



Server-Library/C Reference Manual

**Open Server™**

15.7

DOCUMENT ID: DC35400-01-1570-02

LAST REVISED: June 2012

Copyright © 2012 by Sybase, Inc. All rights reserved.

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

Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at <http://www.sybase.com/detail?id=1011207>. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

Java and all Java-based marks are trademarks or registered trademarks of Oracle and/or its affiliates in the U.S. and other countries.

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

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

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

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

# Contents

<b>About This Book</b> .....	<b>xix</b>	
<b>CHAPTER 1</b>	<b>Introducing Open Server</b> .....	<b>1</b>
	Client/Server overview .....	1
	Types of clients .....	2
	Types of servers.....	2
	Open Server configurations .....	3
	Standalone open server application .....	4
	Auxiliary open server application.....	4
	Gateway Open Server application.....	5
	Open Server .....	5
	The Open Server libraries .....	5
	Network services .....	6
	Using Open Server.....	7
	The CS_CONTEXT structure .....	7
	Steps in a simple program.....	7
	Basic Open Server program.....	8
	Open Server events .....	13
	Default event handlers .....	14
	Non client-initiated events .....	14
	Registered procedures .....	14
	Returning results to clients .....	15
	Types of result data .....	15
	Order of results.....	16
	Error handling.....	16
	Multithread programming .....	17
<b>CHAPTER 2</b>	<b>Topics</b> .....	<b>19</b>
	Attention events .....	20
	Interrupt-level activity .....	20
	Coding recommendations for attention events.....	21
	Handling disconnects .....	21
	Example .....	22

- Browse mode ..... 22
  - Example ..... 24
- Capabilities ..... 24
  - Request capabilities ..... 25
  - Response capabilities ..... 28
  - Transparent negotiation ..... 30
  - Server-wide defaults ..... 31
  - Explicit negotiation ..... 35
  - Ad hoc retrieval of capability information ..... 37
  - A note on pre-10.0 clients ..... 37
  - Example ..... 37
- Client command errors ..... 38
  - Sending messages with `srv_sendinfo` ..... 38
  - Sequencing long messages ..... 38
  - Extended error data ..... 39
- Connection migration ..... 40
  - In-batch migration and idle migration ..... 40
  - Context migration ..... 41
  - APIs used in connection migration ..... 41
  - Instructing clients to migrate to a different server ..... 48
  - Accepting connections from migrated clients ..... 52
  - Error messages ..... 52
- CS\_BROWSEDESC structure ..... 52
- CS\_DATAFMT structure ..... 54
- CS\_IODESC structure ..... 57
- CS-Library ..... 59
  - Common routines ..... 59
  - Common data structures ..... 60
  - Error handling ..... 60
- CS\_SERVERMSG structure ..... 60
- Cursors ..... 63
  - Cursor overview ..... 63
  - Advantages of cursors ..... 63
  - Open Server applications and cursors ..... 64
  - Handling cursor requests ..... 72
  - Key data ..... 76
  - Update columns ..... 76
  - Example ..... 76
- Scrollable cursors ..... 77
  - SRV\_CURDESC2 structure ..... 77
- Data stream messages ..... 80
  - Data stream messages overview ..... 80
  - Retrieving client data stream messages ..... 80
  - Sending data stream messages to a client ..... 81



Directory services.....	81
Specifying a directory driver .....	82
Registering an Open Server application with a directory .....	82
Dynamic SQL .....	83
Advantages of dynamic SQL.....	84
Handling dynamic SQL requests.....	84
Example .....	88
Dynamic listeners.....	88
Configuration .....	88
Starting a dynamic listener .....	89
Errors .....	90
Types of errors .....	90
Severity of errors .....	91
Error numbers and corresponding message text .....	92
Example .....	92
Events .....	93
Event overview .....	93
What is an event handler?.....	93
Standard events .....	94
Programmer-defined events.....	99
Example .....	99
Gateway applications.....	100
Passthrough mode .....	101
International support .....	101
Localizing an Open Server application.....	102
Supporting localized clients.....	103
Using a CS_LOCALE structure to set custom localization values	
103	
Responding to client requests.....	106
Localization properties .....	108
Localization examples .....	109
Language calls.....	109
Login redirection and extended HA failover support .....	110
Messages.....	111
Multithread programming .....	111
What is a thread? .....	111
Thread types .....	112
Scheduling.....	115
Tools and techniques .....	117
Programming considerations.....	120
Example .....	121
Negotiated behavior .....	121
Login negotiations .....	122
Ad hoc negotiations.....	123

- Example ..... 124
- Options ..... 124
  - Inside the SRV\_OPTION event handler ..... 125
  - Option descriptions and default values ..... 125
  - Example ..... 129
- Parameters ..... 129
  - Batched parameters ..... 129
- Partial update ..... 130
  - Open Server set-up ..... 130
- Passthrough mode ..... 131
  - Regular passthrough mode ..... 132
  - Event handler passthrough mode ..... 134
- Processing parameter and row data ..... 136
  - A note on terminology ..... 136
  - The Open Server data processing model ..... 136
  - Retrieving parameters ..... 137
  - Returning rows ..... 138
  - Returning return parameters ..... 138
  - A closer look at describing, binding, and transferring ..... 139
  - Returning parameters in a language data stream ..... 141
  - Example ..... 141
- Properties ..... 141
  - Context properties ..... 142
  - Server properties ..... 143
  - Thread properties ..... 151
- Registered procedures ..... 164
  - Standard remote procedure calls ..... 165
  - Advantages of registered procedures ..... 165
  - Notification procedures ..... 166
  - Creating registered procedures ..... 166
  - The mechanics of registered procedures ..... 166
  - System registered procedures ..... 168
  - Using callback handlers with registered procedures ..... 169
  - Example ..... 171
- Remote procedure calls ..... 171
  - Example ..... 172
- Retrieving large object parameters ..... 172
- Security services ..... 174
  - Security service properties ..... 175
  - How do security services work with Open Server? ..... 184
  - Using security mechanisms with Open Server applications.. 185
  - Determining which security services are active ..... 189
  - Scenarios for using security services with Open Server applications ..... 189

Text and image .....	201
Processing text and image data .....	201
Example .....	204
Types .....	204
Routines that manipulate datatypes .....	207
Open Server datatypes .....	207

**CHAPTER 3**

<b>Routines.....</b>	<b>217</b>
srv_alloc .....	221
srv_alt_bind.....	223
srv_alt_descfmt .....	227
srv_alt_header .....	231
srv_alt_xferdata.....	234
srv_bind.....	236
srv_bmove.....	241
srv_bzero .....	243
srv_callback .....	244
srv_capability .....	248
srv_capability_info.....	249
srv_createmsgq.....	253
srv_createmutex.....	255
srv_createproc .....	257
srv_cursor_props .....	259
srv_cursor_props2 .....	262
srv_dbg_stack .....	263
srv_dbg_switch .....	265
srv_define_event.....	266
srv_deletemsgq.....	268
srv_deletemutex.....	270
srv_descfmt.....	272
srv_dynamic .....	275
srv_envchange.....	280
srv_event.....	282
srv_event_deferred .....	285
srv_free .....	287
srv_freeserveraddr .....	288
srv_get_data .....	289
srv_get_text.....	290
srv_getloginfo.....	292
srv_getmsgq.....	294
srv_getobjid.....	297
srv_getobjname.....	300
srv_getserverbyname.....	302
srv_handle.....	303

srv_init.....	306
srv_langcpy .....	308
srv_langlen .....	310
srv_lockmutex .....	312
srv_log.....	315
srv_mask.....	317
srv_msg.....	319
srv_negotiate.....	322
srv_numparams .....	329
srv_options.....	331
srv_orderby .....	337
srv_poll (UNIX only) .....	339
srv_props .....	342
srv_putmsgq.....	348
srv_realloc.....	351
srv_recvpass thru .....	352
srv_regcreate .....	354
srv_regdefine .....	356
srv_regdrop .....	360
srv_regexec.....	362
srv_reginit.....	364
srv_reglist.....	366
srv_reglistfree.....	368
srv_regnowatch .....	369
srv_regparam .....	371
srv_regwatch .....	374
srv_regwatchlist .....	377
srv_rpcdb .....	379
srv_rpcname .....	380
srv_rpcnumber .....	383
srv_rpcoptions.....	384
srv_rpcowner.....	386
srv_run .....	388
srv_s_ssl_local_id.....	389
srv_select (UNIX only) .....	389
srv_send_ctlinfo .....	393
srv_send_data.....	394
srv_send_text.....	398
srv_senddone.....	401
srv_sendinfo.....	406
srv_sendpassthru .....	409
srv_sendstatus .....	412
srv_setcolotype .....	413
srv_setcontrol.....	415

srv_setloginfo .....	417
srv_setpri.....	419
srv_signal (UNIX only) .....	421
srv_sleep.....	424
srv_spawn .....	427
srv_symbol .....	430
srv_tabcolname.....	434
srv_tabname .....	437
srv_termproc .....	438
srv_text_info.....	440
srv_thread_props .....	443
srv_timedsleep .....	448
srv_ucwakeup .....	449
srv_unlockmutex .....	451
srv_version.....	452
srv_wakeup .....	454
srv_xferdata .....	456
srv_yield .....	458

**CHAPTER 4                   System Registered Procedures..... 461**

sp_ps.....	461
sp_regcreate .....	464
sp_regdrop .....	471
sp_reglist.....	472
sp_regnowatch.....	473
sp_regwatch.....	473
sp_regwatchlist .....	475
sp_serverinfo.....	475
sp_terminate .....	476
sp_who.....	478

**APPENDIX A                Open Server Error Messages..... 481**

Error 16000 .....	481
Error 16001 .....	481
Error 16002 (Deprecated) .....	482
Error 16003 .....	482
Error 16004 .....	482
Error 16005 .....	483
Error 16006 (Deprecated) .....	483
Error 16007 (Deprecated) .....	483
Error 16008 .....	483
Error 16009 .....	484
Error 16010 .....	484

Error 16011 (Deprecated) .....	485
Error 16012 .....	485
Error 16013 .....	485
Error 16014 (Deprecated) .....	486
Error 16015 (Deprecated) .....	486
Error 16016 .....	486
Error 16017 (Deprecated) .....	486
Error 16018 (Deprecated) .....	487
Error 16019 .....	487
Error 16020 (Deprecated) .....	487
Error 16021 (Deprecated) .....	488
Error 16022 (Deprecated) .....	488
Error 16023 (Deprecated) .....	488
Error 16024 (Deprecated) .....	489
Error 16025 (Deprecated) .....	489
Error 16026 (Deprecated) .....	489
Error 16027 (Deprecated) .....	490
Error 16028 (Deprecated) .....	490
Error 16029 .....	490
Error 16030 (Deprecated) .....	490
Error 16031 (Deprecated) .....	491
Error 16032 .....	491
Error 16033 .....	491
Error 16034 .....	492
Error 16035 (Deprecated) .....	492
Error 16036 (Deprecated) .....	492
Error 16037 (Deprecated) .....	493
Error 16038 (Deprecated) .....	493
Error 16039 (Deprecated) .....	493
Error 16040 (Deprecated) .....	494
Error 16041 (Deprecated) .....	494
Error 16042 (Deprecated) .....	494
Error 16043 .....	494
Error 16044 .....	495
Error 16045 .....	495
Error 16046 .....	495
Error 16047 .....	496
Error 16048 .....	496
Error 16049 .....	497
Error 16050 .....	497
Error 16051 (Deprecated) .....	497
Error 16052 .....	498
Error 16053 .....	498
Error 16054 .....	498

Error 16055 .....	498
Error 16056 .....	499
Error 16057 .....	499
Error 16058 .....	499
Error 16059 (Deprecated) .....	500
Error 16060 (Deprecated) .....	500
Error 16061 .....	500
Error 16062 .....	501
Error 16063 .....	501
Error 16064 (Deprecated) .....	501
Error 16065 .....	502
Error 16066 (Deprecated) .....	502
Error 16067 .....	502
Error 16068 (Deprecated) .....	503
Error 16069 (Deprecated) .....	503
Error 16070 (Deprecated) .....	503
Error 16071 (Deprecated) .....	504
Error 16072 .....	504
Error 16073 .....	504
Error 16074 .....	505
Error 16075 .....	505
Error 16076 .....	505
Error 16077 (Deprecated) .....	506
Error 16078 (Deprecated) .....	506
Error 16079 (Deprecated) .....	506
Error 16080 .....	506
Error 16081 (Deprecated) .....	507
Error 16082 (Deprecated) .....	507
Error 16083 (Deprecated) .....	508
Error 16084 (Deprecated) .....	508
Error 16085 (Deprecated) .....	508
Error 16086 (Deprecated) .....	508
Error 16087 (Deprecated) .....	509
Error 16088 (Deprecated) .....	509
Error 16089 (Deprecated) .....	509
Error 16090 (Deprecated) .....	510
Error 16091 (Deprecated) .....	510
Error 16092 (Deprecated) .....	510
Error 16093 (Deprecated) .....	511
Error 16094 .....	511
Error 16095 (Deprecated) .....	511
Error 16096 (Deprecated) .....	512
Error 16097 .....	512
Error 16098 (Deprecated) .....	512

Error 16099 (Deprecated) .....	513
Error 16100 .....	513
Error 16101 (Deprecated) .....	513
Error 16102 (Deprecated) .....	514
Error 16103 .....	514
Error 16104 .....	514
Error 16105 .....	515
Error 16106 .....	515
Error 16107 (Deprecated) .....	515
Error 16108 .....	516
Error 16109 .....	516
Error 16110 .....	516
Error 16111 .....	517
Error 16112 .....	517
Error 16113 .....	517
Error 16114 (Deprecated) .....	518
Error 16115 .....	518
Error 16116 (Deprecated) .....	518
Error 16117(Deprecated) .....	519
Error 16118 .....	519
Error 16119 .....	519
Error 16120 .....	520
Error 16121 .....	520
Error 16122 .....	520
Error 16123 .....	521
Error 16124 .....	521
Error 16125 .....	521
Error 16126 .....	522
Error 16127 .....	522
Error 16128 .....	522
Error 16129 .....	523
Error 16130 .....	523
Error 16131 .....	523
Error 16132 .....	524
Error 16133 .....	524
Error 16134 .....	525
Error 16135 .....	525
Error 16136 .....	525
Error 16137 .....	526
Error 16138 .....	526
Error 16139 (Deprecated) .....	526
Error 16140 .....	527
Error 16141 .....	527
Error 16142 .....	527



Error 16143 .....	528
Error 16144 .....	528
Error 16145 .....	529
Error 16146 (Deprecated) .....	529
Error 16147 .....	529
Error 16148 .....	530
Error 16149 .....	530
Error 16150 .....	530
Error 16151 .....	531
Error 16152 (Deprecated) .....	531
Error 16153 (Deprecated) .....	532
Error 16154 (Deprecated) .....	532
Error 16155 (Deprecated) .....	532
Error 16156 .....	532
Error 16157 .....	533
Error 16158 .....	533
Error 16159 .....	533
Error 16160 .....	534
Error 16161 (Deprecated) .....	534
Error 16162 .....	534
Error 16163 .....	535
Error 16164 (Deprecated) .....	535
Error 16165 .....	535
Error 16166 .....	535
Error 16167 .....	536
Error 16168 .....	536
Error 16169 .....	536
Error 16170 .....	537
Error 16171 .....	537
Error 16172 .....	537
Error 16173 (Deprecated) .....	538
Error 16174 .....	538
Error 16175 .....	538
Error 16176 .....	539
Error 16177 .....	539
Error 16178 .....	539
Error 16179 .....	540
Error 16180 .....	540
Error 16181 .....	540
Error 16182 .....	540
Error 16183 (Reserved) .....	541
Error 16184 (Deprecated) .....	541
Error 16185 (Deprecated) .....	541
Error 16186 (Deprecated) .....	542

Error 16187 (Deprecated) .....	542
Error 16188 .....	542
Error 16189 (Deprecated) .....	543
Error 16190 (Deprecated) .....	543
Error 16191 (Deprecated) .....	543
Error 16192 .....	543
Error 16193 (Deprecated) .....	544
Error 16194 (Deprecated) .....	544
Error 16195 .....	545
Error 16196 (Deprecated) .....	545
Error 16197 (Deprecated) .....	545
Error 16198 .....	545
Error 16199 (Deprecated) .....	546
Error 16200 .....	546
Error 16201 .....	547
Error 16202 .....	547
Error 16203 .....	547
Error 16204 .....	548
Error 16205 (Deprecated) .....	548
Error 16206 (Deprecated) .....	548
Error 16207 .....	549
Error 16208 .....	549
Error 16209 (Deprecated) .....	549
Error 16210 .....	550
Error 16211 (Deprecated) .....	550
Error 16212 .....	550
Error 16213 .....	551
Error 16214 (Deprecated) .....	551
Error 16215 (Deprecated) .....	551
Error 16216 .....	552
Error 16217 .....	552
Error 16218 .....	552
Error 16219 .....	553
Error 16220 .....	553
Error 16221 .....	553
Error 16222 .....	554
Error 16223 .....	554
Error 16224 (Deprecated) .....	554
Error 16225 (Deprecated) .....	555
Error 16226 (Deprecated) .....	555
Error 16227 .....	555
Error 16228 (Deprecated) .....	556
Error 16229 .....	556
Error 16230 .....	556

Error 16231 .....	557
Error 16232 .....	557
Error 16233 .....	557
Error 16234 .....	558
Error 16235 .....	558
Error 16236 .....	558
Error 16237 .....	559
Error 16238 .....	559
Error 16239 (Deprecated) .....	559
Error 16240 .....	560
Error 16241 .....	560
Error 16242 (Deprecated) .....	560
Error 16243 (Deprecated) .....	561
Error 16244 .....	561
Error 16245 (Deprecated) .....	561
Error 16246 (Deprecated) .....	562
Error 16247 .....	562
Error 16248 .....	562
Error 16249 (Deprecated) .....	563
Error 16250 (Deprecated) .....	563
Error 16251 .....	563
Error 16252 .....	563
Error 16253 .....	564
Error 16254 .....	564
Error 16255 .....	565
Error 16256 .....	565
Error 16257 .....	565
Error 16258 (Deprecated) .....	566
Error 16259 (Deprecated) .....	566
Error 16260 .....	566
Error 16261 .....	566
Error 16262 .....	567
Error 16263 .....	567
Error 16264 (Deprecated) .....	567
Error 16265 .....	568
Error 16266 .....	568
Error 16267 .....	568
Error 16268 (Deprecated) .....	569
Error 16269 .....	569
Error 16270 (Deprecated) .....	570
Error 16271 .....	570
Error 16272 .....	570
Error 16273 .....	570
Error 16274 .....	571

Error 16275 (Deprecated) .....	571
Error 16276 .....	571
Error 16277 (Deprecated) .....	572
Error 16278 .....	572
Error 16279 .....	572
Error 16280 .....	573
Error 16281 .....	573
Error 16282 .....	573
Error 16283 .....	574
Error 16284 .....	574
Error 16285 .....	574
Error 16286 .....	575
Error 16287 (Deprecated) .....	575
Error 16288 (Deprecated) .....	576
Error 16289 .....	576
Error 16290 (Deprecated) .....	576
Error 16291 .....	577
Error 16292 (Deprecated) .....	577
Error 16293 .....	577
Error 16294 (Deprecated) .....	578
Error 16295 .....	578
Error 16296 .....	578
Error 16297 .....	579
Error 16298 .....	579
Error 16299 (Deprecated) .....	579
Error 16300 .....	580
Error 16301 .....	580
Error 16302 .....	580
Error 16303 .....	581
Error 16304 .....	581
Error 16305 .....	581
Error 16306 .....	582
Error 16307 (Deprecated) .....	582
Error 16308 (Deprecated) .....	582
Error 16309 .....	583
Error 16310 .....	583
Error 16311 (Deprecated) .....	584
Error 16312 (Deprecated) .....	584
Error 16313 (Deprecated) .....	584
Error 16314 (Deprecated) .....	584
Error 16315 .....	585
Error 16316 (Deprecated) .....	585
Error 16317 .....	586
Error 16318 (Deprecated) .....	586

Error 16319 .....	586
Error 16320 .....	587
Error 16321 .....	587
Error 16322 .....	587
Error 16323 .....	588
Error 16324 .....	588
Error 16325 .....	589
Error 16326 .....	589
Error 16327 .....	589
Error 16328 .....	590
Error 16329 .....	590
Error 16330 .....	591
Error 16331 .....	591
Error 16332 .....	591
Error 16333 .....	592
Error 16334 .....	592
Error 16335 .....	592
Error 16336 .....	593
Error 16337 .....	593
Error 16338 .....	594
Error 16339 .....	594
Error 16340 .....	594
Error 16341 (Deprecated) .....	595
Error 16342 (Deprecated) .....	595
Error 16343 (Deprecated) .....	595
Error 16344 (Deprecated) .....	595
Error 16345 (Deprecated) .....	596
Error 16346 (Deprecated) .....	596
Error 16347 (Deprecated) .....	596
Error 16348 (Deprecated) .....	597
Error 16349 (Deprecated) .....	597
Error 16350 (Deprecated) .....	597
Error 16351 (Deprecated) .....	597
Error 16352 (Deprecated) .....	598
Error 16353 (Deprecated) .....	598
Error 16354 (Deprecated) .....	598
Error 16355 (Deprecated) .....	599
Error 16356 (Deprecated) .....	599
Error 16357 (Deprecated) .....	599
Error 16358 (Deprecated) .....	599
Error 16359 (Deprecated) .....	600
Error 16360 .....	600
Error 16361 .....	600
Error 16362 .....	601

Error 16363 ..... 601  
Error 16364 ..... 601  
Error 16365 ..... 602  
Error 16366 (Deprecated) ..... 602  
Error 16367 ..... 602  
Error 16368 ..... 603  
Error 16369 ..... 603  
Error 16370 (Deprecated) ..... 604  
Error 16371 ..... 604  
Error 16372 ..... 604  
Error 16373 ..... 605  
Error 16374 ..... 605  
Error 16375 ..... 605  
Error 16376 (Deprecated) ..... 606  
Error 16377 (Deprecated) ..... 606  
Error 16378 ..... 606  
Error 16379 ..... 606  
Error 16380 ..... 607  
Error 16381 ..... 608  
Error 16382 ..... 608  
Error 16383 ..... 608  
Error 16384 (Deprecated) ..... 609  
Error 16385 ..... 609  
Error 16386 ..... 609  
Error 16387 (Deprecated) ..... 610  
Error 16388 ..... 610  
Error 16389 ..... 610  
Error 16390 ..... 611  
Error 16391 ..... 611  
Error 16392 ..... 612  
Error 16393 ..... 612  
Error 16394 ..... 612  
Error 16395 ..... 613  
Error 16396 ..... 613  
Error 16397 ..... 614  
Error 16398 ..... 614

**Glossary ..... 615**

**Index ..... 623**

# About This Book

This book 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

This book contains these chapters:

- 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.
- Appendix A, “Open Server Error Messages,” lists and describes Open Server error messages.

## Related documents

You can see these books for more information:

- *The Open Server and SDK New Features for 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 Server Release Bulletin* for your platform contains important last-minute information about Open Server.

- 
- The *Software Developer's Kit Release Bulletin* for your platform contains important last-minute information about Open Client™ and SDK.
  - The *jConnect™ for JDBC™ Release Bulletin* contains important last-minute information about jConnect.
  - The *Open Client and Open Server Configuration Guide* for your platform contains information about configuring your system to run Open Client and Open Server.
  - The *Open Client Client-Library/C Programmers Guide* contains information on how to design and implement Client-Library applications.
  - The *Open Client Client-Library/C Reference Manual* contains reference information for Open Client Client-Library™.
  - The *Open Client and Open Server Common Libraries Reference Manual* contains reference information for CS-Library, which is a collection of utility routines that are useful in both Client-Library and Server-Library applications.
  - The *Open Server DB-Library/C Reference Manual* contains reference information for the C version of Open Client DB-Library™.
  - The *Open Client and Open Server Programmers Supplement* for your platform contains platform-specific information for programmers using Open Client and Open Server. This document includes information about:
    - Compiling and linking an application
    - The sample programs that are included with Open Client and Open Server
    - Routines that have platform-specific behaviors
  - The *Installation and Release Bulletin Sybase® SDK DB-Library Kerberos Authentication Option* contains information about installing and enabling the MIT Kerberos security mechanism to be used on DB-Library. DB-Library only supports network authentication and mutual authentication in the Kerberos security mechanism.
  - The *Open Client Client-Library Migration Guide* contains information on how to migrate Open Client™ DB-Library™ applications to Open Client Client-Library
  - The *Open Client and Open Server International Developers Guide* provides information about creating internationalized and localized applications.



- The *Open Client Embedded SQL™/C Programmers Guide* explains how to use Embedded SQL and the Embedded SQL precompiler with C applications.
- The *Open Client Embedded SQL™/COBOL Programmers Guide* explains how to use Embedded SQL and the Embedded SQL precompiler with COBOL applications.
- The *jConnect for JDBC Programmers Reference* describes the jConnect for JDBC product and explains how to access data stored in relational database management systems.
- The *Adaptive Server® Enterprise ADO.NET Data Provider Users Guide* provides information on how to access data in Adaptive Server using any language supported by .NET, such as C#, Visual Basic .NET, C++ with managed extension, and J#.
- The *Adaptive Server Enterprise ODBC Driver by Sybase Users Guide* for Microsoft Windows and UNIX, provides information on how to access data from Adaptive Server on Microsoft Windows and UNIX platforms, using the Open Database Connectivity (ODBC) Driver.
- The *Adaptive Server Enterprise OLE DB Provider by Sybase Users Guide for Microsoft Windows* provides information on how to access data from Adaptive Server on Microsoft Windows platforms, using the Adaptive Server OLE DB Provider.
- The *Adaptive Server Enterprise Database Driver for Perl Programmers Guide* provides information for Perl developers to connect to an Adaptive Server database and query or change information using a Perl script.
- The *Adaptive Server Enterprise extension module for PHP Programmers Guide* provides information for PHP developers to execute queries against an Adaptive Server database.
- The *Adaptive Server Enterprise extension module for Python Programmers Guide* provides information about Sybase-specific Python interface that can be used to execute queries against an Adaptive Server database.

**Other sources of information**

Use the Sybase Product Documentation Web site to learn more about your product:

- The Sybase Product Documentation Web site is accessible using a standard Web browser. In addition to product documentation, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

---

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

## **Sybase certifications on the Web**

Technical documentation at the Sybase Web site is updated frequently.

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

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

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

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

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

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

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

## **Sybase EBFs and software maintenance**

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

- 1 Point your Web browser to the Sybase Support Page at <http://www.sybase.com/support>.
- 2 Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
- 3 Select a product.

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

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

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

## Conventions

**Table 1: Syntax conventions**

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

## Accessibility features

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

---

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

---

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

---

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

**If you need help**

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

# Introducing Open Server

Topic	Page
Client/Server overview	1
Types of clients	2
Types of servers	2
Open Server configurations	3
Open Server	5
Using Open Server	7
Basic Open Server program	8
Open Server events	13
Registered procedures	14
Returning results to clients	15
Error handling	16
Multithread programming	17

## 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 100 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 131 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*
- *libsybcomm.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 141.

## 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 <b>properties</b> .	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 <code>srv_sendinfo</code> routine on an ad hoc basis to send error messages to the client, or <code>srv_log</code> to write to the log file. See “Errors” on page 90 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.	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.  
**
```

```
**      opublic.h: Public Open Server structures, typedefs,
**      defines, and function prototypes.
**
**      oserver.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      <opublic.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_155, &cp) != CS_SUCCEEDED)
{
    (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_155) != CS_SUCCEEDED)
{
    /*
    ** 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_SUCCEEDED)
{
    /*
```

```
    ** 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

See “Events” on page 93.

## 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 164 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 136 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 204. 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 90 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 111 for details.



# 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:

<b>Topic</b>	<b>Page</b>
Attention events	20
Browse mode	22
Capabilities	24
Client command errors	38
Connection migration	40
CS_BROWSEDESC structure	52
CS_DATAFMT structure	54
CS_IODESC structure	57
CS-Library	59
CS_SERVERMSG structure	60
Cursors	63
Scrollable cursors	77
Data stream messages	80
Directory services	81
Dynamic SQL	83
Dynamic listeners	88
Errors	90
Events	93
Gateway applications	100
International support	101
Language calls	109

<b>Topic</b>	<b>Page</b>
Login redirection and extended HA failover support	110
Messages	111
Multithread programming	111
Negotiated behavior	121
Options	124
Parameters	129
Partial update	130
Passthrough mode	131
Processing parameter and row data	136
Properties	141
Registered procedures	164
Remote procedure calls	171
Retrieving large object parameters	172
Security services	174
Text and image	201
Types	204

## 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_cancel` 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_ATTREASON`.

## 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. 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 `origname` and `origlen` 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`.

---

See the *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**

<b>CS_REQUEST capability</b>	<b>Meaning</b>	<b>Capability relates to</b>
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_BIGDATETIME	bigdatetime datatype	Datatypes
CS_DATA_BIGTIME	bigtime datatype	Datatypes
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

<b>CS_REQUEST capability</b>	<b>Meaning</b>	<b>Capability relates to</b>
CS_DATA_INT8	Big integer datatype	Datatypes
CS_DATA_INTN	Null integers	Datatypes
CS_DATA_MNY4	Short money datatype	Datatypes
CS_DATA_MNY8	Money datatype	Datatypes
CS_DATA_MONEYN	Null money values	Datatypes
CS_DATA_NUM	Numeric datatype	Datatypes
CS_DATA_SENSITIVITY	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_SRPKTSIZE	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**

<b>CS_RESPONSE capability</b>	<b>Meaning</b>	<b>Capability relates to</b>
CS_CON_NOINBAND	No in-band (non-expedited) attentions	Connections
CS_CON_NOOOB	No out-of-band (expedited) attentions	Connections
CS_DATA_NOBIGDATETIME	No bigdatetime datatype	Datatypes
CS_DATA_NOBIGTIME	No bigtime datatype	Datatypes
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

<b>CS_RESPONSE capability</b>	<b>Meaning</b>	<b>Capability relates to</b>
CS_DATA_NOTIME	No time datatype	Datatype
CS_DATA_NOUCHAR	No 2-byte character datatype	Datatypes
CS_DATA_NOUNITEXT	No Unitext datatype	Datatypes
CS_DATA_NOXML	No XML datatype	Datatypes
CS_NO_SRVPKTSIZE	No server-specified packetsize	Connections
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_RES_NOXNLMETADATA	No table metadata.	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 342.

