props (except SRV_S_TRACEFLAG, SRV_S_LOGFILE, and SRV_S_TRUNCATELOG) must be set before srv_init is called.
- After srv_init is called, setting the SRV_S_LOGFILE property with *bufp* set to an empty string ("") and *buflen* set to zero will close the log file.
- Table 3-83 summarizes the server properties, whether they can be set or retrieved, and the datatype of each property value:

Table 3-83: Server properties and their datatypes (srv_props)

Property	SET/ CLEAR	GET	bufp when cmd is CS_SET	bufp when cmd is CS_GET
SRV_S_ALLOCFUNC	Yes	Yes	A function pointer	The address of a function pointer
SRV_S_APICHK	Yes	Yes	A CS_BOOL	A CS_BOOL
SRV_S_ATTNREASON	No	Yes	Not applicable	A CS_INT
SRV_S_CERT_AUTH	Yes	Yes	char*	char*
SRV_S_CURTHREAD	No	Yes	Not applicable	The address of a thread pointer
SRV_S_DISCONNECT	Yes	Yes	A CS_BOOL	A CS_BOOL
SRV_S_DEFQUEUESIZE	Yes	Yes	A CS_INT	A CS_INT
SRV_S_DS_PROVIDER	Yes	Yes	A pointer to a character string	A pointer to a character string
SRV_S_DS_REGISTER	Yes	Yes	A CS_BOOL	A CS_BOOL
SRV_S_ERRHANDLE	Yes	Yes	A function pointer	The address of a function pointer
SRV_S_FREEFUNC	Yes	Yes	A function pointer	The address of a function pointer
SRV_S_IFILE	Yes	Yes	A character string	A character string

Property	SET/ CLEAR	GET	bufp when cmd is CS_SET	bufp when cmd is CS_GET
SRV_S_LOGFILE	Yes	Yes	A character string	A character string
SRV_S_LOGSIZE	Yes	Yes	A CS_INT	A CS_INT
SRV_S_MSGPOOL	Yes	Yes	A CS_INT	A CS_INT
SRV_S_NETBUFSIZE	Yes	Yes	A CS_INT	A CS_INT
SRV_S_NETTRACEFILE	Yes	Yes	A character string	A character string
SRV_S_NUMCONNECTIONS	Yes	Yes	A CS_INT	A CS_INT
SRV_S_NUMMSGQUEUES	Yes	Yes	A CS_INT	A CS_INT
SRV_S_NUMMUTEXES	Yes	Yes	A CS_INT	A CS_INT
SRV_S_NUMREMBUF	Yes	Yes	A CS_INT	A CS_INT
SRV_S_NUMREMSITES	Yes	Yes	A CS_INT	A CS_INT
SRV_S_NUMTHREADS	Yes	Yes	A CS_INT	A CS_INT
SRV_S_NUMUSEREVENTS	Yes	Yes	A CS_INT	A CS_INT
SRV_S_PREEMPT	Yes	Yes	A CS_BOOL	A CS_BOOL
SRV_S_REALLOCFUNC	Yes	Yes	A function pointer	The address of a function pointer
SRV_S_RETPARMS	Yes	Yes	A CS_BOOL	A CS BOOL
SRV_S_REQUESTCAP	Yes	Yes	A CS_CAP_TYPE structure	A CS_CAP_TYPE structure
SRV_S_RESPONSECAP	Yes	Yes	A CS_CAP_TYPE structure	A CS_CAP_TYPE structure
SRV_S_SEC_KEYTAB	Yes	Yes	A pointer to a character string	A pointer to a character string
SRV_S_SEC_PRINCIPAL	Yes	Yes	A pointer to a character string	A pointer to a character string
SRV_S_SERVERNAME	No	Yes	A character string	A character string
SRV_S_SSL_CIPHER	Yes	No	char*	
SRV_S_SSL_LOCAL_ID	Yes	Yes	struct	char*
SRV_S_SSL_VERSION	Yes	No	CS_INT	
SRV_S_STACKSIZE	Yes	Yes	A CS_INT	A CS_INT
SRV_S_TDSVERSION	Yes	Yes	A CS_INT	A CS_INT
SRV_S_TIMESLICE	Yes	Yes	A CS_INT	A CS_INT

Property	SET/ CLEAR	GET	bufp when cmd is CS_SET	bufp when cmd is CS_GET
SRV_S_TRACEFLAG	Yes	Yes	A CS_INT (bit mask)	A CS_INT (bit mask)
SRV_S_TRUNCATELOG	Yes	Yes	A CS_BOOL	A CS_BOOL
SRV_S_USESRVLANG	Yes	Yes	A CS_BOOL	A CS_BOOL
SRV_S_VERSION	No	Yes	Not applicable	A character string
SRV_S_VIRTCLKRATE	Yes	Yes	A CS_INT	A CS_INT
SRV_S_VIRTTIMER	Yes	Yes	A CS_BOOL	A CS_BOOL

- Table 3-84 lists the default value for each server property:

Table 3-84: Legal properties and their default values (srv_props)

Property	Default
SRV_S_ALLOCFUNC	malloc()
SRV_S_APICHK	CS_TRUE
SRV_S_ATTNREASON	No default
SRV_S_CURTHREAD	N/A.
SRV_S_DEFQUEUESIZE	SRV_DEF_DEFQUEUESIZE
SRV_S_DISCONNECT	CS_FALSE
SRV_S_DS_PROVIDER	Platform dependent. Refer to the Open Client and Open Server *Configuration Guide* for your platform.
SRV_S_DS_REGISTER	CS_TRUE, Server-Library registers itself with a directory on start-up.
SRV_S_ERRHANDLE	No error handler
SRV_S_FREEFUNC	free()
SRV_S_IFILE	*$SYBASE/interfaces*
SRV_S_LOGFILE	*srv.log*
SRV_S_LOGSIZE	Max integer value
SRV_S_MSGPOOL	SRV_DEF_MSGPOOL
SRV_S_NETBUFSIZE	SRV_DEF_NETBUFSIZE
SRV_S_NETTRACEFILE	*sybnet.dbg*
SRV_S_NUMCONNECTIONS	SRV_DEF_NUMCONNECTIONS
SRV_S_NUMMSGQUEUES	SRV_DEF_NUMMSGQUEUES
SRV_S_NUMMUTEXES	SRV_DEF_NUMMUTEXES
SRV_S_NUMREMBUF	SRV_DEF_NUMREMBUF
SRV_S_NUMREMSITES	SRV_DEF_NUMREMSITES
SRV_S_NUMTHREADS	SRV_DEF_NUMTHREADS

Property	Default
SRV_S_NUMUSEREVENTS	SRV_DEF_NUMUSEREVENTS
SRV_S_PREEMPT	CS_FALSE
SRV_S_REALLOCFUNC	realloc()
SRV_S_REQUESTCAP	See “Capabilities” on page 24
SRV_S_RESPONSECAP	See “Capabilities” on page 24
SRV_S_RETPARMS	No default
SRV_S_SEC_KEYTAB	No default
SRV_S_SEC_PRINCIPAL	Security mechanism dependent
SRV_S_SERVERNAME	DSLISTEN environment variable
SRV_S_STACKSIZE	SRV_DEF_STACKSIZE
0	SRV_TDS_5_0
SRV_S_TIMESLICE	SRV_DEF_TIMESLICE
SRV_S_TRACEFLAG	0
SRV_S_TRUNCATELOG	CS_FALSE
SRV_S_USESRVLANG	CS_TRUE
SRV_S_VERSION	Compile-time version string
SRV_S_VIRTCLKRATE	SRV_DEF_VIRTCLKRATE
SRV_S_VIRTTIMER	CS_FALSE

- All server properties that have a default and are settable can be reset back to the default value by calling srv_props with *cmd* set to CS_CLEAR.
- All server properties can be retrieved at any time by calling srv_props with *cmd* set to CS_GET. If the Open Server application has not defined a value for a property, the default value is returned.
- For a description of properties, see the Properties topic page.
- When a property is being retrieved, if *buflen* indicates that the user buffer is not big enough to hold the property value, Open Server will place the number of bytes required in **outlenp*, and the user buffer will not be modified.
- The default stacksize (default value for SRV_S_STACKSIZE) depends on the platform used.

  For native-threaded versions of Open Server, the default stacksize of underlying threads is used. This value can be changed by setting the stacksize with the SRV_S_STACKSIZE property.

  Note that when setting the stacksize, stack overflow errors may occur if the specified stacksize is too small.

See also srv_init, srv_thread_props, srv_spawn, “Properties” on page 139

srv_putmsgq

Description Put a message into a message queue.

Syntax CS_RETCODE srv_putmsgq(msgqid, msgp, putflags)

```
SRV_OBJID       msgqid;
CS_VOID         *msgp;
CS_INT          putflags;
```

Parameters

msgqid

The identifier for the message queue. If you want to reference the message queue by name, call srv_getobjid to look up the name and return the message queue ID.

msgp

A pointer to the message. The message data must be valid until it is received and processed.

putflags

The values for *putflags* can be OR’d together. Table 3-85 describes each value’s significance:

Table 3-85: Values for putflags (srv_putmsgq)

Value	Description
SRV_M_NOWAIT	When this flag is set, the call to srv_putmsgq returns immediately after the message is placed in the message queue.
SRV_M_WAIT	When SRV_M_WAIT is set, srv_putmsgq does not return until either the message is read or the queue is deleted.
SRV_M_URGENT	If this flag is set, the message is put at the head of the list of messages in the message queue instead of at the end. If more than one urgent message is added to a given queue, the urgent messages will appear at the head of the queue in the order in which they were enqueued.

Return value

Table 3-86: Return values (srv_putmsgq)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>

/*
 ** Local Prototype.
 */
CS_RETCODE ex_srv_putmsgq PROTOTYPE((
SRV_OBJID  mqid,
CS_INT     flags
));
/*
 ** EX_SRV_PUTMSGQ
**
**    Example routine to put a message into a message queue.
**
** Arguments:
**   msgqid    Message queue identifier.
**   putflags  Special instructions for srv_putmsgq.
**
** Returns:
**
**   CS_SUCCEED
**   CS_FAIL
 */
CS_RETCODE     ex_srv_putmsgq(mqid, flags)
SRV_OBJID      mqid;
CS_INT         flags;
```

```
{
     CS_CHAR   *msgp;
    msgqp = srv_alloc(20);
     strcpy(msgp, "Hi there");
     return(srv_putmsgq(mqid, msgp, flags));
}
```

Usage

- srv_putmsgq puts the message in **msgp* into the message queue *msgqid*.
- A message is always passed as a pointer. The data the message points to must remain valid even if the thread sending the message changes context.

  In particular, be cautious when passing a message that points to a stack address in the context of the thread that sends the message. If you do this, you must guarantee that the thread that sends the message does not return from the frame in which it sent the message until the message has been removed from the queue. Otherwise, the message may point to a stack that is being used for other purposes.
- The SRV_S_NUMMSGQUEUES server property determines the number of message queues available to an Open Server application. Refer to "Server properties" on page 141 for more information.
- The SRV_S_MSGPOOL server property determines the number of messages available to an Open Server application at runtime. Refer to "Server properties" on page 141 for more information.

See also srv_createmsgq, srv_deletemsgq, srv_getmsgq, srv_getobjid

srv_realloc

Description Reallocate memory.

Syntax

```
CS_VOID* srv_realloc(mp,newsize)

CS_VOID       *mp;
CS_INT        newsize;
```

Parameters

mp
A pointer to the old block of memory.

newsize
The number of bytes to reallocate.

Return value

Table 3-87: Return values (srv_realloc)

Returns	To indicate
A pointer to the newly allocated space	The location of the new space.
A null pointer	Server-Library could not allocate *newsize* bytes.

Examples

```
#include   <ospublic.h>

/*
** Local Prototype.
*/
extern CS_RETCODE ex_srv_realloc PROTOTYPE((
CS_VOID       *mp,
CS_INT       newsize
));

/*
** EX_SRV_REALLOC
**
**   Reallocate a memory chunk.
**
** Arguments:
**    mp        A pointer to existing memory block.
**    newsize   The new size of the memory block.
**
** Returns:
**    CS_SUCCEED   Memory was allocated successfully.
**    CS_FAIL      An error was detected.
*/
CS_RETCODE      ex_srv_realloc(mp, newsize)
CS_VOID         *mp;
CS_INT          newsize;
{
      mp = srv_realloc(mp, newsize);

      if(mp == (CS_VOID *)NULL)
      {
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

Usage

- srv_realloc reallocates memory dynamically.

- It changes the size of the block referenced by *mp* to *newsize*, and returns a pointer to the (possibly moved) block.
- Any memory allocated using srv_realloc should be freed by calling srv_free.
- Use srv_realloc wherever the standard C memory-allocation routines would be used.
- Currently, srv_realloc calls the C routine, *realloc*. An Open Server application, however, can install its own memory management routines using the srv_props routine. The parameter-passing conventions of the user-installed routines must be the same as those of *realloc*. If the application is not configured to use the user-installed routines, Open Server will call *realloc*.

See also srv_alloc, srv_free, srv_props

srv_recvpassthru

Description Receive a protocol packet from a client.

Syntax

```
CS_RETCODE srv_recvpassthru(spp, recv_bufp, infop)

SRV_PROC        *spp;
CS_BYTE         **recv_bufp;
CS_INT          *infop;
```

Parameters

spp
A pointer to an internal thread control structure.

recv_bufp
A pointer to a CS_BYTE pointer that will receive the starting address of the buffer containing the received protocol packet.

infop
A pointer to a CS_INT that is set to SRV_I_UNKNOWN if srv_recvpassthru returns CS_FAIL. Table 3-88 describes the possible values returned in **infop* if srv_recvpassthru returns CS_SUCCEED:

Table 3-88: CS_SUCCEED values (srv_recvpassthru)

Value	Description
SRV_I_PASSTHRU_MORE	A protocol packet was read successfully and is not the end of message packet.
SRV_I_PASSTHRU_EOM	The packet is the end of message packet.

Return value

Table 3-89: Return values (srv_recvpassthru)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include      <ospublic.h>
/*
 ** Local prototype.
 */
CS_RETCODE      ex_srv_recvpassthru PROTOTYPE((
CS_VOID         *spp
));
/*
** EX_SRV_RECVPASSTHRU
**
**   Example routine to receive protocol packets from a client.
**
** Arguments:
**   spp    A pointer to an internal thread control structure.
**
** Returns:
**   CS_SUCCEED If we were able to receive the packets.
**   CS_FAIL If were unsuccessful at receiving the packets.
**
*/
CS_RETCODE    ex_srv_recvpassthru(spp)
SRV_PROC    *spp;
{
     CS_RETCODE    result;
     CS_BYTE       *recvbuf;
     CS_INT        info;

     /*
     ** Read packets until we get the EOM flag.
     */
     do
     {
```

```
                result = srv_recvpassthru(spp, &recvbuf, &info);
        }
        while (result == CS_SUCCEED && info == SRV_I_PASSTHRU_MORE);

        return (result);
    }
```

Usage

- srv_recvpassthru receives a protocol packet without interpreting its contents.
- Once srv_recvpassthru is called, the event handler that called it is in "passthrough" mode. Passthrough mode ends when SRV_I_PASSTHRU_EOM is returned in **infop*.
- No other Server-Library routines can be called while the event handler is in passthrough mode.
- In passthrough mode, the SRV_CONNECT handler for the client must allow the client and remote server to negotiate the protocol packet format by calling srv_getloginfo, ct_setloginfo, ct_getloginfo, and srv_setloginfo. This allows clients and remote servers running on dissimilar platforms to perform any necessary data conversions.
- srv_recvpassthru can be called in all event handlers except SRV_START, SRV_CONNECT, SRV_STOP, SRV_DISCONNECT, SRV_URGDISCONNECT, and SRV_ATTENTION.
- Once it has called srv_recvpassthru, an application cannot call any other routine that does I/O until it has issued a srv_senddone.

See also srv_getloginfo, srv_sendpassthru, srv_setloginfo

srv_regcreate

Description Complete the registration of a registered procedure.

Syntax

```
CS_RETCODE srv_regcreate(spp, infop)

SRV_PROC        *spp;
CS_INT          *infop;
```

Parameters

spp
A pointer to an internal thread control structure.

infop

A pointer to a CS_INT. Table 3-90 describes the possible values returned in **infop* if srv_regcreate returns CS_FAIL:

Table 3-90: Values for infop (srv_regcreate)

Value	Description
SRV_I_PEXISTS	The procedure is al&ready registered.
SRV_I_UNKNOWN	Some other error occurred.

Return value

Table 3-91: Return values (srv_regcreate)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include  <ospublic.h>

/*
** Local Prototype.
*/
CS_INT    ex_srv_regcreate PROTOTYPE((
SRV_PROC  *sproc
));

/*
** EX_SRV_REGCREATE
**    An example routine that completes the registration of a
**    registered procedure using srv_regcreate.
**
** Arguments:
**    sproc  A pointer to an internal thread control structure.
**
** Returns:
**    CS_SUCCEED   If the procedure was registered successfully.
**    CS_FAIL      If the supplied internal control structure is
 **                NULL.
**    SRV_I_EXIST       If the procedure is al&ready registered.
**    SRV_I_UNKNOWN     If some other error occurred.
*/
CS_INT    ex_srv_regcreate(sproc)
SRV_PROC  *sproc;
{
      CS_INT    info;     /* The reason for failure */
```

```
        /*
        ** Check whether the internal control structure is NULL.
        */
        if ( sproc == (SRV_PROC *)NULL )
        {
             return((CS_INT)CS_FAIL);
        }

        /*
        ** Now register the procedure al&ready defined by
        ** srv_regdefine and(or) srv_regparam. If an error
        ** occurred, return the cause of error.
        */
        if ( srv_regcreate(sproc, &info) == CS_FAIL )
        {
               return(info);
        }

        /* The procedure is registered. */
        return((CS_INT)CS_SUCCEED);
    }
```

Usage

- After all information needed to register a procedure has been provided, srv_regcreate completes the registration.
- The procedure's name and parameters must have been previously defined with srv_regdefine and srv_regparam respectively.
- Once registered, the procedure can be invoked by a client application or from within an Open Server application program.
- See srv_regdefine, for an example that registers a procedure.

See also srv_regdefine, srv_regdrop, srv_reglist, srv_regparam

srv_regdefine

Description Initiate the process of registering a procedure.

Syntax

```
CS_RETCODE srv_regdefine(spp, procnamep,
                namelen, funcp)
SRV_PROC        *spp;
CS_CHAR         *procnamep;
CS_INT          namelen;
SRV_EVENTHANDLE_FUNC(*funcp)();
```

Parameters

spp
A pointer to an internal thread control structure.

procnamep
A pointer to the name of the procedure.

namelen
The length of the procedure name. If the string in **proc_namep* is null terminated, *namelen* can be CS_NULLTERM.

funcp
A pointer to the function to be called each time the procedure is executed. Setting this parameter to null registers a "notification" procedure. Notification procedures are useful for inter-client communication. For more information on notification procedure, see "Registered procedures" on page 162.

Return value

Table 3-92: Return values (srv_regdefine)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>
#include    <stdio.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_regdefine PROTOTYPE((
SRV_SERVER      *server
));
CS_RETCODE      stop_serv PROTOTYPE((
SRV_PROC        *spp
));

/*
** Local defines.
*/
#define   STOP_SERV   "stop_serv"

/*
** STOP_SERV
**    This function is called when the client sends the stop_serv
**    registered procedure.
**
```

```
** Arguments:
**    spp A pointer to internal thread control structure.
**
** Returns:
**    SRV_CONTINUE
*/
CS_INT     stop_serv(spp)
SRV_PROC   *spp;
{
 /* Queue a SRV_STOP event. */
 (CS_VOID)srv_log((SRV_SERVER *)NULL, CS_TRUE,
           "Stopping Server\n", CS_NULLTERM);

     /* Send a final DONE to client to acknowledge the command. */
     if (srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED,
                (CS_INT)0)
                == CS_FAIL)
     {
            fprintf(stderr, "srv_senddone failed\n");
     }

     /* Queue a SRV_STOP event to shut down the server. */
 if (srv_event(spp, SRV_STOP, (CS_VOID *)NULL)
           == CS_FAIL)
     {
           fprintf(stderr, "Error queuing SRV_STOP event\n");
     }
 return (SRV_CONTINUE);
}

/*
** EX_SRV_REGDEFINE
**
**    Example routine to illustrate the use of srv_regdefine to
**    register a procedure.
**
** Arguments:
**   server       A pointer to the Open Server control structure.
**
** Returns:
**
**   CS_SUCCEED  If procedure was registered successfully.
**   CS_FAIL     If an error occurred in registering the
**                  procedure.
*/
CS_RETCODE      ex_srv_regdefine (server)
```

```
SRV_SERVER      *server;
{
      SRV_PROC  *spp;
      CS_INT    info;

      /* Create a thread. */
      spp = srv_createproc(server);

      if (spp == (SRV_PROC *)NULL)
           return (CS_FAIL);

      /* Define the procedure. */
      if (srv_regdefine(spp, STOP_SERV, CS_NULLTERM, stop_serv)
           == CS_FAIL)
           return (CS_FAIL);

      /* Complete the registration. */
      if (srv_regcreate(spp, &info) == CS_FAIL)
           return (CS_FAIL);

      /*
      ** Terminate the thread created here. We do not care about
      ** the return code from srv_termproc here.
      */
      (CS_VOID)srv_termproc(spp);

      return (CS_SUCCEED);
}
```

Usage

- srv_regdefine is the first step in the process of registering a procedure. Once it is registered, a procedure can be invoked by clients or from within the Open Server application program.
- After calling srv_regdefine, define the procedure's parameters with srv_regparam.
- Complete the processing of registering the procedure by calling srv_regcreate.
- If a registered procedure exists with a name identical to the one in *procnamep*, the error is detected and reported when srv_regcreate is called.
- All requested procedures should return SRV_CONTINUE.

See also

srv_regcreate, srv_regdrop, srv_reglist, srv_regparam

srv_regdrop

Description

Drop or “unregister” a procedure.

Syntax

```
CS_RETCODE srv_regdrop(spp, procnamep,
                namelen, info)