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**

<b>CS_REQUEST capability</b>	<b>4.0</b>	<b>4.0.2</b>	<b>4.2</b>	<b>4.6</b>	<b>5.0</b>
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_BIGDATETIME	0	0	0	0	1
CS_DATA_BIGTIME	0	0	0	0	1
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

<b>CS_REQUEST capability</b>	<b>4.0</b>	<b>4.0.2</b>	<b>4.2</b>	<b>4.6</b>	<b>5.0</b>
CS_DATA_SENSITIVITY	0	0	0	0	0
CS_DATA_TEXT	1	1	1	1	1
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_SRPKTSIZE	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**

<b>CS_RESPONSE capability</b>	<b>4.0</b>	<b>4.0.2</b>	<b>4.2</b>	<b>4.6</b>	<b>5.0</b>
CS_CON_NOINBAND	1	1	1	1	1
CS_CON_NOOOB	0	0	0	0	0
CS_DATA_NOBIGDATETIME	1	1	1	1	0
CS_DATA_NOBIGTIME	1	1	1	1	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

<b>CS_RESPONSE capability</b>	<b>4.0</b>	<b>4.0.2</b>	<b>4.2</b>	<b>4.6</b>	<b>5.0</b>
CS_RES_NOPARAM	1	1	1	1	1
CS_RES_NOTDSDEBUG	1	1	1	1	1
CS_RES_NOXNLMETADATA	1	1	1	1	0
CS_NO_LARGEIDENT	1	1	1	1	0
CS_NO_SRPKTSIZE	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. 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
<code>CS_TST_CAPMASK</code>	Test to see whether a specific capability is set to <code>CS_TRUE</code> or <code>CS_FALSE</code>
<code>CS_SET_CAPMASK</code>	Set a specific capability to <code>CS_TRUE</code>
<code>CS_CLR_CAPMASK</code>	Set a specific capability to <code>CS_FALSE</code> .

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.

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 Enterprise 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. 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.

Return value	Description
CS_FAIL	<p>The migration request failed due to one of the following reasons:</p> <ul style="list-style-type: none"> <li>• The Open Server thread does not support connection migration.</li> <li>• An earlier migration request was sent and the client has started migrating to the new server.</li> </ul>

### *Canceling 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_SUCCEEDED)
{
    ...
}
```

`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_SUCCEEDED	The migration request was cancelled successfully.
CS_FAIL	The migration cancellation failed due to one of these reasons: <ul style="list-style-type: none"> <li>• There is no migration in progress.</li> <li>• The client has started migrating to the new server.</li> </ul>

---

**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_SUCCEEDED;
}
...

```



```

/*
** 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: <ul style="list-style-type: none"> <li>• Make the context available for the other servers.</li> <li>• Cancel the migration if the application decides that migration is no longer needed.</li> <li>• Request another migration if a new migration target has been selected.</li> </ul>
SRV_MIG_FAILED	The client has sent a message to the server indicating that the migration failed.	One of the following: <ul style="list-style-type: none"> <li>• Access the client context and continue serving the connection.</li> <li>• Request another migration.</li> </ul>

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
<code>srv_thread_props()</code> : Property - <code>SRV_T_SESSIONID</code> is not available	You try to retrieve a session ID that the client has not yet received.
<code>srv_send_ctlinfo(SRV_CTL_MIGRATE)</code> : Connection cannot migrate	The client does not support migration.
<code>srv_send_ctlinfo(SRV_CTL_MIGRATE)</code> : Migration can no longer be cancelled	You requested for a cancellation of a migration that has already started.
Migration failed but no <code>SRV_MIGRATE_STATE</code> handler was installed	The default <code>SRV_MIGRATE_STATE</code> 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         origlen;
    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.
- origlen 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`.
- `datatype` is the datatype of the data, which is one of the Open Server datatypes listed in “Types” on page 204.

---

**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 Enterprise command creates the Adaptive Server Enterprise 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 <i>timestamp</i> 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 Enterprise 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_txtlen` 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 Enterprise id.*/
    CS_INT     timestamplen;         /* Length of timestamp.*/
    CS_BYTE    textptr[CS_TP_SIZE];  /* Adaptive Server Enterprise 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 141 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    svrnlenn;

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

    /*
    ** Other information.
    */
    CS_INT    status;
    CS_BYTE   sqlstate[CS_SQLSTATE_SIZE];
    CS_INT    sqlstatelenn;
} 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.
- svrnlenn is the length, in bytes, of svrname.
- proc is the name of the **stored procedure** (if any) that caused the message.
- proclenn 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”.

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

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. 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.

See `srv_cursor_props` on page 259.

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	<p>fetchtype is described when a CS_CURSOR_FETCH command is received from the client. The valid fetch types and their meanings are as follows:</p> <ul style="list-style-type: none"> <li>• CS_NEXT – next row</li> <li>• CS_PREV – previous row</li> <li>• CS_FIRST – first row</li> <li>• CS_LAST – last row</li> <li>• CS_ABSOLUTE – row identified in the rowoffset field</li> <li>• CS_RELATIVE – current row plus or minus value in the rowoffset field.</li> </ul> <p>Requests to an Adaptive Server Enterprise will always have a fetchtype of CS_NEXT.</p>
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.



Field name	Description	Notes
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)**

<b>Value</b>	<b>Meaning</b>	<b>Legal values for cmdoptions</b>
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 <code>srv_langlen</code> and <code>srv_langcpy</code> .	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.

<b>Value</b>	<b>Meaning</b>	<b>Legal values for cmdoptions</b>
CS_CURSOR_INFO	<p>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.</p>	<p>SRV_CUR_SETROWS when the client describes the current row fetch count. The fetchcnt field contains the requested fetch count.</p> <p>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.</p> <p>SRV_CUR_INFORMSTAT US when the Open Server application responds to a CS_CURSOR_INFO command. The curstatus field contains the cursor status.</p>
CS_CURSOR_OPEN	Cursor open command.	SRV_CUR_HASARGS or SRV_CUR_UNUSED.
CS_CURSOR_UPDATE	<p>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.</p>	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 ( $\rightarrow$ ) indicates that `cmd` is set to `CS_GET`—the Open Server application retrieves information from the client. The backward arrow ( $\leftarrow$ ) 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**

<b>Client action</b>	<b>Open Server application response</b>
<p>Declares a cursor (curcmd field of SRV_CURDESC contains CS_CURSOR_DECLARE)</p>	<ul style="list-style-type: none"> <li>-&gt;Retrieve curcmd value from SRV_CURDESC (srv_cursor_props)</li> <li>-&gt;Retrieve number of cursor parameters, if any (srv_numparams)</li> <li>-&gt;Retrieve format of cursor parameters, if any (srv_descfmt with type argument set to SRV_CURDATA)</li> <li>-&gt;Retrieve update column information, if any (srv_descfmt with type argument set to SRV_UPCOLDATA)</li> <li>-&gt;Retrieve actual text of cursor command (srv_langlen and srv_langcpy)</li> <li>&lt;- Set cursor ID. Set curcmd field to CS_CURSOR_INFO and curid field to unique cursor ID (srv_cursor_props)</li> <li>&lt;- Send a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)</li> </ul>
<p>Requests the status of the current cursor or sends a fetch count (curcmd field of SRV_CURDESC contains CS_CURSOR_INFO)</p>	<ul style="list-style-type: none"> <li>-&gt;Retrieve curcmd and curid cmdoptions values from SRV_CURDESC structure (srv_cursor_props)</li> <li>&lt;- 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)</li> <li>&lt;- 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)</li> <li>&lt;- Send a DONE packet (srv_senddone with status argument set to SRV_DONE_FINAL)</li> </ul>

<b>Client action</b>	<b>Open Server application response</b>
<p>Opens a cursor (curcmd field of SRV_CURDESC contains CS_CURSOR_OPEN)</p>	<ul style="list-style-type: none"> <li>- &gt; Retrieve curcmd and curid values from SRV_CURDESC structure (srv_cursor_props)</li> <li>- &gt; Retrieve number of cursor parameters, if any (srv_numparams)</li> <li>- &gt; Retrieve format of cursor parameters and actual parameters, if any (srv_descfmt, srv_bind, srv_xferdata with type argument set to SRV_CURDATA)</li> <li>&lt; - Send cursor status. Set curid to current cursor ID and curcmd to CS_CURSOR_INFO (srv_cursor_props)</li> <li>&lt; - Describe result row formats (srv_descfmt with type argument set to SRV_ROWDATA)</li> <li>&lt; - Send a DONE packet (srv_senddone with status argument set to SRV_DONE_FINAL)</li> </ul>
<p>Fetches rows (curcmd field of SRV_CURDESC contains CS_CURSOR_FETCH)</p>	<ul style="list-style-type: none"> <li>- &gt; Retrieve curcmd and curid values from SRV_CURDESC structure (srv_cursor_props)</li> <li>&lt; - Send result rows, fetchcnt times (srv_bind, srv_xferdata with type argument set to SRV_ROWDATA)</li> <li>&lt; - Send a DONE packet (srv_senddone with status argument set to SRV_DONE_FINAL)</li> </ul>

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

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.

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\_CURDESC 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
<code>currow_pos</code>	Current row position of a cursor.
<code>curtotalrowcount</code>	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)**

<b>Value</b>	<b>Meaning</b>
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.
CS_CUR_RELOCKS _ONCLOSE	The cursor was closed and the shared read locks are released.

### 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.  <b>Note</b> 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 version 15.0 and later applications that support scrollable cursors on Open Server, use the `SRV_CURDESC2` structure, and set the application to `CS_VERSION_xxx`, where `xxx` corresponds to the Open Server version.

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.

See `srv_msg` on page 319.

- 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. See the “Processing parameter and row data” on page 136.

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.

See `srv_msg` on page 319.

- 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*.

## 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. See the *Open Client and Open Server Configuration Guide* for each platform. See `srv_props` on page 342 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 `sv_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.

See the *Embedded SQL/C Programmers Guide*.

## 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. See `srv_dynamic` on page 275. 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 ( $\rightarrow$ ) indicates that `cmd` is set to `CS_GET`—the Open Server application retrieves information from the client. The backward arrow ( $\leftarrow$ ) 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**

<b>Client action</b>	<b>Open Server application response</b>
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
<p>Requests a description of the statement's output parameters (Operation type is CS_DESCRIBE_OUTPUT)</p>	<p>→ Retrieves the operation type. (srv_dynamic)</p> <p>→ Retrieves the statement ID length. (srv_dynamic)</p> <p>→ Retrieves the statement ID. (srv_dynamic)</p> <p>← Acknowledges the client command. (srv_dynamic)</p> <p>← Sends the statement ID length. (srv_dynamic)</p> <p>← Sends the statement ID. (srv_dynamic)</p> <p>← 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)</p> <p>← Sends a DONE packet. (srv_senddone with status argument set to SRV_DONE_FINAL)</p>

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

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

## Example

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

## Dynamic listeners

Dynamic listeners allow an Open Server application to create a new listener after a call has been made to the *srv\_run* routine. Consequently, an Open Server application can be started with a minimal number of entries in an interface file, and an Open Server application can restart listeners.

## Configuration

The *SRV\_LISTENER\_PREBIND* Open Server event is used to further configure a listener identified by the specified *SRV\_PROC* control structure. For example, an alternate SSL certificate can be specified. This configuration is accomplished using thread properties.

The `SRV_LISTENER_POSTBIND` Open Server event allows the final configuration of a dynamic listener to be identified using the specified `SRV_PROC` control structure. For example, the address to which the listener was bound can be determined using thread properties.

## Properties

Two Open Server properties are used in support of dynamic listeners.

- The `SRV_S_NUMLISTENERS` property returns the number of `SRV_PROC` control structures involved in listening for client connections.
- The `SRV_S_MAXLISTENERS` property is used to limit the maximum number of listener threads.

## Thread type

The `SRV_TLISTENER` client thread type is used for dynamic listeners.

## Thread properties

Three thread properties are used expressly for dynamic listener support:

- The `SRV_T_LISTENADDR` property returns the address of the listener identified by the specified `SRV_PROC` control structure. If the `SRV_PROC` control structure is a listener, this property returns the address on which the listener accepts connections.
- The `SRV_T_LOCALID` property specifies the SSL certificate to use for a listener.
- The `SRV_T_REMOTEADDR` property returns the address of a `SRV_PROC` peer.

## Starting a dynamic listener

A dynamic listener can be started by calling the `srv_spawn` routine with the following parameter values:

- *sppp* – The pointer to the thread structure pointer should be null.
- *stacksize* – The stack size should be specified as `CS_UNUSED`.

- *funcp* – The entry-point function pointer should be `SRV_C_START_LISTENER`.
- *argp* – This parameter should point to a `CS_TRANADDR` structure.
- *priority* – The priority should be specified as `CS_UNUSED`.

## 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. See `srv_props` on page 342.

To troubleshoot error messages, see Appendix A, “Open Server Error Messages.”

## 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 (10)	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 <code>srv_xferdata</code> to send a row before describing all the columns with <code>srv_descfmt</code> generates a <code>SRV_INFO</code> error.	Open Server application error Client command error

Severity	Meaning	Applicable error type
SRV_FATAL_PROCESS (15)	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 <code>srv_senddone</code> . An error of this severity causes Open Server to queue a <code>SRV_DISCONNECT</code> 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 (20)	A fatal server error. Open Server has detected an internal error from which it cannot recover. This causes Open Server to queue a <code>SRV_STOP</code> event for the Open Server application, which causes <code>srv_run</code> to return <code>CS_FAIL</code> .	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. For troubleshooting error messages, see Appendix A, “Open Server Error Messages.”

## 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 94, and "Programmer-defined events" on page 99.

### 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. See `srv_handle` on page 303.

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**

<b>Event</b>	<b>Description</b>	<b>Argument to handler</b>	<b>Default event handler</b>
SRV_ATTENTION	An attention has been received. This event usually occurs when a client calls <code>ct_cancel</code> to stop results processing prematurely. <code>SRV_ATTENTION</code> is an immediate event; Open Server services it as soon as it occurs rather than adding it to the client's event queue. A <code>SRV_ATTENTION</code> 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 <code>DONE ERROR</code> to the client.
SRV_CONNECT	A Client-Library client has called <code>ct_connect</code> .	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 <code>SRV_CURSOR</code> handler installed" to the client. Open Server returns <code>DONE ERROR</code> to the client.

Event	Description	Argument to handler	Default event handler
SRV_DISCONNECT	<p>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 <code>srv_event</code> made from within the application explicitly to disconnect a client.</p> <p>Client-Library programs call <code>ct_close</code> or <code>ct_exit</code> to log out of the Open Server application. Remote Adaptive Server Enterprise connections terminate when the remote procedure call has completed.</p>	SRV_PROC*	The default handler takes no action.
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	<p>A network read for the connection has completed.</p> <p>(The <code>SRV_T_FULLPASSTHRU</code> property for the thread must have been set to <code>CS_TRUE</code> for this event to occur.</p>	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 <code>ct_command</code> and <code>ct_send</code> . <code>isql</code> 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.

<b>Event</b>	<b>Description</b>	<b>Argument to handler</b>	<b>Default event handler</b>
SRV_LISTENER_PREBIND	Allows further configuration of the listener identified by the specified SRV_PROC control structure. For example, an alternate SSL certificate can be specified. This configuration is accomplished using thread properties.	SRV_PROC*	The default handler takes no action.
SRV_LISTENER_POSTBIND	Allows the final configuration of the listener identified by the specified SRV_PROC control structure to be determined. For example, the address to which the listener was bound can be determined. Configuration is determined using thread properties.	SRV_PROC*	The default handler takes no action.
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 Enterprise has issued a remote procedure call (RPC).	SRV_PROC*	The default handler sends the message “RPC < <i>rpcname</i> > received. No remote procedure call handler installed” to the client. Open Server returns a DONE ERROR to the client.
SRV_START	A call to <code>srv_run</code> 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 <code>srv_event</code> or by an Open Server fatal server error. The Open Server application is stopped. <code>srv_run</code> 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	<p>This event is only triggered by an Open Server application calling <code>srv_event</code>. In response to this event, Open Server calls a threads <code>SRV_DISCONNECT</code> event handler. Open Server places the event at the top of the threads event queue, so that it is processed as the next event.</p> <p>An application should raise this event if it wants to terminate a thread immediately, bypassing other events in the queue. When a <code>SRV_URGDISCONNECT</code> event is raised, the I/O channel associated with the thread is marked dead.</p>	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 Enterprise 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 Enterprise. 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 Enterprise 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 Enterprise” gateway. The gateway demonstrates how to pass data from a remote Adaptive Server Enterprise 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 Enterprise, Client-Library and DB-Library provide a set of application protocol passthrough routines that allow the Open Server to pass 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. See “Passthrough mode” on page 131.

---

**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 Developers 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 103.

## 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 104.

- 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 105.

- 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 106.

## 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 generates 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 106.

## 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 Enterprise 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 Enterprise can convert between ISO 8859-1 and CP850, because both of these character sets are in the Western European language group; however, Adaptive Server Enterprise 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 <code>cs_loc_alloc</code> .	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 <code>srv_thread_props(GET)</code> with <code>property</code> as <code>SRV_T_LOCALE</code> .	Copy the client thread's existing localization values into the new CS_LOCALE structure.	
3	Call <code>cs_locale</code> .	Overwrite the CS_LOCALE structure with the requested language or character set.	See "Localizing a CS_CONTEXT structure" on page 106.

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

See “Registered procedures” on page 164.

## 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.

See “Properties” on page 141.

## 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 393, “`srv_getserverbyname`” on page 302, and “`srv_freeserveraddrs`” on page 288.

The following properties support the routines:

- `SRV_S_HASERVER`, a read-only server property that returns the `HAFILOVER` 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. 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 111.
- Error messages – Open Server reports error conditions by means of error messages. See “Errors” on page 90.

## 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. See "Scheduling" on page 115, and "Tools and techniques" on page 117.

## 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 93 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 Enterprise 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 Enterprise 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 419 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 <code>srv_yield</code> 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 123.

### 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 174 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 188, and “Security services” on page 174.
- 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 195 and the *Open Client Client-Library/C Reference Manual*.

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 124.

“International support” on page 101 covers negotiation of national language and character set in detail.

See the *Adaptive Server Enterprise Reference Manual* and “Security services” on page 174.

## Example

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

## Options

Adaptive Server Enterprise permits clients to determine how to handle query processing. It provides a variety of configurable options that govern aspects of query processing behavior. 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 Enterprise 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 Enterprise 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**

<b>Symbolic constant</b>	<b>What the option does</b>	<b>Default value</b>
CS_OPT_ANSINULL	<p>If this option is set to CS_TRUE, Adaptive Server Enterprise enforces the ANSI behavior that “=NULL” and “is NULL” are not equivalent. In standard Transact SQL, “=NULL” and “is NULL” are treated as equivalent.</p> <p>This option affects “&lt;&gt; NULL” and “is not NULL” behavior in a similar fashion.</p>	CS_FALSE
CS_OPT_ANSIPERM	If this option is set to CS_TRUE, Adaptive Server Enterprise 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 Enterprise aborts a query when an arithmetic exception occurs during its execution.	CS_FALSE
CS_OPT_ARITHIGNORE	If this option is set, Adaptive Server Enterprise substitutes NULL for selected or updated values when an arithmetic exception occurs during query execution. Adaptive Server Enterprise does not return a warning message. If neither CS_OPT_ARITHABORT nor CS_OPT_ARITHIGNORE is set, Adaptive Server Enterprise 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	<p>If this option is set to CS_TRUE, Adaptive Server Enterprise uses chained transaction behavior.</p> <p>Chained transaction behavior means that each server command is considered to be a distinct transaction.</p> <p>Unchained transaction behavior requires an explicit commit transaction statement to define a transaction.</p>	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

Symbolic constant	What the option does	Default value
CS_OPT_CURWRITE	Sets a security label specifying the current write level.	NULL
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 Enterprise flags any nonstandard SQL commands that are sent.	CS_FALSE
CS_OPT_FORCEPLAN	If this option is set to CS_TRUE, Adaptive Server Enterprise 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 Enterprise 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 Enterprise returns information on every insert, delete, or update command. Adaptive Server Enterprise 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. See the set command in your Adaptive Server Enterprise documentation.	Not applicable
CS_OPT_IDENTITYON	Enables inserts into a table’s identity column. See the set command in your Adaptive Server Enterprise 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 Enterprise to stop sending back information about the number of rows affected by each SQL statement.	CS_FALSE

<b>Symbolic constant</b>	<b>What the option does</b>	<b>Default value</b>
CS_OPT_NOEXEC	If this option is set to CS_TRUE, Adaptive Server Enterprise 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 Enterprise treats all strings enclosed in double quotes as identifiers.	CS_FALSE
CS_OPT_RESTREES	If this option is set, Adaptive Server Enterprise 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 Enterprise returns only a maximum specified number of regular rows for <b>select</b> 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 <b>select</b> 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 Enterprise generates 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 Enterprise internal I/O statistics are returned to the client after each query.	CS_FALSE
CS_OPT_STATS_TIME	This option determines whether Adaptive Server Enterprise 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 Enterprise 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 Enterprise global variable @@ <i>textsize</i> , which limits the size of text or image values that Adaptive Server Enterprise 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 Enterprise should return.	32,768 bytes
CS_OPT_TRUNCIGNORE	If this option is set to CS_TRUE, Adaptive Server Enterprise ignores truncation errors, which is standard ANSI behavior.  If this option is set to CS_FALSE, Adaptive Server Enterprise raises an error when conversion results in truncation.	CS_FALSE

srv\_options on page 331 lists the legal values and datatype for each option.

## Example

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

## Parameters

### Batched parameters

Open Server can accept multiple sets of command parameters during execution of the command. To enable batched parameter support, set SRV\_S\_PARAM\_BATCHING property to CS\_TRUE.

Then, srv\_xferdata() has two new return codes when a command contains multiple sets of command parameters.

- CS\_PARAMS\_MORE - indicates parameters have been successfully copied and there are more parameters in the batch.
- CS\_PARAMS\_END - indicates parameters have been successfully copied. This is the last set of parameters in the batch.

## 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. See the *Open Client Client-Library /C Reference Manual*.

---

**Note** Currently, Adaptive Server Enterprise 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
<i>clienttype</i>	5	5 indicates that the client is Client-Library.
<i>mdaversion</i>	1	
<i>clientversion</i>	0	<i>clientversion</i> is an optional parameter that indicates the client version. The default is 0.

If the server supports partial updates, *sp\_mda* returns:

Parameter	Value
<i>mdinfo</i>	“UPDATETEXT”
<i>querytype</i>	2

Parameter	Value
query	<i>updatetext_syntax</i> Example: updatetext ? ? ? {NULL   ?} {NULL   ?} where “?” indicates the updatetext parameters.

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`. See “`SRV_T_BULKTYPE`” on page 159.

## 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()`.

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 Enterprise 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 Enterprise, 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.

See “Regular passthrough mode” on page 132.

- 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.

See “Event handler passthrough mode” on page 134.

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 121.

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 Enterprise—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_LOGININFO` structure and fill it with login information from the client thread.
- 3 `ct_setloginfo` – prepare a `CS_LOGININFO` 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_LOGININFO` 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_LOGININFO` structure.
- 7 `srv_setloginfo` – send the remote server’s response, retrieved in step 6, to the client, then release the `CS_LOGININFO` 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 Enterprise 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. See “Coding and installing a full passthrough event handler” on page 135.

- Enable event handler passthrough mode for a thread by setting `SRV_T_FULLPASSTHRU` to `CS_TRUE` in the Open Server connection handler. See “Enabling event handler passthrough mode for a thread” on page 135.
- Call routines to negotiate the TDS protocol level between the client and the target server. See “Negotiating the TDS protocol level” on page 136.

## 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 attention event 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_getloginf`, `ct_setloginf`, `ct_getloginf`, and `srv_setloginf` routines.

See “Negotiating the TDS protocol level in passthrough mode” on page 133.

# 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 141, 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 141, 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 `clmtp` 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 `osfmt` 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 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 142, “Server properties” on page 143 and “Thread properties” on page 151.

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.

## 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*.
- 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. 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 342 provides this information.

**Table 2-25: Server properties**

Property name	Definition	Notes
<code>SRV_S_ALLOCFUNC</code>	The address of the routine Open Server will use to allocate memory.	

Property name	Definition	Notes
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.
SRV_S_ATTREASON	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_DEFQUEUE SIZE	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 <i>Open Client and Open Server Configuration Guide</i> 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.	

Property name	Definition	Notes
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.
SRV_S_LOGFILE	The name of the log file Open Server writes to.	The SRV_S_LOGFILE property can be set after calling <code>srv_init</code> . After <code>srv_init</code> is called, setting the SRV_S_LOGFILE property with <code>bufpset</code> to an empty string ("") and <code>buflen</code> 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 <code>currentfilename_old</code> and will truncate the current log to 0 bytes.	
SRV_S_MAXLISTENERS	Limits the maximum number of listener threads.	The default value is CS_MAX_NOMAX. This value is only used when trying to create a new listener thread. Setting this property to a value less than the current value of SRV_S_NUMLISTENERS does not cause any listeners to fail.

Property name	Definition	Notes
SRV_S_MSGPOOL	The number of messages available to an Open Server application at runtime.	Open Server applications use messages through <code>srv_putmsgq</code> . A message remains in use until it is received through <code>srv_getmsgq</code> . The value of an application's <code>SRV_S_MSGPOOL</code> 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, <code>SRV_S_NETBUFSIZE</code> 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.
SRV_S_NETTRACEFILE	Net-Library tracing written to this file.	The maximum allowable length for this property is <code>SRV_MAXCHAR</code> bytes.
SRV_S_NUMCONNECTIONS	The maximum number of physical network connections the Open Server 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 <code>CS_MAX_CONNECT</code> property. <code>CS_MAX_CONNECT</code> can be set using <code>ct_config()</code> .
SRV_S_NUMLISTENERS	Returns the number of <code>SRV_PROC</code> control structures involved in listening for client connections.	This is a retrieve-only property.

Property name	Definition	Notes
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.	
SRV_S_PARAM_BATCHING	A Boolean. If CS_TRUE, Open Server can accept multiple sets of command parameters. If CS_FALSE, Open Server only accepts a single set of parameters for each command.	
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 <code>srv_capability</code> 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.

Property name	Definition	Notes
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.	<p>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 174.</p> <p>The maximum allowable length for this property is SRV_MAXCHAR bytes.</p>
SRV_S_SEC_PRINCIPAL	<p>The principal name to use when acquiring credentials for the Open Server application.</p> <p>The value of this property defaults to the Open Server application’s network name, which can be specified through srv_init.</p>	<p>The maximum allowable length for this property is SRV_MAXCHAR bytes.</p> <p>See “Security services” on page 174.</p>
SRV_S_SERVERNAME	The name of the Open Server application.	<p>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.</p> <p>The maximum allowable length for this property is SRV_MAXCHAR bytes.</p>



Property name	Definition	Notes
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: <ul style="list-style-type: none"> <li>• CS_SSLVER_20</li> <li>• CS_SSLVER_30</li> <li>• CS_SSLVER_TLS1</li> </ul> Adaptive Server Enterprise 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 TDS protocol version that Open Server uses to negotiate all client connections.	See “SRV_S_TDSVERSION” on page 150 for a list of values.
SRV_S_TIMESLICE	The number of clock ticks an active thread consumes before the time slice callback routine is called.	See the <code>srv_callback</code> reference page for information on time slice callbacks.
SRV_S_TRACEFLAG	The type of tracing desired.	See “SRV_S_TRACEFLAG” on page 150 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 <code>srv_init</code> .
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.	

Property name	Definition	Notes
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 151 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 <b>message queue</b> 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 111.

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 443 for a list of defaults.

**Table 2-28: Thread properties**

Property name	Definition	Notes
SRV_T_APPLNAME	The client application's name.	

Property name	Definition	Notes
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 159 for a list of legal values.
SRV_T_CHARTYPE	The type of character data representation.	See "SRV_T_CHARTYPE" on page 159 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.	This property can be set from within the SRV_LISTENER_PREBIND Open Server event.
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.
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 159 for an example of using SRV_T_ENDPOINT.

Property name	Definition	Notes
SRV_T_EVENT	The Open Server event the thread is currently in.	See “SRV_T_EVENT” on page 160 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 <code>srv_event</code> .
SRV_T_FULLPASSTHRU	A Boolean. When set to <code>CS_TRUE</code> , the <code>SRV_FULLPASSTHRU</code> event handler is activated for the thread.	Can only be set inside the Open Server application’s connect handler.  The value of the <code>SRV_T_EVENT</code> property is <code>SRV_FULLPASSTHRU</code> 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 161 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.
SRV_T_IODEAD	A Boolean. Indicates whether a thread’s I/O channel is valid.	<code>CS_TRUE</code> means a thread cannot successfully perform I/O, <code>CS_FALSE</code> means it can. Open Server always returns <code>CS_FALSE</code> for service threads.
SRV_T_LISTENADDR	Returns the address of the listener identified by the specified <code>SRV_PROC</code> control structure. If the <code>SRV_PROC</code> control structure is a listener, this property returns the address on which the listener accepts connections.	This property requires a pointer to a <code>CS_TRANADDR</code> structure as an argument.  This is a retrieve-only property.
SRV_T_LOCALE	A pointer to a <code>CS_LOCALE</code> structure allocated by the Open Server application.	Use this property to retrieve or set localization information.

Property name	Definition	Notes
SRV_T_LOCALID	Specifies the SSL certificate to use for a listener.	This property allows a listener to use a different SSL certificate than the global server-level SSL certificate. This is a set-only property.
SRV_T_LOGINTYPE	The type of login record received.	See “SRV_T_LOGINTYPE” on page 161 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 161 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 162 for more details.
SRV_T_MACHINE	The host name of the machine the client thread is running on.	
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: <ul style="list-style-type: none"> <li>• SRV_CHALLENGE signals the client’s intent to negotiate through a challenge/response exchange.</li> <li>• SRV_ENCRYPT signals the client’s intent to pass a symmetrically encrypted password.</li> <li>• SRV_SECLABEL indicates that the client will send security labels.</li> <li>• SRV_APPDEFINED indicates that an application-defined login handshake is in use.</li> <li>• SRV_EXTENDED_ENCRYPT signals the client’s intent to pass an asymmetrically encrypted password.</li> </ul>

Property name	Definition	Notes
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.
SRV_T_PASSTHRU	A Boolean. Indicates whether the client thread is operating in passthrough mode.	This property can be set inside the application's connect handler.  When set to CS_TRUE, the <code>srv_getloginf</code> and <code>ct_setloginf</code> 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 <code>srv_setpri</code> .
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_REMOTEADDR	Returns the address of a SRV_PROC peer. This property value is only valid for a client SRV_PROC.	This property requires a pointer to a CS_TRANADDR structure as an argument.  This is a retrieve-only property.
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.

Property name	Definition	Notes
SRV_T_RMTCERTIFICATE	CS_SSLCERT * A pointer that describes the client certificate.	
SRV_T_RMTPWDS	An array of SRV_RMTPWDSs.	See “SRV_T_RMTPWDS” on page 163 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.
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: <ul style="list-style-type: none"> <li>• CS_NO_LIMIT – never expires</li> <li>• CS_UNEXPIRED – unexpired</li> <li>• 0 – expired</li> <li>• A positive number – the number of seconds remaining</li> </ul>
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.



Property name	Definition	Notes
SRV_T_SEC_DELEGCREC	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_DELEGCREC 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.	
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: <ul style="list-style-type: none"> <li>• CS_NO_LIMIT – never expires</li> <li>• CS_UNEXPIRED – unexpired</li> <li>• 0 – expired</li> <li>• A positive number – the number of seconds remaining</li> </ul>

Property name	Definition	Notes
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 163 for more details.
SRV_T_SSL_VERSION	The SSL/TLS protocol version that was negotiated during the connection handshake.	This property can be set from within the SRV_LISTENER_PREBIND Open Server event.
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.	
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 163 for a list of legal values.
SRV_T_TYPE	The thread type.	See “SRV_T_TYPE” on page 164 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
<code>SRV_BULKLOAD</code>	The client is preparing to transfer bulk copy data.
<code>SRV_TEXTLOAD</code>	The client is preparing to transfer text data.
<code>SRV_IMAGELOAD</code>	The client is preparing to transfer image data.
<code>SRV_UNITEXTLOAD</code>	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.

See the *Open Client and Open Server Common Libraries Reference Manual*. See “Text and image” on page 201.

## 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
<code>SRV_CHAR_ASCII</code>	ASCII character format
<code>SRV_CHAR_EBCDIC</code>	EBCDIC character format
<code>SRV_CHAR_UNKNOWN</code>	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_LISTENER_PREBIND`
- `SRV_LISTENER_POSTBIND`
- `SRV_MSG`
- `SRV_OPTION`
- `SRV_RPC`
- `SRV_START`
- `SRV_STOP`

- User-defined events

See “Events” on page 93.

## SRV\_T\_FLTTYPER

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_FLTTYPER` 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
<code>SRV_SITEHANDLER</code>	A site handler login request from a remote server.
<code>SRV_SUBCHANNEL</code>	A site handler subchannel login from a remote server.
<code>SRV_CLIENT</code>	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 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 models 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_SUCCEEDED)
{
    ...
}
```

See “Connection migration” on page 40 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);
```

## 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 151 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**

<b>SRV_T_TDSVERSION value</b>	<b>Meaning</b>
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**

<b>Value</b>	<b>Thread type</b>
SRV_TCLIENT	A client thread
SRV_TLISTENER	A thread that accepts client connections
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 111.

## 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 169, 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 Enterprise 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. See "Registered procedures" on page 164.

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 136.
- 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 136.

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.

## Retrieving large object parameters

Open Server can retrieve the complete large object (LOB) parameter data at the same time using `srv_xferdata()`, or in chunks using `srv_get_data`. Open Server retrieves LOB parameters in chunks when the parameter length has been set to `CS_UNUSED`. See “`srv_get_data`” on page 289.

Example

Retrieves description of LOB parameters:

```
/*
** Retrieve the description of the parameters coming
** from client
*/

for (paramnum = 1; paramnum <= numparams; paramnum++)
{
    /*
```



```

** Get a description of the parameter.
*/
ret = srv_descfmt(spp, CS_GET, SRV_RPCDATA,
    paramnum, &(paramfmtp[paramnum - 1]));

/*
** Allocate space for the parameters and bind the
** data.
*/
if (paramfmtp[paramnum-1].maxlength >= 0)
{
    if (paramfmtp[paramnum-1].maxlength > 0)
    {
        data[paramnum-1] = calloc(1,
            paramfmtp[paramnum-1].maxlength);
    }
    else
    {
        ind[paramnum-1] = CS_NULLDATA;
    }
}
else
{
    /*
    ** Allocate a large size buffer for BLOB data
    ** (which length is unknown yet)
    */
    blobbuf[blobnum] = malloc(BUFSIZE);
    blobnum++;
}

srv_bind(spp, CS_GET, SRV_RPCDATA, paramnum,
    &(paramfmtp[paramnum-1]), data[paramnum-1],
    &(len[paramnum-1]), &(ind[paramnum-1]))

/*
** For every LOB parameter, call srv_get_data() in
** a loop as long as it succeeds
*/
for (i = 0; i < blobnum ; i++)
{
    bufp = blobbuf[i];
    bloblen[i] = 0;
    do
    {
        ret = srv_get_data(spp, bufp, BUFSIZE,

```

```
        &outlen);
        bufp += outlen;
        bloblen[i] += outlen;
    } while (ret == CS_SUCCEED);

    /*
    ** Check for the correct return code
    */
    if (ret != CS_END_DATA)
    {
        return CS_FAIL;
    }
}

/*
** And receive remaining client data srv_xferdata()
*/
ret = srv_xferdata(spp, CS_GET, SRV_RPCDATA);
}
```

## 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.

### FIPS 140-2 compliance of login password encryption

Encryption of login and remote passwords in Open Client and Open Server is accomplished with the Sybase Common Security Infrastructure (CSI). CSI 2.6 complies with the Federal Information Processing Standard (FIPS) 140-2.

To support FIPS encryption, a Certicom Security Builder shared library named *libsbgse2.so* (UNIX and Linux platforms) or *libsbgse2.dll* (Microsoft Windows platforms) is installed on platforms that do not already use the Certicom Security Builder. Also, the *sybcsi* subdirectory found in *\$\$SYBASE/\$SYBASE\_OCS/lib3p* or *\$\$SYBASE/\$SYBASE\_OCS/lib3p64* has been removed.

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.

Also see the *Open Client Client-Library/C Reference Manual*.

## **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*.

## 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 Enterprise, 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 Enterprise 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 Enterprise 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 Enterprise 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 183.
- 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 182.

When establishing a connection to an SSL-enabled Adaptive Server Enterprise, Adaptive Server Enterprise 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 Enterprise as specified on the command line when starting the server with the -S flag or from the server's environment variable \$DSLSTEN.

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 Adaptive Server Enterprise 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 Adaptive Server Enterprise 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 Adaptive Server Enterprise 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 Enterprise 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 187.

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 Programmers 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.

## 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 187.

## 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 151 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 190, 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_SUCCEEDED);
}

```

## 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 198 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_LOGININFO   *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_SECSSESSION_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_SECSSESSION)`
  - `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_SECSSESSION)`.
- 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_SECSSESSION)
        != 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_SUCCEEDED)
{
    return(CS_FAIL);
}
/* Complete this portion of the exchange */
if (srv_senddone(spp, SRV_DONE_FINAL, CS_TRAN_UNDEFINED, 0)
    != CS_SUCCEEDED)
{
    return(CS_FAIL);
}
/* Wait until the client responds */
if (srv_negotiate(spp, CS_GET, SRV_NEG_SECSSESSION)
    != CS_SUCCEEDED)
{
    return(CS_FAIL);
}
/* Get the number of parameters in the client's response */
if (srv_numparams(spp, outnumparams) != CS_SUCCEEDED)
{
    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_SUCCEEDED)
        {
            return(CS_FAIL);
        }
    if (srv_bind(spp, CS_GET, SRV_NEGDATA, i + 1, &outfmt[i],
        outbuf[i], &outlen[i], (CS_SMALLINT *)NULL)
        != CS_SUCCEEDED)
        {
            return(CS_FAIL);
        }
}
if (srv_xferdata(spp, CS_GET, SRV_NEGDATA) != CS_SUCCEEDED)
{
    return(CS_FAIL);
}

```

```
    }  
    /* Return success */  
    return(CS_SUCCEED);  
}
```

## Text and image

The text and image Adaptive Server Enterprise 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 136 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. 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.

## Suppressing commands

To simplify the process of updating a text or image column and potentially improve performance, a client can suppress the generation of the SQL command (`update` or `writetext`) and send data directly to the server bulk handler. To do this, the client must initiate the send-data command by calling the `ct_command` routine with the *type* parameter set to `CS_SEND_DATA_NOCMD`. The client application can then use send-data commands to send only text or image data to the server bulk handler. When a bulk event occurs at the server, a 4-byte field is sent indicating the total number of bytes to be sent, followed by the text or image data. The bulk handler reads the total number of bytes expected using `srv_text_info` and the data using `srv_get_data`.

The server must define a stored procedure, `sp_mda`, to indicate whether or not it supports the `ct_send_data` routine sending only text or image data without a SQL command. The server `sp_mda` procedure is called only if the client application sets the `CS_SENDDATA_NOCMD` connection property—for example, `ct_con_props(CS_SENDDATA_NOCMD)`—before the `ct_connect` routine is called. If `CS_SENDDATA_NOCMD` is set, the server `sp_mda` procedure is called during execution of `ct_connect`. If `sp_mda` indicates that the server does not support the `ct_send_data` routine sending only text or image data without a SQL command, any calls to the `ct_command` routine with the `type` parameter set to `CS_SEND_DATA_NOCMD` fail.

If the server can receive text or image data without a SQL command, `sp_mda` returns the following:

Parameter	Value
<code>mdinfo</code>	<code>"SENDDATA_NOCMD"</code>
<code>querytype</code>	2
<code>query</code>	<code>senddata no cmd</code>

---

**Note** Adaptive Server cannot receive image or text data without a SQL command.

---

## 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 136 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 Enterprise datatypes.

Table 2-37 lists the Open Server type definitions, together with their corresponding type constants and Adaptive Server Enterprise 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**

<b>Type</b>	<b>Open Client and Open Server type constant</b>	<b>Description</b>	<b>Corresponding Open Client and Open Server type definition</b>	<b>Corresponding Adaptive Server Enterprise datatype</b>
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
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
	CS_BIGDATETIME_TYPE	8-byte binary type	CS_BIGDATETIME	bigdatetime
	CS_BIGTIME_TYPE	8-byte binary type	CS_BIGTIME	bigtime

<b>Type</b>	<b>Open Client and Open Server type constant</b>	<b>Description</b>	<b>Corresponding Open Client and Open Server type definition</b>	<b>Corresponding Adaptive Server Enterprise datatype</b>
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
	Large object (LOB) locator types	CS_TEXTLOCATOR_TYPE	Textlocator type	CS_TEXTLOCATOR
CS_IMAGELOCATOR_TYPE		Imagelocator type	CS_IMAGELOCATOR	imagelocator
CS_UNITEXTLOCATOR_TYPE		Unsigned variable-length character type	CS_UNITEXTLOCATOR	unitextlocator
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 Enterprise 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 Enterprise 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 Enterprise 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 Enterprise 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 Enterprise 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 Enterprise 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 Enterprise 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 Enterprise 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 Enterprise 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 six datetime types, CS\_DATE, CS\_TIME, CS\_DATETIME, CS\_DATETIME4, CS\_BIGDATETIME, and CS\_BIGTIME. These datatypes are intended to hold 4-byte and 8-byte datetime values.

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

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 Enterprise 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 Enterprise 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 Enterprise 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 Enterprise 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.

- CS\_BIGDATETIME corresponds to the Adaptive Server Enterprise bigdatetime datatype and contains the number of microseconds that have passed since January 1, 0000 00:00:00.000000. The range of legal CS\_BIGDATETIME values is from January 1, 0001 00:00:00.000000 to December 31, 9999 23:59:59.999999.

---

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

---

The definition of CS\_BIGDATETIME can be found in *cstypes.h*:

```
typedef CS_UBIGINT CS_BIGDATETIME;
```

- CS\_BIGTIME corresponds to the Adaptive Server Enterprise bigtime datatype and indicates the number of microseconds that have passed since the beginning of the day. The range of legal CS\_BIGTIME values is from 00:00:00.000000 to 23:59:59.999999. The definition of CS\_BIGTIME can be found in *cstypes.h*:

```
typedef CS_UBIGINT CS_BIGTIME;
```

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

## Datetime minimum and maximum values

The following table describes minimum and maximum values for the datetime types.

**Table 1: Datetime minimum and maximum values**

Datatype	Minimum value	Maximum value
CS_BIGDATETIME	January 1, 0001 00:00:00.000000	December 31, 9999 23:59:59.999999
CS_BIGTIME	00:00:00.000000	23:59:59.999999
CS_DATE	January 1, 0001	December 31, 9999
CS_DATETIME	January 1, 1753 00:00:00.000	December 31, 9999 23:59:59.999



Datatype	Minimum value	Maximum value
CS_DATETIME4	January 1, 1900 00:00:00.000	June 6, 2079 23:59:59.999
CS_TIME	00:00:00.000	23:59:59.999

## 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 Enterprise 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 < 1844670000000000000.0

Values outside of these ranges cause overflow errors.

---

- CS\_FLOAT corresponds to the Adaptive Server Enterprise 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:

- $-9223372036854770000.0 < \text{bigint} < 9223372036854770000.0$
- $0 < \text{ubigint} < 18446744073709500000.0$

Values outside of these ranges cause overflow errors.

---

- CS\_NUMERIC and CS\_DECIMAL correspond to the Adaptive Server Enterprise datatypes numeric and decimal. These types provide platform-independent support for numbers with precision and scale.

The Adaptive Server Enterprise datatypes numeric and decimal are equivalent; and CS\_DECIMAL is defined as CS\_NUMERIC:

```
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.

## Large object locator datatypes

Open Server supports large object (LOB) locators. A LOB locator contains a logical pointer to LOB data in Adaptive Server rather than the data itself, thereby reducing the amount of data that passes through the network between Adaptive Server and its clients.

The `CS_LOCATOR` datatype supports LOB locator. The `CS_LOCATOR` datatype is an opaque datatype that stores locator values and optional prefetched data. Use `cs_locator_alloc()` to allocate memory for a `CS_LOCATOR` variable before binding the incoming locator to the variable, otherwise, an error occurs. When the variable is no longer needed, free its memory using `cs_locator_drop()`.

`CS_LOCATOR` variables can be reused. However, the current locator value in Adaptive Server is valid only until the transaction ends. The `CS_LOCATOR` datatype has three type constants—`CS_TEXTLOCATOR_TYPE`, `CS_IMAGELOCATOR_TYPE`, and `CS_UNITEXTLOCATOR_TYPE`.

## 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 Enterprise 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 Enterprise 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 Enterprise 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 Enterprise `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;
```

# Routines

This chapter contains a reference page for each Server-Library routine.

<b>Routine</b>	<b>Description</b>	<b>Page</b>
srv_alloc	Allocate memory.	221
srv_alt_bind	Describe and bind the source data for a compute row column.	223
srv_alt_descfmt	Describe the aggregate operator of a compute row column and the format of the column data returned to the client.	227
srv_alt_header	Describe a compute row's row identifier and bylist.	231
srv_alt_xferdata	Send a compute row to a client.	234
srv_bind	Describe and bind a program variable for a column or parameter.	236
srv_bmove	Copy bytes from one memory location to another.	241
srv_bzero	Set the contents of a memory location to zero.	243
srv_callback	Install a state transition handler for a thread.	244
srv_capability	Determine whether the Open Server supports a platform-dependent service.	248
srv_capability_info	Define or retrieve capability information on a client connection.	249
srv_createmsgq	Create a message queue.	253
srv_createmutex	Create a mutual exclusion semaphore.	255
srv_createproc	Create a nonclient, event-driven thread.	257
srv_cursor_props	(For pre-15.0 applications) retrieve or set information about the current cursor.	259
srv_cursor_props2	(For 15.0 and later applications) retrieve or set information about the current cursor.	262
srv_dbg_stack	Display the call stack of a thread.	263
srv_dbg_switch	Temporarily restore another thread context for debugging.	265
srv_define_event	Define a user event.	266

---

<b>Routine</b>	<b>Description</b>	<b>Page</b>
srv_deletemsgq	Delete a message queue.	268
srv_deletemutex	Delete a mutex created by <code>srv_createmutex</code> .	270
srv_descfmt	Describe or retrieve the description of a column or a parameter going to, or coming from, a client.	272
srv_dynamic	Read or respond to a client dynamic SQL command.	275
srv_envchange	Notify the client of an environment change.	280
srv_event	Add an event request to a thread's request-handling queue.	282
srv_event_deferred	Add an event request to the event queue of a thread as the result of an asynchronous event.	285
srv_free	Free previously allocated memory.	287
srv_freeserveraddrs	Frees memory allocated by <code>srv_getserverbyname</code> .	288
srv_get_data	Read a text, unitext, or image parameter stream from a client, in chunks.	289
srv_get_text	Read a text or image datastream from a client, in chunks.	290
srv_getlogininfo	Obtain login information from a client thread to prepare a passthrough connection with a remote server.	292
srv_getmsgq	Get the next message from a message queue.	294
srv_getobjid	Look up the object ID for a message queue or mutex with a specified name.	297
srv_getobjname	Get the name of a message queue or mutex with a specified name.	300
srv_getserverbyname	Returns the connection information for <i>server_name</i> , allocating memory as needed.	302
srv_handle	Install an event handler into an Open Server application.	303
srv_init	Initialize an Open Server application.	306
srv_langcpy	Copy a client's language request into an application buffer.	308

<b>Routine</b>	<b>Description</b>	<b>Page</b>
srv_langlen	Return the length of the language request buffer.	310
srv_lockmutex	Lock a mutex.	312
srv_log	Write a message to the Open Server application log file.	315
srv_mask	Initialize, check, set, or clear bits in a SRV_MASK_ARRAY structure.	317
srv_msg	Send or receive a message datastream.	319
srv_negotiate	Send and receive negotiated login information to or from a client.	322
srv_numparams	Return the number of parameters contained in the current client command.	329
srv_options	Send or receive option information to or from a client.	331
srv_orderby	Return an order-by list to a client.	337
srv_poll (UNIX only)	Check for I/O events on a set of open streams file descriptors.	339
srv_props	Define and retrieve Open Server properties.	342
srv_putmsgq	Put a message into a message queue.	348
srv_realloc	Reallocate memory.	351
srv_recvpass thru	Receive a protocol packet from a client.	352
srv_regcreate	Complete the registration of a registered procedure.	354
srv_regdefine	Initiate the process of registering a procedure.	356
srv_regdrop	Unregister a procedure.	360
srv_regexec	Execute a registered procedure.	362
srv_reginit	Begin executing a registered procedure.	364
srv_reglist	Obtain a list of all of the procedures registered in the Open Server.	366
srv_reglistfree	Free a previously allocated SRV_PROCLIST structure.	368
srv_regnowatch	Remove a client thread from the notification list for a registered procedure.	369
srv_regparam	Describe a parameter for a registered procedure being defined; or supply data for the execution of a registered procedure.	371
srv_regwatch	Add a client thread to the notification list for a specified procedure.	374

<b>Routine</b>	<b>Description</b>	<b>Page</b>
srv_regwatchlist	Return a list of all registered procedures for which a client thread has notification requests pending.	377
srv_rpcdb	Return the database component of the current remote procedure call's designation.	379
srv_rpcname	Return the name component of the current remote procedure call's designation.	380
srv_rpcnumber	Return the number component of the current remote procedure's designation.	383
srv_rpcoptions	Return the runtime options for the current remote procedure call.	384
srv_rpcowner	Return the owner component of the current remote procedure call's designation.	386
srv_run	Start an Open Server.	388
srv_s_ssl_local_id	Used to specify the path to the local ID (certificates) file.	389
srv_select (UNIX only)	Check to see if a file descriptor is &ready for a specified I/O operation.	389
srv_send_ctlinfo	Sends control messages to Client-Library.	393
srv_send_data	Transfers rows containing multiple columns to clients.	394
srv_send_text	Send a text or image datastream to a client, in chunks.	398
srv_senddone	Send a results completion message or flush some results to a client.	401
srv_sendinfo	Send error or informational messages to the client.	406
srv_sendpassthru	Send a protocol packet to a client.	409
srv_sendstatus	Send a status value to a client.	412
srv_setcolotype	Define the user datatype to be associated with a column.	413
srv_setcontrol	Describe user control or format information for columns.	415
srv_setloginfo	Return protocol format information from a remote server to a client.	417
srv_setpri	Modify the scheduling priority of a thread.	419
srv_signal (UNIX only)	Install a UNIX signal handler for the SIGIO or SIGURG signals, using the same interface as signal.	421



<b>Routine</b>	<b>Description</b>	<b>Page</b>
srv_sleep	Suspend the currently executing thread.	424
srv_spawn	Allocate a service thread.	427
srv_symbol	Convert an Open Server token value to a readable string.	430
srv_tabcolname	Associate browse mode result columns with result tables.	434
srv_tabname	Provide the name of the table or tables associated with a set of browse mode results.	437
srv_termproc	Terminate the execution of a thread.	438
srv_text_info	Set or get a description of text or image data.	440
srv_thread_props	Define and retrieve thread properties.	443
srv_timedsleep	Sleep until an event is signalled.	448
srv_unlockmutex	Unlock a mutex.	451
srv_version	Define the version of Server-Library an application is using, and define the application's default national language and character set.	452
srv_wakeup	Enable sleeping threads to run.	454
srv_xferdata	Send parameters or data to a client, or receive parameters or data from a client.	456
srv_yield	Allow another thread to run.	458

## srv\_alloc

Description	Allocate memory.
Syntax	<pre>CS_VOID *srv_alloc(size) CS_INT  size;</pre>
Parameters	<p><i>size</i> The number of bytes to allocate.</p>

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 <i>size</i> 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, osfmt, varaddr, varlenp, indp)`

```

SRV_PROC      *spp;
CS_INT        altid;
CS_INT        item;
CS_DATAFMT    *osfmt;
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 Enterprise'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 Enterprise.
- `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 “`osfmp`” is a pointer to the structure.)

**Table 3-4: CS\_DATAFMT fields used (srv\_alt\_bind)**

Field	CS_SET	CS_GET
<i>osfmp</i> → <i>datatype</i>	Datatype of application program variable	Datatype of application program variable
<i>osfmp</i> → <i>maxlength</i>	Unused	Maximum length of program variable
<i>osfmp</i> → <i>count</i>	0 or 1	0 or 1

- If the format described by *osfmp* differs from the client format set with `srv_alt_descfmt` (*clfmp*), 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, clfmt)

SRV\_PROC \*spp;  
 CS\_INT altid;  
 CS\_INT optype;  
 CS\_TINYINT operand;  
 CS\_INT item;  
 CS\_DATAFMT \*clfmt;

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.

*clfmt*

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   clfmt[]
));
/*
** 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.
**   clfmt[] 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, clfmt)
SRV_PROC     *sproc;
CS_INT       altid;
CS_DATAFMT   clfmt[];
{
/*
** 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 Enterprise'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 Enterprise.
- `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
<i>clfmtp</i> → <i>namelen</i>	Length of name	Length of name
<i>clfmtp</i> → <i>status</i>	Parameter/column status	Parameter status
<i>clfmtp</i> → <i>name</i>	Parameter/column name	Parameter name
<i>clfmtp</i> → <i>datatype</i>	Remote datatype set here	Remote datatype retrieved from here
<i>clfmtp</i> → <i>maxlength</i>	Maximum length of remote datatype set here	Maximum length of remote datatype retrieved from here
<i>clfmtp</i> → <i>format</i>	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 Enterprise’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 Enterprise.
- `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 Enterprise, 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 Enterprise’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 Enterprise'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 Enterprise.
- `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, osfmt, varaddrp, varlenp, indp)`

```
SRV_PROC      *spp;
CS_INT        cmd;
CS_INT        type;
CS_INT        item;
CS_DATAFMT   *osfmt;
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 <i>*varaddrp</i> is sent to a client when <code>srv_xferdata</code> is called.
CS_GET	<i>*varaddrp</i> is initialized with data from a client after a call to <code>srv_xferdata</code> .

*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)	<ul style="list-style-type: none"> <li>• Cannot be NULL</li> <li>• Points to the actual length of the data in *varaddrp</li> <li>• Need not be valid until srv_xferdata is called</li> </ul>
CS_GET (data coming in from client)	<ul style="list-style-type: none"> <li>• Can be NULL (indicating that the Open Server application already knows the length of the data)</li> <li>• Is a pointer to the program variable in which Open Server places the actual length of the data</li> <li>• Is filled in after a call to srv_xferdata</li> </ul>

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_SUCCEEDED   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 “*osfmp*” 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:
<i>osfmp</i> → <i>datatype</i>	Datatype of application program variable	Datatype of application program variable
<i>osfmp</i> → <i>maxlength</i>	Actual length of program variable	Maximum length of program variable
<i>osfmp</i> → <i>count</i>	0 or 1	0 or 1
<i>osfmp</i> → <i>status</i>	<i>CS_CANBENULL</i> 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 *osfmp* differs from the format of the data received from the client (*cmd* set to *CS\_GET*), Open Server automatically converts the data to *osfmp*. 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 (*clfmp*).

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 136.

## 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_SUCCEEDED  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_SUCCEEDED);
}

```

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 <code>srv_spawn</code> 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 <code>SRV_S_TIMESLICE</code> , <code>SRV_S_VIRTCLKRATE</code> , and <code>SRV_S_VIRTTIMER</code> server properties. See <code>srv_props</code> on page 342 and "Properties" on page 141.

*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 164 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_SUCCEEDED   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_SUCCEEDED, if it is
    ** CS_FALSE we return CS_FAIL.
    */
    return(supported ? CS_SUCCEEDED : 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 141

## 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 already 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.
**      msgqnm     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 143.
- 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 143.

**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 successfully.
** 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 already 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.<br>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.

---

**Note** For pre-15.0 applications, use the `SRV_CURDESC` structure and `srv_cursor_props` routine. For version 15.0 and later applications that support scrollable cursors on Open Server, use the `SRV_CURDESC2` structure and `srv_cursor_props2` routine.

---

**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_SUCCEEDED);
}

```

**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\_cursor\_props2

Description Retrieve or set information about the current cursor.

---

**Note** For version 15.0 and later applications that support scrollable cursors on Open Server, use the `SRV_CURDESC2` structure and `srv_cursor_props2` routine. For pre-15.0 applications, use the `SRV_CURDESC` structure and `srv_cursor_props` routine.

---

Syntax `CS_RETCODE srv_cursor_props2(spp, cmd, cdp)`

```
SRV_PROC      *spp;
CS_INT        cmd;
SRV_CURDESC2 *cdp;
```

Parameters

*spp*

A pointer to an internal thread control structure.

*cmd*

Indicates whether `srv_cursor_props2` sends cursor information to the client or retrieves cursor information from the client.

**Table 3-29: Values for *cmd* (*srv\_cursor\_props2*)**

| Value  | Description                                                                                            |
|--------|--------------------------------------------------------------------------------------------------------|
| CS_SET | <code>srv_cursor_props2</code> sends information about the current cursor to the client.               |
| CS_GET | <code>srv_cursor_props2</code> retrieves information about the current cursor command from the client. |

*cdp*

A pointer to a `SRV_CURDESC2` structure. When the application is setting cursor information, the `SRV_CURDESC2` structure describes the current cursor. When the application is retrieving information, Open Server updates the `SRV_CURDESC2` 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_CURDESC2` structure” on page 77.

Return value

**Table 3-30: Return values (*srv\_cursor\_props2*)**

| Returns    | To indicate                         |
|------------|-------------------------------------|
| CS_SUCCEED | The routine completed successfully. |
| CS_FAIL    | The routine failed.                 |

Usage

See the Usage section of the *srv\_cursor\_props* routine.

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 3-31: 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(spид)`

`CS_INT` `spид;`

Parameters *spид*  
The server process ID (*spид*) of the thread whose context should be temporarily restored.

Return value **Table 3-32: Return values (`srv_dbg_switch`)**

| Returns                 | To indicate                         |
|-------------------------|-------------------------------------|
| <code>CS_SUCCEED</code> | The routine completed successfully. |
| <code>CS_FAIL</code>    | 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_QUEUED`.

*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 3-33: 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.<br>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_QUEUED,
                                     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 93

## 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 3-34: Return values (srv\_deletemsgq)**

| Returns                   | To indicate                         |
|---------------------------|-------------------------------------|
| <code>CS_SUCCEEDED</code> | The routine completed successfully. |
| <code>CS_FAIL</code>      | 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 143.
  - 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 143.
- 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 3-35: 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 3-36 describes the legal values for *cmd*:

**Table 3-36: Values for *cmd* (`srv_descfmt`)**

| Value  | Description                                                                                    |
|--------|------------------------------------------------------------------------------------------------|
| CS_SET | <code>srv_descfmt</code> describes the format the data will be in when the client receives it. |
| CS_GET | <code>srv_descfmt</code> 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 3-37 describes the valid types and their appropriate context:

**Table 3-37: 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.

*clfmt*

A pointer to a CS\_DATAFMT structure containing a description of the data.

Return value

**Table 3-38: 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_SUCCEEDED 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 already raised.
        */
        return CS_FAIL;
    }

    /*
    ** All done.
    */
    return CS_SUCCEEDED;
}

```

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 “clfmt” is a pointer to the structure.

**Table 3-39: 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 3-40 describes the legal values for *cmd*:

**Table 3-40: 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 3-41 describes the legal values for *item*:

**Table 3-41: 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 3-42 summarizes the required buffer sizes:



**Table 3-42: 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 <i>srv_dynamic</i> with item set to CS_DYN_IDLEN and then allocate buffer size accordingly.   |
| SRV_DYN_STMTLEN | sizeof(CS_INT).                                                                                                                         |
| SRV_DYN_STMT    | Varies. Determine length by first calling <i>srv_dynamic</i> 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 3-43: 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 3-44: 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 Enterprise 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 Enterprise. Table 3-45 lists the message number and `oserror.h` #define that correspond to each changing value.

**Table 3-45: Environment variables (`srv_envchange`)**

| Changing value    | Message number | #define in <code>oserror.h</code> |
|-------------------|----------------|-----------------------------------|
| 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 93 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 3-46: 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 93



## srv\_event\_deferred

|              |                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Description  | Add an event request to the event queue of a thread as the result of an asynchronous event.                                                                                                                                                                                                                                                                                                               |
| Syntax       | <pre>CS_INT srv_event_deferred(spp, event, datap) SRV_PROC *spp; CS_INT event; CS_VOID *datap;</pre>                                                                                                                                                                                                                                                                                                      |
| Parameters   | <p><i>spp</i><br/>A pointer to an internal thread control structure.</p> <p><i>event</i><br/>The event to add to the thread's event queue.</p> <p><i>datap</i><br/>A pointer (CS_VOID) to data supplied by the Open Server programmer. An application can retrieve the data by calling <code>srv_thread_props</code> with property set to <code>SRV_T_EVENTDATA</code> from within the event handler.</p> |
| Return value | The requested event. If there was an error, -1 is returned.                                                                                                                                                                                                                                                                                                                                               |

**Table 3-47: 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 successfully 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 93

## 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 3-48: 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\_freesserveraddrs

Description            Free memory allocated by `srv_getserverbyname`.  
Syntax                 CS\_RETCODE **srv\_freesserveraddrs**(void \*resultptr)

Parameters *resultptr*  
A pointer to memory returned by `srv_getserverbyname`.