SRV_PROC        *spp;
CS_CHAR         *procnamep;
CS_INT           namelen;
CS_INT          *infop;
```

Parameters

spp

A pointer to an internal thread control structure.

procnamep

A pointer to the name of the procedure.

namelen

The length of the registered procedure name. If the name is null terminated, *namelen* can be CS_NULLTERM.

infop

A pointer to a CS_INT. If srv_regdrop returns CS_FAIL, the flag is set to one of the following values:

- SRV_I_PNOTKNOWN – the procedure was not registered.
- SRV_I_UNKNOWN – some other error occurred.

Return value

Table 3-93: Return values (srv_regdrop)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE      ex_srv_regdrop PROTOTYPE((
SRV_PROC        *spp,
CS_CHAR         *name,
CS_INT          namelen,
CS_INT          *infop
));

/*
```

```
** EX_SRV_REGDROP
**
**    Example routine to unregister a registered procedure using
**    srv_regdrop.
**
** Arguments:
**   spp       A pointer to an internal thread control structure.
**   name      The name of the registered procedure to drop.
**   namelen   The length of the registered procedure name.
**   infop     A return pointer to an integer containing more
**             descriptive error information if this routine
**             returns CS_FAIL.
**
** Returns:
**   CS_SUCCEED    Registered procedure was successfully deleted.
**   CS_FAIL       Registered procedure was not deleted or does
**                  not exist.
*/
CS_RETCODE     ex_srv_regdrop(spp, name, namelen, infop)
SRV_PROC       *spp;
CS_CHAR        *name;
CS_INT         namelen;
CS_INT         *infop;
{
      /* Initialization. */
      *infop = (CS_INT)0;
      /* Execute the procedure. */
      if (srv_regdrop(spp, name, namelen, infop) != CS_SUCCEED)
      {
           /* Open Server has set infop to a specific error. */
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

Usage

- srv_regdrop drops a procedure previously registered with srv_regcreate.
- Any client threads waiting for notification of this procedure are informed that the procedure has been dropped.

See also

srv_regcreate, srv_regdefine, srv_reglist, srv_regparam

srv_regexec

Description

Execute a registered procedure.

Syntax

```
CS_RETCODE srv_regexec(spp, infop)

SRV_PROC        *spp;
CS_INT          *infop;
```

Parameters

spp

A pointer to an internal thread control structure.

infop

A pointer to a CS_INT. Table 3-94 describes the possible values returned in **infop* if srv_regexec returns CS_FAIL:

Table 3-94: Values for infop (srv_regexec)

Value	Description
SRV_I_PNOTKNOWN	The procedure is not registered.
SRV_I_PPARAMERR	There is a parameter error.
SRV_I_PNOTIFYERR	An error occurred while sending notifications.

Return value

Table 3-95: Return values (srv_regexec)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE      ex_srv_regexec PROTOTYPE((
SRV_PROC        *spp,
CS_INT          &infop
));

/*
** EX_SRV_REGEXEC
**
**   Example routine to complete the execution of a registered
**   procedure using srv_regexec. This routine should be called
**   after srv_reginit and srv_regparam.
**
** Arguments:
```

```
**   spp       A pointer to an internal thread control structure.
**   infop     A return pointer to an integer containing more
**             descriptive error information if this routine
**             returns CS_FAIL.
**
** Returns:
**    CS_SUCCEED    Registered procedure executed successfully.
**    CS_FAIL       Registered procedure not executed, or
**                  notifications not completed successfully.
*/
CS_RETCODE ex_srv_regexec(spp, infop)
SRV_PROC   *spp;
CS_INT     &infop;
{
      /* Initialization. */
      &infop = (CS_INT)0;

      /* Execute the procedure. */
      if (srv_regexec(spp, infop) != CS_SUCCEED)
      {
           /*
           ** Open Server has set the argument to a specific
           ** error.
           */
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

Usage

- srv_regexec executes a registered procedure.
- The procedure name and its parameters must be specified with srv_reginit and srv_regparam before calling srv_regexec.

Warning! Open Server system registered procedures send a final DONE. If an application executes a system registered procedure from an event handler using srv_regexec, the application must not send a final DONE from the event handler code. Doing so will cause Open Server to raise a state error.

See also

srv_reginit, srv_regparam

srv_reginit

Description

Begin executing a registered procedure.

Syntax

```
CS_RETCODE srv_reginit(spp, procnamep,
                namelen, options)

SRV_PROC        *spp;
CS_CHAR         *procnamep;
CS_INT           namelen;
unsigned short   options;
```

Parameters

spp

A pointer to an internal thread control structure.

procnamep

A pointer to the name of the registered procedure.

namelen

The length of the procedure name. If the name is null terminated, *namelen* can be CS_NULLTERM.

options

A flag that determines which threads to notify. Table 3-96 describes the legal values for *options*:

Table 3-96: Values for options (srv_reginit)

Value	Description
SRV_M_PNOTIFYALL	Notify all waiting threads in the notification list.
SRV_M_PNOTIFYNEXT	Notify only the thread that has been waiting the longest.

Return value

Table 3-97: Return values (srv_reginit)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE     ex_srv_reginit  PROTOTYPE((
SRV_PROC       *sp,
CS_CHAR        *pname,
CS_INT         nlen
));
```

```
/*
** EX_SRV_REGINIT
**
**    This routine demonstrates how to use srv_reginit to
**    initiate the execution of a registered procedure.
**
** Arguments:
**         sp       A pointer to an internal thread control
                    structure.
**         pname    The name of the procedure to execute.
**         nlen     The length of the procedure name.
** Returns
**
**    CS_SUCCEED    If the registered procedure began execution.
**    CS_FAIL       If an error was detected.
**
*/
CS_RETCODE      ex_srv_reginit(sp, pname, nlen)
SRV_PROC        *sp;
CS_CHAR         *pname;
CS_INT          nlen;
{
      /*
      ** Call srv_reginit to initiate the execution of this
      ** registered procedure; ask that all threads waiting for
      ** notification of this event be notified.
      */
      if( srv_reginit(sp, pname, nlen, SRV_M_PNOTIFYALL) ==
            CS_FAIL )
      {
           /*
           ** An error was al&ready raised.
           */
           return CS_FAIL;
      }

      /*
      ** All done.
      */
      return CS_SUCCEED;
}
```

Usage

- srv_reginit is the first step in the process of executing a registered procedure.
- The procedure's parameters are defined with srv_regparam after srv_reginit has been called.

- Call srv_regexec to execute the registered procedure.
- If the procedure does not exist, the error is detected and reported by srv_regexec.
- When a registered procedure is executed, Open Server notifies the threads in the procedure's notification list. The *options* parameter specifies whether notifications are sent to all threads in the list, or just the one that has been waiting the longest.
- An Open Server application can nest registered procedures up to a maximum of 16 levels.

See also srv_regexec, srv_regparam

srv_reglist

Description Obtain a list of all of the procedures registered in the Open Server.

Syntax

```
CS_RETCODE srv_reglist(spp, proclistp)

SRV_PROC        *spp;
SRV_PROCLIST    **proclistp;
```

Parameters

spp
A pointer to an internal thread control structure.

proclistp
A pointer to a SRV_PROCLIST pointer that will be set to the address of a SRV_PROCLIST containing the results. The Open Server allocates the space for this structure at the time srv_reglist is called.

Return value

Table 3-98: Return values (srv_reglist)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include   <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_reglist PROTOTYPE((
SRV_PROC        *spp,
SRV_PROCLIST    **proclp
```

```
));

/*
** EX_SRV_REGLIST
**
** Arguments:
**
**   spp       Pointer to an internal thread control structure.
**   proclp    Pointer to a SRV_PROCLIST pointer that will be set
**             to point to the result.
**
** Returns
**
**   CS_SUCCEED    srv_reglist was successful.
**   CS_FAIL       An argument was invalid or srv_reglist failed.
**
*/
CS_RETCODE      ex_srv_reglist (spp, proclp)
SRV_PROC        *spp;
SRV_PROCLIST    **proclp;
{
      /* Check arguments. */
      if(spp == (SRV_PROC *)NULL)
      {
            return(CS_FAIL);
      }
      return(srv_reglist(spp,proclp));
}
```

Usage

- srv_reglist returns a list of all currently registered procedures for the thread.
- The parameter *proclistp* is set to point to a structure that is allocated and initialized by the Open Server. The SRV_PROCLIST structure is defined as follows:

```
typedef struct srv__proclist
 {
      CS_INT        num_procs;      /* The number of procedures */
      CS_CHAR       **proc_list;    /* Array of procedure names */
 } SRV_PROCLIST;
```

- The SRV_PROCLIST structure should be deallocated with srv_reglistfree when it is no longer needed.

See also srv_reglistfree

srv_reglistfree

Description

Free a previously allocated SRV_PROCLIST structure.

Syntax

CS_RETCODE srv_reglistfree(spp, proclistp)

```
SRV_PROC        *spp;
SRV_PROCLIST    *proclistp;
```

Parameters

spp

A pointer to an internal thread control structure.

proc_list

A pointer to a SRV_PROCLIST structure previously allocated by srv_reglist or srv_regwatchlist.

Return value

Table 3-99: Return values (srv_reglistfree)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE      ex_srv_reglistfree PROTOTYPE((
SRV_PROC        *srvproc,
SRV_PROCLIST    *reglistp
));

/*
** EX_SRV_REGLISTFREE
**
**   Example routine to free a previously allocated reglist.
**
** Arguments:
**  srvproc    A pointer to an internal thread control structure.
**  reglistp   A pointer to the list to free.
**
** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE      ex_srv_reglistfree(srvproc, reglistp)
```

```
SRV_PROC        *srvproc;
SRV_PROCLIST    *reglistp;
{
       return(srv_reglistfree(srvproc, reglistp));
}
```

Usage

srv_reglistfree deallocates a SRV_PROCLIST structure allocated by srv_reglist or srv_regwatchlist.

See also

srv_reglist, srv_regwatchlist

srv_regnowatch

Description

Remove a client thread from the notification list for a registered procedure.

Syntax

```
CS_RETCODE srv_regnowatch(spp, procnamep,
                namelen, infop)

SRV_PROC        *spp;
CS_CHAR        *procnamep;
CS_INT          namelen;
CS_INT         *infop;
```

Parameters

spp

A pointer to an internal thread control structure.

procnamep

A pointer to the name of the procedure.

namelen

The length of the procedure name. If the name is null terminated, *namelen* can be CS_NULLTERM.

infop

A pointer to a CS_INT. Table 3-100 describes the possible values returned in **infop* if srv_regnowatch returns CS_FAIL:

Table 3-100: Values for infop (srv_regnowatch)

Value	Description
SRV_I_PNOTCLIENT	A non-client thread was specified.
SRV_I_PNOTKNOWN	The procedure is not known to the Open Server application.
SRV_I_PNOPENDING	The thread is not on the notification list for this procedure.
SRV_I_PPARAMERR	A parameter error occurred.
SRV_I_UNKNOWN	Some other error occurred.

Return value

Table 3-101: Return values (srv_regnowatch)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include   <ospublic.h>

/*
** Local Prototype.
*/
extern CS_RETCODE  ex_srv_regnowatch PROTOTYPE((
CS_VOID         *spp,
CS_CHAR         *procnamep,
CS_INT          namelen
));

/*
** EX_SRV_REGNOWATCH
**
**    Remove a client thread from the notification list for the
**    specified registered procedure.
**
** Arguments:
**   spp           A pointer to an internal thread control
                   structure.
**   procnamep     A pointer to the name of the registered
                   procedure.
**   namelen      The length of the registered procedure name.
**
** Returns:
**   CS_SUCCEED    The thread was removed from notification list.
**   CS_FAIL       An error was detected.
*/
```

```
CS_RETCODE      ex_srv_regnowatch(spp, procnamep, namelen)
SRV_PROC        *spp;
CS_CHAR         *procnamep;
CS_INT          namelen;
{
      if(srv_regnowatch(spp, procnamep, namelen, (CS_INT *)NULL)
           == CS_FAIL)
      {
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

Usage

- srv_regnowatch removes a client thread from the list of threads to notify when the specified procedure executes.
- The maximum length of a procedure name is SRV_MAXNAME.

See also srv_regwatch, srv_regwatchlist

srv_regparam

Description Describe a parameter for a registered procedure being defined, or supply data for the execution of a registered procedure.

Syntax

```
CS_RETCODE srv_regparam(spp, param_namep, namelen,
                type, datalen, datap)

SRV_PROC        *spp;
CS_CHAR         *param_namep;
CS_INT           namelen;
CS_INT           type;
CS_INT          datalen;
CS_BYTE         *datap;
```

Parameters

spp

A pointer to an internal thread control structure.

param_namep

A pointer to the name of the parameter. When registering the procedure, this parameter is mandatory. When invoking the procedure, this parameter can be null if the parameters are given in the same sequence they were defined when the procedure was registered.

namelen
The length of the parameter name. If the *param_namep* is null terminated, *namelen* can be CS_NULLTERM.

type
The datatype of the parameter. See “Types” on page 199 for a list of Open Server datatypes.

datalen
The length of the parameter’s data. This parameter is ignored for fixed length datatypes. Set *datalen* to 0 to indicate a null data value. If a client fails to provide parameter values, the Open Server application can set the length of a default value here. To define a parameter with no default value, set *datalen* to SRV_NODEFAULT.

datap
A pointer to the data. If registering the procedure, the value in **datap* is the default value for future invocations of the procedure. If invoking the procedure, set *datap* to NULL to accept the default value.

Return value

Table 3-102: Return values (srv_regparam)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <ospublic.h>

/*
** Local prototype.
*/
CS_RETCODE  ex_srv_regparam PROTOTYPE((
SRV_PROC    *spp
));

/*
** Local defines.
*/
#define PARAMNAME (CS_CHAR *)”myparam” /* Parameter name. */

#define PARAMDEFAULT (CS_INT)100

/*
**The default value for the parameter.
*/
```

```
#define PARAMVAL (CS_INT)20 /* The value for this invocation. */

/*
** EX_SRV_REGPARAM
**
**    Example routine to describe a parameter for a registered
**    procedure.
**
** Arguments:
**   spp   A pointer to an internal thread control structure.
**
** Returns:
**    CS_SUCCEED If we were able to describe the parameter.
**    CS_FAIL If an error was detected.
*/
CS_RETCODE      ex_srv_regparam(spp)
SRV_PROC        *spp;
{
      CS_RETCODE    result;
      CS_INT        param;

      /* Define the parameter with a default. */
      param = PARAMDEFAULT;
      result = srv_regparam(spp, PARAMNAME, CS_NULLTERM,
             CS_INT_TYPE, sizeof(CS_INT), (CS_BYTE *)&param);

      if (result == CS_FAIL)
      {
            return (CS_FAIL);
      }

      /* Define the parameter with no default. */
      result = srv_regparam(spp, PARAMNAME, CS_NULLTERM,
               CS_INT_TYPE, SRV_NODEFAULT, (CS_BYTE *)NULL);

      if (result == CS_FAIL)
      {
           return (CS_FAIL);
      }

      /* Give a non-default value for the parameter. */
      param = PARAMVAL;
      result = srv_regparam(spp, PARAMNAME, CS_NULLTERM,
                CS_INT_TYPE, sizeof(CS_INT), (CS_BYTE *)&param);
```

```
        return (result);
    }
```

Usage

- srv_regparam specifies a procedure parameter for an invocation of, or the registration of, a procedure. A call to srv_reginit or srv_regdefine must precede srv_regparam.
- A registered procedure can have a maximum of 1024 parameters.
- When registering a procedure, use srv_regparam to define the properties of the procedure's parameters and any default values.
- When invoking a procedure, call srv_regparam for each parameter except those with acceptable default values.
- To indicate a null data value, set *datalen* to 0.
- To accept the default value for a parameter when executing a procedure, set *datap* to NULL.
- It is not necessary to call srv_regparam for a parameter if a default value has been provided and that value is to be used for the execution of the procedure.

See also srv_regcreate, srv_regdefine, srv_reginit, srv_regexec, "Types" on page 199

srv_regwatch

Description Add a client thread to the notification list for a specified procedure.

Syntax

```
CS_RETCODE srv_regwatch(spp, proc_namep, namelen,
                options, infop)

SRV_PROC        *spp;
CS_CHAR         *proc_namep;
CS_INT           namelen;
CS_INT           options;
CS_INT          *infop;
```

Parameters

spp
A pointer to an internal thread control structure.

proc_namep
The name of the procedure.

namelen
The length of the procedure name. If the procedure name is null terminated, *namelen* can be CS_NULLTERM.

options

A flag that specifies whether this is a one-time notification request, or a permanent request. Table 3-103 describes the legal values for *options*:

Table 3-103: Values for options (srv_regwatch)

Value	Description
SRV_NOTIFY_ONCE	After the first notification, the client thread is removed from the notification list for the procedure.
SRV_NOTIFY_ALWAYS	The client thread will be notified each time the procedure executes until srv_regnowatch is used to remove the thread from the procedure's notification list.

infop

Table 3-104 describes the possible values returned in *infop if srv_regwatch returns CS_FAIL:

Table 3-104: Values for infop (srv_regwatch)

Value	Description
SRV_I_PNOTKNOWN	The procedure is not known to the Open Server application. The thread was not added to the notification list.
SRV_I_PINVOPT	An invalid *options* value was specified. The thread was not added to the notification list.
SRV_I_PNOTCLIENT	A non-client thread was specified. The thread was not added to the notification list.
SRV_I_PNOTIFYEXISTS	The thread is al&ready on the notification list for the specified procedure.

Return value

Table 3-105: Return values (srv_regwatch)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include     <ospublic.h>

/*
** Local Prototype.
*/
CS_INT      ex_srv_regwatch PROTOTYPE((
SRV_PROC    *sproc,
CS_CHAR     *procedure_name
```

```
));

/*
** EX_SRV_REGWATCH
**   An example routine to add a client thread to the
**   notification list for a specified procedure.
**
** Arguments:

**   sproc              A pointer to an internal thread control
**                      structure.
**   procedure_name     The null terminated procedure name.
**
** Returns:
**   CS_SUCCEED            If the thread was added to the
**                         notification list.
**   SRV_I_PNOTKNOWN       The procedure is not known to the Open
**                         Server application.
**   SRV_I_PNOTCLIENT      A non-client thread was specified.
**   SRV_I_PNOTIFYEXISTS   The thread is al&ready on the
**                         notification list for the specified
**                         procedure.
**   CS_FAIL               The attempt to add the thread to the
**                         notification failed due to other
**                         errors.
*/
CS_INT      ex_srv_regwatch(sproc, procedure_name)
SRV_PROC    *sproc;
CS_CHAR     *procedure_name;
{
      CS_INT    info;

      if ( srv_regwatch(sproc, procedure_name, CS_NULLTERM,
            SRV_NOTIFY_ALWAYS, &info) == CS_FAIL )
      {
            if ( (info == SRV_I_PNOTKNOWN)
            || (info == SRV_I_PNOTCLIENT)
            || (info == SRV_I_PNOTIFYEXISTS) )
            {
                return(info);
           }
           else
           {
                return((CS_INT)CS_FAIL);
           }
      }
```

```
        return((CS_INT)CS_SUCCEED);
}
```

Usage

- srv_regwatch adds a thread to the list of threads to notify when the specified procedure executes.
- The *options* flag specifies whether the thread is notified every time the procedure executes or just once—the next time the procedure executes.
- Use srv_regnowatch to cancel a notification request.

See also

srv_regnowatch, srv_regwatchlist

srv_regwatchlist

Description

Return a list of all registered procedures for which a client thread has notification requests pending.

Syntax

CS_RETCODE srv_regwatchlist(spp, proclistp)

```
SRV_PROC        *spp;
SRV_PROCLIST    **proclistp;
```

Parameters

spp

A pointer to an internal thread control structure.

proclistp

A pointer to a pointer to a structure that contains the number of registered procedures and the names of each registered procedure. Open Server allocates the space for this structure.

Return value

Table 3-106: Return values (srv_regwatchlist)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include   <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE     ex_srv_regwatchlist PROTOTYPE((
SRV_PROC       *spp
```

```
));

/*
** EX_SRV_REGWATCHLIST
**
**    Example routine to get a list of all registered procedures
**    for which a client thread has notifications pending.
**
** Arguments:
**    spp      A pointer to an internal thread control structure.
**
** Returns:
**
**    CS_SUCCEED    The list returned successfully.
**    CS_FAIL       An error was detected.
*/
CS_RETCODE      ex_srv_regwatchlist(spp)
SRV_PROC        *spp;
{
     SRV_PROCLIST  *listp;

     if (srv_regwatchlist(spp, &listp) == CS_FAIL)
           return (CS_FAIL);

      /*
      **   Process the information in the list and free the
      **   memory allocated for the list.
      */
      (CS_VOID)srv_reglistfree(spp, listp);

      return (CS_SUCCEED);
}
```

Usage

- srv_regwatchlist returns a list of registered procedures for which the client thread has requested notification.
- The *proclistp* parameter points to a SRV_PROCLIST structure that is allocated and initialized by Open Server. The SRV_PROCLIST structure looks like this:

```
typedef struct srv__proclist
{
    CS_INT       num_procs;     /* The number of procedure names */
     CS_CHAR      **proc_list;   /* The list of procedure names */
} SRV_PROCLIST;
```

- An application deallocates the SRV_PROCLIST structure by calling srv_reglistfree.

See also srv_reglistfree

srv_rpcdb

Description Return the database component of the current remote procedure designation.

Syntax

```
CS_CHAR *srv_rpcdb(spp, lenp)

SRV_PROC       *spp;
CS_INT         *lenp;
```

Parameters

spp
A pointer to an internal thread control structure.

lenp
A pointer to an *int* variable that will contain the length of the database name. *lenp* can be NULL, in which case the length of the database name is not returned.

Return value

Table 3-107: Return values (srv_rpcdb)

Returns	To indicate
A pointer to a null terminated string containing the database component of the current RPC's designation.	The location of the database component of the current RPC's designation.
(CS_CHAR *) NULL	There is no current RPC. Open Server sets *lenp* to -1 and raises an informational error.

Examples

```
#include        <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE      ex_srv_rpcdb PROTOTYPE((
SRV_PROC        *spp,
CS_CHAR         **dbp,
CS_INT          *lenp
));
/*
 ** EX_SRV_RPCDB
 **
**   Example routine to return the database component name of the
 **   current remote procedure call designation, using srv_rpcdb.
```

```
**
** Arguments:
**   spp   A pointer to an internal thread control structure.
 **   dbp   A return pointer to the null terminated database name.
 **   lenp  A return pointer to an integer containing the length
 **         of the database name.
**
** Returns:
**   CS_SUCCEED   Database component name returned successfully.
**   CS_FAIL      An error was detected.
 */
CS_RETCODE     ex_srv_rpcdb(spp, dbp, lenp)
SRV_PROC       *spp;
CS_CHAR        **dbp;
CS_INT         *lenp;
{
      /* Initialization.*/
      *lenp = (CS_INT)0;
      /* Retrieve the database component name. */
      if ((*dbp = (CS_CHAR *)srv_rpcdb(spp, lenp)) == (CS_CHAR
            *)NULL)
      {
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

Usage

- srv_rpcdb returns a CS_CHAR pointer to a null terminated string containing the database name component of the current remote procedure call designation.
- srv_rpcdb returns only the database name part of the RPC's designation and does not include anything else, such as optional specifiers for owner or RPC number. A fully qualified stored procedure designation takes the form *database.owner.rpcname;number*. To get the other parts of the RPC's designation, if any, use srv_rpcname, srv_rpcowner, and srv_rpcnumber.

See also srv_numparams, srv_rpcname, srv_rpcnumber, srv_rpcoptions, srv_rpcowner

srv_rpcname

Description Return the name component of the current remote procedure call's designation.