Return value **Table 3-49: Return values (*srv\_freeserveraddrs*)**

| Returns    | To indicate                                                    |
|------------|----------------------------------------------------------------|
| CS_SUCCEED | The call to <code>srv_freeserveraddrs</code> ran successfully. |
| CS_FAIL    | <i>resultptr</i> is NULL or deallocation failed.               |

See also `srv_getserverbyname`, `srv_send_ctlinfo`

## srv\_get\_data

Description Read a text, unitext, or image parameter stream from a client, in chunks.

Syntax CS\_RETCODE `srv_get_data(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, indicating 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 3-50: Return values (srv\_get\_data)**

| Returns     | To indicate                                                                      |
|-------------|----------------------------------------------------------------------------------|
| CS_SUCCEED  | srv_get_data ran successfully, more data is pending.                             |
| CS_FAIL     | The routine failed.                                                              |
| CS_END_DATA | srv_get_data has completed reading the entire text, unitext, or image parameter. |

## 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 3-51: 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 101, “Text and image” on page 201

## 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 3-52: 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_LOGININFO` structure and fill it with protocol information from the client thread.
  - b `ct_setloginfo` – prepare a `CS_LOGININFO` 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_LOGININFO` structure.
  - d `srv_setloginfo` – send the remote server’s response, retrieved in step 3, to the client, then release the `CS_LOGININFO` 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 3-53 lists the legal values for *getflags*, and their significance:

**Table 3-53: Values for *getflags* (*srv\_getmsgq*)**

| Value           | Significance                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SRV_M_WAIT      | If no message is available, <code>srv_getmsgq</code> sleeps until a message is delivered.                                                                                                                                                                                                                                                                                                                                                                                        |
| SRV_M_NOWAIT    | <code>srv_getmsgq</code> returns immediately whether a message is available or not.                                                                                                                                                                                                                                                                                                                                                                                              |
| SRV_M_READ_ONLY | The default behavior of <code>srv_getmsgq</code> 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 <code>SRV_M_READ_ONLY</code> 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 3-54 describes the possible values returned in *infop* if `srv_getmsgq` returns `CS_FAIL`:

**Table 3-54: 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 3-55: 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_SUCCEEDED)
{
    /* 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 3-56 describes the possible values returned in \*infop if srv\_getobjid returns CS\_FAIL:

**Table 3-56: 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 3-57: Return values (srv\_getobjid)**

| Returns      | To indicate                         |
|--------------|-------------------------------------|
| CS_SUCCEEDED | 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_QUEUE 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_SUCCEEDED  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\_QUEUE).

*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 3-58: 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 occurred.
*/
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 | Return the connection information for <i>server_name</i> , allocating memory as needed. Memory allocated by <code>srv_getserverbyname</code> can be freed by calling <code>srv_freeserveraddrs</code> .                                                                                                                                                                                                                                                                                                                       |
| Syntax      | <code>CS_RETCODE srv_getserverbyname(CS_CHAR *server_name, CS_INT namelen, CS_INT querytype, CS_INT result_type, void *resultptr, CS_INT *result_cnt)</code>                                                                                                                                                                                                                                                                                                                                                                  |
| Parameters  | <p><i>server_name</i><br/>Name of the server to be looked up.</p> <p><i>namelen</i><br/>Length of <i>server_name</i>. Can be specified as <code>CS_NULLTERM</code>.</p> <p><i>querytype</i><br/>Selects master (<code>CS_ACCESS_CLIENT_MASTER</code>) or query (<code>CS_ACCESS_CLIENT_QUERY</code>) entries for <i>server_name</i>.</p> <p><i>result_type</i><br/>Indicates the data format of connection information. <i>result_type</i> can be specified as <code>SRV_C_GETADDRS</code> or <code>SRV_C_GETSTRS</code>.</p> |

*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_freeserveradds`, `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 93.

*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 3-59: 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_SUCCEEDED`.

See also

`srv_define_event`, `srv_event`, `srv_event_deferred`, “Events” on page 93

## 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 `DSLISITEN` will be the server’s name. If `DSLISITEN` 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 3-60: 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. See the *Open Client and Open Server Programmers 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 3-61: 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)`

Parameters            `SRV_PROC *spp;`  
                          *spp*  
                          A pointer to an internal thread control structure.

Return value            **Table 3-62: 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_SUCCEEDED 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_SUCCEEDED);
}
```

- 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.

*info*

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 3-63: 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 already 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 already 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	<pre>CS_RETCODE srv_log(ssp, datestamp, msgp, msglen) SRV_SERVER *ssp; CS_BOOL datestamp; CS_CHAR *msgp; CS_INT msglen;</pre>
Parameters	<p><i>ssp</i> The handle to the Open Server. This argument is optional. It is only present for backward compatibility.</p> <p><i>datestamp</i> If <i>datestamp</i> is CS_TRUE, the current date and time is added to the beginning of the log message. If <i>datestamp</i> is CS_FALSE, the log message is not timestamped.</p> <p><i>msgp</i> A pointer to the actual text of the message.</p> <p><i>msglen</i> The length in bytes of <i>msg</i>. If the string in <i>*msgp</i> is null terminated, <i>msglen</i> can be CS_NULLTERM.</p>

Return value

**Table 3-64: 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 3-65 summarizes the legal values for *cmd*:

**Table 3-65: Legal values for *cmd* (*srv\_mask*)**

Value	Action
CS_SET	Set the bit in the <code>SRV_MASK_ARRAY</code> in <i>*maskp</i> .
CS_GET	Find out whether the bit is currently set in the <code>SRV_MASK_ARRAY</code> in <i>*maskp</i> . If bit is set, <i>*infop</i> is set to <code>CS_TRUE</code> . Otherwise, it is set to <code>CS_FALSE</code> .
CS_CLEAR	Clear the bit in the <code>SRV_MASK_ARRAY</code> in <i>*maskp</i> .
CS_ZERO	Initialize the <code>SRV_MASK_ARRAY</code> in <i>*maskp</i> so that all the bits are off. When <i>cmd</i> is set to <code>CS_ZERO</code> , <i>bit</i> and <i>infop</i> 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 3-66: 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 3-67 describes the legal values for `cmd`:

**Table 3-67: Values for *cmd* (*srv\_msg*)**

Value	Description
CS_SET	<code>srv_msg</code> is setting the values for <i>status</i> and <i>msgid</i> prior to sending the message to the client.
CS_GET	<code>srv_msg</code> is retrieving the <i>status</i> and <i>msgid</i> 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 3-68 describes the legal values for *\*statusp*:

**Table 3-68: Values for statusp (srv\_msg)**

Value	Description
SRV_HASPARAMS	The message has parameters.
SRV_NOPARAMS	The message has no parameters.

Return value

**Table 3-69: 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 3-70 describes the legal values for *cmd*:

**Table 3-70: Values for cmd (srv\_negotiate)**

Value	Description
CS_SET	The negotiated login information defined by <i>type</i> is to be sent to the client.
CS_GET	The negotiated login information defined by <i>type</i> is to be read from the client.

*type*

The type of negotiated login information to be sent to or read from a client. Table 3-71 describes the legal values for *type*:

**Table 3-71: 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 <i>cmd</i> 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 <i>cmd</i> 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 <i>cmd</i> 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 <i>cmd</i> 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 <i>cmd</i> 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 <i>cmd</i> 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 174.
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 <i>type</i> argument itself.



Return value

**Table 3-72: 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 3-73 lists the parameter or parameters that accompany each type of challenge sent to a client:

**Table 3-73: Required challenge parameters (*srv\_negotiate*)**

Negotiated login type	Parameters required
<code>SRV_NEG_CHALLENGE</code>	One parameter – Challenge-data value. Datatype is <code>CS_BINARY_TYPE</code> with the <code>CS_DATAFMT status</code> field set to <code>CS_CANBENULL</code> .
<code>SRV_NEG_ENCRYPT</code>	One parameter – Encryption key data value. Datatype is <code>CS_BINARY_TYPE</code> with the <code>CS_DATAFMT status</code> field set to <code>CS_CANBENULL</code> .
<code>SRV_NEG_SECLABEL</code>	No parameters.
<code>SRV_NEG_SECSSESSION</code>	The security session callback specifies the number of parameters and their data formats. Refer to “Security session callbacks” on page 198 and to the <i>Open Client Client-Library/C Reference Manual</i> .
An integer value between <code>CS_USER_MSGID</code> and <code>CS_USER_MAX_MSGID</code> , inclusive.	One parameter – Application-defined login handshake data value.

- Table 3-74 lists the parameter that should be read from a client for each type of negotiated login challenge:

**Table 3-74: 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_SECSSESSION	The security session callback specifies the number of parameters and their data formats. Refer to “Security session callbacks” on page 198 and to the <i>Open Client Client-Library/C Reference Manual</i> .
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 fails.
- 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 198 and to the *Open Client Client-Library/C Reference Manual*.

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 3-75: 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 3-76 lists those events and their parameters:

**Table 3-76: 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. <i>srv_numparams</i> 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 <i>srv_numparams</i> if the TDS version is less than <i>SRV_TDS_5_0</i> . You can use the <i>SRV_S_TDSVERSION</i> property of the <i>srv_props</i> routine to get the TDS protocol version on the connection (see Table 2-25 on page 143).
After a <i>srv_negotiate</i> (CS_GET) call.	Parameters in the client's response. For example, in the sample program, <i>ctos.c</i> .

See also

*srv\_bind*, *srv\_cursor\_props*, *srv\_descfmt*, *srv\_dynamic*, *srv\_msg*, *srv\_xferdata*, "Processing parameter and row data" on page 136

## 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 3-77 describes the legal values for *cmd*:

**Table 3-77: 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 3-78 summarizes the legal values for *\*optcmdp*:

**Table 3-78: Values for optcmdp (srv\_options)**

Value	Description	Cmd
SRV_SETOPTION	The client is requesting that the option be set. The value associated with <i>optionp</i> is returned in <i>*bufp</i> . Open Server will set <i>bufsize</i> to the size, in bytes, of the data returned. If <i>*bufp</i> 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 <i>*outlenp</i> , and the values of <i>optionp</i> and <i>bufp</i> will remain undefined.	CS_GET
SRV_CLEAROPTION	The client is requesting that <i>optionp</i> be set to its default value. The <i>bufp</i> and <i>optionp</i> values will remain undefined.	CS_GET
SRV_GETOPTION	A client is requesting information on the current value in <i>*optionp</i> . The <i>bufp</i> and <i>optionp</i> 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. <i>bufp</i> points to the argument associated with the option, and <i>bufsize</i> holds the size, in bytes, of the data in <i>*bufp</i> .	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 3-79: 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 receive 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 3-80 summarizes the valid options, their legal values, and the datatype of the *optionp* parameter:

**Table 3-80: Description of options (srv\_options)**

<b>Option</b>	<b>Legal value</b>	<b>bufp points to</b>
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 124 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_SEDOPTION` 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 124

## 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 3-81: 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 Enterprise'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, nfd, waitflag)`

```

SRV_POLLFD    *fdsp;
CS_INT        nfd;
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. */

```

*nfd*

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 3-82: 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.
**     nfd - 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, nfd)
struct pollfd   *fdp;
CS_INT          nfd;
{
```



```

/*
** 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, nfd, 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 3-83 summarizes legal values for `srv_events` and `srv_revents`:

**Table 3-83: 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 <code>SRV_POLLFD</code> .

- `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 3-84 summarizes the legal values for *cmd*:

**Table 3-84: Values for cmd (srv\_props)**

Value	Meaning
CS_SET	The Open Server application is setting the property. In this case, <i>bufp</i> should contain the value the property is to be set to, and <i>buflen</i> should specify the size, in bytes, of that value.
CS_GET	The Open Server application is retrieving the property. In this case, <i>bufp</i> should point to the buffer where the property value is placed, and <i>buflen</i> 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, <i>bufp</i> , <i>buflen</i> , and <i>outlenp</i> 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 3-85: 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.
- Use the SRV\_S\_TCP\_RCVBUF and SRV\_S\_TCP\_SNDBUF server properties for setting TCP socket input and output buffer sizes. Open Server applications use these property settings to set buffer sizes with the operating system setsockopt command. Because setsockopt must be invoked before the TCP connect and accept commands, set these properties before attempting to create a connection.

Set the properties as appropriate for your application. For example, if the server is expected to be receiving a large amount of data from the client, set SRV\_S\_TCP\_RCVBUF to a large value to increase the corresponding buffer size.

---

**Note** Use the CS\_TCP\_RCVBUF and CS\_TCP\_SNDBUF context/connection properties to set TCP socket input and output buffer sizes on the client side. See the *Open Client Client-Library/C Reference Manual*.

---

A server example:

```

    srv_props(cp, CS_SET, SRV_S_TCP_SNDBUF, bufp,
        CS_SIZEOF(CS_INT), (CS_INT *)NULL);

```

- Table 3-86 summarizes the server properties, whether they can be set or retrieved, and the datatype of each property value:

**Table 3-86: Server properties and their datatypes (srv\_props)**

<b>Property</b>	<b>SET/ CLEAR</b>	<b>GET</b>	<b>bufp when cmd is CS_SET</b>	<b>bufp when cmd is CS_GET</b>
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
SRV_S_LOGFILE	Yes	Yes	A character string	A character string
SRV_S_LOGSIZE	Yes	Yes	A CS_INT	A CS_INT
SRV_S_MAXLISTENERS	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_NUMLISTENERS	No	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

Property	SET/ CLEAR	GET	bufp when cmd is CS_SET	bufp when cmd is CS_GET
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_TCP_RCVBUF	Yes	Yes	A CS_INT	A CS_INT
SRV_S_TCP_SNDBUF	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
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 3-87 lists the default value for each server property:

**Table 3-87: Legal properties and their default values (srv\_props)**

Property	Default
SRV_S_ALLOCFUNC	malloc()
SRV_S_APICLK	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 <i>Open Client and Open Server Configuration Guide</i> 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	<i>\$\$YBASE/interfaces</i>
SRV_S_LOGFILE	<i>srv.log</i>
SRV_S_LOGSIZE	Max integer value
SRV_S_MAXLISTENERS	CS_MAX_NOMAX
SRV_S_MSGPOOL	SRV_DEF_MSGPOOL
SRV_S_NETBUFSIZE	SRV_DEF_NETBUFSIZE
SRV_S_NETTRACEFILE	<i>sybnet.dbg</i>
SRV_S_NUMCONNECTIONS	SRV_DEF_NUMCONNECTIONS
SRV_S_NUMLISTENERS	N/A.
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
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	DSLISITEN environment variable

Property	Default
SRV_S_STACKSIZE	SRV_DEF_STACKSIZE
SRV_S_TCP_RCVBUF	Platform dependent.
SRV_S_TCP_SNDBUF	Platform dependent.
SRV_S_TDSVERSION	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 141

## 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 3-88 describes each value's significance:

**Table 3-88: Values for putflags (srv\_putmsgq)**

Value	Description
SRV_M_NOWAIT	When this flag is set, the call to <code>srv_putmsgq</code> returns immediately after the message is placed in the message queue.
SRV_M_WAIT	When SRV_M_WAIT is set, <code>srv_putmsgq</code> 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 3-89: 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;
    msgpp = 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 143.
- 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 143.

See also

srv\_createmsgq, srv\_deletemsgq, srv\_getmsgq, srv\_getobjid

## srv\_realloc

**Description**                 Reallocate memory.

**Syntax**                     CS\_VOID\* srv\_realloc(mp,newsiz)

                              CS\_VOID         \*mp;  
                              CS\_INT          newsiz;

**Parameters**                *mp*  
                              A pointer to the old block of memory.

*newsiz*  
                              The number of bytes to reallocate.

**Return value**               **Table 3-90: 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 <i>newsiz</i> bytes.

### Examples

```
#include <ospublic.h>

/*
** Local Prototype.
*/
extern CS_RETCODE ex_srv_realloc PROTOTYPE((
CS_VOID         *mp,
CS_INT          newsiz
));

/*
** EX_SRV_REALLOC
**
**    Reallocate a memory chunk.
**
** Arguments:
**    mp            A pointer to existing memory block.
**    newsiz        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, newsiz)
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 *newsiz*e, 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.

*infp*

A pointer to a CS\_INT that is set to SRV\_I\_UNKNOWN if *srv\_recvpasssthr* returns CS\_FAIL. Table 3-91 describes the possible values returned in *\*infp* if *srv\_recvpasssthr* returns CS\_SUCCEED:

**Table 3-91: CS\_SUCCEED values (*srv\_recvpasssthr*)**

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 3-92: Return values (*srv\_recvpasssthr*)**

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

## Examples

```
#include      <ospublic.h>
/*
** Local prototype.
**/
CS_RETCODE   ex_srv_recvpasssthr 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 we were unsuccessful at receiving the packets.
**
**/
CS_RETCODE   ex_srv_recvpasssthr(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_SUCCEEDED && 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 *\*info*.
- 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, the executing thread cannot call any other routine that performs I/O until it has issued a srv\_senddone. No network I/O can occur on the event handler that executes srv\_recvpassthru.

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 3-93 describes the possible values returned in \**infop* if `srv_regcreate` returns CS\_FAIL:

**Table 3-93: Values for infop (srv\_regcreate)**

Value	Description
SRV_I_PEXISTS	The procedure is already registered.
SRV_I_UNKNOWN	Some other error occurred.

Return value **Table 3-94: 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 already 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 already 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. See “Registered procedures” on page 164.

## Return value

**Table 3-95: 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 3-96: Return values (srv\_regdrop)**

Returns	To indicate
CS_SUCCEEDED	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_SUCCEEDED 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_SUCCEEDED)
    {
        /* Open Server has set infop to a specific error. */
        return(CS_FAIL);
    }
    return(CS_SUCCEEDED);
}

```

**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\_regex

**Description** Execute a registered procedure.

**Syntax** `CS_RETCODE srv_regex(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 3-97 describes the possible values returned in \*infop if srv\_regex returns CS\_FAIL:

**Table 3-97: Values for infop (srv\_regex)**

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 3-98: Return values (srv\_regex)**

Returns	To indicate
CS_SUCCEED	The routine completed successfully.
CS_FAIL	The routine failed.

**Examples**

```
#include <ospublic.h>

/*
** Local Prototype
*/
CS_RETCODE      ex_srv_regex PROTOTYPE((
SRV_PROC        *spp,
CS_INT          &infop
));

/*
** EX_SRV_REGEX
**
```

```

** 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 3-99 describes the legal values for *options*:

**Table 3-99: 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 3-100: 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 already 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 3-101: Return values (srv\_reglist)**

Returns	To indicate
<code>CS_SUCCEED</code>	The routine completed successfully.
<code>CS_FAIL</code>	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 *proclstp* 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, proclstp)  
 SRV\_PROC \*spp;  
 SRV\_PROCLIST \*proclstp;

**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 3-102: 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 3-103 describes the possible values returned in *\*infop* if `srv_regnowatch` returns `CS_FAIL`:

**Table 3-103: 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 3-104: 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 204 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 3-105: 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 204

## 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 3-106 describes the legal values for *options*:

**Table 3-106: 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 <i>srv_regwatch</i> is used to remove the thread from the procedure's notification list.

*infp*

Table 3-107 describes the possible values returned in *\*infp* if *srv\_regwatch* returns CS\_FAIL:

**Table 3-107: Values for infp (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 <i>options</i> 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 already on the notification list for the specified procedure.

Return value

**Table 3-108: 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_SUCCEEDED    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 already 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 3-109: 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_reglstfree`

## 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 3-110: 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 <i>lenp</i> 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)`



Parameters

```

SRV_PROC   *spp;
CS_INT     *lenp;

```

*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 3-111: 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 <i>lenp</i> 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_SUCCEEDED      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 already raised.
        */
        return CS_FAIL;
    }

    /*
    ** Copy the RPC name to the output buffer.
    */
    (void)strncpy(buf, np, buflen);

    /*
    ** All done.
    */
    return CS_SUCCEEDED;
}

```

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 Enterprise 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 3-112: 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 <i>lenp</i> 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)`

Parameters      `SRV_PROC      *spp;`  
                   `spp`  
                   A pointer to an internal thread control structure.

Return value      **Table 3-113: 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 Enterprise.
- 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 3-114: 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 <i>lenp</i> 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 3-115: Return values (srv\_run)**

Returns	To indicate
CS_SUCCEED	The server is stopped.
CS_FAIL	Open Server could not start the server. If <i>srv_run</i> returns CS_FAIL, an application must call <i>srv_init</i> before calling <i>srv_run</i> 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 93

## srv\_s\_ssl\_local\_id

Description	Properties used to specify the path to the local ID (certificates) file.
Syntax	<pre>typedef struct _cs_sslid {     CS_CHAR    *identity_file;     CS_CHAR    *identity_password; } CS_SSLIDENTITY</pre>
Parameters	<p><i>identity_file</i> provides a path to the file containing a digital certificate and the associated private key.</p> <p><code>CS_GET</code> only returns the <i>identity_file</i> used, and only if it is set with <code>CS_CONNECTION</code>.</p> <p><i>identity_password</i> used to decrypt the private key.</p>

## srv\_select (UNIX only)

Description	Check to see if a file descriptor is &ready for a specified I/O operation.
Syntax	<code>CS_INT</code> <code>srv_select(nfds, &amp;readmaskp, writemaskp, exceptmaskp, waitflag)</code>

```

CS_INT          nfd;
SRV_MASK_ARRAY *readmask;
SRV_MASK_ARRAY *writemask;
SRV_MASK_ARRAY *exceptmask;
CS_INT          waitflag;
    
```

Parameters

*nfd*

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 3-116: 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 3-117 summarizes the legal values for *waitflag*:

**Table 3-117: 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 <b>srv_send_ctlinfo</b> (SRV_PROC * <i>srvproc</i> , CS_INT <i>ctl_type</i> , SRV_CTL_MIGRATE <i>ctl_type</i> , CS_INT <i>paramcnt</i> , SRV_CTLITEM * <i>param</i> )
Parameters	<p><i>srvproc</i> A pointer to an internal thread control structure.</p> <p><i>ctl_type</i> The type of control message to send.</p> <p><i>paramcnt</i> The number of elements in the <i>param</i> array.</p> <p><i>param</i> Parameters for library control message.</p>
Usage	<ul style="list-style-type: none"> <li>• <i>ctl_type</i> has the following values: <ul style="list-style-type: none"> <li>• 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.</li> <li>• 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.</li> <li>• SRV_CTL_HAUPDATE Valid at any time <i>srv_sendinfo</i> 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 <i>param</i>.</li> </ul> </li> <li>• <i>param</i> has the following fields: <ul style="list-style-type: none"> <li>• SRV_CTLTYPES <i>srv_ctlitemtype</i>, where <i>srv_ctlitemtype</i> indicates the parameter type. The following types are available: <ul style="list-style-type: none"> <li>• <i>SRV_CT_SERVERNAME</i>, which indicates that <i>srv_ctlptr</i> points to a CS_CHAR string containing the name of the server whose address will be looked up.</li> </ul> </li> </ul> </li> </ul>

- *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 formatted 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_freesserveraddrs`, `srv_getserverbyname`

## srv\_send\_data

Description	<i>srv_send_data</i> 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	<pre> CS_RETCODE <i>srv_send_data</i>(<i>spp</i>, <i>column</i>, <i>buf</i>, <i>buflen</i>) SRV_PROC   *<i>spp</i>; CS_INT     *<i>column</i>; CS_BYTE    *<i>buf</i>; CS_INT     <i>buflen</i>;         </pre>
Parameters	<p><i>spp</i> A pointer to an internal thread control structure.</p> <p><i>column</i> The number of the column in a row set.</p> <p><i>buf</i> A pointer to a buffer containing the data to send to the client. This determines the size of a section.</p> <p><i>buflen</i> The length of the <i>*buf</i> buffer.</p>

Return value

**Table 3-118: 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 *rlen;          /* 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 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 3-119: 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_SUCCEEDED)
{
    /* Write text to client a chunk at a time */
    if (srv_send_text(spp, data, len_read) != CS_SUCCEEDED)
    {
        ok = CS_FALSE;
        break;
    }
}

switch(ret)
{
case CS_SUCCEEDED: /* 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_SUCCEEDED : 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 201

## 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 3-120 describes each flag:

**Table 3-120: 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 3-121 describes the legal values for *transtate*:

**Table 3-121: 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 3-122: Return values (srv\_senddone)**

Returns	To indicate
CS_SUCCEEDED	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_SUCCEEDED)
{
    /* 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_SUCCEEDED)
{
    /* 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_SUCCEEDED)
{
    /* 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_SUCCEEDED);

```



}

## 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 3-123 describes the legal values for *transtate*:

**Table 3-123: 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 3-124: 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.
- See “Client command errors” on page 38.
- Also 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 3-125 describes the possible values returned in *\*infop* if the routine returns `CS_SUCCEED`:

**Table 3-125: 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 3-126: 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_SENDDPASSTHRU
**
** 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 Enterprise 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 3-127: 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 3-128: 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_setcolotype` 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*, *ctrlinfo*, *ctrlrlen*)

```

SRV_PROC    *spp;
CS_INT      colnum;
CS_BYTE     *ctrlinfo;
CS_INT      ctrlrlen;

```

**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.

*ctrlinfo*

A pointer to the control data. Its length is given by the *ctrlrlen* parameter.

*ctrlrlen*

The length, in bytes, of the control data. There are, at most, `SRV_MAXCHAR` bytes of control information per column.

**Return value**

**Table 3-129: 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 3-130: 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 (TDS) 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_LOGININFO` structure and fill it with TDS information from the client thread.
  - b `ct_setloginfo` – Prepare a `CS_LOGININFO` 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_LOGININFO` structure.
  - d `srv_setloginfo` – Send the remote server’s response, retrieved in step 3, to the client, then release the `CS_LOGININFO` 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 3-131: 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 new
**             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 3-132: 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)srvc_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.

## srv\_sleep

Description Suspend the currently executing thread.

Syntax CS\_RETURNCODE 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 3-133 summarizes the legal values for *sleepflags*:

**Table 3-133: 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 3-134 describes the possible values returned in *\*infop* if *srv\_sleep* returns CS\_FAIL:

**Table 3-134: Values for *infop* (*srv\_sleep*)**

Value	Description
SRV_I_INTERRUPTED	The thread was woken unconditionally by <i>srv_ucwakeup</i> .
SRV_I_UNKNOWN	Some other error occurred. For example, the thread is already 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 3-135: 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      *sleepevt,
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	<ul style="list-style-type: none"> <li>• <code>srv_sleep</code> suspends the currently executing thread and initiates rescheduling. The thread will sleep until <code>srv_wakeup</code> is called on the same event.</li> <li>• Depending on the value of <i>sleepflags</i>, a thread that is sleeping can also wake up by receiving an attention.</li> <li>• A thread resumes execution on the statement just following the call to <code>srv_sleep</code>.</li> <li>• <code>srv_sleep</code> cannot be used in a <code>SRV_START</code> handler.</li> <li>• <code>srv_sleep</code> should not be called from interrupt level code. Any number of problems could occur if this rule is violated.</li> <li>• Call <code>srv_capability</code> to determine whether your platform supports preemptive scheduling.</li> <li>• The <i>reserved1</i> 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 <i>Open Client and Open Server Programmers Supplement</i> for your platform for an example of preemptive scheduling.</li> </ul>
See also	<code>srv_wakeup</code>

## srv\_spawn

Description	Allocate a service thread.
Syntax	<pre>CS_RETCODE srv_spawn(sppp, stacksize, funcp,                     argp, priority)  SRV_PROC      **sppp; CS_INT        stacksize; CS_RETCODE    (*funcp)(); CS_VOID       *argp; CS_INT        priority;</pre>
Parameters	<p><i>sppp</i> A pointer to a thread structure pointer. If the call is successful, the address of an internal thread structure is returned in <i>sppp</i>.</p> <p><i>stacksize</i> The size of the stack; it must be at least <code>SRV_C_MINSTACKSIZE</code>. Specify <code>SRV_DEFAULT_STACKSIZE</code> to use the stack size set with <code>srv_props(SRV_S_STACKSIZE)</code>.</p>

*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. The entry-point function pointer should be *SRV\_C\_START\_LISTENER*.

*argp*

A pointer that is passed to the routine in *\*funcp* when the thread begins execution. If the entry-point function pointer (*funcp*) is *SRV\_C\_START\_LISTENER*, *argp* should point to a *CS\_TRANADDR* structure.

*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 3-136: Return values (*srv\_spawn*)**

Returns	To indicate
<i>CS_SUCCEED</i>	The routine completed successfully.
<i>CS_FAIL</i>	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.
- The `SRV_TLISTENER` thread type is used for dynamic listeners.
- A given host name can translate into multiple IPv4 and IPv6 addresses. Starting a dynamic listener can therefore cause multiple threads to be created. The only way to obtain `SRV_PROC` pointers for these threads is by using the `SRV_LISTENER_PREBIND` and `SRV_LISTENER_POSTBIND` events.
- If the entry-point function pointer (*funcp*) is `SRV_C_START_LISTENER`, both *stacksize* and *priority* must be specified as `CS_UNUSED`, and *sppp* must be set to null.

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 3-137 describes the legal token types:

**Table 3-137: 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 3-138: 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 <i>type</i> or <i>symbol</i> . Open Server sets <i>lenp</i> 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 3-139 summarizes the tokens `srv_symbol` can convert:

**Table 3-139: 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). See “*CS\_BROWSEDESC* structure” on page 52.

**Return value****Table 3-140: Return values (*srv\_tabcolname*)**

Returns	To indicate
<i>CS_SUCCEED</i>	The routine completed successfully.
<i>CS_FAIL</i>	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_SUCCEEDED   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	<pre>CS_RETCODE srv_tabname(spp, tablenum, tablenamep,                         namelen)  SRV_PROC   *spp; CS_INT     tablenum; CS_CHAR    *tablenamep; CS_INT     namelen;</pre>
Parameters	<p><i>spp</i> A pointer to an internal thread control structure.</p> <p><i>tablenum</i> The number used to identify the table in subsequent calls to <code>srv_tabcolname</code>.</p> <p><i>tablenamep</i> A pointer to the name of the table. It cannot be null, as tables always have names.</p> <p><i>namelen</i> The length, in bytes, of the table name. If <i>namelen</i> is <code>CS_NULLTERM</code>, then Server Library expects the table name to be null terminated.</p>

Return value

**Table 3-141: 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 3-142: 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 3-143 summarizes the legal values for *cmd*:

**Table 3-143: 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 <code>srv_text_info</code> call will update a text or image column (inside Open Server) with the information in <code>iodescp</code> . (The application must have previously described the column using <code>srv_descfmt</code> .) Typically, this will be followed by a call to <code>srv_send_text</code> , or <code>srv_bind</code> and <code>srv_xferdata</code> .
CS_GET	Open Server is updating the <code>iodescp</code> 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 <code>srv_get_text</code> . 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 3-144: 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 structure 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 201

## 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 3-145 summarizes the legal values for *cmd*:

**Table 3-145: Values for *cmd* (srv\_thread\_props)**

Value	Meaning
CS_SET	The Open Server application is setting the property. In this case, <i>bufp</i> should contain the value the property is to be set to, and <i>buflen</i> should specify the size, in bytes, of that value.
CS_GET	The Open Server application is retrieving the property. In this case, <i>bufp</i> should point to the buffer where the property value is placed, and <i>buflen</i> 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, <i>bufp</i> , <i>buflen</i> , and <i>outlenp</i> are ignored.

*property*  
The property being set, retrieved or cleared.

*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 3-146: 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 already raised.
    */
    return CS_FAIL;
}
/*
** All done.
*/
return CS_SUCCEED;
}

```

**Usage**

- `srv_thread_props` is called to define, retrieve, and reset thread properties.
- Table 3-147 summarizes legal property values, whether they can be set or retrieved, and each value's datatype.

Refer to Table 2-28 on page 151 for descriptions of each thread property.

**Table 3-147: 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_LISTENADDR	No	Yes	Not applicable	A pointer to a CS_TRANADDR structure
SRV_T_LOCALE	Yes	Yes	A pointer to a CS_LOCALE pointer	A pointer to a CS_LOCALE pointer
SRV_T_LOCALID	Yes	No	Not applicable	A listener SSL certificate
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_REMOTEADDR	No	Yes	Not applicable	A pointer to a CS_TRANADDR structure
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_DELEGCREC	No	Yes	Not applicable	A pointer to a CS_VOID

<b>Property</b>	<b>SET/ CLEAR</b>	<b>GET</b>	<b>bufp when cmd is CS_SET:</b>	<b>bufp when cmd is CS_GET:</b>
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
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 3-148 lists the default values for the thread properties that can be defined (CS\_SET).

**Table 3-148: 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 151 for descriptions of each thread property.

See also `srv_props`, “Properties” on page 141

## srv\_timsleep

**Description** Sleep until an event is signalled or until the specified time expires. `srv_timsleep` is available in the reentrant libraries only.

**Syntax** `CS_RETCODE srv_timsleep(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 3-149: Return values (*srv\_timedsleep*)**

Returns	To indicate
CS_SUCCEED	The routine succeeded.
CS_FAIL	The routine failed. See the <i>infop</i> parameter.

## 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 3-150: 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 3-151: 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. See “CS-Library” on page 59.

*version*

The version of Open Server the application assumes is in effect.

## Return value

**Table 3-152: Return values (*srv\_version*)**

Returns	To indicate
CS_SUCCEEDED	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_155, &cp) != CS_SUCCEEDED)
{
    fprintf(stderr, "cs_ctx_alloc failed \n");
    exit(1);
}
if(srv_version(cp, CS_VERSION_155) != CS_SUCCEEDED)
{
    /*
    ** Release the context structure already 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 3-153 describes the legal values for *wakeflags*:

**Table 3-153: Values for wakeflags (srv\_wakeup)**

Value	Description
SRV_M_WAKE_INTR	The call to <code>srv_wakeup</code> is from interrupt level code. Failure to use this flag when calling <code>srv_wakeup</code> 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 3-154: 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_SUCCEEDED)
    {
        return(CS_FAIL);
    }
    return(CS_SUCCEEDED);
}

```

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 Programmers 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 3-155 describes the legal values for *cmd*:

**Table 3-155: Values for *cmd* (*srv\_xferdata*)**

Value	Description
CS_SET	The application is calling <code>srv_xferdata</code> to send data to a client.
CS_GET	The application is calling <code>srv_xferdata</code> to retrieve data from a client.

*type*

The type of data stored into or read from the program variable. Table 3-156 describes the valid types and their appropriate context:

**Table 3-156: 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 3-157: Return values (srv\_xferdata)**

Returns	To indicate
<i>CS_SUCCEED</i>	The routine completed successfully.
<i>CS_FAIL</i>	The routine failed.
<i>CS_PARAMS_MORE</i>	(for batched parameters) Parameters successfully copied. There are more parameters in the batch.
<i>CS_PARAMS_END</i>	(for batched parameters) Parameters successfully copied. This is the last set of parameters in the batch.

**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_SUCCEEDED
**      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).
- To enable batched parameter support, set the `SRV_S_PARAM_BATCHING` server property to `CS_TRUE`. Then, `srv_xferdata()` can receive multiple sets of parameters. For example, before `srv_run()`:

```

    if (srv_props (ctos_ctx->cx_context, CS_SET,
        SRV_S_PARAM_BATCHING, (CS_VOID *)&cs_true,
        sizeof (cs_true), NULL) != CS_SUCCEEDED)
    { ... }

```

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 111.

- 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`



# 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 164.

System registered procedure	Page
sp_ps	461
sp_regcreate	464
sp_regdrop	471
sp_reglist	472
sp_regnowatch	473
sp_regwatch	473
sp_regwatchlist	475
sp_serverinfo	475
sp_terminate	476
sp_who	478

## 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 Enterprise, 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 Enterprise.
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	<code>sp_regcreate proc_name, parm1, parm2, ...</code>
Parameters	<p><i>proc_name</i> The value of <i>proc_name</i> specifies the name of the registered procedure to be created.</p> <p><i>parm1, parm2, ...</i> (Optional) If the client application passes additional parameters, they specify the names, datatypes, and default values of the new procedure's parameters.</p>
Examples	<p>Calling <code>sp_regcreate</code> from a Client-Library Client</p> <p>This example creates a registered procedure <code>np_test</code> that takes parameters:</p> <ul style="list-style-type: none"> <li>• <code>@p1</code>, datatype <code>CS_INT</code>, no default value (that is, the value defaults to <code>NULL</code>)</li> </ul>

- @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_SUCCEEDED - 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_SUCCEEDED))
        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_SUCCEEDED)
{
    fprintf(stdout, "np_create: ct_param() @p3 failed\n");
    ok = CS_FALSE;
}
/*
** Send the RPC command.
*/
if (ok && ct_send(cmd) != CS_SUCCEEDED)
    ok = CS_FALSE;

/*
** Process the results from the RPC execution.
*/
if (ok && rpc_results(cmd, CS_FALSE) != CS_SUCCEEDED)
    ok = CS_FALSE;

return (ok ? CS_SUCCEEDED : 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_SUCCEEDED -- 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 Enterprise 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\_reglis

Description

List all registered procedures in Open Server.

Syntax

sp\_reglis

Examples

```
1>execute utility...sp_reglis
2>go
```

```
Procedure Name
-----
sp_who
```

```
sp_regwatch
sp_ps
sp_regdrop
sp_reglis
sp_regwatchlist
sp_regcreate
sp_regnowatch
```