Syntax CS_CHAR *srv_rpcname(spp, lenp)

```
SRV_PROC        *spp;
CS_INT          *lenp;
```

Parameters

spp

A pointer to an internal thread control structure.

lenp

A pointer to the buffer that will contain the length of the RPC name. *lenp* can be NULL, in which case the length of the RPC name is not returned.

Return value

Table 3-108: Return values (srv_rpcname)

Returns	To indicate
A pointer to the null terminated name component of the current RPC's designation.	The location of the database component of the current RPC's designation.
A null pointer	There is no current RPC. Open Server sets *lenp* to -1 and raises an informational error.

Examples

```
#include        <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_rpcname PROTOTYPE((
SRV_PROC        *sp,
CS_CHAR         *buf,
CS_INT          buflen,
CS_INT          *lenp
));

/*
** EX_SRV_RPCNAME
**
**    This routine demonstrates how to use srv_rrpcname to obtain
 **    the name of the remote procedure call received by this
 **    thread.
**
** Arguments:
**   sp       A pointer to an internal thread control
 **             structure.
**   buf      The address of the buffer in which the RPC
 **             name will be returned.
**   buflen   The size of the name buffer.
**   lenp     The address of an integer variable, which
 **             will be set to the length of the name
```

```
 **            returned.
**
** Returns
**   CS_SUCCEED       If the RPC name is returned.
**   CS_FAIL          If an error occurred.
*/
CS_RETCODE      ex_srv_rpcname(sp, buf, buflen, lenp)
SRV_PROC        *sp;
CS_CHAR         *buf;
CS_INT          buflen;
CS_INT          *lenp;
{
CS_CHAR         *np;    /* The procedure name pointer. */

     /*
     ** Initialization.
     */
     np = (CS_CHAR *)NULL;
     *lenp = (CS_INT)0;

     /*
     ** Get the procedure name.
     */
     np = srv_rpcname(sp, lenp);

     if( np == (CS_CHAR *)NULL )
     {
           /*
           ** An error was al&ready raised.
           */
           return CS_FAIL;
     }

     /*
     ** Copy the RPC name to the output buffer.
     */
     (void)strncpy(buf, np, buflen);

     /*
     ** All done.
     */
     return CS_SUCCEED;
}
```

Usage

- srv_rpcname returns a CS_CHAR pointer to a null terminated string containing the name component of the current remote procedure call (“RPC”) designation.

- srv_rpcname returns only the RPC name and does not include anything else, such as optional specifiers for database, owner, or RPC number. For example, a fully qualified object name for an RPC in the Adaptive Server is *database.owner.rpcname;number*. To get the other parts of the RPC's designation, if any, use srv_rpcdb, srv_rpcowner, and srv_rpcnumber.
- A user can determine whether an RPC exists by calling srv_rpcname. If the RPC does not exist, Open Server will return a SRV_ENORPC error. A user can code his or her error handler to ignore this error if detected.

See also

srv_numparams, srv_rpcdb, srv_rpcnumber, srv_rpcoptions, srv_rpcowner

srv_rpcnumber

Description

Return the number component of the current remote procedure call's designation.

Syntax

CS_INT srv_rpcnumber(spp)

SRV_PROC *spp;

Parameters

spp

A pointer to an internal thread control structure.

Return value

Table 3-109: Return values (srv_rpcnumber)

Returns	To indicate
A non-zero integer	The number component of the current RPC's designation.
-1	There is no current RPC. Open Server sets *lenp* to -1 and raises an informational error.
0	The client did not include a number component when it invoked the RPC.

Examples

```
#include    <ospublic.h>
/*
** Local Prototype.
*/
CS_INT     ex_srv_rpcnumber PROTOTYPE((
SRV_PROC   *spp
));

/*
** EX_SRV_RCPNUMBER
```

```
**
**    Example routine to show hiw to get the number of the
 **    current RPC designation.
**
**
** Arguments:
**
**   spp   A pointer to an internal thread control structure.
**
** Returns:
**
**    The number component of the current RPC's designation. If
 **    the client used no number component when it invoked the
 **    RPC, 0 is returned. If there is not a current RPC, -1 is
 **    returned and Open Server raises an informational error.
 */
CS_INT     ex_srv_rpcnumber(spp)
SRV_PROC   *spp;
{
      /* Check arguments. */
      if(spp == (SRV_PROC *)NULL)
      {
            return(-1);
      }
      return((CS_INT)srv_rpcnumber(spp));
}
```

Usage

- srv_rpcnumber returns the number component of the current remote procedure call ("RPC") designation.
- srv_rpcnumber returns only the number component of the RPC's designation and does not include anything else, such as optional specifiers for owner or RPC name. A fully qualified designation for an RPC takes the form *database.owner.rpcname;number*. To get the other parts of the RPC's designation, if any, use srv_rpcname, srv_rpcowner, and srv_rpcdb.

See also srv_numparams, srv_rpcdb, srv_rpcname, srv_rpcoptions, srv_rpcowner

srv_rpcoptions

Description Return the runtime options for the current remote procedure call.

Syntax

```
CS_INT      srv_rpcoptions(spp)
SRV_PROC    *spp;
```

Parameters

spp

A pointer to an internal thread control structure.

Return value

Table 3-110: Return values (srv_rpcoptions)

Returns	To indicate
A non-zero integer containing the runtime flags for the current RPC.	The current RPC's runtime flags.
0	There is no current RPC. Open Server raises an error.

Examples

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_rpcoptions PROTOTYPE((
SRV_PROC        *spp
));

/*
** EX_SRV_RPCOPTIONS
**
**    Example routine to retrieve RPC runtime options
**
** Arguments:
**   spp   A pointer to an internal thread control structure.
**
** Returns:
**
**   CS_SUCCEED
**   CS_FAIL
*/
CS_RETCODE      ex_srv_rpcoptions(spp)
SRV_PROC        *spp;
{
   CS_INT        options;

   if ( (options = srv_rpcoptions(spp)) == 0 )
         return(CS_FAIL);

   return(CS_SUCCEED);
}
```

Usage

- srv_rpcoptions returns a CS_INT value containing the runtime flags for the current remote procedure call.
- Currently, the only flag is SRV_PARAMRETURN. If SRV_PARAMRETURN is CS_TRUE, the RPC must be recompiled before it is executed. This is significant only if the RPC is a stored procedure executing on an Adaptive Server.

See also

srv_numparams, srv_rpcdb, srv_rpcname, srv_rpcnumber, srv_rpcowner

srv_rpcowner

Description

Return the owner component of the current remote procedure call's designation.

Syntax

CS_CHAR *srv_rpcowner(spp, lenp)

```
SRV_PROC     *spp;
CS_INT       *lenp;
```

Parameters

spp

A pointer to an internal thread control structure.

lenp

A pointer to a buffer that will contain the length of the owner name. *lenp* can be NULL, in which case the length of the database owner is not returned.

Return value

Table 3-111: Return values (srv_rpcowner)

Returns	To indicate
A pointer to the null terminated owner component of the current RPC's designation.	The location of the database component of the current RPC's designation.
A null pointer	There is no current RPC. Open Server sets *lenp* to -1 and raises an informational error.

Examples

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
extern CS_RETCODE    ex_srv_rpcowner PROTOTYPE((
CS_VOID              *spp,
```

```
CS_CHAR        *ownerp
));

/*
** EX_SRV_RPCOWNER
**
**   Determine the owner component of an RPC destination.
**
** Arguments:
**   spp     A pointer to an internal thread control structure.
**   ownerp  A pointer to the buffer to which Open Server
 **           returns the owner component.
**
** Returns:
**   CS_SUCCEED      Owner component returned successfully.
**   CS_FAIL         An error was detected.
*/
CS_RETCODE     ex_srv_rpcowner(spp, ownerp)
SRV_PROC       *spp;
CS_CHAR        *ownerp;
{
      CS_INT    len;

      ownerp = srv_rpcowner(spp, &len);

      if(len == (CS_INT)(-1))
      {
            return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

Usage

- srv_rpcowner returns a CS_CHAR pointer to a null terminated string containing the owner component of the current remote procedure call (“RPC”) designation.
- srv_rpcowner returns only the owner component of the RPC’s designation and does not include anything else, such as optional specifiers for database or RPC number. A fully qualified designation for an RPC takes the form *database.owner.rpcname;number*. To get the other parts of the RPC’s designation, if any, use srv_rpcname, srv_rpcdb, and srv_rpcnumber.

See also

srv_numparams, srv_rpcdb, srv_rpcname, srv_rpcnumber, srv_rpcoptions

srv_run

Description

Start up the Open Server application.

Syntax

```
CS_RETCODE srv_run(ssp)
SRV_SERVER    *ssp;
```

Parameters

ssp

A pointer to the Open Server control structure. This is an optional argument.

Return value

Table 3-112: Return values (srv_run)

Returns	To indicate
CS_SUCCEED	The server is stopped.
CS_FAIL	Open Server could not start the server. If srv_run returns CS_FAIL, an application must call srv_init before calling srv_run again.

Examples

```
#include     <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE       ex_srv_run PROTOTYPE((CS_VOID));

/*
** EX_SRV_RUN
**   An example routine to start up an Open Server using srv_run.
**
** Arguments:
**   None.
**
** Returns:
**    SRV_STOP    If the server is stopped.
**    CS_FAIL     If the server can't be brought up.
*/
CS_RETCODE ex_srv_run()
{
      return(srv_run((SRV_SERVER *)NULL));
}
```

Usage

- srv_run starts, or restarts, the Open Server application.
- srv_run returns when the server is stopped by a SRV_STOP event.

- Once started, the server listens for a client request, calls the function defined to handle the request, and then continues listening for further requests.
- If a server has stopped, it must be re-initialized using srv_init before it is restarted.

Note If srv_run is called in the entry functions of a DLL, a deadlock may arise. srv_run creates operating system threads and tries to synchronize them using system utilities. This synchronization conflicts with the operating system's serialization process.

See also srv_init, srv_props, "Events" on page 92

srv_s_ssl_local_id

Description Properties used to specify the path to the local ID (certificates) file.

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

Parameters

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

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

identity_password
used to decrypt the private key.

srv_select (UNIX only)

Description Check to see if a file descriptor is &ready for a specified I/O operation.

Syntax
```
CS_INT srv_select(nfds, &readmaskp, writemaskp,
                  exceptmaskp, waitflag)
```

```
CS_INT                  nfds;
SRV_MASK_ARRAY          *&readmaskp;
SRV_MASK_ARRAY          *writemaskp;
SRV_MASK_ARRAY          *exceptmaskp;
CS_INT                  waitflag;
```

Parameters

nfds
The highest number file descriptor to check.

&readmaskp
A pointer to a SRV_MASK_ARRAY structure initialized with the mask of file descriptors to check for read availability.

writemaskp
A pointer to a SRV_MASK_ARRAY structure initialized with the mask of file descriptors to check for write availability.

exceptmaskp
A pointer to a SRV_MASK_ARRAY structure initialized with the mask of file descriptors to check for exceptions.

waitflag
A CS_INT that indicates whether the thread should be suspended until any file descriptor is available for the desired operation. See the "Comments" section for a description of the legal values for *waitflag*.

Return value

The total number of file descriptors that are &ready for any of the indicated operations. If an error occurs, -1 is returned.

Table 3-113: Return values (srv_select)

Returns	To indicate
An integer	The total number of file descriptors &ready for any of the indicated operations.
-1	The routine failed.

Examples

```
#include    <ospublic.h>

 /*
 ** Local Prototype.
 */
 CS_RETCODE     ex_srv_select PROTOTYPE((
 CS_INT         readfd
 ));
 /*
 ** EX_SRV_SELECT
 **
```

```
**   Example routine to illustrate the use of srv_select.
**
** Arguments:
**   readfd  -  fd to be checked if it is &ready for a read **
                operation.
**
** Returns:
**  CS_SUCCEED      If readfd is &ready for a read operation.
**  CS_FAIL         If readfd is not &ready for a read operation.
*/
CS_RETCODE      ex_srv_select(readfd)
CS_INT          readfd;
{
      SRV_MASK_ARRAY       &readmask;
      CS_BOOL              &ready;

      /* Initialization. */
     (CS_VOID)srv_mask(CS_ZERO, &&readmask, (CS_INT)0, (CS_BOOL
           *)NULL);
      &ready = CS_FALSE;

      /* Set readfd in the mask. */
      (CS_VOID)srv_mask(CS_SET, &&readmask, readfd, (CS_BOOL
            *)NULL);

      /*
      ** Check whether the descriptor is &ready for a read
      ** operation. If it is not, return.
      */
      if (srv_select(readfd+1, &&readmask, (SRV_MASK_ARRAY *)NULL,
           (SRV_MASK_ARRAY *)NULL, SRV_M_NOWAIT) <= 0 )
           return (CS_FAIL);

      /*
      ** A file descriptor is &ready for a read operation.
      */
      (CS_VOID)srv_mask(CS_GET, &&readmask, readfd, &&ready);
      return ((&ready) ? CS_SUCCEED : CS_FAIL);

}
```

Usage

- Use srv_select when you want to know if a network I/O operation can be performed on a file descriptor without requesting the I/O.
- Open Server will include the designated file descriptor in the global mask that it uses when it checks for file descriptor availability.

- A SRV_MASK_ARRAY is defined as follows:

```
#define SRV_MASK_SIZE            (CS_INT)40
#define SRV_MAXMASK_LENGTH
(CS_INT)(SRV_MASK_SIZE*CS_BITS_PER_LONG)
     typedef struct        srv_mask_array
{
  long      mask_bits[SRV_MASK_SIZE];
} SRV_MASK_ARRAY;
```

  SRV_MASK_SIZE indicates the number of elements in the SRV_MASK_ARRAY and SRV_MAXMASK_LENGTH indicates the maximum number of file descriptors that can be represented in the SRV_MASK_ARRAY.

- An Open Server application that uses external file descriptors must close them in an orderly fashion. An application thread must wait for a pending srv_select call to complete before closing an external file descriptor. If not, Open Server will exit.
- Table 3-114 summarizes the legal values for waitflag:

Table 3-114: Values for waitflag (srv_select)

Value	Meaning
SRV_M_WAIT	The thread is suspended and will wake up when any file descriptor represented in the masks is available for the specified operation. The return status indicates whether any file descriptors are available for the desired operations.
SRV_M_NOWAIT	The routine will return immediately after the next network check. The return status indicates whether any file descriptors are available for the desired operations.

- An application can use srv_select to poll the file descriptor and return immediately or not return until one of the file descriptors is &ready.
- srv_select cannot be used in a SRV_START or SRV_ATTENTION handler.

See also srv_mask

srv_send_ctlinfo

Description

Sends control messages to Client-Library.

Syntax

CS_RETCODE **srv_send_ctlinfo**(SRV_PROC *srvproc, CS_INT *ctl_type*, SRV_CTL_MIGRATE *ctl_type*, CS_INT *paramcnt*, SRV_CTLITEM **param*)

Parameters

srvproc
A pointer to an internal thread control structure.

ctl_type
The type of control message to send.

paramcnt
The number of elements in the *param* array.

param
Parameters for library control message.

Usage

- *ctl_type* has the following values:
  - SRV_CTL_MIGRATE

    Sends migration request to the client or cancel a previous migration request. SRV_CTL_MIGRATE can be used only if the client supports migration and has received a session ID when it first connected to the session.

    See “SRV_CTL_MIGRATE” on page 42 for more information.
  - SRV_CTL_LOGINREDIRECT

    Only valid during a connect handler. When used, a SRV_PROC for which SRV_T_REDIRECT is true will instruct the client to restart login using the passed-in server addresses.
  - SRV_CTL_HAUPDATE

    Valid at any time srv_sendinfo is valid. When the server sends this message to a client, the client will replace its current HA failover target information with the server connection information as expressed via *param*.
- *param* has the following fields:
  - SRV_CTLTYPES *srv_ctlitemtype*, where *srv_ctlitemtype* indicates the parameter type. The following types are available:
    - *SRV_CT_SERVERNAME*, which indicates that *srv_ctlptr* points to a CS_CHAR string containing the name of the server whose address will be looked up.

- *SRV_CT_TRANADDR*, which indicates that *srv_ctlptr* points to a CS_TRANADDR structure containing the server address information.
- *SRV_CT_ADDRSTR*, which indicates that *srv_ctlptr* points to a string formated by srv_getserverbyname.
- *SRV_CT_OPTION*, which indicates that *srv_ctlptr* points to a *CS_INT* bitmask that contains a set of options for this message.

- CS_INT *srv_ctllength*, which is the length of variable size parameters. When *srv_ctlitemtype* is SRV_CT_SERVERNAME or SRV_CT_ADDRSTR, it is the length of the string pointed to by *srv_ctlptr* or CS_NULLTERM. When *srv_ctlitemtype* is SRV_CT_TRANADDR, it is the number of elements in the CS_TRANADDR array pointed to by *srv_ctlptr*.
- void **srv_ctlptr*, where *srv_ctlptr* points to the actual parameter data.

See also srv_freeserveraddrs, srv_getserverbyname

srv_send_data

Description srv_send_data allows Open Server applications to transfer rows containing multiple columns to clients. It allows Open Server applications to send text, image, and XML data in chunks, preventing the excessive use of memory.

Syntax
```
CS_RETCODE srv_send_data(spp, column, buf, buflen)
SRV_PROC     *spp;
CS_INT       *column;
CS_BYTE      *buf;
CS_INT        buflen;
```

Parameters

spp
A pointer to an internal thread control structure.

column
The number of the column in a row set.

buf
A pointer to a buffer containing the data to send to the client. This determines the size of a section.

buflen
The length of the **buf* buffer.

Return value

Table 3-115: Return values (srv_send_data)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <ctpublic.h>
#include <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE ex_srv_send_data PROTOTYPE((
SRV_PROC *spp,
CS_COMMAND *cmd,
CS_INT cols
));
#define MAX_BULK 51200

/*
** EX_SRV_SEND_DATA

** Example routine to demonstrate how to write columns
** of data in a row set to a client using srv_send_data.
** This routine will send all the columns of data read
** from a server back to the client.
** Arguments:
** spp        - A pointer to an internal thread control structure.
** cmd        - The command handle for the command that is returning
**              text data.
** cols       - The number of columns in a row set.
** Returns:
** CS_SUCCEED - Result set sent successfully to client.
** CS_FAIL    - An error was detected.
*/
CS_RETCODE ex_srv_send_data(spp, cmd, cols)
SRV_PROC *spp;
CS_COMMAND *cmd;
CS_INT cols;
{
    CS_INT  *len;          /* Length of column data. */
    CS_INT  *outlen;       /* Number of bytes received. */
    CS_BYTE **data;        /* Column data. */
    CS_BYTE buf[MAX_BULK]; /* Buffer for text data. */
    CS_BOOL ok;           /* Error control flag. */
    CS_INT i;
    CS_INT ret;
```

```
/* Initialization. */
ok = CS_TRUE;

/*
** Transfer a row.
*/
for (i = 0; i < cols; i++)
{
    if ((fmt[i].datatype != CS_TEXT_TYPE) &&
        (fmt[i].datatype != CS_IMAGE_TYPE))
    {
         /*
         ** Transfer a non TEXT/IMAGE column.
         */

         /*
         ** Read the data of a non-text/image column
         ** from the server.
         */
         ret = ct_get_data(cmd, i+1, data[i],
               len[i], &outlen[i]);
         if ((ret != CS_SUCCEED) && (ret != CS_END_DATA)
          && (ret != CS_END_ITEM))
         {
              ok = CS_FALSE;
              break;
         }

         /*
         ** Write the data of a non-text/image column
         ** to client.
         */
         if (ret = srv_send_data(srvproc, i+1, NULL, 0)
              != CS_SUCCEED)
         {
             ok = CS_FALSE;
             break;
         }
    }
    else
    {
       /*
       ** Transfer a TEXT/IMAGE column in small trunks.
       */
```

```
            /*
            ** Read a chunk of data of a text/image column
            ** from the server.
            */
            while ((ret = ct_get_data(cmd, i+1, buf, MAX_BULK, &len[i]))
                    == CS_SUCCEED)
            {
                 /*
                 ** Write the chunk of data to client.
                 */
                 if (ret = srv_send_data(srvproc, i+1, buf, len[i])
                     != CS_SUCCEED)
                 {
                      ok = CS_FALSE;
                      break;
                  }
             }
        }
    }

    switch(ret)
    {
        case CS_SUCCEED:
        /* The routine completed successfully. */
        case CS_END_ITEM:
        /* Reached the end of this item's value. */
        case CS_END_DATA:
        /* Reached the end of this row's data. */
        break;
        case CS_FAIL:
        /* The routine failed. */
        case CS_CANCELED:
        /* The get data operation was cancelled. */
        case CS_PENDING:
        /* Asynchronous network I/O is in effect. */
        case CS_BUSY:
        /* An asynchronous operation is pending. */
        default:
        ok = CS_FALSE;
    }
    return (ok ? CS_SUCCEED : CS_FAIL);
}
```

Usage

- srv_send_data sends data of a row set column by column to a client.

- When sending columns with text, image or XML data, Open Server applications must call srv_text_info before srv_send_data. This ensures the data stream is correctly set to the total length of data being sent. The application then calls srv_send_data to send the data in chunks, and continues to call the routine until there is no remaining data to be sent.
- Open Server applications can send text, image and XML data to clients using srv_bind and srv_xferdata. However, these routines require all data columns to be sent at once. srv_send_data allows applications to send text and image data in chunks.
- Because srv_send_data reads and sends data one column at a time, the data format for a whole row needs to be sent to the client together with the first column in the row set. To retrieve fixed I/O fields, such as object name before a column is read, call ct_data_info(). Note that the changeable fields in I/O descriptors such as pointers to text data, and length of text data are retrievable only after the column is read.
- Open Server applications treat text, image and XML data streams identically, with the exception of character set conversions. These conversions are only performed on text data.

See also

Related srv_bind, srv_get_text, srv_text_info, srv_xferdata, srv_get_data, and srv_send_text routines in the Open Server 15.0 *Server Library/C Reference Manual*.

srv_send_text

Description

Send a text or image data stream to a client, in chunks.

Syntax

```
CS_RETCODE srv_send_text(spp, bp, buflen)

SRV_PROC        *spp;
CS_BYTE         *bp;
CS_INT          buflen;
```

Parameters

spp

A pointer to an internal thread control structure.

bp

A pointer to a buffer containing the data to send to the client. This determines the size of a section.

buflen

The size of the **bp* buffer.