```
(0 rows affected)
```

This isql example lists all of the currently registered procedures.

Usage

- sp\_reglis 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\_reglis 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	<code>sp_regnowatch proc_name</code>
Parameters	<i>proc_name</i> The name of the registered procedure.
Examples	<pre>dbrpcinit(dbproc, "sp_regnowatch", (DBUSMALLINT)         0); dbrpcparam(dbproc, "@proc_name", 0, SYBCHAR, 15,         15, "pricechange"); dbrpcsend(dbproc);</pre> <p>This example removes the client from the notification list for the pricechange registered procedure.</p>
Usage	<ul style="list-style-type: none"> <li>• This registered procedure executes when a client calls <code>dbregnowatch</code>.</li> <li>• A <code>SRV_C_PROCEXEC</code> callback handler can use <code>srv_rpcname</code> to determine that <code>sp_regnowatch</code> is executing and <code>sp_paramdata</code> to obtain the name of the procedure for which the notification request is to be removed.</li> </ul>
Messages	<pre>Notification request removed.</pre> <p>The notification request was removed successfully.</p> <pre>proc_name is not a registered procedure.</pre> <p>The procedure specified by <i>proc_name</i> is not registered in Open Server.</p> <pre>No requests pending.</pre> <p>The client had no notification requests pending for the procedure.</p> <pre>Unable to remove notification request.</pre> <p>Open Server failed to remove the notification request.</p>
See also	<code>sp_recreate</code> , <code>sp_regdrop</code> , <code>sp_regwatch</code> , <code>sp_regnowatch</code> , <code>sp_regwatch</code>

## sp\_regwatch

Description	Add the client to the notification list for a registered procedure.
Syntax	<code>sp_regwatch proc_name [options]</code>
Parameters	<i>proc_name</i> 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_reglis`, `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
<i>server_csname</i>	The name of the character set for the Open Server application will be sent as a one, single column, character row to the client.
<i>server_soname</i>	The name of the Open Server application sort order will be sent as one, single column, character row to the client.
<i>csdefinition</i>	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.
<i>sodefinition</i>	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 Enterprise `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 Enterprise, 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 Enterprise.

*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`



# Open Server Error Messages

Resolve configuration, development, deployment, and production problems that you may encounter when using Open Server.

See “Errors” on page 90 for information about Open Server error types, severity, and handling.

## Error 16000

Severity	10
Message text	Unknown option '%1!'
Explanation	The <i>options</i> parameter passed to <code>srv_reginit()</code> is invalid.
Action	Pass a valid option to <code>srv_reginit()</code> . See <code>srv_reginit</code> on page 364.
Additional information	Symbolic constant: <code>SRV_EOPTION</code>

## Error 16001

Severity	10
Message text	No free server processes.
Explanation	Open Server cannot allocate a service thread. This error may be caused by insufficient resources.
Action	Check the server log. Messages that indicate the underlying cause of the problem generally precede error 16001.
Additional information	Symbolic constant: <code>SRV_ENOPROC</code>

## Error 16002 (Deprecated)

Severity	N/A
Message text	Attempt to drop client server process '%1!' with active i/o.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ELIVEPROC

## Error 16003

Severity	10 or 15
Message text	Unknown event '%1!' requested for process '%2!'.
Explanation	An unknown event has been passed to <code>srv_event()</code> . This error may also occur during thread creation, which means that Open Server cannot assign an event to the new <code>SRV_PROC</code> .
Action	Pass only these events to <code>srv_event()</code> : <code>SRV_DISCONNECT</code> , <code>SRV_URGDISCONNECT</code> , <code>SRV_STOP</code> , user-defined events.
Additional information	Symbolic constant: <code>SRV_ENOEVENT</code>

## Error 16004

Severity	10
Message text	Unknown event '%1!' requested for process '%2!'.
Explanation	<code>srv_event()</code> received an event other than <code>SRV_DISCONNECT</code> , <code>SRV_URGDISCONNECT</code> , <code>SRV_STOP</code> , or a user-defined event.
Action	Pass only valid events to <code>srv_event()</code> .
Additional information	Symbolic constant: <code>SRV_EBADEVENT</code>

## Error 16005

Severity	10
Message text	Unknown event '%1!' in %2!.
Explanation	An application attempted to install an invalid event handler using <code>srv_handle()</code> .
Action	Install a standard Open Server event such as <code>SRV_CONNECT</code> or <code>SRV_LANGUAGE</code> , or a user-defined event that has been defined using <code>srv_define_event()</code> .
Additional information	Symbolic constant: <code>SRV_EBADUEVENT</code>

## Error 16006 (Deprecated)

Severity	N/A
Message text	Corrupt send buffer, <code>sleft &lt; 0 (%1!)</code> .
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_EIOBADBUF</code>

## Error 16007 (Deprecated)

Severity	N/A
Message text	Unknown i/o operation (%1!) requested in <code>srv__buf_io()</code> .
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_EIOBADTYPE</code>

## Error 16008

Severity	10, 15, or 20
----------	---------------

Message text	Memory allocation for '%1!' bytes failed in '%2!' for '%3!' allocation.
Explanation	Typically, this error occurs when Open Server cannot allocate the requested number of bytes in the specified function. Different Open Server routines can cause this error.
Action	Verify that the kernel and operating system parameters (ulimit -a) allow Open Server to allocate the memory it needs. Monitor the memory usage of Open Server to determine if there is a memory leak.
Additional information	Symbolic constant: SRV_EMEMORY

## Error 16009

Severity	20
Message text	Can't create server process for '%1!' process.
Explanation	makecontext() failed because of insufficient memory during the allocation of a stack for a new thread.
Action	Verify that the kernel and operating system parameters (ulimit -a) allow Open Server to allocate the memory it needs. Monitor the memory usage of Open Server to determine if there is memory leak.
Additional information	Symbolic constant: SRV_ENOPPROC

## Error 16010

Severity	10
Message text	Initialization failed for network '%1!' on machine '%2!'.
Explanation	The <i>sql.ini</i> or <i>interfaces</i> file includes an unknown network interface.
Action	Check the <i>sql.ini</i> and <i>interfaces</i> files to verify that Open Server is listening to a valid entry.
Additional information	Symbolic constant: SRV_EINITNET



## Error 16011 (Deprecated)

Severity	N/A
Message text	Bad interfaces file.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EIFILE

## Error 16012

Severity	10
Message text	Can't open %1! file '%2!'.
Explanation	Open Server cannot open the specified log file.
Action	Ensure that: <ul style="list-style-type: none"><li>• The file is not being used by another process.</li><li>• The file path and name is correct.</li><li>• Open Server has permission to read the file.</li></ul>
Additional information	Symbolic constant: SRV_EFOPEN

## Error 16013

Severity	20
Message text	Could not find server name '%1!' in interfaces file.
Explanation	Open Server cannot locate the specified server name in the <i>sql.ini</i> or <i>interfaces</i> file.
Action	Check the DSLISTEN environment variable and the server name specified at the command line. Ensure that the server name matches the case and spelling of the server specified in the <i>sql.ini</i> or <i>interfaces</i> file.
Additional information	Symbolic constant: SRV_ESRVNAME

## Error 16014 (Deprecated)

Severity	N/A
Message text	Could not find the user's (%!) home directory.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EHOMEDIR

## Error 16015 (Deprecated)

Severity	N/A
Message text	Unknown process type '%!'' in routine '%2!''.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EPROCTYPE

## Error 16016

Severity	15
Message text	No free messages.
Explanation	Open Server cannot allocate or send a message into a queue, or retrieve a message from a queue.
Action	Use the Open Server trace flag SRV_TR_MSGQ to trace message queue activity. Adjust the srv_props(SRV_S_MSGPOOL) property as appropriate.
Additional information	Symbolic constant: SRV_ENOMSGS

## Error 16017 (Deprecated)

Severity	N/A
----------	-----

Message text	%1! process '%2!' for message from process '%3!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EMMSGPROC

## Error 16018 (Deprecated)

Severity	N/A
Message text	Bad message %1! process '%2!' for message from process '%3!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EMMSGADDRESS

## Error 16019

Severity	10
Message text	Bad message type received by %1!
Explanation	srv_negotiate() received an invalid message type while reading negotiated login information from a client application.
Action	Run Ribo between the client application and Open Server to see the message type sent by the client. Check this against the valid message types listed in Table 3-71 on page 324.
Additional information	Symbolic constant: SRV_EMMSGTYPE

## Error 16020 (Deprecated)

Severity	N/A
Message text	Bad event type '%1!' from srv__runnet() for client

## Error 16021 (Deprecated)

---

	'%2!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBADIOEVENT

## Error 16021 (Deprecated)

Severity	N/A
Message text	I/O error '%1!' (%2!) for client process '%3!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EIOERROR

## Error 16022 (Deprecated)

Severity	N/A
Message text	Bad server process id '%1!' in %2!
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EQPROC

## Error 16023 (Deprecated)

Severity	N/A
Message text	No free run queue structures.
Explanation	N/A
Action	N/A

Additional information      Symbolic constant: SRV\_EQFREE

## **Error 16024 (Deprecated)**

Severity                    N/A  
Message text                Non-event message received by client process spid %1!  
Explanation                N/A  
Action                       N/A  
Additional information      Symbolic constant: SRV\_ESRVEVENT

## **Error 16025 (Deprecated)**

Severity                    N/A  
Message text                Unknown message type '%1!' given in %2!  
Explanation                N/A  
Action                       N/A  
Additional information      Symbolic constant: SRV\_ECLNTMSG

## **Error 16026 (Deprecated)**

Severity                    N/A  
Message text                Can't create listen socket.  
Explanation                N/A  
Action                       N/A  
Additional information      Symbolic constant: SRV\_ESOCKET

## Error 16027 (Deprecated)

Severity	N/A
Message text	Can't bind socket.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBIND

## Error 16028 (Deprecated)

Severity	N/A
Message text	Can't find host '%1!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EGETHOST

## Error 16029

Severity	20
Message text	Failed to start any network listeners.
Explanation	Open Server cannot listen for client connections on the specified port.
Action	Use <code>netstat -an</code> to ensure that the port associated with your server's <i>interfaces</i> entry is not in use. If the port is not in use, check the error log for other errors.
Additional information	Symbolic constant: SRV_ELISTEN

## Error 16030 (Deprecated)

Severity	N/A
----------	-----

Message text	Can't connect client socket.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECNTSOCKET

## **Error 16031 (Deprecated)**

Severity	N/A
Message text	Server process '%1!' is not active in '%2!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EPROCDEAD

## **Error 16032**

Severity	10
Message text	Bad STRCHR (%1!) passed to '%2!'.
Explanation	This error may occur when the internal function pointer array used for translating between different data formats (for example, byte-swapping) has specified an invalid index value.
Action	Verify that the LOGINREC passed to Open Server has not been corrupted.
Additional information	Symbolic constant: SRV_EBADINDEX

## **Error 16033**

Severity	10
Message text	Can't convert from type '%1!' to type '%2!'.

Explanation	The client application sent data that cannot be converted to the datatype that CS_DATAFMT described in <code>srv_bind()</code> or <code>srv_alt_bind()</code> .
Action	Ensure that client applications send only supported datatypes. See the datatype conversion chart in <i>Common Libraries Reference Manual</i> .
Additional information	Symbolic constant: SRV_ENOCONVERT

## Error 16034

Severity	10
Message text	%1! names may not be longer than %2! bytes (%3! given).
Explanation	The column name is longer than CS_MAX_NAME.
Action	See <i>cstypes.h</i> for the current definition of CS_MAX_NAME.
Additional information	Symbolic constant: SRV_EMAXNAME

## Error 16035 (Deprecated)

Severity	N/A
Message text	Conversion from type '%1!' to type '%2!' resulted in underflow.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECUFLOW

## Error 16036 (Deprecated)

Severity	N/A
Message text	Conversion from type '%1!' to type '%2!' resulted in overflow.
Explanation	N/A



Action	N/A
Additional information	Symbolic constant: SRV_ECOFLOW

## **Error 16037 (Deprecated)**

Severity	N/A
Message text	Conversion to a binary variable resulted in overflow.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECBINOFLOW

## **Error 16038 (Deprecated)**

Severity	N/A
Message text	Conversion from type '%1!' to type '%2!' resulted in loss of precision.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECPREC

## **Error 16039 (Deprecated)**

Severity	N/A
Message text	Conversion from type '%1!' to type '%2!' stopped by syntax error in source field.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECSYNTAX

## Error 16040 (Deprecated)

Severity	N/A
Message text	Illegal integer column length '%1!'. Legal values are 1, 2, and 4 bytes.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBADLEN

## Error 16041 (Deprecated)

Severity	N/A
Message text	Unknown variable-length datatype '%1!' encountered.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EUVDT

## Error 16042 (Deprecated)

Severity	N/A
Message text	Unknown fixed-length datatype '%1!' encountered.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EUFDT

## Error 16043

Severity	10
----------	----

Message text	Attempted to send row without row description.
Explanation	srv_alt_xferdata(), srv_send_text(), or srv_send_data() attempted to send a row without sending the corresponding row description.
Action	If srv_alt_xferdata() failed, call srv_alt_descfmt() to describe the row. Otherwise, call srv_descfmt().
Additional information	Symbolic constant: SRV_ENOROWDESC

## Error 16044

Severity	10
Message text	The %1! datatype '%2!' is not a known datatype.
Explanation	An invalid datatype has been defined in the CS_DATAFMT or CS_IODESC structure.
Action	Validate the CS_DATAFMT or CS_IODESC structure datatypes.
Additional information	Symbolic constant: SRV_ENOTYPE

## Error 16045

Severity	10
Message text	%1!: There are no dataitems for %2!
Explanation	The client application sent a query containing a COMPUTE clause, but Open Server did not call srv_alt_descfmt().
Action	Ensure that Open Server calls srv_alt_descfmt() to describe the COMPUTE column format to the client application.
Additional information	Symbolic constant: SRV_ENOITEM

## Error 16046

Severity	10 or 20
----------	----------

Message text	<code>%1!: results state error - %2! already sent to client.</code>
Explanation	This problem may occur during a call to <code>srv_negotiate()</code> when a negotiated login directive is sent without sending <code>srv_senddone()</code> between calls to <code>srv_negotiate()</code> .
Action	Send <code>srv_senddone(SRV_DONE_FINAL)</code> or <code>srv_senddone(SRV_DONE_MORE)</code> between calls to <code>srv_negotiate()</code> . Also, check the Open Server application code to ensure that parameters are not returned after the final <code>srv_senddone()</code> call.
Additional information	Symbolic constant: <code>SRV_ESTATSTATE</code>

## Error 16047

Severity	10
Message text	<code>No such trace flag (%1!)</code>
Explanation	<code>srv_props(SRV_S_TRACEFLAG)</code> received an invalid trace flag.
Action	Verify that only these trace flags are passed to <code>srv_props(SRV_S_TRACEFLAG)</code> : <code>SRV_TR_TDSHDR</code> , <code>SRV_TR_TDSDATA</code> , <code>SRV_TR_MSGQ</code> , <code>SRV_TR_ATTEN</code> , <code>SRV_TR_TDSTOKEN</code> , <code>SRV_TR_EVENT</code> , <code>SRV_TR_NETPARAM</code> , <code>SRV_TR_NETREQ</code> , <code>SRV_TR_NETDRIVER</code> , <code>SRV_TR_NETMEM</code> , <code>SRV_TR_NETDATA</code> , <code>SRV_TR_NETWAKE</code> , <code>SRV_TR_DEFQUEUE</code> .
Additional information	Symbolic constant: <code>SRV_ENOTRACE</code>

## Error 16048

Severity	10
Message text	<code>There is no current remote procedure call.</code>
Explanation	Open Server has attempted to process a remote procedure call (RPC) from outside an <code>SRV_RPC</code> event handler, and an RPC or RPC parameter has not been defined.
Action	Call <code>srv_rpcoptions()</code> only from the <code>SRV_RPC</code> event handler.
Additional information	Symbolic constant: <code>SRV_ENORPC</code>

## Error 16049

Severity	10
Message text	%1!: There is no dataitem #%2! for %3!
Explanation	The column number specified in a routine that returns column or parameter information, such as blk_colval(), is invalid.
Action	Specify a column number that exists in the table definition.
Additional information	Symbolic constant: SRV_ENOITEMNO

## Error 16050

Severity	10
Message text	Rpc parameter #%1! is not a return parameter.
Explanation	A parameter that has been designated as a return parameter during a call to srv_bind(), srv_descfmt(), or srv_xferdata() is not a return parameter.
Action	Check the definition of the RPC and ensure that the parameter is a return parameter.
Additional information	Symbolic constant: SRV_ENORETPARAM

## Error 16051 (Deprecated)

Severity	N/A
Message text	Illegal length '%1!' for type '%2!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBADPARMLEN

## Error 16052

Severity	10
Message text	Actual length '%1!' cannot be greater than maximum length '%2!'.
Explanation	The column name specified in <code>srv_describe()</code> is longer than <code>CS_MAX_NAME</code> .
Action	Ensure that the column name length is smaller than or equal to <code>CS_MAX_NAME</code> .
Additional information	Symbolic constant: <code>SRV_EBADPARAMMAXLEN</code>

## Error 16053

Severity	10
Message text	Unknown %1! '%2!' passed to <code>srv_symbol()</code> .
Explanation	<code>srv_symbol()</code> received an unknown symbol.
Action	See Table 3-139 on page 433 for a list of convertible symbols.
Additional information	Symbolic constant: <code>SRV_ESYMBOL</code>

## Error 16054

Severity	10
Message text	No such server field (%1!).
Explanation	The field specified in a call to <code>srv_sfield()</code> does not exist.
Action	<code>srv_sfield()</code> is not intended for customer use; contact Sybase Technical Support.
Additional information	Symbolic constant: <code>SRV_ESRVTYPE</code>

## Error 16055

Severity	10
----------	----

Message text	No such server process field (%1!).
Explanation	The field specified during a call to <code>srv_pfield()</code> does not exist.
Action	<code>srv_pfield()</code> is not intended for customer use; contact Sybase Technical Support.
Additional information	Symbolic constant: <code>SRV_ESRVPROCTYPE</code>

## **Error 16056**

Severity	10
Message text	NULL %1! pointer passed to %2!
Explanation	An API that does not accept NULL parameters has received a NULL pointer.
Action	Verify that the parameter passed to the specified API is not NULL.
Additional information	Symbolic constant: <code>SRV_ENULLPTR</code>

## **Error 16057**

Severity	10
Message text	Down level TDS -- attempt to send status, ENVCHANGE or parameters ignored.
Explanation	The client application does not support the status, TDS environment change token, or parameter that Open Server attempted to send.
Action	Update the client application's Tabular Data Stream™ (TDS) level if the objects sent by Open Server are required.
Additional information	Symbolic constant: <code>SRV_ETDSVERSION</code>

## **Error 16058**

Severity	20
Message text	Could not install network ATTENTION/OOB handler -- fatal error.

## Error 16059 (Deprecated)

---

Explanation	This error is obsolete and no longer used.
Action	Upgrade the Open Server application to a supported version.
Additional information	Symbolic constant: SRV_EOOB

## Error 16059 (Deprecated)

Severity	N/A
Message text	TCP Select found no sockets had sent SIGURG -- SIGURG ignored.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENO_OOB

## Error 16060 (Deprecated)

Severity	N/A
Message text	TCP socket SIGURG setup failed due to condition %!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ESIGURG_SETUP

## Error 16061

Severity	15
Message text	Thread %! is not a site-handler. Type = %2!, function = %3!.
Explanation	Open Server expected a SRV_SITEHANDLER thread for the specified function.



Action	Use the <code>SRV_T_LOGINTYPE</code> property to verify that the login record received during the login process is <code>SRV_SITEHANDLER</code> .
Additional information	Symbolic constant: <code>SRV_ENOTSITE</code>

## Error 16062

Severity	10
Message text	Bad configuration option value (%1!) for option %2!.
Explanation	<code>srv_config()</code> , a deprecated command, received an invalid option.
Action	Correct the datatype of the configuration option passed to <code>srv_config()</code> . For example, if you are setting a boolean option, pass <code>CS_TRUE</code> or <code>CS_FALSE</code> .
Additional information	Symbolic constant: <code>SRV_EBADCONFIGOPT</code>

## Error 16063

Severity	10
Message text	Object name syntax error. Use <code>site.db.owner.object;1</code> .
Explanation	Open Server received an RPC that did not match the form <code>site.db.owner.object;1</code> .
Action	Enable the Open Server trace flags <code>SRV_TR_EVENT</code> and <code>SRV_TR_TDSDATA</code> to see the format of the RPC, and implement the necessary corrections.
Additional information	Symbolic constant: <code>SRV_EOBSYNTAX</code>

## Error 16064 (Deprecated)

Severity	N/A
Message text	Network termination failed.
Explanation	N/A

Action	N/A
Additional information	Symbolic constant: SRV_ENETTERM

## Error 16065

Severity	10
Message text	<code>%1!: illegal status - %2!</code>
Explanation	A structure has been assigned an invalid status.
Action	Certain structures contain a <i>status</i> field. For valid status values, see: <ul style="list-style-type: none"><li>• Table 2-9 on page 57 for the CS_DATAFMT structure</li><li>• Table 2-10 on page 62 for CS_SERVERMSG</li><li>• Table 2-13 on page 69 for SRV_CURDESC</li></ul>
Additional information	Symbolic constant: SRV_EBADSTATUS

## Error 16066 (Deprecated)

Severity	N/A
Message text	<code>The current client command is an rpc -- use srv__paramset() to return parameters.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EREVALRPC

## Error 16067

Severity	10
Message text	<code>Illegal length (%1!) for datatype '%2!'.</code>
Explanation	Specified datatype length is illegal.

Action	See Table 2-37 on page 205 for a description of the Open Server and Open Client datatypes.
Additional information	Symbolic constant: SRV_EBADTYPELEN

## **Error 16068 (Deprecated)**

Severity	N/A
Message text	Attempt to drop a thread that is currently free.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EPROCFREE

## **Error 16069 (Deprecated)**

Severity	N/A
Message text	Read message for an invalid channel in %1!
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EINVCHAN

## **Error 16070 (Deprecated)**

Severity	N/A
Message text	No server log file open.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOLOGFILE

## Error 16071 (Deprecated)

Severity	N/A
Message text	Column number must be greater than %1!
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECOLNUMBER

## Error 16072

Severity	10
Message text	%1!: Item no '%2!' has already been described.
Explanation	Open Server has called <code>srv_descfmt()</code> more than once to retrieve the information of the same column or parameter.
Action	Check the parameter or column name passed to <code>srv_descfmt()</code> to verify that it is correct.
Additional information	Symbolic constant: SRV_EDUPITEM

## Error 16073

Severity	10
Message text	Description is not complete for %1! %2! for %3!.
Explanation	<code>srv_send_data()</code> , <code>srv_alt_xferdata()</code> , or <code>srv_xferdata()</code> has detected incomplete descriptions for some of the columns in the row being sent.
Action	Ensure that Open Server completely describes every column that it sends.
Additional information	Symbolic constant: SRV_EINCDESC

## Error 16074

Severity	10
Message text	The configured maximum number of user events (%1!) have already been defined.
Explanation	srv_define_event() has failed because adding a new event exceeds the maximum number of events the server is configured for.
Action	Use srv_props() to increase SRV_S_NUMUSEREVENTS.
Additional information	Symbolic constant: SRV_EMAXUEVENT

## Error 16075

Severity	10
Message text	Bad event type (%1!).
Explanation	Either an invalid event handler has called srv_event() or an invalid event type had been passed to srv_define_event().
Action	If the cause is an invalid event handler, see Table 3-76 on page 331 for a list of valid events. If the cause is an invalid event type, pass SRV_EQUEUED, which, is currently the only valid event type for srv_define_event().
Additional information	Symbolic constant: SRV_EEVENTTYPE

## Error 16076

Severity	10
Message text	Can't call %1! from an attention-handler.
Explanation	Open Server has called the specified function from the SRV_ATTENTION handler.
Action	Call only these commands (with certain restrictions) from the attention handler: srv_wakeup(), srv_ucwakeup(), srv_thread_props(), srv_props() and srv_event_deferred(). See “Attention events” on page 20.
Additional information	Symbolic constant: SRV_EINATTN

## Error 16077 (Deprecated)

Severity	N/A
Message text	Can't call %1! from a SRV_CONNECT handler.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EINCONNECT

## Error 16078 (Deprecated)

Severity	N/A
Message text	Received blocking read from site-handler '%1!'. Received blocking read from site-handler '%1!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBLOCKREAD

## Error 16079 (Deprecated)

Severity	N/A
Message text	Could not install network IO INTERRUPT handler -- fatal error.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ESIGIO

## Error 16080

Severity	20
----------	----

Message text	Deferred wakeup queue is full. Increase its size using SRV_S_DEFQUEUESIZE server property.
Explanation	Open Server has attempted to use <code>srv_deferred_event()</code> to add an event to a full deferred event queue. This error may also be generated if the Open Server scheduler does not run long enough to process all the deferred events in the queue.
Action	Increase the value of the SRV_S_DEFQUEUESIZE property. If you have a time-consuming routine that does not sleep or perform network I/O, occasionally call <code>srv_yield()</code> to prevent the routine from monopolizing the server.
Additional information	Symbolic constant: SRV_EDEFQFULL

## Error 16081 (Deprecated)

Severity	N/A
Message text	Warning: %1! for signal %2! ignored because a Sybase installed handler exists for this signal.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EFLAGIGN

## Error 16082 (Deprecated)

Severity	N/A
Message text	The signal handler for %1! will be executed by a Sybase installed handler for the same signal.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ESIGSUPERCEDE

## Error 16083 (Deprecated)

Severity	N/A
Message text	Unused error define.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDEFUNUSED

## Error 16084 (Deprecated)

Severity	N/A
Message text	Unknown task type %1! encountered in %2! while dropping server.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBADPROCTYPE

## Error 16085 (Deprecated)

Severity	N/A
Message text	Could not allocate memory for network control structure for DECNET.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDNAALLFAIL

## Error 16086 (Deprecated)

Severity	N/A
----------	-----



Message text	Failed to create mailbox for DECNET listener.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDNACRELSNMBX

## **Error 16087 (Deprecated)**

Severity	N/A
Message text	Assign to _NET failed for DECNET. Cannot start network.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDNANETASSIGN

## **Error 16088 (Deprecated)**

Severity	N/A
Message text	Port name %1! already in use for DECNET. Cannot start network.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDNADUPOBJ

## **Error 16089 (Deprecated)**

Severity	N/A
Message text	Declare object for object %1! failed for DECNET. Cannot start network.
Explanation	N/A

Action	N/A
Additional information	Symbolic constant: SRV_EDNADECLARE

## **Error 16090 (Deprecated)**

Severity	N/A
Message text	Read failed for listen on DECNET.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDNALISTEN

## **Error 16091 (Deprecated)**

Severity	N/A
Message text	Failed to assign a new channel for a client connection.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDNAASSIGN

## **Error 16092 (Deprecated)**

Severity	N/A
Message text	Failed to complete new client connection.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDNAACCESS

## Error 16093 (Deprecated)

Severity	N/A
Message text	No link table entries available for client.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDNANOLINK

## Error 16094

Severity	10
Message text	Attempt to drop non-service process %1!
Explanation	srv_dropproc(), which is a deprecated command, has attempted to drop a thread type other than SRV_TSERVICE.
Action	Use srv_termproc() instead of srv_dropproc(). For a list of thread types, see Table 2-33 on page 164.
Additional information	Symbolic constant: SRV_ENOTSERVICE

## Error 16095 (Deprecated)

Severity	N/A
Message text	Attempt to send non-user-defined-event from process %1! to service process %2!
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EEVENTNOSERVICE

## Error 16096 (Deprecated)

Severity	N/A
Message text	Error processing deferred event notice for process %1!
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDEFHANDLER

## Error 16097

Severity	10
Message text	Rpc parameters must all either be passed by name or positionally.
Explanation	This error occurs during a call to <code>srv_numparams()</code> when one parameter has a name and the others do not.
Action	When defining parameters, supply a name to either all or none of the parameters.
Additional information	Symbolic constant: SRV_ERPCINCONSIST

## Error 16098 (Deprecated)

Severity	N/A
Message text	A subchannel setup failed because a process could not be allocated.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECHANSETUP

## Error 16099 (Deprecated)

Severity	N/A
Message text	Couldn't create thread for the scheduler.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOSCHEDPROC

## Error 16100

Severity	20
Message text	Deferred interrupt handler message queue failure in <code>srv__rundef()</code> , <code>status = %1!</code> .
Explanation	An internal Open Server call to <code>srv_getmsgq(SRV_M_WAIT)</code> has failed.
Action	Check the <i>infop</i> parameter for the cause of the failure, and correct it. Table 3-56 on page 298 lists the <i>infop</i> values.
Additional information	Symbolic constant: SRV_EDEFERFAIL

## Error 16101 (Deprecated)

Severity	N/A
Message text	Invalid message received by deferred interrupt handler in <code>srv__rundef()</code> . <code>type = %1!</code> , <code>from = %2!</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDEFERINVMMSG

## Error 16102 (Deprecated)

Severity	N/A
Message text	Invalid event type received by a site handler.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EINVEVENT

## Error 16103

Severity	10
Message text	Invalid stack size, %!!, passed to <code>srv_spawn()</code> .
Explanation	The stack size passed to <code>srv_spawn()</code> is too small.
Action	Ensure that the value of the <i>stacksize</i> parameter is at least equal to the value of <code>SRV_MINSTACKSIZE</code> .
Additional information	Symbolic constant: SRV_EINVSTACK

## Error 16104

Severity	10 or 20
Message text	Unable to allocate stack, size %!!, for new thread.
Explanation	The stack size passed to <code>srv_spawn()</code> is too large.
Action	Ensure that the value of the <i>stacksize</i> parameter is smaller than or equal to the maximum stack size, which is the amount of memory that can be allocated by a call to <code>malloc</code> .
Additional information	Symbolic constant: SRV_ESTACKALLOC

## Error 16105

Severity	10
Message text	Invalid priority, %1!, passed to %2!. Valid priorities are in the range from %3! to %4!.
Explanation	srv_spawn() or srv_setpri() has received an invalid <i>priority</i> value.
Action	Pass a <i>priority</i> value that is between SRV_C_LOWPRIORITY and SRV_C_MAXPRIORITY.
Additional information	Symbolic constant: SRV_EINVPRIORITY

## Error 16106

Severity	10
Message text	Failed to allocate memory for free process queue.
Explanation	Open Server ran out of memory while extracting security mechanism names.
Action	Increase or remove any restrictions for memory allocation for the user running the Open Server application.
Additional information	Symbolic constant: SRV_EALLOCFAIL

## Error 16107 (Deprecated)

Severity	N/A
Message text	Unable to allocate message for protocol acknowledge.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOPRACK

## Error 16108

Severity	10
Message text	%1! was called with a NULL %2! parameter.
Explanation	A procedure expecting a non-NULL parameter has received a NULL parameter.
Action	Ensure that the parameter identified in the error message is not NULL.
Additional information	Symbolic constant: SRV_ENULLPARAM

## Error 16109

Severity	10
Message text	%1! may only be called after %2! has returned successfully.
Explanation	The specified procedure has been called before <code>srv_regcreate()</code> , <code>srv_regdefine()</code> , or <code>srv_reginit()</code> has completed.
Action	Write the application so the specified procedure is not called until the proper initialization commands complete.
Additional information	Symbolic constant: SRV_EPMSTAT

## Error 16110

Severity	10
Message text	%1! was called with a bad %2! parameter.
Explanation	A parameter passed to an Open Server API has an invalid value.
Action	Many parameters passed to Open Server APIs are limited to specific values or ranges. Confirm that the parameter passed to the specified API falls into the specific range or values.
Additional information	Symbolic constant: SRV_EBADPARAM



## Error 16111

Severity	10
Message text	<code>SRV_PROC passed to %! is not a Client thread.</code>
Explanation	<code>srv_regnowatch()</code> , <code>srv_regwatch()</code> , <code>srv_regwatchlist()</code> , or <code>srv_sendinfo()</code> has received an <code>SRV_PROC</code> parameter that is not <code>SRV_TCLIENT</code> .
Action	Use <code>srv_thread_props(SRV_T_TYPE)</code> to retrieve the <code>SRV_PROC</code> thread type. Thread type must be <code>SRV_TCLIENT</code> .
Additional information	Symbolic constant: <code>SRV_EPNOTCLIENT</code>

## Error 16112

Severity	10
Message text	<code>Invalid object name found in %!.</code>
Explanation	An API that manipulates message queues or mutexes has received an object name with an invalid length.
Action	Change the name of your message queue or mutex so that its length is greater than zero but smaller than <code>SRV_MAXNAME</code> .
Additional information	Symbolic constant: <code>SRV_EOBJNAME</code>

## Error 16113

Severity	10
Message text	<code>Invalid object id found in %!.</code>
Explanation	The specified message queue or mutex identifier cannot be located.
Action	Determine the correct object ID by calling <code>srv_getobjid()</code> and passing the message queue or mutex name as a parameter.
Additional information	Symbolic constant: <code>SRV_EOBJID</code>

## Error 16114 (Deprecated)

Severity	N/A
Message text	Invalid lock mode found in %!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ELOCKMODE

## Error 16115

Severity	10
Message text	Could not start thread.
Explanation	A call to <code>srv_createproc()</code> or <code>srv_spawn()</code> prompted Open Server to create more threads than it is configured for.
Action	Call <code>srv_props(SRV_S_NUMTHREADS)</code> to configure Open Server to allow more threads than the default, which is defined by <code>SRV_DEF_NUMTHREADS</code> .
Additional information	Symbolic constant: SRV_ENOSPAWN

## Error 16116 (Deprecated)

Severity	N/A
Message text	Tried to restart thread %! which was not stopped.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EINRESTART

## Error 16117(Deprecated)

Severity	N/A
Message text	Tried to stop thread %1! which was executing.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EINSTOP

## Error 16118

Severity	10
Message text	Tried to a unlock a mutex that is not owned by thread %1!.
Explanation	The thread attempting to unlock a mutex through <code>srv_unlockmutex()</code> is not the current owner of the mutex.
Action	Track the mutexes that each thread creates. Currently, Open Server allows only one thread to lock or unlock a given mutex.
Additional information	Symbolic constant: SRV_ENOTOWNER

## Error 16119

Severity	10
Message text	Too many parameters for procedure %1! were received.
Explanation	<code>srv_regexec()</code> has detected an inconsistency between the number of parameters passed and the ones defined.
Action	Check the definition of the registered procedure and compare the defined number of parameters with the number of parameters that the client application passes.
Additional information	Symbolic constant: SRV_EPRNUM

## Error 16120

Severity	10
Message text	Must give name for parameter %1! for procedure %2!, once names have been used for other parameters.
Explanation	The client application passed an unnamed parameter after initially passing named parameters. When defining parameters, names must be supplied for either all or none of the parameters.
Action	Run Ribo between the client application and Open Server to identify named and unnamed parameters. Make appropriate corrections to the client application.
Additional information	Symbolic constant: SRV_EPARAMNAME

## Error 16121

Severity	10
Message text	Got bad datatype for parameter %1! for procedure %2!
Explanation	The client application has defined a parameter datatype that does not match the stored procedure definition.
Action	Ensure that the client application's definition of the parameter datatype matches that of the stored procedure.
Additional information	Symbolic constant: SRV_EPTYPE

## Error 16122

Severity	10
Message text	Attempted to unregister unknown procedure %1!
Explanation	Open Server cannot find the specified procedure during a call to <code>srv_regdrop()</code> .
Action	Check the <code>srv_regdrop()</code> <i>*infop</i> parameter for details.
Additional information	Symbolic constant: SRV_EPUNREGNOPROC.

## Error 16123

Severity	10
Message text	Attempted to execute procedure %!!, which was not registered.
Explanation	Open Server has attempted to execute a registered procedure that has not been registered.
Action	Check the <code>srv_regexec()</code> <i>*infop</i> parameter for details.
Additional information	Symbolic constant: <code>SRV_EPROCNOTREG</code>

## Error 16124

Severity	10
Message text	Cannot create %! %2! because it already exists.
Explanation	Open Server has attempted to create a named mutex, message queue, or registered procedure that already exists.
Action	Create a unique name for your mutex, message queue, or registered procedure.
Additional information	Symbolic constant: <code>SRV_EPEXISTS</code>

## Error 16125

Severity	10
Message text	Requested notification for unknown procedure %!!
Explanation	<code>srv_regwatch()</code> received a procedure that could not be found.
Action	Ensure that the procedure name passed to <code>srv_regwatch()</code> exists in Open Server, and that its case is consistent with that of the procedure name definition.
Additional information	Symbolic constant: <code>SRV_EPNOPROC</code>

## Error 16126

Severity	10
Message text	Client does not have a notification request for procedure %1!.
Explanation	srv_regnowatch() cannot find a thread that is waiting for notification from the specified procedure.
Action	Check the srv_regnowatch() <i>*infop</i> parameter for details.
Additional information	Symbolic constant: SRV_EPNONOTICEREQ

## Error 16127

Severity	10
Message text	Could not allocate a SRV_PROC to register system procedures.
Explanation	Open Server cannot allocate a new SRV_TSERVICE thread while the system registered stored procedures are being created or deleted.
Action	Look at preceding error messages that might explain the underlying cause of the error. One possible solution is to update SRV_S_NUMTHREADS to increase the maximum number of threads available to Open Server.
Additional information	Symbolic constant: SRV_EPNOSRVPROC

## Error 16128

Severity	10
Message text	Buffer size insufficient to hold %1! value in procedure %2!.
Explanation	Open Server routines such as srv_dynamic() and srv_options() return information through a buffer pointer parameter. If the buffer passed to the specified routine is not large enough, the error is generated. This error can also occur during a call to blk_colval() if the <i>*valuep</i> buffer is too small.

Action	For the correct buffer size, see Table 3-42 on page 277 (for <code>srv_dynamic()</code> ) or Table 3-80 on page 335 (for <code>srv_options()</code> ).
Additional information	Symbolic constant: <code>SRV_EBUFSIZE</code>

## Error 16129

Severity	10
Message text	<code>Cannot cancel notification request for procedure %! because this procedure does not exist.</code>
Explanation	<code>srv_regnowatch()</code> cannot find the specified procedure.
Action	Verify that the procedure name passed to <code>srv_regnowatch()</code> exists, and that its spelling and case are correct.
Additional information	Symbolic constant: <code>SRV_EPNOPROCEX</code>

## Error 16130

Severity	10