Return value

Table 3-116: Return values (srv_send_text)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ctpublic.h>
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE ex_srv_send_text PROTOTYPE((
SRV_PROC    *spp,
CS_COMMAND  *cmd
));

/*
** EX_SRV_SEND_TEXT
**
**   Example routine to demonstrate how to write text to a client
 **   using srv_send_text. This routine will send all the text
 **   read from a server back to the client.
**
** Arguments:
**   spp   A pointer to an internal thread control structure.
**   cmd   The command handle for the command that is returning
 **         text data.
**
** Returns:
**   CS_SUCCEED   Result set sent successfully to client.
**   CS_FAIL      An error was detected.
*/
CS_RETCODE      ex_srv_send_text(spp, cmd)
SRV_PROC       *spp;
CS_COMMAND     *cmd;
{
   CS_BOOL     ok;            /* Error control flag.    */
   CS_INT      ret;           /* ct_fetch return value.  */
   CS_INT      len_read;      /* Amount of data read.    */
   CS_BYTE     data[1024];    /* Buffer for text data.    */

   /* Initialization. */
   ok = CS_TRUE;
```

```
        /* Read the text from the server. */
        while ((ret = ct_get_data(cmd, 1, data, CS_SIZEOF(data),
                &len_read))
              == CS_SUCCEED)
        {
              /* Write text to client a chunck at a time */
              if (srv_send_text(spp, data, len_read) != CS_SUCCEED)
              {
                     ok = CS_FALSE;
                     break;
              }
        }

        switch(ret)
        {
        case CS_SUCCEED:    /* The routine completed successfully. */
        case CS_END_ITEM:   /* Reached the end of this item's value. */
        case CS_END_DATA:   /* Reached the end of this item's value. */
             break;
        case CS_FAIL:       /* The routine failed.  */
        case CS_CANCELED:  /* The get data operation was cancelled. */
        case CS_PENDING:    /* Asynchronous network I/O is in effect. */
        case CS_BUSY:       /* An asynchronous operation is pending. */
        default:
               ok = CS_FALSE;
        }

        return (ok ? CS_SUCCEED : CS_FAIL);
    }
```

Usage

- srv_send_text is used to send a single column of text or image data to a client.
- The Open Server application must always call srv_text_info prior to the first call to srv_send_text for the data stream, to set the total length of the data to be sent. The application then calls srv_send_text to send a chunk. srv_send_text is called as many times as there are chunks.
- The item being sent to the client must have previously been described using srv_descfmt.
- An Open Server application can also write text and image data to a client using srv_bind and srv_xferdata. srv_send_text allows the application to send the data in chunks, whereas the standard srv_bind/srv_xferdata method requires that all the data in the column be sent at once.
- A column sent with srv_send_text must be of type text or image.

- Open Server treats text and image data streams identically except for character set conversion, which is only performed on text data.

Warning! An Open Server application can only use srv_send_text to send a row if that row contains a single column and that column contains text or image data.

See also

srv_bind, srv_descfmt, srv_get_text, srv_text_info, srv_xferdata, "Text and image" on page 196

srv_senddone

Description

Send a results completion message or flush results to a client.

Syntax

```
CS_RETCODE srv_senddone(spp, status, transtate, count)

SRV_PROC        *spp;
CS_INT          status;
CS_INT          transtate;
CS_INT          count;
```

Parameters

spp

A pointer to an internal thread control structure.

status

A 2-byte bit mask composed of one or more flags OR'd together. Table 3-117 describes each flag:

Table 3-117: Values for status (srv_senddone)

Status	Description
SRV_DONE_FINAL	The current set of results is the final set of results.
SRV_DONE_MORE	The current set of results is not the final set of results.
SRV_DONE_COUNT	The count parameter contains a valid count.
SRV_DONE_ERROR	The current client command got an error.
SRV_DONE_FLUSH	The current result set will be sent to the client without waiting for a full packet.

transtate

The current state of the transaction. Table 3-118 describes the legal values for *transtate*:

Table 3-118: Values for transtate (srv_senddone)

Transaction State	Description
CS_TRAN_UNDEFINED	Not currently in a transaction.
CS_TRAN_COMPLETED	The current transaction completed successfully.
CS_TRAN_FAIL	The current transaction failed.
CS_TRAN_IN_PROGRESS	Currently in a transaction.
CS_TRAN_STMT_FAIL	The current transaction statement failed.

count

A 4-byte field containing a count for the current set of results. The count is valid if the SRV_DONE_COUNT flag is set in the *status* field.

Return value

Table 3-119: Return values (srv_senddone)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include     <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_senddone PROTOTYPE((
SRV_PROC        *spp
));

/*
** Constants and data definitions.
*/
#define    NUMROWS         3
#define    MAXROWDATA      6

CS_STATIC CS_CHAR  *row_data[NUMROWS] = {
                    "Larry",
                    "Curly",
                    "Moe"
                    };

/*
** EX_SRV_SENDDONE
**
**    Example routine illustrating the use of srv_senddone. This
 **    routine will send a set of results to the client
```

```
 **    application, and then send the results completion message.
**
** Arguments:
**    spp     A pointer to an internal thread control structure.
**
** Returns:
**    CS_SUCCEED    Results set sent successfully to client.
**    CS_FAIL       An error was detected.
*/
CS_RETCODE      ex_srv_senddone(spp)
SRV_PROC        *spp;
{
      CS_DATAFMT      fmt;
      CS_INT          row_len;
      CS_INT          idx;

      /*
      ** Describe the format of the row data, with the single
       ** dummy column.
      */
      srv_bzero((CS_VOID *)&fmt, (CS_INT)sizeof(fmt));
      fmt.datatype = CS_CHAR_TYPE;
      fmt.maxlength = MAXROWDATA;

      if (srv_descfmt(spp, (CS_INT)CS_SET, (CS_INT)SRV_ROWDATA,
                (CS_INT)1, &fmt) != CS_SUCCEED)
      {
             (CS_VOID)srv_senddone(spp,
                (CS_INT)(SRV_DONE_FINAL | SRV_DONE_ERROR),
                (CS_INT)CS_TRAN_FAIL, (CS_INT)0);
             return(CS_FAIL);
       }

      for (idx = 0; idx < NUMROWS; ++idx)
      {
           /*
           ** Bind the row_data array element.
           */
           row_len = (CS_INT)strlen(row_data[idx]);
           if (srv_bind(spp, (CS_INT)CS_SET, (CS_INT)SRV_ROWDATA,
                (CS_INT)1, &fmt, (CS_BYTE *)(row_data[idx]),
                 &row_len, (CS_SMALLINT *)NULL) != CS_SUCCEED)

           {
                /* Communicate failure, and number of rows sent. */
                (CS_VOID)srv_senddone(spp,
```

```
                    (CS_INT)(SRV_DONE_FINAL |
                             SRV_DONE_ERROR | SRV_DONE_COUNT),
                    (CS_INT)CS_TRAN_FAIL, (CS_INT)idx);
                return(CS_FAIL);
            }

            /*
            ** Transfer the row data.
            */
            if (srv_xferdata(spp, (CS_INT)CS_SET, SRV_ROWDATA)
                                  != CS_SUCCEED)
            {
                /* Communicate failure, and number of rows sent. */
                (CS_VOID)srv_senddone(spp,
                    (CS_INT)(SRV_DONE_FINAL |
                        SRV_DONE_ERROR | SRV_DONE_COUNT),
                    (CS_INT)CS_TRAN_FAIL, (CS_INT)idx);
                return(CS_FAIL);
             }
        }

        /* Send a status value. */
        if (srv_sendstatus(spp, (CS_INT)0) != CS_SUCCEED)
        {
              /* Communicate failure, and number of rows sent. */
              (CS_VOID)srv_senddone(spp,
                  (CS_INT)(SRV_DONE_FINAL |
                      SRV_DONE_ERROR | SRV_DONE_COUNT),
                  (CS_INT)CS_TRAN_FAIL, (CS_INT)NUMROWS);
              return(CS_FAIL);
        }

        /* Send the final DONE message, with the row count. */
        if (srv_senddone(spp, (CS_INT)(SRV_DONE_FINAL |
               SRV_DONE_COUNT),
                  (CS_INT)CS_TRAN_COMPLETED,
                  (CS_INT)NUMROWS) != CS_SUCCEED)
        {
              /* Communicate failure, and number of rows sent. */
              (CS_VOID)srv_senddone(spp,
                  (CS_INT)(SRV_DONE_FINAL |
                           SRV_DONE_ERROR | SRV_DONE_COUNT),
                  (CS_INT)CS_TRAN_FAIL, (CS_INT)NUMROWS);
              return(CS_FAIL);
        }
    return(CS_SUCCEED);
```

```
}
```

Usage

- srv_senddone sends a message to the client that the current set of results is complete. A client request can cause the server to execute a number of commands and to return a number of results sets. For each set of results, a completion message must be returned to the client with srv_senddone.
- If the current results are not the last set of results for the client command batch, the Open Server must set the *status* mask's SRV_DONE_MORE field. Otherwise, the Open Server application must set the *status* field to SRV_DONE_FINAL to indicate that there are no more results for the current command batch.
- The *count* field indicates how many rows were affected by a particular command. If *count* actually contains a count, the SRV_DONE_COUNT bit should be set in the *status* field. This enables the client to distinguish between an actual count of 0 and an unused *count* field.
- If the SRV_CONNECT handler rejects the client login, the Open Server application must call srv_senddone with the *status* parameter set to the SRV_DONE_ERROR flag. The SRV_CONNECT handler must then send a DONE packet to the client with srv_senddone. In any case, srv_senddone must be called only once before the SRV_CONNECT handler returns and the SRV_DONE_FINAL *status* flag must be set.
- When a write is in progress and the network buffer fills up, Open Server flushes its contents. Issuing srv_senddone with status set to SRV_DONE_FINAL or SRV_DONE_FLUSH causes a flush of the network buffer, regardless of how full it is. SRV_DONE_FLUSH can be set with or without SRV_DONE_MORE.
- Setting *status* to SRV_DONE_FLUSH allows an application to flush to a client results that have accumulated over a long period of time.
- An application cannot set the *status* argument to SRV_DONE_FLUSH inside a SRV_CONNECTION error handler.
- Open Server does not provide any transaction management. It is the responsibility of the Open Server application to use the *transtate* argument as required to notify a client of the current transaction state.

Note The *transtate* argument replaces the *info* argument in the Open Server 2.0 version of srv_senddone.This change will cause runtime errors in existing applications if the value of *info* in the existing application is not 0.

See also

srv_bind, srv_descfmt, srv_sendstatus, srv_xferdata

srv_sendinfo

Description

Send error messages to the client.

Syntax

CS_RETCODE srv_sendinfo(spp, errmsgp, transtate)

```
SRV_PROC            *spp;
CS_SERVERMSG        *errmsgp;
CS_INT              transtate;
```

Parameters

spp

A pointer to an internal thread control structure.

errmsgp

A pointer to the CS_SERVERMSG structure containing the error message information to be sent to the client. See “CS_SERVERMSG structure” on page 60.

transtate

The current state of the transaction. Table 3-120 describes the legal values for *transtate*:

Table 3-120: Values for transtate (srv_sendinfo)

Transaction State	Description
CS_TRAN_UNDEFINED	Not currently in a transaction.
CS_TRAN_COMPLETED	The current transaction completed successfully.
CS_TRAN_FAIL	The current transaction failed.
CS_TRAN_IN_PROGRESS	Currently in a transaction.
CS_TRAN_STMT_FAIL	The current transaction statement failed.

Return value

Table 3-121: Return values (srv_sendinfo)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>
/*
** Local Prototype.
*/

CS_RETCODE      ex_srv_sendinfo  PROTOTYPE((
SRV_PROC        *sp,
CS_CHAR         *msg,
CS_INT          msglen,
CS_INT          msgnum
```

```
));

/*
** EX_SRV_SENDINFO
**
**    This routine demonstrates how to use srv_sendinfo to send
 **    an error message to a client.
**
** Arguments:
**         sp       A pointer to an internal thread control
 **                  structure.
**         msg      The message text to send.
**         msglen   The length of the message text to send.
**         msgnum   The message number to send.
**
** Returns
**         CS_SUCCEED     If the message is sent.
**         CS_FAIL        If an error occurred.
*/
CS_RETCODE      ex_srv_sendinfo(sp, msg, msglen, msgnum)
SRV_PROC        *sp;
CS_CHAR         *msg;
CS_INT          msglen;
CS_INT          msgnum;
{
      CS_SERVERMSG    &mrec;

      /*
      ** Initialization.
      */
      srv_bzero(&&mrec, sizeof(CS_SERVERMSG));

      /*
      ** First, determine if the message string will fit
       ** in the message structure. If not, truncate it.
      */
      if( msglen > CS_MAX_MSG )
      {
          msglen = CS_MAX_MSG;
      }

      /*
      ** Now copy the message string over.
      */
      srv_bmove(msg, &mrec.text, msglen);
      &mrec.textlen = msglen;
```

```
        /*
        ** Set the message number we want to send.
        */
        &mrec.msgnumber = msgnum;

        /* Set the message status so that &mrec.text contains
        ** the entire message
        */
        &mrec.status = CS_FIRST_CHUNK | CS_LAST_CHUNK;

        /*
        ** Now we're &ready to send the message.
        */
        if( srv_sendinfo(sp, &&mrec, CS_TRAN_UNDEFINED) == CS_FAIL )
        {
              /*
              ** An error was al&ready raised.
              */
              return CS_FAIL;
        }

        /*
        ** All done.
        */
        return CS_SUCCEED;
    }
```

Usage

- srv_sendinfo sends error messages to the client. It must be called once for each message sent.
- An application can call srv_sendinfo before or after it sends result rows. However, an application cannot call srv_sendinfo between calls to srv_descfmt or between a call to srv_descfmt and a call to srv_xferdata.
- If an Open Server application wants to send parameter data pertaining to an error message, it must set the *status* field of the CS_SERVERMSG structure to CS_HASEED. The application must describe, bind and send the error parameters immediately after calling srv_sendinfo, before sending other results and before a call to srv_senddone. The application must invoke srv_descfmt, srv_bind and srv_xferdata with a *type* argument of SRV_ERRORDATA.
- If an application calls srv_sendinfo with the *status* field of the CS_SERVERMSG structure set to CS_HASEED but fails to send error parameters, a fatal process error is raised when the application calls srv_senddone.

- When an application calls srv_sendinfo with the status field of the CS_SERVERMSG structure set to CS_HASEED, Open Server will verify that the CS_RES_NOEED response capability is *not* set. If it is set, Open Server will raise an error. Any subsequent calls to srv_descfmt to describe error parameters will also provoke an error.
- For more information on sending error messages to clients, see "Client command errors" on page 38.
- For more information on extended error data, see "Client command errors" on page 38.
- For more information on the CS_SERVERMSG structure, see the "CS_SERVERMSG structure" on page 60.

See also

srv_bind, srv_descfmt, srv_senddone, srv_xferdata, "Client command errors" on page 38

srv_sendpassthru

Description

Send a protocol packet to a client.

Syntax

```
CS_RETCODE srv_sendpassthru(spp, send_bufp, infop)

SRV_PROC      *spp;
CS_BYTE       *send_bufp;
CS_INT        *infop;
```

Parameters

spp

A pointer to an internal thread control structure.

send_bufp

A pointer to a buffer that contains the protocol packet.

infop

A pointer to a CS_INT that is set to SRV_I_UNKNOWN if srv_sendpassthru returns CS_FAIL. Table 3-122 describes the possible values returned in **infop* if the routine returns CS_SUCCEED:

Table 3-122: CS_SUCCEED values (srv_sendpassthru)

Value	Description
SRV_I_PASSTHRU_MORE	The protocol packet was sent successfully and it is not the end of message packet.
SRV_I_PASSTHRU_EOM	The end of message protocol packet was sent successfully.

Return value

Table 3-123: Return values (srv_sendpassthru)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <stdio.h>
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_sendpassthru PROTOTYPE((
SRV_PROC        *spp
));

/*
** EX_SRV_SENDPASSTHRU
**
**  Example routine to send a protocol packet to a client.
**
** Arguments:
**  spp  A pointer to an internal thread control structure.
**
** Returns:
**
**  CS_SUCCEED
**  CS_FAIL
*/
CS_RETCODE      ex_srv_sendpassthru(spp)
SRV_PROC        *spp;
{
      CS_BYTE    sendbuf[20];
      CS_INT     info;

      strcpy(sendbuf, "Here's what to send");
```

```
        if (srv_sendpassthru(spp, sendbuf, &info) == CS_FAIL)
        {
              return(CS_FAIL);
        }
        else
        {
              if (info == SRV_I_PASSTHRU_MORE)
              {
                  printf("more to come...\n");
                  return(CS_SUCCEED);
              }
              else if (info == SRV_I_PASSTHRU_EOM)
              {
                  printf("That's all.\n");
                  return(CS_SUCCEED);
              }
              else
              {
                  printf("Unknown flag returned.\n");
                  return(CS_FAIL);
              }
        }
    }
```

Usage

- srv_sendpassthru sends a protocol packet received from a client program or Adaptive Server without interpreting its contents.
- srv_sendpassthru performs byte ordering on protocol header fields.
- Once called, the thread that called it is in *passthrough* mode. Passthrough mode ends when the SRV_PASSTHRU_EOM is returned.
- No other Server-Library routines can be called while the event handler is in passthrough mode.
- To use passthrough mode, the SRV_CONNECT handler for the client must allow the client and remote server to negotiate the protocol format by calling srv_getloginfo, ct_setloginfo, ct_getloginfo, and srv_setloginfo. This allows clients and remote servers running on dissimilar platforms to perform any necessary data conversions.
- srv_sendpassthru can be used in all event handlers except SRV_CONNECT, SRV_DISCONNECT, SRV_START, SRV_STOP, SRV_URGDISCONNECT, and SRV_ATTENTION.

See also

srv_getloginfo, srv_recvpassthru, srv_setloginfo

srv_sendstatus

Description

Send a status value to a client.

Syntax

```
CS_RETCODE srv_sendstatus(spp, value)

SRV_PROC       *spp;
CS_INT         value;
```

Parameters

spp

A pointer to an internal thread control structure.

value

The status of the request. By convention, 0 means the request completed normally.

Return value

Table 3-124: Return values (srv_sendstatus)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>

/*
** Local prototype.
*/
CS_RETCODE        ex_srv_sendstatus PROTOTYPE((
SRV_PROC          *spp
));

/*
** EX_SRV_SENDSTATUS
**
**  Example routine to send a status value to a client.
**
** Arguments:
**    spp      A pointer to an internal thread control structure.
**
** Returns:
**  CS_SUCCEED if we were able to send the status.
**  CS_FAIL if an error was detected.
**
*/
CS_RETCODE    ex_srv_sendstatus(spp)
SRV_PROC      *spp;
{
```

```
        CS_RETCODE      result;

        /*
        ** Send an OK status.
        */
        result = srv_sendstatus(spp, (CS_INT)0);

        return (result);
    }
```

Usage

- srv_sendstatus sends a return status value to the client in response to a client request. When a request is received, the programmer-installed event handler routine is called to service it. Part of the response to a request can be to return a status value.
- The status value sent by srv_sendstatus is both optional and application-specific. It is not related to the srv_senddone *status* parameter.
- A status value can be sent after all rows, if any, have been sent to the client with srv_xferdata and before the completion status is sent with srv_senddone. A status value cannot be sent between a call to srv_descfmt and srv_bind, and a call to srv_xferdata.
- Only one status value can be sent for each set of results.

See also

srv_senddone

srv_setcolutype

Description

Define the user datatype to be associated with a column.

Syntax

CS_RETCODE srv_setcolutype(spp, column,utype)

```
SRV_PROC       *spp;
CS_INT          column;
CS_INT          utype;
```

Parameters

spp

A pointer to an internal thread control structure.

column

The column number of the column with which to associate the user datatype. The first column is 1.

utype

The user-defined datatype to be associated with the column.

Return value

Table 3-125: Return values (srv_setcolutype)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE       ex_srv_setcolutype PROTOTYPE((
SRV_PROC         *spp,
CS_INT           column,
CS_INT            utype
));

/*
** EX_SRV_SETCOLUTYPE
**
**   Example routine to define the user datatype to be associated
 **   with a column using srv_setcolutype.
**
** Arguments:
**  spp        A pointer to an internal thread control structure.
**  column     The column number associated with the type.
**  utype      The type to be associated with the column.
**
** Returns:
**
**  CS_SUCCEED      The datatype was successfully associated with
 **                  the column.
**  CS_FAIL         An error was detected.
*/
CS_RETCODE     ex_srv_setcolutype(spp, column, utype)
SRV_PROC       *spp;
CS_INT         column;
CS_INT         utype;
{
      /*
      ** Associate the type with the column.
      */
      if (srv_setcolutype(spp, column, utype) != CS_SUCCEED)
      {
            return(CS_FAIL);
```

```
        }
        return(CS_SUCCEED);
    }
```

Usage

The datatype set through srv_setcolutype is the datatype the client application will receive through the DB-Library call dbcolutype or through the Client-Library call ct_describe.

srv_setcontrol

Description

Describe user control or format information for columns.

Syntax

```
CS_RETCODE srv_setcontrol(spp, colnum, ctrlinfop,
        ctrllen)

SRV_PROC        *spp;
CS_INT          colnum;
CS_BYTE         *ctrlinfop;
CS_INT          ctrllen;
```

Parameters

spp

A pointer to an internal thread control structure.

colnum

The number of the column to which the control information applies. The first column in a row is column number 1.

ctrlinfop

A pointer to the control data. Its length is given by the ctrllen parameter.

ctrllen

The length, in bytes, of the control data. There are, at most, SRV_MAXCHAR bytes of control information per column.

Return value

Table 3-126: Return values (srv_setcontrol)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
```

```
CS_RETCODE     ex_srv_setcontrol PROTOTYPE((
SRV_PROC      *spp
));

/*
** Constants.
*
#define MAXROWDATA        20
#define COLCONTROL        "Emp name: %s"

/*
** EX_SRV_SETCONTROL
**
**    Example routine to describe format information for a column
 **    using srv_setcontrol. In this example, a simple character
 **    column contains an employee name.
**
** Arguments:
**    spp    A pointer to an internal thread control structure.
**
** Returns:
**    CS_SUCCEED      Control information successfully defined.
**    CS_FAIL    An error was detected.
*/
CS_RETCODE       ex_srv_setcontrol(spp)
SRV_PROC         *spp;
{
      CS_DATAFMT fmt;

      /* Describe the format of the row data for the column. */

      srv_bzero((CS_VOID *)&fmt, (CS_INT)sizeof(fmt));
      fmt.datatype = CS_CHAR_TYPE;
      fmt.maxlength = MAXROWDATA;

      if (srv_descfmt(spp, (CS_INT)CS_SET, (CS_INT)SRV_ROWDATA,
            (CS_INT)1, &fmt) != CS_SUCCEED)
      {
            return(CS_FAIL);
      }

      /* Define the control information for the column. */
      if (srv_setcontrol(spp, (CS_INT)1, (CS_BYTE *)COLCONTROL,
            (CS_INT)strlen(COLCONTROL)) != CS_SUCCEED)
      {
            return(CS_FAIL);
```

```
        }
        return(CS_SUCCEED);
    }
```

Usage

- An Open Server application uses srv_setcontrol to tell a client about any user-defined format information pertinent to a particular column. For example, a client may want to send a particular string along with a particular column.
- srv_setcontrol must be called after a call to srv_descfmt and before calls to srv_xferdata. If called from any other context, it will return CS_FAIL.
- Control information can be associated with columns in any order. The only requirement is that the column must first be defined with srv_descfmt.
- It is not necessary to call srv_setcontrol for every column in a row. If an Open Server application does not set control information for a column, a null control string is returned for the column.
- An application should not return control information unless the client has specifically requested such information, through the client option toggle, CS_OPT_CONTROL.

See also

srv_bind, srv_descfmt, srv_xferdata

srv_setloginfo

Description

Return protocol format information from a remote server to a client.

Syntax

```
CS_RETCODE srv_setloginfo(spp, loginfop)

SRV_PROC        *spp;
CS_LOGINFO      *loginfop;
```

Parameters

spp

A pointer to an internal thread control structure.

loginfop

A pointer to a CS_LOGINFO structure that has been updated by ct_getloginfo.

Return value

Table 3-127: Return values (srv_setloginfo)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_setloginfo     PROTOTYPE((
SRV_PROC        *spp,
CS_LOGINFO      *loginfop
));

/*
** EX_SRV_SETLOGINFO
**
**    Return protocol format information from a remote server to
 **    a client.
**
**
** Arguments:
**
**    spp      A pointer to an internal thread control structure.
**    loginfop A pointer to a CS_LOGINFO structure that has been
 **             updated by ct_getloginfo.
**
** Returns
**
**    CS_SUCCEED
**    CS_FAIL
**
*/
CS_RETCODE      ex_srv_setloginfo(spp, loginfop)
SRV_PROC        *spp;
CS_LOGINFO      *loginfop;
{
      /* Check arguments. */
      if(spp == (SRV_PROC *)NULL)
      {
            return(CS_FAIL);
      }
      return(srv_setloginfo(spp,loginfop));
```

```
}
```

Usage

- Use srv_setloginfo in gateway server applications that pass protocol (Tabular Data Stream) packets between clients and remote Sybase servers without interpreting the contents of the packet.
- When a client connects directly to a server, the two programs negotiate the protocol format they will use to send and receive data. When you use protocol passthrough in a gateway application, the Open Server forwards protocol packets between the client and a remote server.
- srv_setloginfo is the fourth of four calls, two of them are CT-Library calls, that allow a client and remote server to negotiate a TDS format. The calls, which can only be made in a SRV_CONNECT event handler, are:
  a srv_getloginfo – Allocate a CS_LOGINFO structure and fill it with TDS information from the client thread.
  b ct_setloginfo – Prepare a CS_LOGINFO structure with the protocol information retrieved in step 1, then log in to the remote server with ct_connect.
  c ct_getloginfo – Transfer protocol login response information from a CS_CONNECTION structure to the newly allocated CS_LOGINFO structure.
  d srv_setloginfo – Send the remote server's response, retrieved in step 3, to the client, then release the CS_LOGINFO structure.

See also srv_getloginfo, srv_recvpassthru, srv_sendpassthru

srv_setpri

Description Modify the scheduling priority of a thread.

Syntax

```
CS_RETCODE srv_setpri(spp, mode, priority_value)

SRV_PROC      *spp;
CS_INT         mode;
CS_INT        priority_value;
```

Parameters

spp
A pointer to an internal thread control structure.

mode
SRV_C_DELTAPRI, if *priority_value* is to adjust the current priority, or SRV_C_NEWPRI, if *priority_value* is the new priority.

priority_value
If *mode* is SRV_C_NEWPRI, *priority_value* is the new priority for the thread. If *mode* is SRV_C_DELTAPRI, a negative *priority_value* reduces the current priority by its absolute value and a positive *priority_value* increases the current priority.

Return value

Table 3-128: Return values (srv_setpri)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE       ex_srv_setpri PROTOTYPE((
SRV_PROC         *spp,
CS_INT           mode,
CS_INT           priority
));

/*
** EX_SRV_SETPRI
**
**    Example routine to change a thread's scheduling priority.
**
** Arguments:
**  spp        A pointer to an internal thread control structure.
**  mode       Indicates whether a priority is relative or
 **              absolute.
**  priority   The change in priority value or the nrew
                priority value.
**
** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE    ex_srv_setpri(spp, mode, priority)
SRV_PROC      *spp;
CS_INT        mode;
CS_INT        priority;
{
```

```
        return(srv_setpri(spp, mode, priority));
    }
```

Usage

- When a thread is started as the result of a client logging into the Open Server or as the result of a call to srv_createproc or srv_spawn, it has a priority of SRV_C_DEFAULTPRI.
- srv_setpri can change the priority by specifying the new value or by adjusting the current value up or down by a specified value.
- If a thread sets the priority of another thread to a level higher than its own, the other thread is scheduled to run immediately. Otherwise, the new priority of the affected thread takes effect the next time the scheduler runs.
- If a thread that never sleeps has a priority higher than other threads, the lower priority threads will never have a chance to execute.
- Internal Open Server threads run with a priority of SRV_C_DEFAULTPRI. If you raise the priority of a thread above SRV_C_DEFAULTPRI, it must sleep occasionally to allow these internal processes to run.
- It is an error to reduce the priority to less than SRV_C_LOWPRIORITY or to increase it to a value greater than SRV_C_MAXPRIORITY.
- srv_setpri cannot be used in a SRV_START handler.

See also

srv_createproc, srv_spawn

srv_signal (UNIX only)

Description

Install a signal handler.

Syntax

```
SRV_SIGNAL_FUNC srv_signal(sig, handler)

CS_INT                  sig;
SRV_SIGNAL_FUNC handler;
```

Parameters

sig

The number of the UNIX signal for which a handler is installed. This is defined in *sgs/signal.h.*

handler

A pointer to a function that is called when *sig* is delivered to Open Server. Setting *handler* to SIG_DFL restores the default handler. Setting *handler* to SIG_IGN cause *sig* to be ignored.

Return value

Table 3-129: Return values (srv_signal)

Returns	To indicate
A pointer to the previously installed handler function.	The location of the function.
A null pointer	The routine failed.

Examples

```
#include    <errno.h>
#include    <stdio.h>
#include    <signal.h>
#include    <ospublic.h>
/*
** Local Prototype.
*/
CS_STATIC CS_VOID         ex_sigio_handler PROTOTYPE((
CS_INT                    sig
));

CS_RETCODE       ex_srv_signal PROTOTYPE((
CS_INT           *uerrno
));

/*
** Static storage.
*/
CS_STATIC CS_INT io_events = 0;

/*
** EX_SRV_SIGNAL
**
**    Example routine to install a UNIX signal handler for SIGIO,
 **    using srv_signal.
**
** Arguments:
**    uerrno    A pointer to a user's error number indicator.
**
** Returns:
**
**    CS_SUCCEED    Handler successfully installed.
**    CS_FAIL       Handler not installed, UNIX global errno set.
*/
CS_RETCODE      ex_srv_signal(uerrno)
CS_INT          *uerrno;
{
      /*
```

```
        ** Install the handler.
        */
        (CS_VOID)srv_signal((int)SIGIO,
                            (SRV_SIGNAL_FUNC)ex_sigio_handler);

      /* Was there an error condition? */
      if ((*uerrno = errno) != 0)
          return(CS_FAIL);

       return(CS_SUCCEED);
}

/*
** EX_SIGIO_HANDLER
**
**    Example signal handler to count I/O events. It prints a
 **    message when the Open Server application has been up long
 **    enough to get 100,000 I/O events.
**
** Arguments:
**    sig  The signal number, always SIGIO.
**
** Returns:
**    Nothing.
*/
CS_STATIC CS_VOID     ex_sigio_handler(sig)
CS_INT                sig;
{
     if (io_events == 100000)
      {
           fprintf(stderr, "The server has been up a long
                    time!!\n");
           io_events = 0;
      }
      else
      {
           io_events++;
      }
}
```

Usage

- Open Server installs UNIX signal handlers for SIGIO and SIGURG. These handlers must always be active once an Open Server is started. If they are not active, the Server-Library I/O and attention handling routines will either fail to function or will be unreliable.

  Warning! Installing a UNIX signal handler using sigvec(2) or signal(2) can cause unpredictable results. Applications should use srv_signal.

- Open Server guarantees that all other signals are blocked while the application is in the signal handler.
- UNIX documentation on signal for more information.

srv_sleep

Description

Suspend the currently executing thread.

Syntax

```
CS_RETCODE srv_sleep(sleepeventp, sleeplabelp,
                sleepflags, infop, reserved1,
               reserved2)

CS_VOID        *sleepeventp;
CS_CHAR         *sleeplabelp;
CS_INT          sleepflags;
CS_INT         *infop;
CS_VOID        *reserved1;
CS_VOID        *reserved2;
```

Parameters

sleepeventp

A generic void pointer that srv_wakeup uses to wake up the thread or threads. The pointer should be unique for the operating system event the threads are sleeping on. For example, if a message is passed to another thread, the sending thread could sleep until the message was processed. The pointer to the message would be a useful *sleepevent* that the receiving thread could pass to srv_wakeup to wake up the sender.

sleeplabelp

A pointer to a null terminated character string that identifies the event that the thread is sleeping on. This is useful for determining why a thread is sleeping. An application can display this information using the Open Server system registered procedure sp_ps.

sleepflags

The value of this flag determines the manner in which the thread will wake up. Table 3-130 summarizes the legal values for *sleepflags*:

Table 3-130: Values for sleepflags (srv_sleep)

Value	Description
SRV_M_ATTNWAKE	The thread wakes up if it receives an attention.
SRV_M_NOATTNWAKE	Attentions cannot wake up the thread.

infop

A pointer to a CS_INT. Table 3-131 describes the possible values returned in **infop* if srv_sleep returns CS_FAIL:

Table 3-131: Values for infop (srv_sleep)

Value	Description
SRV_I_INTERRUPTED	The thread was woken unconditionally by srv_ucwakeup.
SRV_I_UNKNOWN	Some other error occurred. For example, the thread is al&ready sleeping or is invalid.

reserved1

A platform-dependent handle to a mutex. This argument is ignored on non-preemptive platforms. Set it to (CS_VOID*)0 on non-preemptive platforms.

reserved2

This parameter is not currently used. Set it to 0.

Return value

Table 3-132: Return values (srv_sleep)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include     <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_sleep  PROTOTYPE((
CS_VOID         *sleepevnt,
CS_CHAR         *sleeplbl,
CS_INT          *infop
));

/*
```

```
** EX_SRV_SLEEP
**
**   This routine will suspend the currently executing thread.
**
**
** Arguments:
**
**
**   sleepevnt A void pointer that srv_wakeup uses to wake up
 **             the thread.
**   sleeplbl  A pointer to a null terminated string that
 **             identifies the event being the thread is sleeping
 **             on. This is primarily used for debugging.
 **    infop    A pointer to a CS_INT that is set to one of the
 **             following values:
 **             SRV_I_INTERRUPTED - srv_ucwakeup
 **             unconditionally woke the thread.
 **             SRV_I_UNKNOWN - Some other error occurred.
**
**
** Returns
**
**    CS_SUCCEED
**    CS_FAIL
**
*/
CS_RETCODE      ex_srv_sleep(sleepevnt,sleeplbl,infop)
CS_VOID         *sleepevnt;
CS_CHAR         *sleeplbl;
CS_INT          *infop;
{
     /* Check arguments.  */
     if(sleepevnt == (CS_VOID *)NULL)
     {
           return(CS_FAIL);
     }
     /*
     ** Using SRV_M_ATTNWAKE means the thread should wake up
      ** unconditionally if it receives an attention.
     */

return(srv_sleep(sleepevnt,sleeplbl,SRV_M_ATTNWAKE,infop,(CS_VOID*)0,(CS_V
OID*)0));
}
```

Usage

- srv_sleep suspends the currently executing thread and initiates rescheduling. The thread will sleep until srv_wakeup is called on the same event.
- Depending on the value of *sleepflags*, a thread that is sleeping can also wake up by receiving an attention.
- A thread resumes execution on the statement just following the call to srv_sleep.
- srv_sleep cannot be used in a SRV_START handler.
- srv_sleep should not be called from interrupt level code. Any number of problems could occur if this rule is violated.
- Call srv_capability to determine whether your platform supports preemptive scheduling.
- The *reserved1* parameter prevents a race condition that could occur with preemptive scheduling if the wakeup event occurred before the thread finished going to sleep. See the Open Client and Open Server *Programmer's Supplement* for your platform for an example of preemptive scheduling.

See also

srv_wakeup

srv_spawn

Description

Allocate a service thread.

Syntax

```
CS_RETCODE srv_spawn(sppp, stacksize, funcp,
        argp, priority)

SRV_PROC        **sppp;
CS_INT          stacksize;
CS_RETCODE      (*funcp)();
CS_VOID         *argp;
CS_INT          priority;
```

Parameters

sppp

A pointer to a thread structure pointer. If the call is successful, the address of an internal thread structure is returned in *sppp*.

stacksize

The size of the stack; it must be at least SRV_C_MINSTACKSIZE. Specify SRV_DEFAULT_STACKSIZE to use the stack size set with srv_props(SRV_S_STACKSIZE).

funcp
A pointer to a function that is the entry point for the newly created thread. The thread begins by executing the routine located at *funcp*. The thread is freed when that routine returns or srv_termproc is called.

argp
A pointer that is passed to the routine in **funcp* when the thread begins execution.

priority
An integer between SRV_C_LOWPRIORITY and SRV_C_MAXPRIORITY that indicates the base priority of the spawned thread. The default priority is SRV_C_DEFAULTPRI.

Return value

srv_spawn returns CS_SUCCEED if the thread is successfully spawned. This guarantees only that sufficient Open Server internal resources are available. It does not validate the correctness of the entry point routine or its argument. If the thread cannot be spawned, srv_spawn returns CS_FAIL.

Table 3-133: Return values (srv_spawn)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <stdio.h>
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE       entryfunc PROTOTYPE((
CS_CHAR          *message
));

CS_RETCODE       ex_srv_spawn PROTOTYPE((
SRV_PROC         *spp,
CS_INT           stacksize,
CS_INT           priority
));

CS_RETCODE    entryfunc(message)
CS_CHAR       *message;
{
      printf("Welcome to a new thread - %s!\n", message);
      return(CS_SUCCEED);
```

```
}

/*
** EX_SRV_SPAWN
**
**    Example routine to allocate a service thread
**
** Arguments:
**    spp      A pointer to an internal thread control
 **              structure.
**    stacks   The desired thread stack size.
**    priority The desired thread scheduling priority.
**
** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE    ex_srv_spawn(spp, stacksize, priority)
SRV_PROC      *spp;
CS_INT        stacksize;
CS_INT        priority;
{
    CS_CHAR      msgarg[20];

    strcpy(msgarg, "come in");

    return(srv_spawn(&spp, stacksize, entryfunc, msgarg,
              priority));
}
```

Usage

- srv_spawn allocates a "service" thread—one that is neither event-driven nor associated with any client. The thread runs under the control of the scheduler.
- Threads created by srv_spawn are called service threads because they often provide services required by the event-driven threads, such as accessing shared devices and data objects.
- srv_spawn informs the Open Server about a new thread and makes the thread runnable. The thread does not begin execution immediately. The moment that it actually does start execution is determined by many factors, such as the priority of the spawned thread and the priorities of other runnable threads.

- If you do not call srv_props to configure the stacksize with SRV_S_STACKSIZE, a new thread is created with the default stacksize. This default stacksize depends on the platform used. For native-threaded versions of Open Server, the default stacksize of underlying threads is used.
- Code executed by multiple threads must be re-entrant.

See also

srv_callback, srv_createproc, srv_props, srv_termproc

srv_symbol

Description

Convert an Open Server token value to a readable string.

Syntax

```
CS_CHAR *srv_symbol(type, symbol, lenp)

CS_INT          type;
CS_INT          symbol;
CS_INT         *lenp;
```

Parameters

type

The type of token. Table 3-134 describes the legal token types:

Table 3-134: Token types corresponding to type (srv_symbol)

Token Type	Description
SRV_DATATYPE	A datatype
SRV_EVENT	An event type
SRV_DONE	A DONE status type
SRV_ERROR	An error severity token

symbol

The actual token value.

lenp

A pointer to a CS_INT variable that will contain the length of the returned string.

Return value

Table 3-135: Return values (srv_symbol)

Returns	To indicate
A pointer to a null terminated character string that is a readable translation of an Open Server token value.	The token value.
A null pointer	Open Server does not recognize the *type* or *symbol*. Open Server sets *lenp* to -1.

Examples

```
#include    <ospublic.h>
/*
 ** Local Prototype
 */
extern CS_RETCODE       ex_srv_symbol PROTOTYPE((
CS_INT type,
CS_INT symbol,
CS_CHAR *namep
));
/*
 ** EX_SRV_SYMBOL
**
**   Retrieve a printable string representation of an Open Server
 **   symbol
**
** Arguments:
**      type          Symbol type
**      symbol        Symbol for which to retrieve string
**      namep         Return symbol string here
** Returns:
 **      CS_SUCCEED       Symbol string was retrieved successfully
 **      CS_FAIL          An error was detected
 */
CS_RETCODE           ex_srv_symbol(type, symbol, namep)
CS_INT               type;
CS_INT               symbol;
CS_CHAR           *namep;
{
     CS_INT          len;
     namep = srv_symbol(type, symbol, &len);
     if(namep == (CS_CHAR *)NULL)
     {
          return(CS_FAIL);
     }
     return(CS_SUCCEED);
```

```
}
```

Usage

- srv_symbol returns a pointer to a readable null terminated string that describes an Open Server token value.
- The pointer srv_symbol returns points to space that is never overwritten, so it is safe to call srv_symbol more than once in the same statement.
- Table 3-136 summarizes the tokens srv_symbol can convert:

Table 3-136: Convertible tokens (srv_symbol)

Token type	Token	Description
SRV_ERROR	SRV_INFO	Error severity type
SRV_ERROR	SRV_FATAL_PROCESS	Error severity type
SRV_ERROR	SRV_FATAL_SERVER	Error severity type
SRV_DONE	SRV_DONE_MORE	DONE packet status field
SRV_DONE	SRV_DONE_ERROR	DONE packet status field
SRV_DONE	SRV_DONE_FINAL	DONE packet status field
SRV_DONE	SRV_DONE_FLUSH	DONE packet status field
SRV_DONE	SRV_DONE_COUNT	DONE packet status field
SRV_DATATYPE	CS_CHAR_TYPE	Char datatype
SRV_DATATYPE	CS_BINARY_TYPE	Binary datatype
SRV_DATATYPE	CS_TINYINT_TYPE	1-byte integer datatype
SRV_DATATYPE	CS_SMALLINT_TYPE	2-byte integer datatype
SRV_DATATYPE	CS_INT_TYPE	4-byte integer datatype
SRV_DATATYPE	CS_REAL_TYPE	Real datatype
SRV_DATATYPE	CS_FLOAT_TYPE	Float datatype
SRV_DATATYPE	CS_BIT_TYPE	Bit datatype
SRV_DATATYPE	CS_DATETIME_TYPE	Datetime datatype
SRV_DATATYPE	CS_DATETIME4_TYPE	4-byte datetime datatype
SRV_DATATYPE	CS_MONEY_TYPE	Money datatype
SRV_DATATYPE	CS_MONEY4_TYPE	4-byte money datatype
SRV_DATATYPE	SRVCHAR	Char datatype
SRV_DATATYPE	SRVVARCHAR	Variable-length char datatype
SRV_DATATYPE	SRVBINARY	Binary datatype
SRV_DATATYPE	SRVVARBINARY	Variable-length binary datatype
SRV_DATATYPE	SRVINT1	1-byte integer datatype
SRV_DATATYPE	SRVINT2	2-byte integer datatype
SRV_DATATYPE	SRVINT4	4-byte integer datatype
SRV_DATATYPE	SRVINTN	Integer datatype, nulls allowed
SRV_DATATYPE	SRVBIT	Bit datatype
SRV_DATATYPE	SRVDATETIME	Datetime datatype
SRV_DATATYPE	SRVDATETIME4	4-byte datetime datatype
SRV_DATATYPE	SRVDATETIMN	Datetime datatype, nulls allowed

Token type	Token	Description
SRV_DATATYPE	SRVMONEY	Money datatype
SRV_DATATYPE	SRVMONEY4	4-byte money datatype
SRV_DATATYPE	SRVMONEYN	Money datatype, nulls allowed
SRV_DATATYPE	SRVREAL	4-byte float datatype
SRV_DATATYPE	SRVFLT8	8-byte float datatype
SRV_DATATYPE	SRVFLTN	8-byte float datatype, nulls allowed
SRV_DATATYPE	SRV_LONGCHAR_TYPE	Long char datatype
SRV_DATATYPE	SRV_LONGBINARY_TYPE	Long binary datatype
SRV_DATATYPE	SRV_TEXT_TYPE	Text datatype
SRV_DATATYPE	SRV_IMAGE_TYPE	Image datatype
SRV_DATATYPE	SRV_NUMERIC_TYPE	Numeric datatype
SRV_DATATYPE	SRV_DECIMAL_TYPE	Decimal datatype
SRV_DATATYPE	SRVVOID	Void datatype
SRV_EVENT	SRV_ATTENTION	Open Server event type
SRV_EVENT	SRV_BULK	Open Server event type
SRV_EVENT	SRV_CONNECT	Open Server event type
SRV_EVENT	SRV_CURSOR	Open Server event type
SRV_EVENT	SRV_DISCONNECT	Open Server event type
SRV_EVENT	SRV_DYNAMIC	Open Server event type
SRV_EVENT	SRV_LANGUAGE	Open Server event type
SRV_EVENT	SRV_MSG	Open Server event type
SRV_EVENT	SRV_OPTION	Open Server event type
SRV_EVENT	SRV_RPC	Open Server event type
SRV_EVENT	SRV_START	Open Server event type
SRV_EVENT	SRV_STOP	Open Server event type
SRV_EVENT	SRV_URGDISCONNECT	Open Server event type

See also srv_descfmt

srv_tabcolname

Description Associate **browse mode** result columns with result tables.

Syntax CS_RETCODE srv_tabcolname(spp, colnum, brwsdescp)

```
SRV_PROC            *spp;
CS_INT               colnum;
CS_BROWSEDESC       *brwsdescp;
```

Parameters

spp

A pointer to an internal thread control structure.

colnum

The number used to identify the column that was previously described using srv_descfmt.

brwsdescp

A pointer to a structure containing browse information about the column in question. Specifically, it should contain the number of the table (previously described through srv_tabname) containing the column and the original column name and name length. Note that the original column name and name length are only needed if the column has been renamed in the select statement (indicated by a status of CS_RENAMED in the CS_BROWSEDESC structure). For more information on the CS_BROWSEDESC structure, see "CS_BROWSEDESC structure" on page 52.

Return value

Table 3-137: Return values (srv_tabcolname)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include          <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE         ex_srv_tabcolname PROTOTYPE((
SRV_PROC           *spp,
CS_INT             colnum,
CS_BROWSEDESC      *bdp
));

/*
** EX_SRV_TABCOLNAME
**
**    Example routine to associate a browse mode result column
 **    with result tables.
**
** Arguments:
```

```
**    spp      A pointer to an internal thread control structure.
**
**    colnum   The column number.
**
**    bdp      A pointer to the browse descriptor for the
 **             column.
**
** Returns:
**     CS_SUCCEED   If we successfully associated this result
 **                  column with its table.
**
**     CS_FAIL      If an error was detected.
**
*/
CS_RETCODE          ex_srv_tabcolname(spp, colnum, bdp)
SRV_PROC           *spp;
CS_INT             colnum;
CS_BROWSEDESC      *bdp;
{
     CS_RETCODE         result;

     result = srv_tabcolname(spp, colnum, bdp);

     return (result);
}
```

Usage

- srv_tabcolname is used to send browse mode result information to a client. The information an application can send includes:
  - The name of the table to which a result column maps
  - The real name of a column that was renamed in the client query's select statement
- The column must have previously been defined using srv_descfmt.
- The table must have previously been defined using srv_tabname.
- srv_tabcolname is called once for each result column that is a column in a result row.