Message text	<code>Procedure %! expects parameter %!, which was not supplied.</code>
Explanation	Open Server cannot find the expected parameter for the stored procedure in the RPC call made by the client application.
Action	Verify the client application executing the procedure provides all required parameters. Run <code>Ribo</code> to see the parameters passed between the client application and Open Server.
Additional information	Symbolic constant: <code>SRV_ENEEDPARAM</code>

## Error 16131

Severity	10
Message text	<code>The named parameter %! does not exist for registered</code>

procedure %2!.

Explanation	Open Server cannot find the named parameter passed by the client application to the specified stored procedure.
Action	Run Ribo to see the parameters passed between the client application and Open Server.
Additional information	Symbolic constant: SRV_EUNKNOWNPARAM

## Error 16132

Severity	10
Message text	There is no current language command.
Explanation	srv_langcpy() or srv_langlen() has been called outside of the SRV_LANGUAGE or SRV_CURSOR event handler.
Action	Call srv_langcpy() or srv_langlen() only in the SRV_LANGUAGE or SRV_CURSOR event handler.
Additional information	Symbolic constant: SRV_ENOLANG

## Error 16133

Severity	10
Message text	Configuration of %1! connections has been exceeded, connection rejected.
Explanation	The last client connection request exceeded the maximum number of physical network connections the Open Server has been configured to accept.
Action	Increase the value of the Open Server property SRV_S_NUMCONNECTIONS to accommodate more client connections. The default is SRV_DEF_NUMCONNECTIONS.
Additional information	Symbolic constant: SRV_EMAXCONNECT



## Error 16134

Severity	10
Message text	The callback routine for %1! aborted the current operation.
Explanation	The callback routine of type SRV_C_PROCEXEC has returned SRV_S_INHIBIT.
Action	See “Using callback handlers with registered procedures” on page 169.
Additional information	Symbolic constant: SRV_EPCALLBACK

## Error 16135

Severity	10
Message text	The maximum number of parameters for procedure %1! have already been defined.
Explanation	A registered procedure can have a maximum of 1024 parameters. An attempt to describe a parameter through <code>srv_regparam()</code> has exceeded this number.
Action	Limit the number of your procedure’s parameter to 1024.
Additional information	Symbolic constant: SRV_EMAXPARAM

## Error 16136

Severity	10
Message text	A bad flag value was passed to routine %1!.
Explanation	<code>srv_getmsgq()</code> , <code>srv_putmsgq()</code> , <code>srv_lockmutex()</code> , or <code>srv_wakeup()</code> has received an invalid flag. This error may also occur if <code>srv_getobjname()</code> has received an invalid object type.
Action	Currently, the valid object types are only SRV_C_Mutex and SRV_C_MQueue. For a list of valid flags, see: <ul style="list-style-type: none"><li>• Table 3-53 on page 295 for <code>srv_getmsgq()</code></li><li>• Table 3-88 on page 349 for <code>srv_putmsgq()</code></li></ul>

- “`srv_lockmutex`” on page 312 for `srv_lockmutex()`
- Table 3-153 on page 454 for `srv_wakeup()`

Additional information      Symbolic constant: `SRV_EBADFLAG`

## Error 16137

Severity	10
Message text	<code>The maximum number of site handlers, %1! has been exceeded.</code>
Explanation	A connection attempt from a remote server site handler has exceeded the currently configured maximum.
Action	Update <code>SRV_S_NUMREMSITES</code> to increase the maximum number of active remote server site handler connections. The default is <code>SRV_DEF_NUMREMSITES</code> .
Additional information	Symbolic constant: <code>SRV_EMAXREMSITES</code>

## Error 16138

Severity	10
Message text	<code>This client already has a notification request for procedure %1!.</code>
Explanation	Through <code>srv_regwatch()</code> , Open Server has attempted to add a thread to the notification list for a registered procedure that is already in the list.
Action	No action is required because the thread is already in the notification list.
Additional information	Symbolic constant: <code>SRV_EPNOTIFYEXISTS</code>

## Error 16139 (Deprecated)

Severity	N/A
----------	-----

Message text	The maximum number of remote channels, %1! has been exceeded.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EMAXREMCONNS

## Error 16140

Severity	10
Message text	Maximum RPC execution stack depth has already been reached.
Explanation	Currently, the maximum number of RPCs that can be called recursively is 16.
Action	Limit the number of times Open Server recursively call RPCs to 16.
Additional information	Symbolic constant: SRV_EMAXRPC

## Error 16141

Severity	10
Message text	NULL Server process passed to '%1!'. The specified function has received a NULL SRV_PROC.
Explanation	
Action	Pass a valid, non-NULL SRV_PROC to the specified API.
Additional information	Symbolic constant: SRV_EPROCNULL

## Error 16142

Severity	10
Message text	Server process address 0x%1! not in pool in '%2!'.

Explanation	The SRV_PROC address does not exist in the array of server processes maintained in the SRV_SERVER structure.
Action	Use the SRV_CURPROC() macro to find the address of the active thread.
Additional information	Symbolic constant: SRV_EPROCINVALID

## Error 16143

Severity	10
Message text	Server process address 0x%1! is on bad boundary in '%2!'.
Explanation	The SRV_PROC address exists within the array of server processes maintained in the SRV_SERVER structure, but is located outside of the valid range.
Action	Use the SRV_CURPROC() macro to find the address of the currently active thread.
Additional information	Symbolic constant: SRV_EPROCBOUND

## Error 16144

Severity	10
Message text	Server process '%1!' is not allocated in '%2!'.
Explanation	This error indicates that <code>srv_termproc()</code> has already terminated the SRV_PROC address passed to an Open Server routine.
Action	Perform one of these: <ul style="list-style-type: none"><li>• Search for other errors that may indicate that a different event caused Open Server to terminate the thread.</li><li>• Write code around <code>srv_termproc()</code> that records which SRV_PROC address is about to be terminated. Compare the recorded address to the address of the SRV_PROC passed to the specified Open Server routine. If <code>srv_termproc()</code> is not called, confirm that <code>srv_createproc()</code> has completed successfully, and that the Open Server thread is fully initialized.</li></ul>

- Ensure that `srv_termproc()` is not called from interrupt-level code because this can lead to unpredictable results.

Additional information      Symbolic constant: `SRV_EPROCNOTUSED`

## Error 16145

Severity                    10

Message text                `Server process '%1!' has invalid I/O in '%2!'.`

Explanation                An `SRV_PROC` I/O can be marked invalid for various reasons, such as when a server shuts down, when an error occurs, or when the process is terminated.

Action                      Analyze preceding error messages to understand the underlying cause of the error. Use `srv_thread_props(SRV_T_IODEAD)` to determine whether the thread's I/O channel is valid.

Additional information      Symbolic constant: `SRV_EPROCIODEAD`

## Error 16146 (Deprecated)

Severity                    N/A

Message text                `The connection service process has failed.`

Explanation                N/A

Action                      N/A

Additional information      Symbolic constant: `SRV_ECONSRVFAIL`

## Error 16147

Severity                    20

Message text                `Failed to allocate a network buffer for thread '%1!'.`

Explanation	The specified thread prompted Open Server to allocate more network buffers than it has been configured for. Open Server bases the maximum number of network buffers to allocate on configuration parameters including SRV_S_NUMCONNECTIONS, SRV_S_NUMREMBUF, and SRV_S_NUMREMSITES.
Action	Check the SRV_T_TYPE property to determine what type of thread caused the problem. Increase the appropriate property.
Additional information	Symbolic constant: SRV_ENONETBUF

## Error 16148

Severity	10
Message text	'%1!' is not a valid capability value.
Explanation	srv_capability() has received an invalid capability value.
Action	See Table 3-19 on page 248 for valid capability values.
Additional information	Symbolic constant: SRV_ENOTSUPPORTED

## Error 16149

Severity	20
Message text	Fatal error: Internal %1! structure is corrupted in %2!.
Explanation	srv_getloginfo() cannot initialize the CS_LOGININFO parameter.
Action	Ensure the SRV_PROC structure is valid before calling srv_getloginfo(). srv_getloginfo() initializes CS_LOGININFO parameter with data that it reads from SRV_PROC.
Additional information	Symbolic constant: SRV_ECORRUPT

## Error 16150

Severity	10
----------	----

Message text	Thread '%1!' is in danger of exceeding allotted stack space of %2! bytes.
Explanation	The specified thread is likely to exceed the allotted stack space.
Action	Increase the SRV_S_STACKSIZE property to allocate more stack for each thread.
Additional information	Symbolic constant: SRV_ELOWSTK

## Error 16151

Severity	20
Message text	Thread '%1!' has overflowed or corrupted its stack -- aborting.
Explanation	The specified thread has exceeded its stack space or has had its stack space overwritten by another thread.
Action	Increase the SRV_S_STACKSIZE property to allocate more stack for each thread.
Additional information	Symbolic constant: SRV_EBADSTK

## Error 16152 (Deprecated)

Severity	N/A
Message text	Fatal error: Error detected during network I/O, error '%1!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EIO

## Error 16153 (Deprecated)

Severity	N/A
Message text	Client disconnected during network read.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDISC

## Error 16154 (Deprecated)

Severity	N/A
Message text	Fatal error: Maximum number of file descriptors exceeded.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENETCLOSE

## Error 16155 (Deprecated)

Severity	N/A
Message text	Fatal error: Maximum number of file descriptors exceeded.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENODEV

## Error 16156

Severity	10
----------	----



Message text	Invalid command argument (0x%1!).
Explanation	Some routines require a <i>cmd</i> argument, which is usually set to CS_GET, CS_SET, or CS_CLEAR.
Action	Seeo Chapter 3, “Routines,” to check the valid <i>cmd</i> arguments for the routine that failed.
Additional information	Symbolic constant: SRV_ECMD

## Error 16157

Severity	10
Message text	Invalid type argument (0x%1!).
Explanation	An invalid datatype constant has been specified.
Action	See Table 2-37 on page 205 for a list of valid type constants.
Additional information	Symbolic constant: SRV_ETYPE

## Error 16158

Severity	10
Message text	%1!: Invalid count (%2!) in CS_DATAFMT.
Explanation	The value assigned to the CS_DATAFMT <i>count</i> field is invalid.
Action	See “CS_DATAFMT structure” on page 54 for the valid <i>count</i> field values.
Additional information	Symbolic constant: SRV_EBADCOUNT

## Error 16159

Severity	10
Message text	%1!: Invalid status (0x%2!) in CS_DATAFMT.
Explanation	The value assigned to the CS_DATAFMT <i>status</i> field is invalid.

Action	See Table 2-9 on page 57 for a list of valid CS_DATAFMT statuses.
Additional information	Symbolic constant: SRV_EBADSTAT.

## Error 16160

Severity	10
Message text	%1!: Invalid format (0x%2!) in CS_DATAFMT.
Explanation	The value assigned to the CS_DATAFMT <i>format</i> field is invalid.
Action	See Table 2-7 on page 56 for a list of valid CS_DATAFMT formats.
Additional information	Symbolic constant: SRV_EBADFORMAT

## Error 16161 (Deprecated)

Severity	N/A
Message text	Must be in SRV_MSG event handler for CS_GET.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOMSG

## Error 16162

Severity	10
Message text	%1!: Bind was not completed for item number(%2!) for %3!.
Explanation	The specified parameter is not bound.
Action	Verify that <code>srv_bind()</code> or <code>srv_alt_bind()</code> successfully binds the specified parameter.
Additional information	Symbolic constant: SRV_ENOBIND

## Error 16163

Severity	10
Message text	Data was truncated from (%1!) to (%2!).
Explanation	The CS_DATAFMT or CS_IODESC <i>name</i> field or the CS_SERVERMSG <i>svrname</i> is longer than CS_MAX_NAME.
Action	Assign <i>name</i> or <i>svrname</i> values that are smaller than or equal to CS_MAX_NAME. Refer to <i>cstypes.h</i> for the value of CS_MAX_NAME.
Additional information	Symbolic constant: SRV_ETRUNC

## Error 16164 (Deprecated)

Severity	N/A
Message text	Bind was attempted with NULL program variable.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENULLADDR

## Error 16165

Severity	10 or 15
Message text	%1!: %2! for %3! not valid.
Explanation	A state error has occurred for the specified Open Server routine.
Action	See Chapter 3, “Routines,” for the specified routine’s valid states.
Additional information	Symbolic constant: SRV_ESTATE

## Error 16166

Severity	10
----------	----

Message text	%!:: Invalid length for variable, oldvallen(%2!), newvallen(%3!).
Explanation	srv_envchange() has received an invalid <i>oldvallen</i> or <i>newvallen</i> value.
Action	Ensure that the length passed is greater than zero but less than CS_MAX_CHAR, or that the length is set to CS_UNUSED or CS_NULLTERM.
Additional information	Symbolic constant: SRV_ELEN

## Error 16167

Severity	10
Message text	Row '%1!' has already been described.
Explanation	srv_alt_header() has detected a duplicate row.
Action	Pass a unique <i>altid</i> value to srv_alt_header().
Additional information	Symbolic constant: SRV_EDUPROW

## Error 16168

Severity	10
Message text	%!:: There is no compute row '%2!' for the current thread.
Explanation	The specified routine cannot find a computed column.
Action	Check the column number (passed as parameter <i>item</i> ) passed to srv_alt_bind() or srv_alt_descfmt(), and verify that it corresponds to a computed column.
Additional information	Symbolic constant: SRV_ENOROW

## Error 16169

Severity	10
----------	----

Message text	%1!: There is no aggregate operator with id# '%2!'.
Explanation	srv_alt_descfmt() has received an invalid aggregate operator.
Action	See Table 3-5 on page 228 for a list of valid aggregate operator types.
Additional information	Symbolic constant: SRV_ENOAGG

## Error 16170

Severity	10
Message text	%1!: Table '%2!' has already been described.
Explanation	srv_tabname() has received a duplicate table name.
Action	Call srv_tabname() only once for each table associated with a set of browse mode results.
Additional information	Symbolic constant: SRV_EDUPTAB

## Error 16171

Severity	10
Message text	%1!: Table '%2!' has not been described.
Explanation	srv_tabcolname() cannot find the specified table.
Action	Define the table using srv_tabname().
Additional information	Symbolic constant: SRV_ENOTAB

## Error 16172

Severity	10
Message text	%1!: Bad compute-id '%2!'.
Explanation	srv_alt_header() has received an invalid compute row identifier.
Action	Pass a compute row identifier that is greater than zero.

Additional information      Symbolic constant: SRV\_EBADROW

## Error 16173 (Deprecated)

Severity                    N/A

Message text                %1!: The compute-by list does not match the order-by list.

Explanation                N/A

Action                      N/A

Additional information      Symbolic constant: SRV\_EBADCOMPUTEBY

## Error 16174

Severity                    10

Message text                %1!: There is no compute-by list in the current statement.

Explanation                srv\_alt\_header() has received a pointer to an empty bylist array.

Action                      Ensure that the number of rows in the bylist array is the same as the number defined in *numbylist*.

Additional information      Symbolic constant: SRV\_ENOCOMPUTEBY

## Error 16175

Severity                    10

Message text                %1!: Bad value '%2!' for number of compute-by columns.

Explanation                srv\_alt\_header() has failed because the number of bylist columns specified is less than zero.

Action                      Ensure that the *numbylist* value is equal to the number of elements in the *bylistarrayp* array pointer.

Additional information      Symbolic constant: SRV\_EBADNUMBYLIST

## Error 16176

Severity                    10

Message text                %1!: There is no order-by list in the current statement.

Explanation                `srv_orderby()` has received a pointer to an empty array of columns.

Action                      Pass a *collistp* parameter that is not NULL.

Additional information      Symbolic constant: SRV\_ENOORDERBY

## Error 16177

Severity                    10

Message text                %1!: Bad value '%2!' for number of order-by columns.

Explanation                The number of columns passed to `srv_orderby()` is less than zero.

Action                      Specify a *numcols* value that is greater than or equal to zero. The value should represent the number of elements in the *collistp* array.

Additional information      Symbolic constant: SRV\_EBADNUMCOLS

## Error 16178

Severity                    10

Message text                %1!: Bad table number '%2!'.

Explanation                The table ID (passed as *tablenum* parameter) passed to `srv_tabname()` is less than one.

Action                      Make sure the *tablenum* parameter is greater than zero.

Additional information      Symbolic constant: SRV\_EBADTABNUM

## Error 16179

Severity	10
Message text	%1!: Bad name length value '%2!'.
Explanation	srv_tabname() has received an invalid <i>namelen</i> value.
Action	Set <i>namelen</i> to CS_NULLTERM or to a value greater than zero.
Additional information	Symbolic constant: SRV_EBADNAMELEN

## Error 16180

Severity	10
Message text	%1!: Bad control-format length value '%2!'.
Explanation	srv_setcontrol() has failed because the control date length is invalid.
Action	Set <i>ctrlen</i> to a value greater than or equal to zero but less than SRV_MAXCHAR.
Additional information	Symbolic constant: SRV_EBADCTRLLEN

## Error 16181

Severity	10
Message text	%1!: '%2!' argument cannot be null.
Explanation	The specified routine has passed a table name with a NULL value.
Action	Ensure that the table name passed to <i>srv_tabname()</i> or defined in the CS_BROWSEDESC <i>name</i> field is not NULL.
Additional information	Symbolic constant: SRV_ENULLNAME

## Error 16182

Severity	10
----------	----



Message text	<code>%1!: column '%2!' must be linked with a table via <code>srv_tabcolname()</code>.</code>
Explanation	The specified column is not linked to a table.
Action	Call <code>srv_tabcolname()</code> once for each result column that is also a column in a result row.
Additional information	Symbolic constant: <code>SRV_EBADTABCOL</code>

## Error 16183 (Reserved)

Severity	10
Message text	<code>%1!: Unsupported option(%2!) for %3!.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_EUNSUPPORTED</code> . Reserved for future use.

## Error 16184 (Deprecated)

Severity	N/A
Message text	<code>Unknown character set %1! in function '%2!'.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_ENOCHARSET</code>

## Error 16185 (Deprecated)

Severity	N/A
Message text	<code>Bad format in csr file '%1!' ('%2!' line).</code>
Explanation	N/A

Action	N/A
Additional information	Symbolic constant: SRV_ECSRFILE

## **Error 16186 (Deprecated)**

Severity	N/A
Message text	Error building character attribute structure for character set '%!'
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECHARATTR

## **Error 16187 (Deprecated)**

Severity	N/A
Message text	Expected hex digit, got %! in function '%2!'
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOTHEX

## **Error 16188**

Severity	10
Message text	Argument %! %2! incorrect in function '%3!'
Explanation	The parameter passed to the specified routine is invalid.
Action	See Chapter 3, “Routines,” for the values you can use for the specified routine’s parameters.
Additional information	Symbolic constant: SRV_EARG

## Error 16189 (Deprecated)

Severity	N/A
Message text	Can't convert between server character set '%1!' and client character set '%2!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECSNOCVT

## Error 16190 (Deprecated)

Severity	N/A
Message text	Character set conversion %1! occurred in '%2!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECSCVT

## Error 16191 (Deprecated)

Severity	N/A
Message text	Unknown return (%1!) in function '%2!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENORETURN

## Error 16192

Severity	10
----------	----

## Error 16193 (Deprecated)

---

Message text	%1! character set conversion error in login record.
Explanation	This error occurs when characters in the client application's login record cannot be converted to the Open Server character set. For example, if the client's character set is utf8 and the Open Server character set is roman8, some characters belonging to utf8 may not be converted to roman8.
Action	Run the client application through the Ribo utility. Review "Login Record" for any question marks, which indicate characters that cannot be converted. Either remove those characters from the client application or change the Open Server character set.
Additional information	Symbolic constant: SRV_ECSLOGREC

## Error 16193 (Deprecated)

Severity	N/A
Message text	'%1!' is not a '%2!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECSMISMATCH

## Error 16194 (Deprecated)

Severity	N/A
Message text	Can't load definition file for %1! '%2!'.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECSLOAD

## Error 16195

Severity	10
Message text	<code>CS_CANBENULL set for invalid data type(%1!).</code>
Explanation	The application set the <code>CS_DATAFMT status</code> bit to <code>CS_CANBENULL</code> .
Action	Set <code>CS_BIT_TYPE</code> , a Boolean, to 0 or 1.
Additional information	Symbolic constant: <code>SRV_EBADNULL</code>

## Error 16196 (Deprecated)

Severity	N/A
Message text	<code>Can't change configuration option '%1!' after srv_init().</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_ECHCONFIG</code>

## Error 16197 (Deprecated)

Severity	N/A
Message text	<code>%1! must be called before srv_alt_header() if a compute- by list is present.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_ELATEORDER</code>

## Error 16198

Severity	10
----------	----

Message text	Invalid item number(%1!).
Explanation	srv_alt_bind(), srv_alt_descfmt(), srv_bind(), srv_descfmt(), or srv_text_info() has received an invalid value for the <i>item</i> parameter.
Action	Specify an <i>item</i> value that is greater than or equal to one but less than the maximum number of columns or parameters. Contact Sybase Technical Support for the current maximum number of columns or parameters.
Additional information	Symbolic constant: SRV_EBADITEM

## Error 16199 (Deprecated)

Severity	N/A
Message text	%!:: must be in SRV_BULK handler to get property %2!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOTINBULK.

## Error 16200

Severity	10
Message text	%!:: invalid transfer type '%2!'.
Explanation	Open Server has invoked blk_srvinit(), blk_gettext(), blk_colval(), blk_getrow(), srv_get_text(), or srv_text_info() but the SRV_T_BULKTYPE value is not SRV_BULKLOAD.
Action	Call the appropriate command for each SRV_T_BULKTYPE value: <ul style="list-style-type: none"><li>• 1 – SRV_BULKLOAD</li><li>• 2 – SRV_TEXTLOAD</li><li>• 3 – SRV_IMAGELOAD</li><li>• 4 – SRV_UNITEXTLOAD</li></ul> See also Table 2-29 on page 159.

Additional information      Symbolic constant: SRV\_EBADIOTYPE

## Error 16201

Severity                    10

Message text                %1! can only be called for single-column rows.

Explanation                The column to be transferred has been bound using `srv_bind()` or `srv_alt_bind()` and cannot be sent with `srv_send_text()`.

Action                      When using `srv_send_text()` to transfer text or image data, call `srv_text_info()` to describe the data.

Additional information      Symbolic constant: SRV\_EGTONECOL

## Error 16202

Severity                    10

Message text                %1!: Nothing left to %2!.

Explanation                `srv_get_text()` does not have anything else to read, or `srv_send_text()` does not have anything else to write.

Action                      In each call to `srv_get_text()` or `srv_send_text()`, specify a length that adds up to the length specified in `srv_text_info()`. If sending a NULL value for a text or image column, specify zero.

Additional information      Symbolic constant: SRV\_ENONELEFT

## Error 16203

Severity                    10

Message text                %1!: Column must be of type TEXT or IMAGE.

Explanation                `srv_send_text()` has received an invalid column type.

Action	Before calling <code>srv_send_text()</code> , ensure that the column type specified in <code>srv_text_info()</code> is <code>CS_TEXT_TYPE</code> , <code>CS_XML_TYPE</code> , <code>CS_IMAGE_TYPE</code> , or <code>CS_UNITEXT_TYPE</code> .
Additional information	Symbolic constant: <code>SRV_EWRONGTYPE</code>

## Error 16204

Severity	10
Message text	<code>%1!: Bad value %2! for structure member '%3!'</code> .
Explanation	The specified member of the <code>CS_SERVERMSG</code> or <code>CS_IODESC</code> structure has an invalid length.
Action	Assign a valid length for the <code>CS_SERVERMSG</code> or <code>CS_IODESC</code> structure members. See <i>Open Client Client-Library/C Reference Manual</i> .
Additional information	Symbolic constant: <code>SRV_EBADNUMVAL</code>

## Error 16205 (Deprecated)

Severity	N/A
Message text	<code>%1!: Parameter '%2!' has already been described.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_EDUPPARAM</code>

## Error 16206 (Deprecated)

Severity	N/A
Message text	RPC parameter description is not complete -- <code>%1! %2!</code> not described.
Explanation	N/A



Action	N/A
Additional information	Symbolic constant: SRV_ERPCINC

## Error 16207

Severity	10
Message text	%1!: rows read in may not exceed %2! bytes.
Explanation	blk_getrow() has retrieved a row that exceeds the specified number of bytes.
Action	Issue dbcc serverlimits on Adaptive Server, and review the “Table-specific row-size limits” section to determine the maximum length of an Adaptive Server row.
Additional information	Symbolic constant: SRV_EBADROWLEN

## Error 16208

Severity	10
Message text	%1!: compute row columns cannot be of type TEXT or IMAGE.
Explanation	srv_alt_descfmt() has received a column of type text, image, or xml, which srv_alt_descfmt() cannot process.
Action	Do not pass text, image, or xml to srv_alt_descfmt().
Additional information	Symbolic constant: SRV_EBADALTCOLTYPE

## Error 16209 (Deprecated)

Severity	N/A
Message text	%1!: srvp_iotype field of SRV_PROC is uninitialized.
Explanation	N/A
Action	N/A

Additional information      Symbolic constant: SRV\_EIOTYPENOTSET

## Error 16210

Severity                    10

Message text                %1!: column %2! should not be bound.

Explanation                srv\_send\_text() has attempted to send a column that is bound using srv\_bind().

Action                      Use srv\_xferdata() to send text or image columns that are bound by srv\_bind(). To send through srv\_send\_text(), bind the column using srv\_text\_info().

Additional information      Symbolic constant: SRV\_EBOUND

## Error 16211 (Deprecated)

Severity                    N/A

Message text                %1!: text/image length for the current column has already been read from the client.

Explanation                N/A

Action                      N/A

Additional information      Symbolic constant: SRV\_ETXTLENREAD

## Error 16212

Severity                    10

Message text                %1!: text/image length for the current column has not yet been read from the client.

Explanation                srv\_get\_text() has failed because srv\_text\_info() has not been called to read the length of the column.

Action                      Call srv\_text\_info() with CS\_GET to get the complete description of the text or image data to retrieve.

Additional information      Symbolic constant: SRV\_ETXTLENNOTREAD

## Error 16213

Severity                    10

Message text                Text or image timestamp not defined.

Explanation                `srv_send_text()` has detected that the *timestamp* field of the `CS_IODESC` structure had not been defined.

Action                      Populate *timestamp* before calling `srv_text_info()`.

Additional information      Symbolic constant: SRV\_ENOTIMESTAMP

## Error 16214 (Deprecated)

Severity                    N/A

Message text                No character set definition file available -- setting server's character set to '%!'

Explanation                N/A

Action                      N/A

Additional information      Symbolic constant: SRV\_ECSFALLBACK

## Error 16215 (Deprecated)

Severity                    N/A

Message text                Out-of-range length (%!) for login field '%2!'

Explanation                N/A

Action                      N/A

Additional information      Symbolic constant: SRV\_ELOGINLEN

## Error 16216

Severity	10
Message text	%!! cannot retrieve TEXT or IMAGE column values.
Explanation	blk_colval() failed when attempting to return the value of a column.
Action	blk_colval() cannot retrieve text, image, boundary, sensitivity, unitext, or XML column values. Use blk_gettext() to retrieve such columns.
Additional information	Symbolic constant: SRV_EBADCOLTYPE

## Error 16217

Severity	10
Message text	%!:: blk_getrow() must be called to retrieve row information.
Explanation	blk_colval() or blk_gettext() is called before blk_getrow().
Action	Use blk_getrow() to populate the CS_BLK_ROW structure before calling blk_colval() or blk_gettext().
Additional information	Symbolic constant: SRV_ENEEDROW

## Error 16218

Severity	10
Message text	%!:: blk_exec() must be called to initialize bulk copy operation.
Explanation	The CS_BLKDESC structure passed to blk_getrow(), blk_gettext(), or blk_srvinit() has not been properly initialized.
Action	Call blk_alloc() to initialize CS_BLKDESC, and ensure that the initialization is successful.
Additional information	Symbolic constant: SRV_ENEEDHANDLE

## Error 16219

Severity	10
Message text	A call to %1! was interrupted by an attention.
Explanation	The specified API has been interrupted while waiting in <code>srv_sleep()</code> . This is an informational message that may occur when a client application disconnects abruptly—such as when you press Ctrl+C, or when an application issues <code>dbcancel()</code> or <code>ct_cancel()</code> —and the thread on which the Open Server is sleeping wakes from a read completion.
Action	Ignore this error unless it causes additional problems.
Additional information	Symbolic constant: <code>SRV_EINTERRUPTED</code>

## Error 16220

Severity	10
Message text	%1!: invalid status argument (0x%2!).
Explanation	<code>srv_senddone()</code> has received an invalid status.
Action	See Table 3-120 on page 401 for a list of valid statuses.
Additional information	Symbolic constant: <code>SRV_ESTATUS</code>

## Error 16221

Severity	10
Message text	%1!: <code>SRV_DONE_FLUSH</code> cannot be set with <code>SRV_DONE_FINAL</code> .
Explanation	You can set <code>SRV_DONE_FLUSH</code> together with <code>SRV_DONE_MORE</code> , but not with <code>SRV_DONE_FINAL</code> .
Action	Ensure that <code>SRV_DONE_FLUSH</code> and <code>SRV_DONE_FINAL</code> are not set at the same time.
Additional information	Symbolic constant: <code>SRV_ENORPC</code>

## Error 16222

Severity	10
Message text	%1!: SRV_DONE_FLUSH cannot be executed from a SRV_CONNECT event.
Explanation	An application has called <code>srv_senddone()</code> in the <code>SRV_CONNECT</code> event handler to set the <code>SRV_DONE_FLUSH</code> bit.
Action	Set the status to <code>SRV_DONE_FLUSH</code> outside <code>SRV_CONNECT</code> .
Additional information	Symbolic constant: <code>SRV_EFLUSHCON</code>

## Error 16223

Severity	10
Message text	%1!: SRV_DONE_MORE cannot be set with <code>SRV_DONE_FINAL</code> .
Explanation	The <code>SRV_DONE_MORE</code> and <code>SRV_DONE_FINAL</code> status bits are mutually exclusive and cannot be set at the same time.
Action	During calls to <code>srv_senddone()</code> , set <code>SRV_DONE_MORE</code> and <code>SRV_DONE_FINAL</code> one at a time.
Additional information	Symbolic constant: <code>SRV_EMORE</code>

## Error 16224 (Deprecated)

Severity	N/A
Message text	%1!: SRV_DONE_MORE cannot be executed from a <code>SRV_CONNECT</code> event.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_EMORECON</code>

## Error 16225 (Deprecated)

Severity	N/A
Message text	<code>%1!: Invalid context.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_ECONTEXT</code>

## Error 16226 (Deprecated)

Severity	N/A
Message text	<code>%1!: Invalid version number(%1!).</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_EVERSION</code>

## Error 16227

Severity	15
Message text	<code>Unknown %1! TDS stream received by spid %2!.</code>
Explanation	Open Server has received an invalid token. Some TDS tokens are valid only in certain versions of TDS, while some are valid only during specific events.
Action	Use Ribo to record the tokens passed between the client application and Open Server, and see which token is invalid. Also, enable these Open Server <code>SRV_S_TRACEFLAG</code> bits: <code>SRV_TR_TDSHDR</code> , <code>SRV_TR_TDSDATA</code> , and <code>SRV_TR_EVENT</code> .
Additional information	Symbolic constant: <code>SRV_EBADTDS</code>

## Error 16228 (Deprecated)

Severity	N/A
Message text	%!:: can only be called in SRV_OPTION handler.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EINOPTION

## Error 16229

Severity	10
Message text	%!:: Wrong TDS version (%2!) in %3!.
Explanation	srv_thread_props() has attempted to set SRV_T_TDSVERSION with an invalid TDS version.
Action	See Table 2-26 on page 150 for a list of valid SRV_T_TDSVERSION values.
Additional information	Symbolic constant: SRV_EWRONGTDSVER

## Error 16230

Severity	15
Message text	%!:: Unknown option cmd from client in func %2!.
Explanation	srv_options() has failed because the client application requested to set an invalid option. See Table 3-78 on page 332 for the valid options.
Action	Use Ribo to identify the invalid option that the client application requested Open Server to set.
Additional information	Symbolic constant: SRV_EWRONGOPTCMD



## Error 16231

Severity	15
Message text	%!!: Unknown option from client in func %2!.
Explanation	Open Server has failed to set an option through <code>srv_options()</code> .
Action	See Table 3-80 on page 335 and ensure that the parameters passed to <code>srv_options()</code> for the specified option have the correct datatype and value.
Additional information	Symbolic constant: <code>SRV_EWRONGOPT</code>

## Error 16232

Severity	10
Message text	%!!: No data left for the current token stream.
Explanation	The internal check to determine if there is enough data left to read in the network stream for the current token has failed. This could be a network issue or an incompatibility issue with older Sybase products (CR #212015 and CR #434920).
Action	If you can reproduce the error, use Ribo to get information about the token.
Additional information	Symbolic constant: <code>SRV_ENODATA</code>

## Error 16233

Severity	10
Message text	Argument %! is NULL in function: %2!.
Explanation	The specified function has received a NULL value.
Action	Check your Open Server code to ensure that the parameter passed is valid.
Additional information	Symbolic constant: <code>SRV_ENULLARG</code>

## Error 16234

Severity	10
Message text	Cannot call %1! in user defined event handler.
Explanation	The user-defined event handler has attempted to send or receive a result. Sending <code>srv_senddone()</code> for a result type, such as <code>ROWDATA</code> , <code>RPCDATA</code> , or <code>DONE_FINAL</code> , from a user-defined event handler is not permitted.
Action	Ensure that <code>srv_senddone()</code> is not called from a user-defined event handler.
Additional information	Symbolic constant: <code>SRV_EINUSEREVENT</code>

## Error 16235

Severity	10
Message text	Invalid bind value (0x%1!) for column (%2!) in function: %3!.
Explanation	The null indicator value for the specified column is invalid.
Action	Set the <code>srv_bind()</code> null indicator variable to <code>CS_GOODDATA</code> or <code>CS_NULLDATA</code> .
Additional information	Symbolic constant: <code>SRV_EBINDARG</code>

## Error 16236

Severity	10
Message text	%1!: State error (0x%2!).
Explanation	This error occurs when an Open Server API is called out of order, typically because an earlier Open Server call failed.
Action	Check your Open Server error logs for earlier errors that might indicate the API that failed first.
Additional information	Symbolic constant: <code>SRV_ESTATECHK</code>

## Error 16237

Severity	10
Message text	<code>%1!: Version not set with srv_version.</code>
Explanation	The Open Server version has not been set using <code>srv_version()</code> .
Action	Verify that the call to <code>srv_version()</code> has successfully completed, and that a valid version is specified for the version parameter.
Additional information	Symbolic constant: <code>SRV_ENOVERSION</code>

## Error 16238

Severity	10
Message text	<code>%1!: Invalid result type(%2!) for event(%3!).</code>
Explanation	An Open Server API cannot be called from the specified event handler.
Action	Do not call the API specified in the error message from the specified event handler. To determine the event handler that called the API, search for the event number in the “Open Server Event Types” section of <i>ospublic.h</i> .
Additional information	Symbolic constant: <code>SRV_EBADRESULT</code>

## Error 16239 (Deprecated)

Severity	N/A
Message text	<code>Connection service thread failed to sleep.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_ECONFAIL</code>

## Error 16240

Severity	10 or 20
Message text	<code>Net-Library routine %1! failed in %2!.</code>
Explanation	This error occurs when various network related functions fail. It can be informational or it can be fatal. Often, additional errors accompany this error, and suggest what problem needs to be resolved.
Action	Resolve the underlying the problem recorded in the error logs. If no other error is recorded, use a utility such as truss or tusc to determine which TCP API failed and why.
Additional information	Symbolic constant: SRV_ENETFUNC

## Error 16241

Severity	20
Message text	<code>Thread %1! failed to get new event.</code>
Explanation	Open Server has attempted to retrieve an event from the message queue but failed.
Action	Let Open Server sleep until a message becomes available. If SRV_I_INTERRUPTED did not cause the failure, call Sybase Technical Support.
Additional information	Symbolic constant: SRV_EPROCFAIL

## Error 16242 (Deprecated)

Severity	N/A
Message text	<code>Failed to close a client connection.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECLOSEFAIL

## Error 16243 (Deprecated)

Severity	N/A
Message text	<code>Read operation for process %1! failed.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_EREADFAIL</code>

## Error 16244

Severity	10
Message text	<code>Write operation for process %1! failed.</code>
Explanation	<code>srv_sendinfo()</code> has failed because it has been called from an error handler while processing a login. If the client login failed abruptly, the <code>SRV_PROC</code> structure corresponding to the client connection may not have been completely initialized, causing <code>srv_sendinfo()</code> to fail.
Action	Call <code>srv_sendinfo()</code> after the connect handler has completed.
Additional information	Symbolic constant: <code>SRV_EWRITEFAIL</code>

## Error 16245 (Deprecated)

Severity	20
Message text	<code>Failed to block interrupts.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_EBLOCK</code>

## Error 16246 (Deprecated)

Severity	20
Message text	Failed to unblock interrupts.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EUNBLOCK

## Error 16247

Severity	10
Message text	Site handler spid %1! failed to setup a new channel.
Explanation	Open Server has received a protocol request to set up another logical channel, but has failed because the channel has already been closed.
Action	Check the log file of the server requesting the site handler for errors that may indicate why the channel has been prematurely closed.
Additional information	Symbolic constant: SRV_ECHANNEL.

## Error 16248

Severity	10
Message text	Error in %1! - file descriptor %2! is no longer active.
Explanation	srv_select() has failed as a result of a lower-level operating system API failing.
Action	Use an operating system command, such as truss, to trace processes and see which operating system API failed and why. Take corrective action to resolve the underlying error. Also, see CR 277236.
Additional information	Symbolic constant: SRV_EPOLL

## Error 16249 (Deprecated)

Severity	N/A
Message text	%1!: Could not access common library global data.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EGLOBAL

## Error 16250 (Deprecated)

Severity	N/A
Message text	%1!: Could not run async manage.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EIOPOST

## Error 16251

Severity	10
Message text	%1! has invalid length value (%2!) in function: %3!.
Explanation	srv_props() or srv_thread_props() has received an invalid value for the <i>buflen</i> parameter.
Action	Ensure that <i>buflen</i> is greater than zero and that it specifies the length of the buffer.
Additional information	Symbolic constant: SRV_EBADLENARG

## Error 16252

Severity	10
----------	----

Message text	Invalid thread property (%1!) in %2!.
Explanation	srv_thread_props() has received an invalid value for the <i>property</i> parameter.
Action	See Table 3-147 on page 445 for a list of valid properties.
Additional information	Symbolic constant: SRV_EPROCPROPS

## Error 16253

Severity	10
Message text	%1!: property %2! not valid for service thread.
Explanation	An Open Server application has called srv_thread_props() on a service thread. However, there are currently no properties that can be set or retrieved on service threads.
Action	Do not call srv_thread_props() on a service thread.
Additional information	Symbolic constant: SRV_EINSERVTHREAD

## Error 16254

Severity	10
Message text	%1!: %2! property is valid only in SRV_DISCONNECT handle.