See also

srv_descfmt, srv_tabname, "Browse mode" on page 22

srv_tabname

Description

Provide the name of the table or tables associated with a set of browse mode results.

Syntax

```
CS_RETCODE srv_tabname(spp, tablenum, tablenamep,
                namelen)

SRV_PROC       *spp;
CS_INT          tablenum;
CS_CHAR        *tablenamep;
CS_INT          namelen;
```

Parameters

spp

A pointer to an internal thread control structure.

tablenum

The number used to identify the table in subsequent calls to srv_tabcolname.

tablenamep

A pointer to the name of the table. It cannot be null, as tables always have names.

namelen

The length, in bytes, of the table name. If *namelen* is CS_NULLTERM, then Server Library expects the table name to be null terminated.

Return value

Table 3-138: Return values (srv_tabname)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include          <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_tabname PROTOTYPE((
SRV_PROC        *sproc,
CS_INT          tablenum,
CS_CHAR         *tablename
));

/*

** EX_SRV_TABNAME
**    An example routine to provide the name of the table
```

```
 **    associated with a set of browse mode results.
**
** Arguments:
**    sproc         A pointer to an internal thread control
 **                  structure.
**    tablenum      The number that will be used to identify
 **                  the table in subsequent calls to

 **                  srv_tabcolname.
**    tablename     A null terminated string specifying the
 **                  table name.
**
** Returns:
**    CS_SUCCEED   If the table is successfully described.
**    CS_FAIL      If an error was detected.
*/
CS_RETCODE         ex_srv_tabname(sproc, tablenum, tablename)
SRV_PROC           *sproc;
CS_INT             tablenum;
CS_CHAR            *tablename;
{
     return( srv_tabname(sproc, tablenum, tablename,
                CS_NULLTERM) );
}
```

Usage

- srv_tabname is used to send to a client the name of the table or tables associated with browse mode results.
- An Open Server application must call srv_tabname once for each table involved in the browse mode results.
- The *tablenum* must be unique for all the tables described. Tables can be described in any order.
- An application links browse mode result columns to particular result tables using the srv_tabcolname routine. A call to srv_tabname must always precede a call to srv_tabcolname.

See also

srv_descfmt, srv_tabcolname, “Browse mode” on page 22

srv_termproc

Description — Terminate the execution of a thread.

Syntax

```
CS_RETCODE srv_termproc(spp)
SRV_PROC      *spp;
```

Parameters

spp
A pointer to an internal thread control structure.

Return value

Table 3-139: Return values (srv_termproc)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include          <ospublic.h>
/*
 ** Local Prototype.
 */
CS_RETCODE         ex_srv_termproc PROTOTYPE((
SRV_PROC           *spp
));

/*
 ** EX_SRV_TERMPROC
**
**   Example routine to terminate the execution of a thread using
 **   srv_termproc.
**
** Arguments:
**   spp    A pointer to an internal thread control structure.
** Returns:
**
**    CS_SUCCEED    Thread successfully terminated
**    CS_FAIL       An error was detected.
 */
CS_RETCODE          ex_srv_termproc(spp)
SRV_PROC            *spp;
{
      /*
       ** Terminate the thread.
       */
      if (srv_termproc(spp) != CS_SUCCEED)
      {
              return(CS_FAIL);
```

```
        }
        return(CS_SUCCEED);
    }
```

Usage

- Using srv_createproc, Open Server applications can create event driver threads that are not associated with a client connection.
- srv_termproc cannot be used in a SRV_START handler.
- Do not call srv_termproc from interrupt level code; the results are unpredictable.
- Mutexes, mutex locks, registered procedures, queued events, and messages associated with a thread are destroyed when the thread terminates.
- The following code fragment illustrates the use of srv_termproc:

See also

srv_createproc, srv_event, srv_event_deferred, srv_spawn

srv_text_info

Description

Set or get a description of text or image data.

Syntax

```
CS_RETCODE srv_text_info(spp, cmd, item, iodescp)

SRV_PROC     *spp;
CS_INT        cmd;
CS_INT        item;
CS_IODESC    *iodescp;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

The direction of data flow. Table 3-140 summarizes the legal values for *cmd*:

Table 3-140: Values for cmd (srv_text_info)

Value	Meaning
CS_SET	The Open Server application is setting internal Server-Library structures to describe text or image data. The srv_text_info call will update a text or image column (inside Open Server) with the information in *iodescp*. (The application must have previously described the column using srv_descfmt.) Typically, this will be followed by a call to srv_send_text, or srv_bind and srv_xferdata.
CS_GET	Open Server is updating the *iodescp* structure with the total length of the text or image data to be read from a client. Typically, this will be followed by a call to srv_get_text. See the comments section below for limitations regarding the CS_GET direction.

item

The column number of the column being described. The first column in a row is column 1. This parameter is ignored when *cmd* is CS_GET.

iodescp

A pointer to a structure that describes the object name, text pointer, and timestamp for a text column. See "CS_IODESC structure" on page 57 for details.

Return value

Table 3-141: Return values (srv_text_info)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include          <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_text_info PROTOTYPE((
SRV_PROC        *spp,
CS_INT          item
CS_IODESC       *iodp
));

/*
** EX_SRV_TEXT_INFO
**
**     Example routine to set a column's text or image data
 **     description before transferring a data row, using
 **     srv_text_info. This example routine would be used in a
```

```
 **     gateway application, where the Open Client application has
 **     initiated an update of text or image data.
**
** Arguments:
**    spp   A pointer to an internal thread control structure.
**    item  The column number of the column being described.
**    iodp  A pointer to a CS_IODESC structure that describes the
 **          text or image data (This stucture is passed from the
 **          Open Client application).
**
** Returns:
**    CS_SUCCEED    Text or image data successfully described.
**    CS_FAIL      An error occurred was detected.
*/
CS_RETCODE      ex_srv_text_info(spp, item, iodp)
SRV_PROC        *spp;
CS_INT          item;
CS_IODESC       *iodp;
{
      /*
      ** Describe the text or image data for the column.
      */
      if (srv_text_info(spp, (CS_INT)CS_SET, item, iodp) !=
            CS_SUCCEED)
      {
           return(CS_FAIL);
      }
      return(CS_SUCCEED);
}
```

Usage

- srv_text_info is used to describe text or image columns for sending a result row or retrieving a parameter.
- If *cmd* is CS_GET, srv_text_info must be called from the SRV_BULK event handler.
- If *cmd* is CS_GET, srv_text_info must be called prior to a call to srv_get_text.
- If *cmd* is CS_SET, srv_text_info must be called for each text or image datatype column in a row before srv_xferdata or srv_send_text is called.
- Text and image data is transferred to a client using either srv_bind followed by srv_xferdata, or srv_send_text.

See also

srv_bind, srv_descfmt, srv_get_text, srv_send_text, srv_xferdata, “Text and image” on page 196

srv_thread_props

Description

Define and retrieve thread properties.

Syntax

```
CS_RETCODE srv_thread_props(spp, cmd, property, bufp, buflen, outlenp)

SRV_PROC *spp;
CS_INT     cmd;
CS_INT     property;
CS_VOID    *bufp;
CS_INT     buflen;
CS_INT     *outlenp;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

The action to take. Table 3-142 summarizes the legal values for *cmd*:

Table 3-142: Values for cmd (srv_thread_props)

Value	Meaning
CS_SET	The Open Server application is setting the property. In this case, *bufp* should contain the value the property is to be set to, and *buflen* should specify the size, in bytes, of that value.
CS_GET	The Open Server application is retrieving the property. In this case, *bufp* should point to the buffer where the property value is placed, and *buflen* should be the size, in bytes, of the buffer.
CS_CLEAR	The Open Server application is resetting the property to its default value. In this case, *bufp*, *buflen*, and *outlenp* are ignored.

property

The property being set, retrieved or cleared. See below for a list of this argument's legal values.

bufp

A pointer to the Open Server application data buffer where property value information from the client is placed or property value information is retrieved.

buflen

The length, in bytes, of the buffer.

outlenp

A pointer to a CS_INT variable, which Open Server will set to the size, in bytes, of the property value retrieved. This argument is only used when *cmd* is CS_GET, and is optional.

Return value

Table 3-143: Return values (srv_thread_props)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE ex_srv_thread_props PROTOTYPE((
SRV_PROC   *sp,
CS_CHAR    *user,
CS_INT     ulen,
CS_INT     *lenp
));
/*
** EX_SRV_THREAD_PROPS
**
** Example routine to obtain a client thread's user name through
** srv_thread_props.
**
** Arguments:
** sp     A pointer to an internal thread control structure.
** user   A pointer to the address of the user name buffer.
** ulen   The size of the user name buffer.
** lenp  A pointer to an integer variable, that will be set to the length
**         of the user name string.
**
** Returns:
** CS_TRUE   If the user name was returned succesfully.
** CS_FALSE  If an error was detected.
*/
CS_RETCODE ex_srv_thread_props(sp, user, ulen, lenp)
SRV_PROC   *sp;
CS_CHAR    *user;
CS_INT     ulen;
CS_INT     *lenp;
{
   /*
    ** Call srv_thread_props to get the user name.
   */
   if( srv_thread_props(sp, CS_GET, SRV_T_USER, user, ulen, lenp)
      == CS_FAIL )
   {
```

```
        /*
        ** An error was al&ready raised.
        */
        return CS_FAIL;
    }
/*
** All done.
*/
return CS_SUCCEED;
}
```

Usage

- srv_thread_props is called to define, retrieve, and reset thread properties.
- Table 3-144 summarizes legal property values, whether they can be set or retrieved, and each value's datatype.

  Refer to Table 2-28 on page 149 for descriptions of each thread property.

Table 3-144: Thread properties and their datatypes (srv_thread_props)

Property	SET/ CLEAR	GET	bufp when cmd is CS_SET:	bufp when cmd is CS_GET:
SRV_T_APPLNAME	No	Yes	Not applicable	A pointer to a character string
SRV_T_BYTEORDER	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_BULKTYPE	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_BYTEORDER	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_CHARTYPE	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_CLIB	No	Yes	Not applicable	A pointer to a character string
SRV_T_CLIBVERS	No	Yes	Not applicable	A pointer to a character string
SRV_T_CLIENTLOGOUT	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_CONVERTSHORT	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_DUMPLOAD	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_ENDPOINT	No	Yes	Not applicable	A CS_VOID pointer to a buffer of sufficient size to hold the end point (file descriptor or file handle).
SRV_T_EVENT	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_EVENTDATA	No	Yes	Not applicable	The address of a CS_VOID pointer
SRV_T_FLTTYPE	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_GOTATTENTION	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_HOSTNAME	No	Yes	Not applicable	A pointer to a character string
SRV_T_HOSTPROCID	No	Yes	Not applicable	A pointer to a character string
SRV_T_IODEAD	No	Yes	Not applicable	A pointer to a CS_BOOL

Property	SET/ CLEAR	GET	bufp when cmd is CS_SET:	bufp when cmd is CS_GET:
SRV_T_LOCALE	Yes	Yes	A pointer to a CS_LOCALE pointer	A pointer to a CS_LOCALE pointer
SRV_T_LOGINTYPE	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_MACHINE	No	Yes	Not applicable	A pointer to a character string
SRV_T_MIGRATED	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_MIGRATE_STATE	No	Yes	Not applicable	A pointer to a SRV_MIG_STATE
SRV_T_NEGLOGIN	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_NOTIFYCHARSET	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_NOTIFYDB	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_NOTIFYLANG	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_NOTIFYPND	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_NUMRMTPWDS	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_PACKETSIZE	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_PASSTHRU	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_PRIORITY	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_PWD	No	Yes	Not applicable	A pointer to a character string
SRV_T_RETPARMS	No	Yes	Not applicable	Return parameters sent if an error occurs during execution
SRV_T_RMTPWDS	No	Yes	Not applicable	A pointer to an array of SRV_RMTPWD structures
SRV_T_RMTSERVER	No	Yes	Not applicable	A pointer to a character string
SRV_T_ROWSENT	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_SEC_CHANBIND	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_SEC_ CONFIDENTIALITY	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_SEC_ CREDTIMEOUT	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_SEC_DATAORIGIN	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_SEC_DELEGATION	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_SEC_DELEGCRED	No	Yes	Not applicable	A pointer to a CS_VOID
SRV_T_SEC_DETECTREPLAY	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_SEC_DETECTSEQ	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_SEC_INTEGRITY	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_SEC_MECHANISM	No	Yes	Not applicable	A pointer to a CS_CHAR

Property	SET/ CLEAR	GET	bufp when cmd is CS_SET:	bufp when cmd is CS_GET:
SRV_T_SEC_ MUTUALAUTH	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_SEC_ NETWORKAUTH	No	Yes	Not applicable	A pointer to a CS_BOOL
SRV_T_SEC_ SESSTIMEOUT	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_SESSIONID	Yes	Yes	A pointer to a CS_SESSIONID	A pointer to a CS_SESSIONID
SRV_T_SPID	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_STACKLEFT	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_TDSVERSION	Yes	Yes	A pointer to a CS_INT	A pointer to a CS_INT
SRV_T_TYPE	No	Yes	Not applicable	A pointer to a CS_INT
SRV_T_USER	No	Yes	Not applicable	A pointer to a character string
SRV_T_USERDATA	Yes	Yes	A CS_VOID pointer	The address of a CS_VOID pointer
SRV_T_USERVLANG	Yes	Yes	A pointer to a CS_BOOL	A pointer to a CS_BOOL
SRV_T_USTATE	Yes	Yes	A pointer to a character string	A pointer to a character string

- Table 3-145 lists the default values for the thread properties that can be defined (CS_SET).

Table 3-145: Definable thread properties and their default values (srv_thread_props)

Property	Default
SRV_T_USERDATA	(CS_VOID *)NULL
SRV_T_USTATE	NULL string
SRV_T_TDSVERSION	Min (client's, server's default)
SRV_T_USESRVLANG	Value of SRV_S_USESRVLANG
SRV_T_LOCALE	(CS_LOCALE *)NULL

- When the property is being retrieved (CS_GET), if *buflen* indicates that the user buffer is not big enough to hold the property value, Open Server will place the required number of bytes in *outlenp, and the application buffer will not be modified.
- See Table 2-28 on page 149 for descriptions of each thread property.

See also srv_props, "Properties" on page 139

srv_timedsleep

Description

Sleep until an event is signalled or until the specified time expires. srv_timedsleep is available in the reentrant libraries only.

Syntax

```
CS_RETCODE srv_timedsleep(sleepevent, sleeplabel,
        sleepflags, infop, srvmutex, timeout)

CS_VOID     *sleepevent;
CS_CHAR     *sleeplabel;
CS_INT      sleepflags;
CS_VOID     *infop;
SRV_OBJID   srvmutex;
CS_INT      timeout;
```

Parameters

sleepevent

A generic pointer to the event to sleep on.

sleeplabel

A pointer to a string for debugging puposes.

sleepflags

This parameter is used and performs the same usage as srv_sleep in suspending currently executing threads.

infop

A pointer to an integer describing the reason for a failure. The following are the integer values for *infop*:

- SRV_I_UNKNOWN — Unknown or no error
- SRV_I_TIMEOUT — The routine timed out
- SRV_I_INTERRUPTED — The srvlib process executing this function was interrupted by a call to srv_ucwakeup().

Note When this function returns SRV_I_INTERRUPTED, the srvlib process is interrupted while waiting on the event or while attempting to lock the mutex.

srvmutex

A srvlib mutex to be released when sleeping, and which will be locked after wakeup. Enter *0* if you do not want srv_timedsleep() to release and lock a mutex.

timeout

A timeout in milliseconds. Pass *0* to block indefinitely.

Return value

Table 3-146: Return values (srv_timedsleep)

Returns	To indicate
CS_SUCCEED	The routine succeeded.
CS_FAIL	The routine failed. See the *infop* parameter for more information.

Usage

It is possible to pass a mutex into this function for synchronization with a wakeup: The mutex will be released at such a point that another thread which obtains the mutex lock and then calls srv_wakeup() , for this event, succeeds in waking up the srvlib process executing this sleep function.

If the routine returns CS_SUCCEED the srvlib mutex will be locked. It will not be locked by this thread if the routine returns CS_FAIL.

See also

srv_wakeup

srv_ucwakeup

Description

Unconditionally wake up a sleeping thread.

Syntax

```
CS_RETCODE srv_ucwakeup(spp, wakeflags)
SRV_PROC    *spp;
CS_INT       wakeflags;
```

Parameters

spp

A pointer to an internal thread control structure.

wakeflags

A bit mask that modifies the way srv_ucwakeup behaves. Just one flag is defined; set *wakeflags* to 0 if it is not used.

SRV_M_WAKE_INTR

This flag indicates that the call to srv_ucwakeup is from interrupt level code. Failure to set this flag when calling srv_ucwakeup from interrupt level code can cause the Open Server application to behave erratically.

Return value

Table 3-147: Return values (srv_ucwakeup)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed, because the thread does not exist or was not sleeping.

Examples

```
&num;include <ospublic.h>
```

```
/*
** Local Protoype.
*/

CS_RETCODE      ex_srv_ucwakeup PROTOTYPE((
SRV_PROC        *sproc
));

/*
** EX_SRV_PROC
** An example routine to wake up a sleeping thread from
** a non-interrupt level by using srv_ucwakeup.
**
** Arguments:
** sproc   A pointer to an internal thread control
** structure.
**
** Returns:
** CS_SUCCEED  The specified thread was woken up.
** CS_FAIL      An error was detected.
*/

CS_RETCODE      ex_srv_ucwakeup(sproc)
SRV_PROC        *sproc;
{
/* Wake up the specified thread. */
return( srv_ucwakeup(sproc, 0));
}
```

Usage

- Waking a thread with srv_ucwakeup causes srv_sleep to return SRV_I_INTERRUPTED.
- Use srv_ucwakeup to wake a thread unconditionally. This may be necessary to break a deadlock condition or for cleanup.
- srv_ucwakeup cannot be used in a SRV_START handler.
- If srv_ucwakeup is called from interrupt level code, *wakeflags* must be set to SRV_M_WAKE_INTR. *wakeflags* must never be set to SRV_M_WAKE_INTR outside of an interrupt level routine.

See also

srv_sleep, srv_wakeup, srv_yield

srv_unlockmutex

Description: Unlock a mutex.

Syntax:

```
CS_RETCODE srv_unlockmutex(mutexid)
SRV_OBJID          mutexid;
```

Parameters:

mutexid
The unique mutex identifier that was returned by srv_createmutex. mutexid can be obtained from the mutex name with srv_getobjid.

Return value:

Table 3-148: Return values (srv_unlockmutex)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_unlockmutex PROTOTYPE((
CS_CHAR         *mutex_name
));

/*
** EX_SRV_UNLOCKMUTEX
**
**    Example routine to illustrate the use of srv_unlockmutex.
**
** Arguments:
**  mutex_name   The name of the mutex to be unlocked.
**
** Returns:
**
**    CS_SUCCEED Mutex successfully unlocked.
**    CS_FAIL      An error was detected.
*/
CS_RETCODE     ex_srv_unlockmutex(mutex_name)
CS_CHAR        *mutex_name;
{
     SRV_OBJID      id;
     CS_INT        info;

      /* Get the object id for the mutex. */
```

```
        if (srv_getobjid(SRV_C_MUTEX, mutex_name, CS_NULLTERM,
              &id,  &info) == CS_FAIL)
            return (CS_FAIL);

       /* Call srv_unlockmutex to unlock it. */
      if (srv_unlockmutex(id) == CS_FAIL)
        return (CS_FAIL);

        return (CS_SUCCEED);
   }
```

Usage

- Unlocking a mutex (mutual exclusion semaphore) releases the lock held on the semaphore, allowing other threads to access the mutex.
- srv_unlockmutex cannot be used in a SRV_START handler.

See also

srv_createmutex, srv_deletemutex, srv_getobjid

srv_version

Description

Define the version of Open Server an application is using.

Syntax

```
CS_RETCODE srv_version(contextp, version)

CS_CONTEXT      *contextp;
CS_INT          version;
```

Parameters

contextp

A pointer to a CS_CONTEXT structure, which the application has obtained through a call to cs_ctx_alloc. The CS_CONTEXT structure serves as a server-wide configuration structure shared with client libraries. For more information on the CS_CONTEXT structure, see “CS-Library” on page 59.

version

The version of Open Server the application assumes is in effect. Currently, the legal values for this parameter are CS_VERSION_100 and CS_VERSION_110, for Server Library versions 10.0 and 11.1, respectively.

Return value

Table 3-149: Return values (srv_version)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include <stdio.h>
```

```
#include <ospublic.h>
......
/*
 ** This code fragment sets the Open Server version.
 */
main()
 {
CS_CONTEXT *cp;
if (cs_ctx_alloc(CS_VERSION_110, &cp) != CS_SUCCEED)
 {
     fprintf(stderr, "cs_ctx_alloc failed \n");
      exit(1);
}
if(srv_version(cp, CS_VERSION_110) != CS_SUCCEED)
{
     /*
      ** Release the context structure al&ready allocated.
      */
     (CS_VOID)cs_ctx_drop(cp);
     (CS_VOID)fprintf(stderr, "srv_version failed \n");
       exit(1);
}
......
}
```

Usage

- An Open Server application must call srv_version prior to calling any other Server-Library routines. It must be preceded by a call to the CS-Library routine cs_ctx_alloc.
- Applications can first set localization configuration parameters in the CS_CONTEXT structure, using cs_config.

See also cs_ctx_alloc, cs_ctx_props

srv_wakeup

Description Enable sleeping threads to run.

Syntax

```
CS_RETCODE srv_wakeup(sleepeventp, wakeflags,
        reserved1, reserved2)

CS_VOID     *sleepeventp;
CS_INT      wakeflags;
CS_VOID     *reserved1;
CS_VOID     *reserved2;
```

Parameters

sleepeventp
A generic void pointer to the operating system event on which the threads are sleeping.

wakeflags
A bit mask that modifies the way that srv_wakeup behaves. If no bits are set, the default action is to wake up all threads sleeping on the event.The bits can be OR'd together. Table 3-150 describes the legal values for *wakeflags*:

Table 3-150: Values for wakeflags (srv_wakeup)

Value	Description
SRV_M_WAKE_INTR	The call to srv_wakeup is from interrupt level code. Failure to use this flag when calling srv_wakeup from interrupt level code can cause the Open Server application to behave erratically. Using this flag at non-interrupt level will cause the Open Server application to behave erratically.
SRV_M_WAKE_FIRST	Only the first thread sleeping on the event is made runnable.
SRV_M_WAKE_ALL	Wake up all threads sleeping on the event.

reserved1
This parameter is not used. It must be set to (CS_VOID*)0.

reserved2
This parameter is not used. It must be set to (CS_VOID*)0.

Return value

srv_wakeup returns CS_FAIL if no sleeping threads were found for the event or if any parameters were in error. If one or more sleeping threads were found, srv_wakeup returns CS_SUCCEED.

Table 3-151: Return values (srv_wakeup)

Returns	To indicate
CS_SUCCEED	One or more sleeping threads were found and enabled to run.
CS_FAIL	The routine failed, or no sleeping threads were found.

Examples

```
#include    <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE ex_srv_wakeup PROTOTYPE((
CS_VOID    *sep
));
```

```
/*
** EX_SRV_WAKEUP
**
**    Example routine using srv_wakeup to make all Open Server
 **    threads, which were previously sleeping on the specified
 **    sleep event, runnable again.
**
** Arguments:
**    sep    A generic void pointer, which was used previously in
 **           calls to srv_sleep to suspend threads.
**
** Returns:
**    CS_SUCCEED    Threads sleeping on the specified sleep event
 **                   are runnable again.
**    CS_FAIL       An error was detected.
*/
CS_RETCODE      ex_srv_wakeup(sep)
CS_VOID         *sep;
{
      /*
      ** Wake up threads for the specified sleep event, passing
       ** zero for reserved fields.
      */
      if (srv_wakeup(sep, (CS_INT)SRV_M_WAKE_ALL,
           (CS_VOID*)0, (CS_VOID*)0) != CS_SUCCEED)
     {
           return(CS_FAIL);
     }
     return(CS_SUCCEED);
}
```

Usage

- srv_wakeup wakes threads that are sleeping on *sleepevent*.
- When srv_wakeup is called from interrupt level code, the actual wakeup is deferred until the scheduler next executes.
- srv_wakeup cannot be used in a SRV_START handler.
- When writing preemptive mode programs with Open Server, srv_wakeup and srv_sleep must use platform-dependent mutexes. See the Open Client and Open Server *Programmer's Supplement* for your platform for an example of preemptive scheduling.

See also

srv_sleep

srv_xferdata

Description

Send parameters or data to a client, or receive parameters or data from a client.

Syntax

```
CS_RETCODE srv_xferdata(spp, cmd, type)

SRV_PROC        *spp;
CS_INT           cmd;
CS_INT          type;
```

Parameters

spp

A pointer to an internal thread control structure.

cmd

Indicates whether the data is going out to a client or coming in from a client. Table 3-152 describes the legal values for *cmd*:

Table 3-152: Values for cmd (srv_xferdata)

Value	Description
CS_SET	The application is calling srv_xferdata to send data to a client.
CS_GET	The application is calling srv_xferdata to retrieve data from a client.

type

The type of data stored into or read from the program variable. Table 3-153 describes the valid types and their appropriate context:

Table 3-153: Values for type (srv_xferdata)

Type	Valid cmd	Description of data
SRV_RPCDATA	CS_SET or CS_GET	RPC parameter
SRV_ROWDATA	CS_SET only	Result row column
SRV_CURDATA	CS_GET only	Cursor parameter
SRV_KEYDATA	CS_GET only	Cursor key column
SRV_ERRORDATA	CS_SET only	Error message parameter
SRV_DYNDATA	CS_SET or CS_GET	Dynamic SQL parameter
SRV_NEGDATA	CS_SET or CS_GET	Negotiated login parameter
SRV_MSGDATA	CS_SET or CS_GET	Message parameter
SRV_LANGDATA	CS_GET only	Language parameter

Return value

Table 3-154: Return values (srv_xferdata)

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

Examples

```
#include   <ospublic.h>
/*
** Local Prototype.
*/
CS_RETCODE      ex_srv_xferdata PROTOTYPE((
SRV_PROC       *spp
));

/*
** EX_SRV_XFERDATA
**
**    This routine will send error message parameters to the
 **    specified client.
**
**
** Arguments:
**
**    spp    A pointer to an internal thread control structure.
**
** Returns
**
**    CS_SUCCEED
**    CS_FAIL
**
*/
```

```
CS_RETCODE      ex_srv_xferdata(spp)
SRV_PROC        *spp;
{
    /* Check arguments.   */
    if(spp == (SRV_PROC *)NULL)
    {
        return(CS_FAIL);
    }
    return(srv_xferdata(spp,CS_SET,SRV_ERRORDATA));
}
```

Usage

- srv_xferdata is used to send parameter or row data to a client (CS_SET), or retrieve parameter or key data from a client. Specifically, it moves data out of local program variables and across the network to the client (CS_SET), or across the network from a client and into local program variables (CS_GET).
- The data as it must appear to the client (CS_SET) or appeared to the client (CS_GET) must have previously been described using srv_descfmt. The application must also have previously called srv_bind to define local program variables.
- srv_xferdata must be called once for each parameter stream (CS_GET, CS_SET) or once for each data row (CS_SET).

See also srv_bind, srv_descfmt

srv_yield

Description Allow another thread to run.

Syntax CS_RETCODE srv_yield()

Return value None.

Examples

```
#include   <stdio.h>
#include   <ospublic.h>

/*
** Local Prototype.
*/
CS_RETCODE ex_srv_yield PROTOTYPE((
    ));
/*
```

```
** EX_SRV_YIELD
**
**   Example routine to suspend the current thread.
** Arguments:
**    None.
**
** Returns:
**
**    CS_SUCCEED
**    CS_FAIL
*/
CS_RETCODE    ex_srv_yield()
{
    printf("I'll wait this one out...\n");
    if (srv_yield() == CS_FAIL)
    {
        printf("srv_yield() failed.\n");
        return(CS_FAIL);
    }
    else
    {
        printf("I'm back!\n");
        return(CS_SUCCEED);
    }
}
```

Usage

- srv_yield suspends the current thread and allows another runnable thread of the same or higher priority to run. The thread is rescheduled at a later time.
- srv_yield is primarily useful when using non-preemptive scheduling.
- If a thread calls srv_yield to allow a new thread which is still being established to run:
  a Open Server completes establishing the new thread.
  b If the new thread does not become runnable it will not gain control and the current thread will seem to get control back immediately.

  Refer to "Multithread programming" on page 109.
- The thread that calls srv_yield will resume execution at the statement following srv_yield.
- srv_yield cannot be used in a SRV_START handler.
- Do not call srv_yield from interrupt level code.

See also srv_sleep, srv_wakeup

CHAPTER 4

System Registered Procedures

This section contains a reference page for each Server-Library system registered procedure. System registered procedures are the registered procedures built into Open Server. When the server initializes, it registers these procedures so that they are available in every Open Server runtime system. The reference pages for the procedures describe their parameters and the results and messages they return.

For additional information on system registered procedures, see "Registered procedures" on page 162.

sp_ps

Description

Return detailed status information on specified Open Server threads.

Syntax

sp_ps [*loginame* | '*spid*']

Parameters

loginame

The user's login name.

spid

The internal identification number of the thread to report on. You can obtain the *spid* from the output of a previous sp_who or sp_ps call. By default, all threads are listed.

Examples

```
1>execute utility...sp_ps
 2>go
```

```
 spid  Login Name Host Name     Program Name  Task Type          ...
 ----  ---------- ---------     ------------  --------------     ...
    1                                        SERVER TASK        ...
    2                                        SERVER TASK        ...
    3                                        SERVER TASK        ...
    4                                        SERVICE TASK       ...
   11             hiram                      SITE HANDLER TASK  ...
   14  bud        sonoma        isql         CHILD TASK         ...
...    Status     Sleep Event   Sleep Label       Current Command    ...

...    -------    -----------   -----------       ---------------    ...
...    runnable   369448                          NETWORK HANDLER    ...
...    sleeping   369544        MSG AVAILABLE     CONNECT HANDLER    ...
...    sleeping   369640        MSG AVAILABLE     DEFERRED HANDLER   ...
...    runnable        0                          SCHEDULER          ...
...    sleeping   369736        MSG AVAILABLE                        ...
...    running    416480                                            ...

 ...  Blocked    Run     Current     Stack      Net       Net
 ...       By  Ticks    Priority    Origin    Writes    Reads
 ...  -------  ------   --------   --------   ------   -----
 ...        0       0          8     2794336        0       0
 ...        0       0          8     2810792        0       0
 ...        0       0          8     2827184        0       0
 ...        0       0         15     2843576        0       0
 ...        0       0          8     2859968        2       7
 ...        0       0          8     2909208        3       0
```

This example shows isql output from the sp_ps procedure. For printing purposes, the report was split where indicated by ellipses.

Usage

- sp_ps reports the detailed status of a specified server thread or all current Open Server threads. The information is useful for debugging during application development.
- *loginame* and *spid* are character string parameters. When using isql to execute sp_ps as a remote procedure call from an Adaptive Server, surround the *spid* in quotes to avoid a syntax error.
- If you do not specify *loginame* or *spid*, sp_ps lists all current threads.
- Table 4-1 summarizes the information sp_ps returns:

Table 4-1: Information returned (sp_ps)

Type of information	Meaning
spid	The internal thread number of the thread.
Login Name	The name of the logged in user. Applies only to client threads.
Host Name	For a client task, this is the name of the client's machine. For site handlers and server-to-server RPC connections, this is the name of the remote Adaptive Server.
Program Name	The name of the client application program.
Task Type	The type of thread. The legal values are NETWORK, CLIENT, SERVER, SITE HANDLER, CHILD, SERVICE, and UNKNOWN.
Status	The current status of the thread. The legal values for this column are running, runnable, sleeping, sick, free, stopped, spawned, terminal, and unknown. The one "running" task is the thread that is executing sp_ps.
Sleep Event	The event that will cause a sleeping thread to become runnable.
Sleep Label	A character string label that describes the sleep event.
Current Command	A character string that describes the state of the thread. The contents of this column are set by the srv_thread_props routine.
Blocked By	(Not currently used.
Run Ticks	(Not currently used.
Current Priority	The priority at which the thread is running.
Stack Origin	The address in memory where the thread's stack begins.
Net Writes	The number of network writes since the thread started. This number applies only to site handler and client threads.
Net Reads	The number of network reads since the thread started. This number applies only to site handler and client threads.

Table 4-2 summarizes the results returned as rows with these columns:

Table 4-2: Format of information returned (sp_ps)

Column name	Datatype	Length
spid	CS_INT_TYPE	4
Login Name	CS_CHAR_TYPE	SRV_MAXNAME
Host Name	CS_CHAR_TYPE	SRV_MAXNAME
Program Name	CS_CHAR_TYPE	SRV_MAXNAME
Task Type	CS_CHAR_TYPE	SRV_MAXNAME
Status	CS_CHAR_TYPE	SRV_MAXNAME
Sleep Event	CS_INT_TYPE	4
Sleep Label	CS_CHAR_TYPE	SRV_MAXNAME
Current Command	CS_CHAR_TYPE	SRV_MAXNAME
Blocked By	CS_INT_TYPE	4
Run Ticks	CS_INT_TYPE	4
Current Priority	CS_INT_TYPE	4
Stack Origin	CS_INT_TYPE	4
Net Writes	CS_INT_TYPE	4
Net Reads	CS_INT_TYPE	4

See also

sp_terminate, sp_who

sp_regcreate

Description

Create a registered procedure in Open Server.

Syntax

sp_regcreate proc_name, parm1, parm2, ...

Parameters

proc_name

The value of *proc_name* specifies the name of the registered procedure to be created.

parm1, parm2, ...

(Optional) If the client application passes additional parameters, they specify the names, datatypes, and default values of the new procedure's parameters.

Examples

```
Calling sp_regcreate from a Client-Library Client
```

This example creates a registered procedure np_test that takes parameters:

- *@p1*, datatype CS_INT, no default value (that is, the value defaults to NULL)

- *@p2*, datatype CS_CHAR, default value is “No value given”
- *@p3*, datatype CS_INT, default value is 0 (zero)

The fragment contains code for functions np_create, which creates the procedure, and rpc_results, which handles the results of the RPC command. The function ex_fetch_data (called by rpc_results) is not shown here. This function is defined in the file *exutils.c* in the Client-Library sample programs.

```
/*
 ** np_create() -- Example function to create a notification
 **   procedure on an Open Server.
 **
 ** Parameters:
 **   cmd - Command handle for sending commands.
 **
 ** Returns:
 **   CS_SUCCEED - The notification procedure was successfully
 **      created.
 **   CS_FAIL - Couldn't do it. This routine fails if the
 **      registered procedure already exists.
 */
 CS_RETCODE np_create(cmd)
 CS_COMMAND *cmd;
 {
   CS_DATAFMT datafmt;
   CS_INT     intval;
   CS_CHAR    charbuf[512];
   CS_BOOL    ok = CS_TRUE;
  /*
   ** Build up an RPC command to create the notification
   ** procedure np_test, defined as follows:
   **   np_test @p1 = <integer value>,
   **           @p2 = <character value>,
   **           @p3 = <integer value>
   */
  if (ok
       && (ct_command(cmd, CS_RPC_CMD,
           "sp_regcreate", CS_NULLTERM,
           CS_UNUSED) != CS_SUCCEED))
     ok = CS_FALSE;
  /*
   ** Name of the created procedure will be 'np_test'.
   */
   strcpy(datafmt.name, "proc_name");
   datafmt.namelen = strlen(datafmt.name);
   datafmt.datatype = CS_CHAR_TYPE;
```

```
 datafmt.status = CS_INPUTVALUE;
 datafmt.maxlength = 255;
 strcpy(charbuf, "np_test");
 if (ok &&
     ct_param(cmd, &datafmt,
             (CS_VOID *)charbuf, strlen(charbuf), 0)
     != CS_SUCCEED)
 {
   fprintf(stdout, "np_create: ct_param() @proc_name failed\n");
   ok = CS_FALSE;
 }
/*
 ** First parameter is named '@p1', is integer type, and has
 ** no default (i.e., defaults to NULL). We pass -1 as the
 ** indicator to ct_param() to specify a NULL value.
 */
 strcpy(datafmt.name, "@p1");
 datafmt.namelen = strlen(datafmt.name);
 datafmt.datatype = CS_INT_TYPE;
 datafmt.status = CS_INPUTVALUE;
 datafmt.maxlength = CS_UNUSED;
 if (ok &&
     ct_param(cmd, &datafmt, (CS_VOID *)NULL, CS_UNUSED, -1)
     != CS_SUCCEED)
 {
   fprintf(stdout, "np_create: ct_param() @p1 failed\n");
   ok = CS_FALSE;
 }
/*
 ** Second parameter is named '@p2', is character type, and has
 ** default "No value given".
 */
 strcpy(datafmt.name, "@p2");
 datafmt.namelen = strlen(datafmt.name);
 datafmt.datatype = CS_CHAR_TYPE;
 datafmt.status = CS_INPUTVALUE;
 datafmt.maxlength = 255;
 strcpy(charbuf, "No value given");
 if (ok &&
     ct_param(cmd, &datafmt,
             (CS_VOID *)&charbuf, strlen(charbuf), 0)
     != CS_SUCCEED)
 {
   fprintf(stdout, "np_create: ct_param() @p2 failed\n");
   ok = CS_FALSE;
 }
```

```
 /*
   ** Third parameter is named '@p3', is integer type, and
   ** has default 0 (zero).
   */
   strcpy(datafmt.name, "@p3");
   datafmt.namelen = strlen(datafmt.name);
   datafmt.datatype = CS_INT_TYPE;
   datafmt.status = CS_INPUTVALUE;
   datafmt.maxlength = CS_UNUSED;
   intval = 0;
   if (ok &&
       ct_param(cmd, &datafmt, (CS_VOID *)&intval, CS_UNUSED, 0)
       != CS_SUCCEED)
   {
     fprintf(stdout, "np_create: ct_param() @p3 failed\n");
     ok = CS_FALSE;
   }

  /*
   ** Send the RPC command.
   */
   if (ok && ct_send(cmd) != CS_SUCCEED)
     ok = CS_FALSE;

  /*
   ** Process the results from the RPC execution.
   */
   if (ok && rpc_results(cmd, CS_FALSE) != CS_SUCCEED)
     ok = CS_FALSE;

 return (ok ? CS_SUCCEED : CS_FAIL);

 } /* np_create */

/*
 ** rpc_results() -- Process results from an rpc.
 **
 ** Parameters
 **    cmd -- The command handle with results pending.
 **    expect_fetchable -- CS_TRUE means fetchable results
 **           are expected. They will be printed w/ the
 **           ex_fetch_data() routine (defined in file exutils.c).
 **           CS_FALSE means fetchable results cause this routine
 **           to fail.
 **
 ** Returns
 **   CS_SUCCEED -- no errors.
```

```
 **   CS_FAIL -- ct_results failed, returned a result_type value
 **      of CS_CMD_FAIL, or returned unexpected fetchable results.
 */
CS_RETCODE rpc_results(cmd, expect_fetchable)
 CS_COMMAND *cmd;
 CS_BOOL    expect_fetchable;
 {
   CS_RETCODE results_ret;
   CS_INT     result_type;
   CS_BOOL    ok = CS_TRUE;
   CS_BOOL    cmd_failed = CS_FALSE;
  while (ok &&
          (results_ret
           = ct_results(cmd, &result_type))
           == CS_SUCCEED)
   {
     switch((int)result_type)
     {
       case CS_STATUS_RESULT:
       case CS_ROW_RESULT:
       case CS_COMPUTE_RESULT:
       case CS_PARAM_RESULT:
          /*
           ** These cases indicate fetchable results.
           */
           if (expect_fetchable)
           {
             /* ex_fetch_data() is defined in exutils.c */
             ok = (ex_fetch_data(cmd) == CS_SUCCEED);
           }
           else
           {
             (CS_VOID)fprintf(stdout,
                              "RPC returned unexpected result\n");
             (CS_VOID)ct_cancel(NULL, cmd, CS_CANCEL_ALL);
             ok = CS_FALSE;
           }
           break;
      case CS_CMD_SUCCEED:
       case CS_CMD_DONE:
          /* No action required */
          break;

      case CS_CMD_FAIL:
          (CS_VOID)fprintf(stdout,
            "RPC command failed on server.\n");
          cmd_failed = CS_TRUE;
```

```
            break;
        default:
            /*
            ** Unexpected result type.
            */
            (CS_VOID)fprintf(stdout,
                             "RPC returned unexpected result\n");
            (CS_VOID)ct_cancel(NULL, cmd, CS_CANCEL_ALL);
            ok = CS_FALSE;
            break;
      } /* switch */
    } /* while */

   switch((int) results_ret)
    {
      case CS_END_RESULTS:
      case CS_CANCELED:
        break;
      case CS_FAIL:
      default:
        ok = 0;
    }
   return ((ok && !cmd_failed) ? CS_SUCCEED : CS_FAIL);

} /* rpc_results() */
```

Calling sp_regcreate from a DB-Library Client

This example creates a registered procedure named pricechange with two parameters. The first parameter is *@current_price* and is represented using the SYBMONEY datatype. The second parameter is *@sequence_num* and is a SYBINT4 datatype. Neither parameter has a default value.

```
dbnpdefine(dbproc, "pricechange", DBNULLTERM);
 dbregparam(dbproc, "@current_price", DBNULLTERM,
            SYBMONEY, DBNODEFAULT, NULL);
 dbregparam(dbproc, "@sequence_num", DBNULLTERM,
            SYBINT4, DBNODEFAULT, NULL);
 status = dbnpcreate(dbproc);

 if (status == FAIL)
 {
 fprintf(stderr,
     "Could not create pricechange procedure.\n");
 }
```

Table 4-3 summarizes the calls a SRV_C_PROCEXEC callback handler would use to find that the pricechange procedure is being registered:

Table 4-3: Returns (sp_regcreate)

Function call	Returns
srv_procname(srvproc, (int *) NULL)	"sp_regcreate"
srv_rpcparams(srvproc)	3
srv_paramdata(srvproc, 1)	"pricechange"
srv_paramdata(srvproc, 2)	"@current_price"
srv_paramdata(srvproc, 3)	"@sequence_num"

Usage

- Client applications call sp_regcreate remotely to create registered procedures.
- Registered procedures that are created by a client application are called *notification procedures*. They cannot contain application-defined code, and are primarily useful for client applications that rely on registered-procedure notifications.
- sp_regcreate's first parameter (*proc_name*) is the name of the procedure to create. If the new registered procedure takes parameters, they are defined by passing additional parameters. The new procedure's first parameter is passed as sp_regcreate's second parameter, the second as sp_regcreate's third, and so forth.
- Client applications built with Client-Library can create registered procedures by sending an RPC command that invokes sp_regcreate.

  An example is provided in "Calling sp_regcreate from a Client-Library Client" on page 418.
- DB-Library programs create registered procedures using dbnpdefine, dbregparam, and dbnpcreate. dbnpdefine internally generates an RPC command to remotely call sp_regcreate. dbnpcreate sends the RPC and processes the results.

  An example is provided in "Calling sp_regcreate from a DB-Library Client" on page 423.
- Server-Library programs can create registered procedures using srv_regdefine, srv_regparam, and srv_regcreate.

Messages

sp_regcreate can return the following messages:

Number	Severity	Text
16505	0	Procedure was registered successfully.
16506	11	Procedure is already registered.
16507	11	Unable to register procedure.

See also

sp_regdrop, sp_regnowatch, sp_regwatch, srv_regdefine, srv_regexec, srv_reginit, srv_regparam

sp_regdrop

Description

Remove a procedure from the list of registered procedures.

Syntax

sp_regdrop *proc_name*

Parameters

proc_name

The name of the registered procedure to remove.

Examples

```
1>execute stock...sp_regdrop pricechange
 2>go
```

In this example, a client logged into Adaptive Server with isql uses a server-to-server remote procedure call to execute sp_regdrop on the stock Open Server application. The procedure deletes the pricechange registered procedure from stock.

```
dbrpcinit(dbproc, "sp_regdrop", NULL);
 dbrpcparam(dbproc, "proc_name", NULL, SYBCHAR, -1,
 11, "pricechange");
 dbrpcsend(dbproc);
```

This example uses the DB-Library RPC routines to execute sp_regdrop with a single parameter “pricechange”. This causes the sp_regdrop system procedure to delete the pricechange registered procedure from Open Server.

Usage

- When a procedure is unregistered, clients that have pending notification requests receive a message to indicate that the procedure is no longer registered.
- sp_regdrop executes when a client executes dbnpdrop. The SRV_C_PROCEXEC callback handler can use srv_rpcname to find that sp_regdrop is executing. Then it can obtain a pointer to parameter number 1, *proc_name*, using srv_bind and srv_xferdata.

Messages

proc_name has been unregistered.

The procedure specified with the *proc_name* parameter was successfully unregistered.

proc_name is not a registered procedure.

The procedure specified with the *proc_name* parameter was not registered with Open Server.

```
Unable to unregister proc_name.
```

Open Server was unable to unregister the procedure for some other reason.

See also sp_regdrop, srv_regexec, srv_reginit, srv_regparam

sp_reglist

Description List all registered procedures in Open Server.

Syntax sp_reglist

Examples

```
1>execute utility...sp_reglist
 2>go

 Procedure Name
  ------------------------------
  sp_who


  sp_regwatch
  sp_ps
  sp_regdrop
  sp_reglist
  sp_regwatchlist
  sp_regcreate
  sp_regnowatch

 (0 rows affected)
```

This isql example lists all of the currently registered procedures.

Usage

- sp_reglist returns, as row data, the names of all of the procedures currently registered in Open Server.
- In a C program, you can also use sp_reglist to list the registered procedures.

Results are returned in rows containing a single char column with a data length of SRV_MAXNAME characters.

See also sp_regcreate, sp_regdrop, sp_regwatch, sp_regwatchlist

sp_regnowatch

Description
Remove a client from the notification list for a procedure.

Syntax
sp_regnowatch proc_name

Parameters
proc_name
The name of the registered procedure.

Examples

```
dbrpcinit(dbproc, "sp_regnowatch", (DBUSMALLINT)
            0);
 dbrpcparam(dbproc, "@proc_name", 0, SYBCHAR, 15,
            15, "pricechange");
 dbrpcsend(dbproc);
```

This example removes the client from the notification list for the pricechange registered procedure.

Usage
- This registered procedure executes when a client calls dbregnowatch.
- A SRV_C_PROCEXEC callback handler can use srv_rpcname to determine that sp_regnowatch is executing and sp_paramdata to obtain the name of the procedure for which the notification request is to be removed.

Messages

```
Notification request removed.
```

The notification request was removed successfully.

```
proc_name is not a registered procedure.
```

The procedure specified by *proc_name* is not registered in Open Server.

```
No requests pending.
```

The client had no notification requests pending for the procedure.

```
Unable to remove notification request.
```

Open Server failed to remove the notification request.

See also
sp_regcreate, sp_regdrop, sp_regwatch, sp_regnowatch, sp_regwatch

sp_regwatch

Description
Add the client to the notification list for a registered procedure.

Syntax
sp_regwatch *proc_name* [options]

Parameters
proc_name
The name of the registered procedure the client wishes notification for.

options

An CS_SMALLINT that specifies whether to notify the client just once or every time the procedure executes, and whether notification is synchronous or asynchronous. Table 4-4, below, shows the values that you can set for *options*. These values are bit flags, so you can set more than one at a time.

Table 4-4: Values for sp_regwatch options parameter

Values for option	Function
CS_NOTIFY_NOWAIT	Indicates asynchronous notification
CS_NOTIFY_WAIT	Indicates synchronous notification
SRV_NOTIFY_ALWAYS	Open Server will notify the client every time the procedure executes until the client disconnects or calls srv_regnowatch or dbregnowatch. This is the default.
SRV_NOTIFY_ONCE	Open Server removes the client from the notification list after it delivers a notification

Examples

```
dbrpcinit(dbproc, "sp_regwatch", (DBUSMALLINT) 0);
 dbrpcparam(dbproc, "@proc_name", 0, SYBCHAR,
                15, 15, "pricechange");
 dbrpcsend(dbproc);
```

This example adds the client to the notification list for a procedure called pricechange. Whenever the procedure executes, this client receives a notification.

```
optionval = SRV_NOTIFY_ONCE;
 dbrpcinit(dbproc, sp_regwatch, (DBUSMALLINT)
           DBWAIT);
 dbrpcparam(dbproc, "@proc_name", 0, SYBCHAR,
            15, 15, pricechange");
 dbrpcparam(dbproc, "@options", 0, SYBINT4, -1,
            -1, &optionval);
 dbrpcsend(dbproc);
```

This example adds the client to the notification list for a procedure called pricechange. It receives notification that the procedure executed just once.

Usage

- Open Server executes sp_regwatch internally when a client calls dbnpwatch.
- If the procedure is dropped while a client is waiting for a notification, the client receives an error message indicating that the procedure is no longer registered.

Messages

```
Notification request added.
```

The notification request was added successfully.

```
proc_name is not a registered procedure.
```

The procedure specified with the *proc_name* parameter is not registered with Open Server.

```
Unable to add notification request.
```

Open Server was unable to add the request for some other reason.

See also sp_regcreate, sp_regnowatch, sp_regdrop

sp_regwatchlist

Description List the registered procedures for which the client has requested notifications.

Syntax sp_regwatchlist

Examples

```
1>execute utility...sp_regwatchlist
 2>go

 Procedure Name
  ------------------------------
  pricechange
```

This isql example of a server-to-server RPC indicates that the client has requested notification for the pricechange registered procedure.

Usage

- Open Server executes sp_regwatchlist internally when a client calls dbregwatchlist.
- A SRV_C_PROCEXEC callback handler can call srv_rpcname to establish that sp_regwatchlist is executing.

Results are returned in rows containing a single char column of SRV_MAXNAME characters.

See also sp_reglist, sp_regwatchlist

sp_serverinfo

Description Send information about a character set or sort order to a client.

Syntax sp_serverinfo function [name]

Parameters

function

Table 4-5 summarizes the legal values for function:

Table 4-5: Values for function (sp_serverinfo)

Value	Meaning
server_csname	The name of the character set for the Open Server application will be sent as a one, single column, character row to the client.
server_soname	The name of the Open Server application sort order will be sent as one, single column, character row to the client.
csdefinition	A row containing the character set definition will be sent to the client. The row consists of three columns: type as a CS_SMALLINT_TYPE, ID as a CS_TINYINT_TYPE, and the character set definition as a CS_IMAGE_TYPE.
sodefinition	A row containing the sort order definition will be sent to the client. The row consists of three columns: type as a CS_SMALLINT_TYPE, ID as a CS_TINYINT_TYPE, and the sort order definition as a CS_IMAGE_TYPE.

name

The character set or sort order name. *name* need only be provided if *function* is set to *csdefinition* or *sodefinition*.

Usage

- The remote procedure sp_serverinfo is automatically registered and handled as a standard system procedure, for example, sp_who. When sp_serverinfo is received as an RPC Open Server handles it automatically. The application code need not be involved.
- If a client sends an sp_serverinfo request through a language request, this stored procedure must be executed using the registered procedure routines to send the correct response.
- The information is sent to a client as a row.

sp_terminate

Description

Terminate an Open Server thread.

Syntax

sp_terminate *spid* [, *options*]

Parameters

spid

The thread ID. This can be obtained with the sp_who procedure or by calling srv_thread_props.

options

Determines whether the thread is terminated immediately or by a queued disconnect event. Specify "deferred" to queue a disconnect event that occurs after previous events are handled. This is the default action. Specify "immediate" to terminate the thread immediately, ignoring current or queued events for the thread.

Examples

```
1> execute utility...sp_who
 2> go
```

```
spid  status     loginame      hostname   blk    cmd
----- ---------- ------------ ---------- ----  ----------------
1     runnable                              0    NETWORK HANDLER
2     sleeping                              0    CONNECT HANDLER
3     sleeping                              0    DEFERRED HANDLER
4     runnable                              0    SCHEDULER
12    runnable    ned           sonoma      0    PRINT TASK
24    running     bud           sonoma      0

(0 rows affected)
```

This example shows how to use isql to locate and terminate an errant server thread. The thread terminates immediately.

```
1> execute utility...sp_terminate 12, "immediate"
 2> go
```

```
spid = 12;
dbrpcinit(dbproc, "sp_terminate", (DBUSMALLINT) 0);
dbrpcparam(dbproc, "@spid", 0, SYBINT4, -1,
           -1, &spid);
dbrpcparam(dbproc, "@options", 0, SYBCHAR, 9,
           9, "deferred");
dbrpcsend(dbproc);
```

This DB-Library example queues a SRV_DISCONNECT event for the thread with the thread. The next time the thread becomes runnable, it receives the disconnect event and terminates.

Usage

- Use sp_who or sp_ps to find the *spid* for the thread to be terminated.
- In a Server-Library program, use srv_termproc to terminate a thread.

Messages

spid terminated.

spid scheduled for termination.

spid not currently in use.

See also

sp_who, srv_termproc

sp_who

Description

Return status information for specified Open Server threads.

Syntax

sp_who [*loginame* | '*spid*']

Parameters

loginame

The user's login name.

spid

The internal identification number of the thread to report on. The *spid* can be obtained from the output of a previous sp_ps or sp_who call. If no *spid* is specified, all threads are listed.

Examples

```
1>execute utility...sp_who
 2>go
```

```
 spid    status     loginame     hostname   blk cmd
-------- ---------- ------------ ---------- --- ----------------
       1 runnable                           0   NETWORK HANDLER
       2 sleeping                           0   CONNECT HANDLER
       3 sleeping                           0   DEFERRED HANDLER
       4 runnable                           0   SCHEDULER
      11 sleeping                hiram      0
      14 running    bud          sonoma     0
```

This example shows output from the sp_who procedure.

Usage

- sp_who reports status information about a specified server thread or all current Open Server threads.
- The output from the sp_who system registered procedure matches the output from the Adaptive Server sp_who system procedure.
- sp_who returns a subset of the information that sp_ps returns.
- *loginame* and *spid* are character string parameters. When using isql to execute sp_who as a remote procedure call from an Adaptive Server, surround the *spid* in quotes to avoid a syntax error.
- If you do not specify *loginame* or *spid*, sp_who lists all current threads.
- sp_who returns the following information:

  spid – the internal thread number of the thread.

  status – the current status of the thread. The values for this column are:

  - running
  - runnable

- sleeping
- sick
- free
- stopped
- spawned
- terminal
- unknown

The one "running" task is the thread that is executing sp_who.

loginame – the name of the logged in user. Applies only to client threads.

hostname – for a client task, this is the name of the client's machine. For a site handler thread, it is the name of the remote Adaptive Server.

blk – this field is unused and is always set to 0.

cmd – a character string that describes the state of the thread. The contents of this column are set by the srv_thread_props routine.

Table 4-6 summarizes the results returned as rows with these columns:

Table 4-6: Format of information returned (sp_who)

Column name	Datatype	Length
spid	CS_INT_TYPE	4
status	CS_CHAR_TYPE	10
loginame	CS_CHAR_TYPE	12
hostname	CS_CHAR_TYPE	10
blk	CS_INT_TYPE	3
cmd	CS_CHAR_TYPE	16

See also

sp_ps, sp_terminate

Glossary

Adaptive Server Enterprise

A server in Sybase's client/server architecture. Adaptive Server Enterprise manages multiple databases and multiple users, keeps track of the actual location of data on disks, maintains mapping of logical data description to physical data storage, and maintains data and procedure caches in memory. Prior to version 11.5, Adaptive Server Enterprise was known as SQL Server.

array

A structure composed of multiple identical variables that can be individually addressed.

array binding

The process of binding a result column to an array variable. At fetch time, multiple rows of the column are copied into the variable.

batch

A group of commands or statements.

A Client-Library command batch is one or more Client-Library commands terminated by an application's call to ct_send. For example, an application can batch together commands to declare, set rows for, and open a cursor.

A Transact-SQL statement batch is one or more Transact-SQL statements submitted to an Adaptive Server by means of a single Client-Library command or Embedded SQL statement.

browse mode

A method that DB-Library and Client-Library applications can use to browse through database rows, updating their values one row at a time. Cursors provide similar functionality and are generally more portable and flexible.

bulk copy

A utility for copying data in and out of databases. Also called bcp.

callback event

In Open Client and Open Server, an occurrence that triggers a callback routine.

callback routine

A routine that Open Client or Open Server calls in response to a triggering event, known as a callback event.

capabilities

A client/server connection's capabilities determine the types of client requests and server responses permitted for that connection.

character set A set of specific (usually standardized) characters with an encoding scheme that uniquely defines each character. ASCII and ISO 8859-1 (Latin 1) are two common character sets.

character set conversion Changing the encoding scheme of a set of characters on the way into or out of a server. Conversion is used when a server and a client communicating with it use different character sets. For example, if Adaptive Server uses ISO 8859-1 and a client uses Code Page 850, character set conversion must be turned on so that both server and client interpret the data passing back and forth in the same way.

client In client/server systems, the client is the part of the system that sends requests to servers and processes the results of those requests.

Client-Library Part of Open Client, a collection of routines for use in writing client applications. Client-Library is a library designed to accommodate cursors and other advanced features in the Sybase product line.

code set See **character set**.

collating sequence See **sort order**.

command In Client-Library, a server request initiated by an application's call to ct_command, ct_dynamic, or ct_cursor and terminated by the application's call to ct_send.

command structure (CS_COMMAND) A hidden Client-Library structure that Client-Library applications use to send commands and process results.

connection structure (CS_CONNECTION) A hidden Client-Library structure that defines a client/server connection within a context.

context structure (CS_CONTEXT) A CS-Library hidden structure that defines an application "context," or operating environment, within a Client-Library or Open Server application. The CS-Library routines cs_ctx_alloc and cs_ctx_drop allocate and drop a context structure.

conversion See character set conversion.

CS-Library Included with both the Open Client and Open Server products, a collection of utility routines that are useful to both Client-Library and Server-Library applications.

current row With respect to cursors, the row to which a cursor points. A fetch against a cursor retrieves the current row.

cursor A symbolic name that is associated with a SQL statement.

In Embedded SQL, a cursor is a data selector that passes multiple rows of data to the host program, one row at a time.

database A set of related data tables and other database objects that are organized to serve a specific purpose.

See also **scrollable cursor**.

datatype A defining attribute that describes the values and operations that are legal for a variable.

DB-Library Part of Open Client, a collection of routines for use in writing client applications.

deadlock A situation that arises when two users, each having a lock on one piece of data, attempt to acquire a lock on the other's piece of data. Adaptive Server detects deadlocks and resolves them by killing one user's process.

default Describes the value, option, or behavior that Open Client and Open Server products use when none is explicitly specified.

default database The database that a user gets by default when he or she logs in to a database server.

default language 1. The language that Open Client and Open Server products use when an application does no explicit localization. The default language is determined by the "default" entry in the locales file.

2. The language that Adaptive Server uses for messages and prompts when a user has not explicitly chosen a language.

dynamic SQL Allows an Embedded SQL or Client-Library application to execute SQL statements containing variables whose values are determined at runtime.

error message A message that an Open Client and Open Server product issues when it detects an error condition.

event An occurrence that prompts an Open Server application to take certain actions. Client commands and certain commands within Open Server application code can trigger events. When an event occurs, Open Server calls either the appropriate event-handling routine in the application code or the appropriate default event handler.

event handler In Open Server, a routine that processes an event. An Open Server application can use the default handlers Open Server provides or can install custom event handlers.

exposed structure — A structure whose internals are exposed to Open Client and Open Server programmers. Open Client and Open Server programmers can declare, manipulate, and deallocate exposed structures directly. The CS_DATAFMT structure is an example of an exposed structure.

extended transaction — In Embedded SQL, a transaction composed of multiple Embedded SQL statements.

FIPS — An acronym for Federal Information Processing Standards. If FIPS flagging is enabled, Adaptive Server or the Embedded SQL precompiler issue warnings when a non-standard extension to a SQL statement is encountered.

gateway — An application that acts as an intermediary for clients and servers that cannot communicate directly. Acting as both client and server, a gateway application passes requests from a client to a server and returns results from the server to the client.

hidden structure — A hidden structure is a structure whose internals are hidden from Open Client and Open Server programmers. Open Client and Open Server programmers must use Open Client and Open Server routines to allocate, manipulate, and deallocate hidden structures. The CS_CONTEXT structure is an example of a hidden structure.

host language — The programming language in which an application is written.

host program — In Embedded SQL, the application program that contains the Embedded SQL code.

host variable — In Embedded SQL, a variable which enables data transfer between Adaptive Server and the application program. See also **indicator variable**, **input variable**, **output variable**, **result variable**, and **status variable**.

indicator variable — A variable whose value indicates special conditions about another variable's value or about fetched data.

When used with an Embedded SQL host variable, indicates when a database value is null.

input variable — A variable that is used to pass information to a routine, a stored procedure, or Adaptive Server.

interfaces file — A file that maps server names to transport addresses. When a client application calls ct_connect or dbopen to connect to a server, Client-Library or DB-Library searches the interfaces file for the server's address. Note that not all platforms use the interfaces file. On these platforms, an alternate mechanism directs clients to server addresses.

isql script file — In Embedded SQL, one of the three files the precompiler can generate. An isql script file contains precompiler-generated stored procedures, which are written in Transact-SQL.

key — A subset of row data that uniquely identifies a row. Key data uniquely describes the **current row** in an open cursor.

keyword — A word or phrase that is reserved for exclusive use in Transact-SQL or Embedded SQL. Also called a reserved word.

listing file — In Embedded SQL, a listing file is one of the three files the precompiler can generate. A listing file contains the input file's source statements and informational, warning, and error messages.

locales file — A file that maps locale names to language/character set pairs. Open Client and Open Server products search the locales file when loading localization information.

locale name — A character string that represents a language/character set pair. Locale names are listed in the locales file. Sybase predefines some locale names, but a system administrator can define additional locale names and add them to the locales file.

locale structure — (CS_LOCALE) A CS-Library hidden structure that defines custom localization values for a Client-Library or Open Server application. An application can use a CS_LOCALE to define the language, character set, datepart ordering, and sort order it will use. The CS-Library routines cs_loc_alloc and cs_loc_drop allocate and drop a locale structure.

localization — The process of setting up an application to run in a particular national language environment. An application that is localized typically generates messages in a local language and character set and uses local datetime formats.

login name — The name a user uses to log in to a server. An Adaptive Server login name is valid if Adaptive Server has an entry for that user in the system table syslogins.

message number — A number that uniquely identifies an error message.

message queue — In Open Server, a linked list of message pointers through which threads communicate. Threads can write messages into and read messages from the queue.

multibyte character set — A character set that includes characters encoded using more than one byte. EUC JIS and Shift-JIS are examples of multibyte character sets.

mutex A mutual exclusion semaphore. This is a logical object that an Open Server application uses to ensure exclusive access to a shared object.

null Having no explicitly assigned value. NULL is not equivalent to zero, or to blank. A value of NULL is not considered to be greater than, less than, or equivalent to any other value, including another value of NULL.

Open Server A Sybase product that provides tools and interfaces for creating custom servers.

Open Server application A custom server constructed with Open Server.

output variable In Embedded SQL, a variable that passes data from a stored procedure to an application program.

parameter 1. A variable that is used to pass data to and retrieve data from a routine.

2. An argument to a stored procedure.

passthrough mode A state of being pertaining to gateway applications.

When in passthrough mode, a gateway relays Tabular Data Stream (TDS) packets between a client and a remote data source without unpacking the packets' contents.

property A named value stored in a structure. Context, connection, thread, and command structures have properties. A structure's properties determine how it behaves.

query 1. A data retrieval request; usually a select statement.

2. Any SQL statement that manipulates data.

registered procedure In Open Server, a collection of C statements stored under a name. Open Server-supplied registered procedures are called **system registered procedures**.

remote procedure call 1. One of two ways in which a client application can execute an Adaptive Server stored procedure. (The other is with a Transact-SQL execute statement.) A Client-Library application initiates a remote procedure call command by calling ct_command. A DB-Library application initiates a remote procedure call command by calling dbrpcinit.

2. A type of request a client can make of an Open Server application. In response, Open Server either executes the corresponding registered procedure or calls the Open Server application's RPC event handler.

3. A **stored procedure** executed on a different server from the server to which the user is connected.

result variable — In Embedded SQL, a variable that receives the results of a select or fetch statement.

scrollable cursor — Allows a current cursor position to be set anywhere in a result set. See also **cursor**.

server — In client/server systems, the part of the system that processes client requests and returns results to clients.

Server-Library — A collection of routines for use in writing Open Server applications.

sort order — Used to determine the order in which character data is sorted. Also called collating sequence.

sqlca — 1. In an Embedded SQL application, a SQLCA is a structure that provides a communication path between Adaptive Server and the application program. After executing each SQL statement, Adaptive Server stores return codes in the SQLCA.

2. In a Client-Library application, a SQLCA is a structure that the application can use to retrieve Client-Library and server error and informational messages.

sqlcode — 1. In an Embedded SQL application, a SQLCODE is a structure that provides a communication path between Adaptive Server and the application program. After executing each SQL statement, Adaptive Server stores return codes in the SQLCODE. A SQLCODE can exist independently or as a variable within a SQLCA structure.

2. In a Client-Library application, a SQLCODE is a structure that the application can use to retrieve Client-Library and server error and informational message codes.

SQL Server — See Adaptive Server Enterprise.

statement — In Transact-SQL or Embedded SQL, an instruction that begins with a keyword. The keyword names the basic operation or command to be performed.

status variable — In Embedded SQL, a variable that receives the return status value of a stored procedure, thereby indicating the procedure's success of failure.

stored procedure — In Adaptive Server, a collection of SQL statements and optional control-of-flow statements stored under a name. Adaptive Server-supplied stored procedures are called **system procedures**.

System Administrator — The user in charge of server system administration, including creating user accounts, assigning permissions, and creating new databases. On Adaptive Server, the System Administrator's login name is "sa."

system descriptor — In Embedded SQL, an area of memory that holds a description of variables used in Dynamic SQL statements.

system procedures — Stored procedures that Adaptive Server supplies for use in system administration. These procedures are provided as shortcuts for retrieving information from system tables, or as mechanisms for accomplishing database administration and other tasks that involve updating system tables.

system registered procedures — Internal registered procedures that Open Server supplies for registered procedure notification and status monitoring.

target file — In Embedded SQL, of the three files the precompiler can generate. A target file is similar to the original input file, except that all SQL statements are converted to Client-Library function calls.

TDS — (Tabular Data Stream) An application-level protocol that Sybase clients and servers use to communicate. It describes commands and results.

thread — A path of execution through Open Server application and library code and the path's associated stack space, state information, and event handlers.

Transact-SQL — An enhanced version of the database language SQL. Applications can use Transact-SQL to communicate with Sybase Adaptive Server.

transaction — One or more server commands that are treated as a single unit for the purposes of backup and recovery. Commands within a transaction are committed as a group; that is, either all of them are committed or all of them are rolled back.

transaction mode — The manner in which Adaptive Server manages transactions. Adaptive Server supports two transaction modes: Transact-SQL mode (also called "unchained transactions") and ANSI mode (also called "chained transactions").

user name — See **login name**.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X