Explanation	The srv_thread_props() <i>property</i> parameter is set to SRV_T_CLIENTLOGOUT, but srv_thread_props() is called from an event other than SRV_DISCONNECT.
Action	Call srv_thread_props(SRV_T_CLIENTLOGOUT) only from the SRV_DISCONNECT event handler.
Additional information	Symbolic constant: SRV_EFROMDISCONNECT



## Error 16255

Severity	10
Message text	%1!: buffer size smaller (%2!) for property %3!.
Explanation	The <i>buflen</i> parameter passed to <code>srv_dynamic()</code> , <code>srv_props()</code> , or <code>srv_thread_props()</code> is too small for the specified property.
Action	If <code>srv_dynamic()</code> caused the failure, see Table 3-42 on page 277 for the required buffer lengths. If <code>srv_props()</code> or <code>srv_thread_props()</code> caused the failure, ensure <i>buflen</i> is appropriate for the specified property's datatype.
Additional information	Symbolic constant: <code>SRV_ESMALLBUF</code>

## Error 16256

Severity	10
Message text	%1!: buffer size may not exceed (%2!) bytes for %3!.
Explanation	The <i>buflen</i> parameter passed to the specified API is larger than the maximum.
Action	Ensure that the <i>buflen</i> passed to the specified API is appropriate for the datatype. For character datatypes, the maximum is typically <code>SRV_MAXCHAR</code> .
Additional information	Symbolic constant: <code>SRV_ELAREGBUF</code>

## Error 16257

Severity	10
Message text	%1!: should be in connect handler to set property %2!.
Explanation	Some properties, such as <code>SRV_T_PASSTHRU</code> and <code>SRV_T_TDSVERSION</code> , may only be set in the <code>SRV_CONNECT</code> event handler.
Action	Set the specified property in <code>SRV_CONNECT</code> .
Additional information	Symbolic constant: <code>SRV_EFROMCONNECT</code>

## Error 16258 (Deprecated)

Severity	N/A
Message text	%1!: Can't set a TDS level (%2!) > current level (%3!).
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EWRONGTDSSET

## Error 16259 (Deprecated)

Severity	N/A
Message text	%1!: (%2!) invalid for property: %3!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EWRONGLANG

## Error 16260

Severity	10
Message text	%1!: Property - %2! cannot be set/cleared.
Explanation	An attempt has been made to set a read-only property.
Action	See Table 3-147 on page 445 for <code>srv_thread_props()</code> property characteristics, or to Table 3-86 on page 345 for that of <code>srv_props()</code> .
Additional information	Symbolic constant: SRV_ENOPROPSET

## Error 16261

Severity	10
----------	----

Message text	<code>Invalid server property (%1!) in %2!.</code>
Explanation	An attempt to set an invalid property through <code>srv_props()</code> has failed.
Action	See Table 3-86 on page 345 for a list of valid properties.
Additional information	Symbolic constant: <code>SRV_ESRVPROPS</code>

## Error 16262

Severity	10
Message text	<code>%1!: Cannot configure server property after <code>srv_init()</code>.</code>
Explanation	Open Server has attempted to set a property through <code>srv_props()</code> after <code>srv_init()</code> has completed.
Action	Except for <code>SRV_S_TRACEFLAG</code> , set all properties that can be set through <code>srv_props()</code> before calling <code>srv_init()</code> .
Additional information	Symbolic constant: <code>SRV_EPROPCFG</code>

## Error 16263

Severity	10
Message text	<code>%1!: Bad value (%2!) for property %3!.</code>
Explanation	The specified API has received a non-integer buffer.
Action	See Table 3-147 on page 445 for <code>srv_thread_props()</code> property characteristics, or to Table 3-86 on page 345 for <code>srv_props()</code> property characteristics.
Additional information	Symbolic constant: <code>SRV_EBADINTPROP</code>

## Error 16264 (Deprecated)

Severity	10
Message text	<code>%1!: invalid capability mask for %2!.</code>

Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECAPMASK

## Error 16265

Severity	10
Message text	<code>%1!: property: %2! not supported on this platform.</code>
Explanation	srv_props() has failed to enable SRV_S_PREEMPT.
Action	Verify that Open Server is linked using thread-safe libraries (named *_r).
Additional information	Symbolic constant: SRV_EPROPNOISUPPORT

## Error 16266

Severity	10
Message text	<code>%1!: Property: %2! cannot be cleared.</code>
Explanation	srv_props() has failed to reset the specified property using CS_CLEAR.
Action	Some properties are read-only. See Table 3-86 on page 345 for a list of properties that can be cleared.
Additional information	Symbolic constant: SRV_ENOPROPCLEAR

## Error 16267

Severity	10
Message text	<code>%1!: property - %2! invalid for current tds version (%3!).</code>
Explanation	Possible causes:

	<ul style="list-style-type: none"> <li>• <code>srv_negotiate()</code> has failed because the client application used a TDS version earlier than 5.0.</li> <li>• <code>srv_thread_props()</code> has failed to set the specified property because the client application's TDS level does not support it.</li> </ul>
Action	Upgrade the client application. If the client application uses DB-Library™ APIs, call <code>dbsetversion(DBVERSION_100)</code> .
Additional information	Symbolic constant: <code>SRV_ENOPROPTDS</code>

## Error 16268 (Deprecated)

Severity	N/A
Message text	<code>%!!: Cannot get property %2! before srv_init.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_ENOSERVNAME</code>

## Error 16269

Severity	15
Message text	<code>Final DONE not sent by event handler.</code>
Explanation	Open Server has detected that a <code>srv_senddone(SRV_DONE_FINAL)</code> was not sent when required. Typically, this is a programming error where a path of execution exists within an event handler, which results in the handler terminating without calling <code>srv_senddone(SRV_DONE_FINAL)</code> .
Action	Look for preceding errors to determine which event handler did not send <code>SRV_DONE_FINAL</code> . Enable the <code>SRV_TR_EVENT</code> trace flag to see the event handlers that are running at any given time.
Additional information	Symbolic constant: <code>SRV_ENOFDONE</code>

## Error 16270 (Deprecated)

Severity	N/A
Message text	%!: SRV_DONE_FLUSH cannot be set with SRV_DONE_ERROR.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EERROR

## Error 16271

Severity	N/A
Message text	Changed database context to '%!''.
Explanation	Used internally for the Adaptive Server 5701 changed database message.
Action	N/A
Additional information	Symbolic constant: SRV_EENVDB

## Error 16272

Severity	N/A
Message text	Changed language setting to '%!''.
Explanation	Used internally for the Adaptive Server 5703 changed language message.
Action	N/A
Additional information	Symbolic constant: SRV_EENVLANG

## Error 16273

Severity	N/A
Message text	Changed client character set setting to '%!''.

Explanation	Used internally for the Adaptive Server 5704 changed character set message.
Action	N/A
Additional information	Symbolic constant: SQLSRV_ENVCHAR

## Error 16274

Severity	10
Message text	<code>%!:</code> no cursor with id %2! or name '%3!'.
Explanation	The specified API cannot find the given cursor.
Action	Open Server is responsible for detecting and handling cursor requests from client applications. Multiple cursors may be open simultaneously; verify that the cursor that Open Server is working on in the SRV_CURSOR event handler is correct.
Additional information	Symbolic constant: SRV_ENOCUR

## Error 16275 (Deprecated)

Severity	N/A
Message text	<code>%!:</code> not in SRV_CURSOR event handler.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOCUREVENT

## Error 16276

Severity	10
Message text	<code>%!:</code> No cursor parameters present.
Explanation	The client application did not send the cursor parameters it declared.

Action	Run Ribo to compare the parameters that the client application declared against the parameters sent to Open Server.
Additional information	Symbolic constant: SRV_ENOCURPARAMS

## Error 16277 (Deprecated)

Severity	N/A
Message text	%1!: No cursor parameter %2! present.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOCURPARAM

## Error 16278

Severity	10
Message text	%1!: No cursor update columns present.
Explanation	srv_descfmt() has failed to retrieve SRV_UPCOLDATA.
Action	Run Ribo between the client application and Open Server to check if the client application declared and sent columns for updating.
Additional information	Symbolic constant: SRV_ENOCURUPCOLS

## Error 16279

Severity	10
Message text	%1!: No cursor update column %2! present.
Explanation	The value in SRV_CURDESC <i>numupcols</i> field is smaller than the <i>item</i> value passed to <i>srv_descfmt()</i> .
Action	When retrieving SRV_UPCOLDATA through <i>srv_descfmt()</i> , make sure <i>item</i> is greater than <i>numupcols</i> .



Additional information      Symbolic constant: SRV\_ENOCURUPCOL

## Error 16280

Severity                      10

Message text                      %! : Cursor '%2!' has already been declared.

Explanation                      A cursor with a duplicate name has been declared.

Action                              Run Ribo between the client application and Open Server. Search the CURDECLARE tokens for the duplicate declaration and make appropriate corrections in the client application's code.

Additional information      Symbolic constant: SRV\_EDUPCUR

## Error 16281

Severity                      10

Message text                      %! : Invalid cursor descriptor.

Explanation                      `srv_cursor_props()` has failed because the current cursor command (*curcmd*) has been associated with an invalid command option (*cmdoptions*).

Action                              See Table 2-14 on page 70 for a list of valid *cmdoptions* for a given *curcmd*.

Additional information      Symbolic constant: SRV\_EBADCUR

## Error 16282

Severity                      10

Message text                      %! : No cursor row description present.

Explanation                      The connection between the client application and Open Server may have failed before Open Server received the TDS token describing the row.

Action                              If you can regenerate the error, run Ribo between Open Server and the client application to see if the client application sends the ROWFMT token.

Additional information      Symbolic constant: SRV\_ENOCURROWDESC

## Error 16283

Severity                    10

Message text                %! : Cursor row fetch count %2! exceeded.

Explanation                `srv_xferdata()` has attempted to send more rows than the client application has set for the `CS_CURSOR_ROWS` property. If the client application sets the `CS_CURSOR_ROWS` property, Open Server must conform to this request.

Action                      See Table 2-15 on page 73 for the proper call sequence to handle this type of request. Review *ctos* for an illustration of how to use `srv_xferdata()` when the client application has set a fetch count.

Additional information      Symbolic constant: SRV\_EROWCNT

## Error 16284

Severity                    10

Message text                %! : Cursor info must sent via `srv_cursor_props()` before sending `DONE`.

Explanation                Open Server has called `srv_senddone()` before a required `srv_cursor_props(CS_SET)` call completed.

Action                      Verify that:

- Open Server has completed all required `srv_cursor_props()` responses listed in Table 2-15 on page 73 before calling `srv_senddone()`.
- You are using a version of Open Server that is not affected by CR #310107.

Additional information      Symbolic constant: SRV\_ENOCURSENT

## Error 16285

Severity                    10

Message text	%!!: Cursor info was already sent via <code>srv_cursor_props()</code> .
Explanation	Calling <code>srv_cursor_props()</code> multiple times can cause this error.
Action	Remove any unnecessary calls to <code>srv_cursor_props()</code> .
Additional information	Symbolic constant: <code>SRV_ECURSENT</code>

## Error 16286

Severity	10
Message text	%!!: Error sending cursor info.
Explanation	<code>srv_cursor_props()</code> has failed when it responded to a client application's <code>CS_CURSOR_INFO</code> request and sent a TDS token to the server. This error is typically caused by transient network errors.
Action	Look for other errors that might indicate a network disconnection. If you can reproduce the error, run <code>Ribo</code> between Open Server and the target server to get more information.
Additional information	Symbolic constant: <code>SRV_ENOSENDCUR</code>

## Error 16287 (Deprecated)

Severity	N/A
Message text	%!!: No key columns were described for the current cursor row.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_ENOKEY</code>

## Error 16288 (Deprecated)

Severity	N/A
Message text	%1!: No key column %2! present.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOKEYCOL

## Error 16289

Severity	10 or 20
Message text	%1!: No dynamic SQL statement '%2!' present.
Explanation	If <code>srv_dynamic()</code> cannot find the specified statement, the severity is 20. If <code>srv_cursor_props()</code> cannot find the specified statement, the severity is 10.
Action	Verify that the <code>SRV_DYN_ID</code> provided to <code>srv_dynamic()</code> is valid. When using <code>srv_cursor_props()</code> , the text of the cursor statement is actually the name of the prepared statement. Make sure the name is the same as when it was declared.
Additional information	Symbolic constant: SRV_ENODYN

## Error 16290 (Deprecated)

Severity	N/A
Message text	%1!: not in SRV_DYNAMIC event handler.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENODYNEVENT

## Error 16291

Severity	10
Message text	%!!: No dynamic SQL parameters present.
Explanation	The remote server returned dynamic input or output parameters to Open Server, but Open Server has failed to describe these parameters when returning them to the client application.
Action	Always call <code>srv_descfmt(SRV_DYNDATA)</code> to describe the parameters.
Additional information	Symbolic constant: <code>SRV_ENODYNPARAMS</code>

## Error 16292 (Deprecated)

Severity	N/A
Message text	%!!: No dynamic SQL parameter %2! present.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_ENODYNPARAM</code>

## Error 16293

Severity	10
Message text	%!!: Dynamic SQL statement '%2!' has already been prepared.
Explanation	Open Server has received a <code>DYNAMIC</code> token from the client application containing a duplicate statement ID. State checking in the client libraries typically prevents this error from occurring.
Action	If you can regenerate the problem, run <code>Ribo</code> between the client application and Open Server to get information about the <code>DYNAMIC</code> tokens that the client application sends.
Additional information	Symbolic constant: <code>SRV_EDUPDYN</code>

## Error 16294 (Deprecated)

Severity	N/A
Message text	<code>%!:: Incomplete dynamic SQL parameter-list -- %2! parameters undescribed.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDYNINC

## Error 16295

Severity	15
Message text	<code>%!:: Error sending dynamic SQL info.</code>
Explanation	Open Server has failed to send the DYNAMIC token to the client application possibly due to a network or character set conversion error
Action	Look for additional errors in the client error log that might indicate which type of failure occurred. If you can regenerate the error, running Ribo between the client application and Open Server might help you determine how to fix the error (changing character sets, for example).
Additional information	Symbolic constant: SRV_ENOSENDDYN

## Error 16296

Severity	10
Message text	<code>%!:: Dynamic SQL statement info must be sent via srv_dynamic() before sending %2!.</code>
Explanation	Open Server did not successfully call one of the required <code>srv_dynamic()</code> calls before attempting to call <code>srv_xferdata()</code> , <code>srv_senddone()</code> , or <code>srv_descfmt()</code> .
Action	See Table 2-19 on page 85 for the appropriate sequencing of <code>srv_dynamic()</code> calls.
Additional information	Symbolic constant: SRV_ENODYNSENT

## Error 16297

Severity	10
Message text	<code>%!!: Dynamic SQL statement info was already sent via srv_dynamic().</code>
Explanation	Open Server has called <code>srv_dynamic()</code> for the same information more than once.
Action	Check the Open Server usage of <code>srv_dynamic()</code> and ensure it does not attempt to send the same information more than once. See Table 2-19 on page 85 for the appropriate sequencing of <code>srv_dynamic()</code> calls.
Additional information	Symbolic constant: <code>SRV_EDYNSENT</code>

## Error 16298

Severity	10
Message text	<code>%!!: Invalid call for this %2!.</code>
Explanation	The specified API has been called inappropriately for the specified type of command.
Action	Review Chapter 3, “Routines,” for the appropriate use of the specified API. For example, if the error is about calling <code>srv_langlen()</code> during a cursor command, the documentation states that <code>srv_langlen()</code> can be used only to process language strings in cursor declare or update statements.
Additional information	Symbolic constant: <code>SRV_EBADCTXT</code>

## Error 16299 (Deprecated)

Severity	N/A
Message text	<code>%!!: Invalid response '%2!' to '%3!'.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_EBADRESP</code>

## Error 16300

Severity	15
Message text	Unknown token 0x%1! received from client.
Explanation	Open Server has received an invalid or unknown token from the client application while trying to read TDS tokens related to cursors.
Action	If you can reproduce the error, run Ribo between the client application and Open Server to see which token is causing the error. If you cannot reproduce the error, it may indicate that the problem is caused by corruption in the network or in the client application's memory.
Additional information	Symbolic constant: SRV_EBADTOKEN

## Error 16301

Severity	10
Message text	%1!: not in SRV_CONNECT event handler.
Explanation	srv_setloginfo() has been called outside the SRV_CONNECT connection event handler.
Action	Call srv_setloginfo() only from the SRV_CONNECT event handler.
Additional information	Symbolic constant: SRV_ENOCONN

## Error 16302

Severity	10
Message text	%1!: Cursor ID %2! has already been assigned.
Explanation	Open Server has detected a duplicate cursor ID while attempting to respond to a client application's request to declare a cursor through srv_cursor_props(). Cursor IDs must be unique on each connection handle.
Action	Run Ribo between the client application and Open Server to see how the client application declares each cursor.
Additional information	Symbolic constant: SRV_EDUPCURID



## Error 16303

Severity	10
Message text	Text/image column %1! was not completely sent.
Explanation	srv_send_text(), srv_text_info(), or srv_senddone() has detected that not all of the specified column's text or image data has been sent. text and image data is transferred to a client application using srv_bind(), followed by srv_xferdata(), srv_send_data(), or srv_send_text().
Action	Review the Open Server usage of these routines to make sure they are reading and sending all of the data for the specified column.
Additional information	Symbolic constant: SRV_ETEXTLEFT

## Error 16304

Severity	10
Message text	1!: Compute-id %2! exceeded limit -- must be between 1 and %3!.
Explanation	srv_alt_header() has received an <i>altid</i> value that is not within the valid range.
Action	Initialize <i>altid</i> properly and ensure that it is within the specified range.
Additional information	Symbolic constant: SRV_EBADALTID

## Error 16305

Severity	10
Message text	%1!: NULL data sent for column %2! -- text pointer length was 0.
Explanation	The CS_IODESC <i>textptrlen</i> field has been set to zero, which indicates that <i>srv_send_text()</i> will be sending a NULL column to the server. However, the <i>buflen</i> value passed to <i>srv_send_text()</i> is not zero, which indicates that there is data to be sent.

Action	Compare the data and data length described in <code>srv_text_info()</code> with the <i>buffer</i> and <i>buflen</i> parameters passed to <code>srv_send_text()</code> . Make sure all of these parameters agree.
Additional information	Symbolic constant: <code>SRV_ENOTXTSENT</code>

## Error 16306

Severity	10
Message text	<code>%!:</code> text or image pointer not defined.
Explanation	<code>srv_send_text()</code> has detected that the <code>CS_IODESC</code> <i>textptrlen</i> field, which should be using the same value as <code>srv_text_info()</code> , is smaller than zero.
Action	Correct the value of <i>textptrlen</i> in the <code>CS_IODESC</code> structure that is passed to <code>srv_text_info()</code> .
Additional information	Symbolic constant: <code>SRV_ENOTEXTPTR</code>

## Error 16307 (Deprecated)

Severity	N/A
Message text	<code>%!:</code> there are no parameters for current rpc: <code>%2!</code> .
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: <code>SRV_ENOPARAMRPC</code>

## Error 16308 (Deprecated)

Severity	N/A
Message text	Invalid capability ( <code>%!!</code> ) in <code>%2!</code> .
Explanation	N/A
Action	N/A

Additional information      Symbolic constant: SRV\_ECAPABILITY

## Error 16309

Severity                    10

Message text                Invalid capability type (%1!) in %2!.

Explanation                Open Server has detected an invalid TDS capability type while reading the TDS\_CAPABILITY token.

Action

- Run Ribo between the client application and Open Server.
- Examine the login record to make sure that a valid Sybase client is attempting to connect to the server.
- Examine the *Type* field of the CAPABILITY token and make sure it is set to either CAP\_REQUEST or CAP\_RESPONSE. If it is not, check for network errors leading to corrupt packets.

Additional information      Symbolic constant: SRV\_ECAPTTYPE

## Error 16310

Severity                    15

Message text                %1!: Fatal error while %2! for spid: %3!.

Explanation                Open Server was reading a stream of data, such as capability token, option command, or parameter format token, when the stream ended prematurely. This error typically occurs when the client application disconnects abruptly while Open Server is trying to read from the client. This error may be caused by network corruption.

Action                        Increase the client application and Open Server login and command timeouts.

Additional information      Symbolic constant: SRV\_EBADIO

## Error 16311 (Deprecated)

Severity	N/A
Message text	%1!: has illegal value (%2!) for msgid.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBADMSGID

## Error 16312 (Deprecated)

Severity	N/A
Message text	%1!: there are no parameters for current MSG.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOPARAMMSG

## Error 16313 (Deprecated)

Severity	N/A
Message text	%1!: there is no parameter #%2! in the current %3!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOPARAMNUM

## Error 16314 (Deprecated)

Severity	N/A
Message text	%1!: parameter description is not complete -- %2! %3!

not described.

Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EPARAMINC

## Error 16315

Severity	15
Message text	<code>%1!: Protocol error occurred: %2!.</code>
Explanation	Typically, this error occurs when a non-Sybase product sends a packet to an Open Server. telnet hostname port or other port monitoring software raises this error. Only Sybase-based products should connect to the Open Server listening port.
Action	If the error occurs because a non-Sybase product is connecting to the server, enter the code for Open Server to ignore or suppress the error. If the error leads to a more severe problem, use a network sniffer to identify the software that is trying to connect to Open Server, then contact Sybase Technical Support.
Additional information	Symbolic constant: SRV_EPROTERR

## Error 16316 (Deprecated)

Severity	N/A
Message text	<code>%1!: there are no parameters for current Error message.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOPARAMEED

## Error 16317

Severity	10
Message text	%!!: Dynamic SQL data should be defined as CS_DESCIN (input parameter format) or CS_DESCOUT (output column format).
Explanation	srv_xferdata() or srv_descfmt() has detected that the CS_DATAFMT <i>status</i> field has not been set to CS_DESCIN or CS_DESCOUT, which is required for dynamic SQL statements that include input or output parameters.
Action	See Table 2-19 on page 85 for the list of valid dynamic SQL requests and responses.
Additional information	Symbolic constant: SRV_ENODYNDESC

## Error 16318 (Deprecated)

Severity	N/A
Message text	%!!: There are still text or image columns left to be read via blk_gettext().
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETXTLEFT

## Error 16319

Severity	10
Message text	%!!: Failed sending table column information.
Explanation	blk_srvinit() has failed to send the table column information. Unless the insert statement from the client application contains the with nodescrbe clause, Open Server must use blk_srvinit() to describe the current server table column descriptions in the CS_BLKDESC structure to the client application.

Action	Look for additional errors, such as network disconnects, that describe the underlying cause of the blk_srvinit() failure.
Additional information	Symbolic constant: SRV_EBLKINIT

## Error 16320

Severity	10
Message text	Could not load localization information for locale %!.
Explanation	The specified character set, language, or locale name cannot be found.
Action	Look in <i>\$\$SYBASE/locales</i> for the specified localization information and make sure that the files' read permission is enabled. Review "International support" on page 101.
Additional information	Symbolic constant: SRV_ENONATLOCALE

## Error 16321

Severity	10
Message text	Invalid CS_LOCALE, or not allocated using Open Server CS_CONTEXT.
Explanation	srv_thread_props() has failed to set or retrieve the SRV_T_LOCALE value because the CS_LOCALE pointer is invalid.
Action	Use cs_loc_alloc() to properly allocate and initialize the CS_LOCALE structure.
Additional information	Symbolic constant: SRV_EBADLOCALE

## Error 16322

Severity	10
Message text	Could not allocate localization information for thread %!.

Explanation	A call to <code>cs_locale()</code> or <code>cs_loc_alloc()</code> has failed.
Action	Use a trace utility such as <code>truss</code> to identify which of these caused the error: <ul style="list-style-type: none"><li>• Insufficient memory to load the locale information.</li><li>• Missing localization files.</li><li>• The <i>action</i> parameter of <code>cs_locale()</code> is set to <code>CS_SET</code> but the <i>*buffer</i> locale name cannot be found in the Sybase <i>locales</i> file.</li><li>• The <i>action</i> parameter of <code>cs_locale()</code> is set to <code>CS_GET</code> but <i>buflen</i> indicates that the <i>*buffer</i> data space is too small.</li></ul>
Additional information	Symbolic constant: <code>SRV_ENOLOCALE</code>

## Error 16323

Severity	10
Message text	<code>Could not load localization information for character set %1! language %2!.</code>
Explanation	<code>sp_serverinfo</code> or <code>cs_locale()</code> cannot find the specified character set or language.
Action	Use a trace utility such as <code>truss</code> to see where Open Server is searching for character set information and if the file permissions are correct. If the <i>locales</i> files exist and are readable, validate the request by reviewing <i>Open Client and Open Server International Developers Guide</i> and “International support” on page 101.
Additional information	Symbolic constant: <code>SRV_ENOCHARLOCALE</code>

## Error 16324

Severity	10
Message text	<code>%1!: data for column no: %2! cannot be NULL.</code>
Explanation	The specified API has detected an attempt to pass <code>NULL</code> data to a column that is not nullable.



Action	Confirm that the remote column definition accepts NULL. If so, make sure that the CS_CANBENULL bit of the CS_DATAFMT <i>status</i> field is set. If the specified API accepts a null value for the <i>indicator</i> parameter, verify that it is set properly to indicate NULL data is being passed.
Additional information	Symbolic constant: SRV_EWRONGNULL

## Error 16325

Severity	10
Message text	<code>srv_init</code> has not been called.
Explanation	An attempt has been made to call an Open Server API before <code>srv_init()</code> completed.
Action	Use <code>srv_init()</code> to initialize Open Server before calling any other API, with the exception of <code>srv_props()</code> and <code>srv_version()</code> .
Additional information	Symbolic constant: SRV_ENOINIT

## Error 16326

Severity	15
Message text	<code>wrong argument length</code> has been passed into %1! when %2!.
Explanation	<code>srv_options()</code> has detected an invalid parameter length while reading an option command from the client application.
Action	Run Ribo between the client application and Open Server. Review “Arg Length” for the OPTIONCMD token and make corrections to the client application as needed.
Additional information	Symbolic constant: SRV_EWRONGARGLEN

## Error 16327

Severity	10
----------	----

Message text	%1!: No remote server information was sent by the client.
Explanation	srv_negotiate() has attempted to retrieve SRV_NEG_REMPWD, although the client application did not send SRV_NEG_LOCPWD.
Action	Ensure that the SRV_NEG_LOCPWD response to a SRV_NEG_ENCRYPT challenge is always present.
Additional information	Symbolic constant: SRV_ENONEGPWD

## Error 16328

Severity	10
Message text	%1!: Failed waiting for client response.
Explanation	srv_negotiate() has failed while waiting for a response from the client application.
Action	See if there are network interruptions preventing Open Server from receiving the client application's response. If you can reproduce the error, run Ribo between the client application and Open Server to view the exchange of tokens during a password encryption challenge.
Additional information	Symbolic constant: SRV_EREADFAILED

## Error 16329

Severity	10
Message text	%1!: Incorrect response type %2! received in response to %3!.
Explanation	srv_negotiate() has received an incorrect response type for the specified type of challenge.
Action	If you can regenerate the problem, run Ribo between the client application and Open Server to review the tokens being exchanged during the challenge. Review <i>Open Client Client-Library/C Reference Manual &gt; Client-Library Topics &gt; Security features &gt; Adaptive Server Enterprise security features</i> for the correct response types.

Additional information      Symbolic constant: SRV\_EWRONGMSG

## Error 16330

Severity                    15

Message text                `Pass-through is incomplete for spid: %!.`

Explanation                Open Server, while cleaning up after the specified spid that has been running in passthrough mode, has detected that not all tokens have been read.

Action                      Call `srv_recvpass thru()` or `srv_sendpass thru()` until the *info*p value is `SRV_I_PASSTHRU_EOM`, which indicates that all tokens have been read from the client application.

Additional information      Symbolic constant: SRV\_EINCPASSTHRU

## Error 16331

Severity                    10

Message text                `Recursive Calls to srv__seterr().`

Explanation                An Open Server error handler is reporting an error when a second error occurs that triggered the error handler a second time (or recursively). For example, an error handler is trying to return an error to the client application, but the client dropped off the network, so the error handler is triggered again to report that it cannot send the error to the client application.

Action                      Correct the situation that raised the original error.

Additional information      Symbolic constant: SRV\_ERECURSIVE

## Error 16332

Severity                    10

Message text                `%!: %! result types are being withheld or not supported.`

Explanation	srv_descfmt() or the srv_msg() is signalling that the specified result type is being withheld.
Action	Run Ribo between the client application and Open Server, and review the CAPABILITY tokens to see the capabilities that the client application and Open Server agreed to support.
Additional information	Symbolic constant: SRV_ENORESCAP

## Error 16333

Severity	10
Message text	%!:: Cannot call in SRV_START handler.
Explanation	The specified routine cannot be called in the SRV_START event handler.
Action	See Chapter 3, “Routines,” to see where you can use the specified routine.
Additional information	Symbolic constant: SRV_EINSTART

## Error 16334

Severity	10
Message text	SRV_START event aborted srv_run.
Explanation	srv_run() has terminated abnormally because the SRV_START event handler failed.
Action	Check additional error messages that might indicate the underlying cause of the SRV_START event handler failure. Correct those failure conditions.
Additional information	Symbolic constant: SRV_ESTARTEVENT

## Error 16335

Severity	10
Message text	%!:: Null indicator not set when datalength = 0 for

itemno %2!.

Explanation	The specified routine has detected that the null indicator parameter is not being set properly when the data length of the specified item is zero.
Action	Review Chapter 3, “Routines,” for the specified routine’s usage of the null indicator parameter.
Additional information	Symbolic constant: SRV_ENONULLIND

## Error 16336

Severity	10
Message text	%1!: Bad value(%2!) for precision in CS_DATAFMT.
Explanation	The specified API has detected an invalid CS_DATAFMT <i>precision</i> value for a CS_NUMERIC_TYPE or CS_DECIMAL_TYPE variable.
Action	See <i>cstypes.h</i> , which defines, from CS_MIN_PREC to CS_MAX_PREC, the valid precision values. The default precision is CS_DEF_PREC.
Additional information	Symbolic constant: SRV_EBADPREC

## Error 16337

Severity	10
Message text	%1!: Bad value(%2!) for scale in CS_DATAFMT.
Explanation	The specified API has detected an invalid CS_DATAFMT <i>scale</i> value for a CS_NUMERIC_TYPE or CS_DECIMAL_TYPE variable. <i>scale</i> must be less than or equal to <i>precision</i> .
Action	See <i>cstypes.h</i> , which defines, from CS_MIN_SCALE to CS_MAX_SCALE, the valid scale values. The default scale is CS_DEF_SCALE.
Additional information	Symbolic constant: SRV_EBADSCALE

## Error 16338

Severity	20
Message text	<code>Fatal error detected by srv_run.</code>
Explanation	A fatal error in <code>srv_run()</code> has prevented Open Server from starting.
Action	Check additional error messages, for example file limit or insufficient memory, which should indicate the underlying cause of the failure.
Additional information	Symbolic constant: <code>SRV_ERUN</code>

## Error 16339

Severity	20
Message text	<code>Could not take coroutine control back in %1!.</code>
Explanation	This is a fatal error for preemptive Open Servers ( <code>SRV_S_PREEMPT=CS_TRUE</code> ).
Action	This error indicates an internal Open Server bug; please report it to Sybase Technical Support.
Additional information	Symbolic constant: <code>SRV_ETAKECOROUTINE</code>

## Error 16340

Severity	20
Message text	<code>Could not take coroutine control back in %1!.</code>
Explanation	This is a fatal error for preemptive Open Servers ( <code>SRV_S_PREEMPT=CS_TRUE</code> ).
Action	This error indicates an internal Open Server bug; please report it to Sybase Technical Support.
Additional information	Symbolic constant: <code>SRV_ERELEASECOROUTINE</code>

## Error 16341 (Deprecated)

Severity	N/A
Message text	Could not take system mutex in %!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETAKEMUTEX

## Error 16342 (Deprecated)

Severity	N/A
Message text	Could not release system mutex in %!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ERELEASEMUTEX

## Error 16343 (Deprecated)

Severity	N/A
Message text	There was an error responding to spid %!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EMMSGSENDREPLY

## Error 16344 (Deprecated)

Severity	N/A
Message text	A bad read buffer was freed.

Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EFREEBUF

## **Error 16345 (Deprecated)**

Severity	N/A
Message text	There are currently no network read buffers available.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ENOREADBUFFERS

## **Error 16346 (Deprecated)**

Severity	N/A
Message text	Received message for a dead or disconnected client.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDEADCLIENT

## **Error 16347 (Deprecated)**

Severity	N/A
Message text	A message for spid %1! was skipped.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EMSGSKIPPED



## Error 16348 (Deprecated)

Severity	N/A
Message text	Could not send login acceptance message to procid %!!
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECLIENTACCEPT

## Error 16349 (Deprecated)

Severity	N/A
Message text	Spid %!! has either died or been disconnected.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ECLIENTDISCONNECT

## Error 16350 (Deprecated)

Severity	N/A
Message text	Error creating direct queue %!!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EQUEUECREATE

## Error 16351 (Deprecated)

Severity	N/A
Message text	Error setting a port with s\$control().

## Error 16352 (Deprecated)

---

Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ESETTINGFASTQ

## Error 16352 (Deprecated)

Severity	N/A
Message text	Could not allocate memory for TCP/IP network control structure.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETCPALLFAIL

## Error 16353 (Deprecated)

Severity	N/A
Message text	Failed to assign TCP/IP device %!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETCPNETASSIGN

## Error 16354 (Deprecated)

Severity	N/A
Message text	Cannot set TCP/IP socket option SO_KEEPLIVE.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETCPSETKEEP

## Error 16355 (Deprecated)

Severity	N/A
Message text	Cannot set TCP/IP socket option SO_REUSEADDR.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETCPSETREUSE

## Error 16356 (Deprecated)

Severity	N/A
Message text	Error stating TCP/IP accept wait.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETCPACCEPTW

## Error 16357 (Deprecated)

Severity	N/A
Message text	Failed to assign a new channel for client connection.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETCPASSIGN

## Error 16358 (Deprecated)

Severity	N/A
Message text	No link entries available for client connection.

## Error 16359 (Deprecated)

---

Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETCPNOLINK

## Error 16359 (Deprecated)

Severity	N/A
Message text	Failed to set attention ast on socket.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETCPATTNAST

## Error 16360

Severity	20
Message text	WaitForSingleObject() failed in %!.
Explanation	WaitForSingleObject() or comn_waitfor_event() has failed in the specified API. The thread that owned the specified mutex object terminated before releasing the mutex.
Action	Look for additional errors that might indicate why the owning thread terminated without releasing the object.
Additional information	Symbolic constant: SRV_EWAITFAIL

## Error 16361

Severity	20
Message text	CreateEvent() failed in %!.
Explanation	CreateEvent() or comn_create_event() has failed in the specified API, possibly due to insufficient resources, such as memory.

Action	Find the underlying operating system error. On UNIX, use a process tracing utility such as truss. On Microsoft Windows, contact Sybase Technical Support and request an instrumented version of the Sybase libraries that call GetLastError().
Additional information	Symbolic constant: SRV_ECREATEEVENT

## Error 16362

Severity	20
Message text	<code>SetEvent() failed in %!!</code> .
Explanation	SetEvent() or comn_signal_event() has failed in the specified API.
Action	Find the underlying operating system error. On UNIX, use a process tracing utility such as truss. On Microsoft Windows, contact Sybase Technical Support and request an instrumented version of the Sybase libraries that call GetLastError().
Additional information	Symbolic constant: SRV_ESETEVENT

## Error 16363

Severity	20
Message text	<code>Could not start coroutine mode.</code>
Explanation	Internal routines try to create and take control of a mutex semaphore.
Action	Contact Sybase Technical Support for assistance with this fatal error.
Additional information	Symbolic constant: SRV_ESTARTCOROUTINE

## Error 16364

Severity	20
Message text	<code>Could not end coroutine mode.</code>

Explanation	Thread clean-up has failed.
Action	Contact Sybase Technical Support for assistance with this fatal error.
Additional information	Symbolic constant: SRV_EENDCOROUTINE

## Error 16365

Severity	20
Message text	<code>CreateMutex() failed in %!.</code>
Explanation	CreateMutex() (Microsoft Windows) or pthread_mutex_init() (UNIX) has failed to create a mutex in the specified API.
Action	Find the underlying operating system error. On UNIX, use a process tracing utility such as truss. On Microsoft Windows, contact Sybase Technical Support and request an instrumented version of the Sybase libraries that call GetLastError().
Additional information	Symbolic constant: SRV_ECREATEMUTEX

## Error 16366 (Deprecated)

Severity	N/A
Message text	<code>Deleting a mutex failed in %!.</code>
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EDELETEMUTEX

## Error 16367

Severity	10
Message text	<code>%! : %2! field of CS_SERVERMSG argument must be set to %3! for initial call.</code>

Explanation	The application has not set the CS_SERVERMSG status mask to CS_FIRST_CHUNK before the first call to <code>srv_sendinfo()</code> .
Action	Review “ <code>srv_sendinfo</code> ” on page 406 and Table 2-10 on page 62.
Additional information	Symbolic constant: <code>SRV_EBADMSGFIELD</code>

## Error 16368

Severity	10
Message text	<code>%1!: A message was initialized, but not completed.</code>
Explanation	<code>srv_senddone()</code> has detected that an extended error data or part of an information message has not been completely sent.
Action	Review “ <code>srv_sendinfo</code> ” on page 406, “ <code>CS_SERVERMSG</code> structure” on page 60, and Table 2-10 on page 62.
Additional information	Symbolic constant: <code>SRV_ENOMSGSENT</code>

## Error 16369

Severity	10 or 20
Message text	<code>Security control layer routine %1! failed in %2!.</code>
Explanation	The specified security routine has failed in the given API.
Action	Use a process trace utility, such as <code>truss</code> , to find supplemental errors specific to the security driver in use. For example, this command returns details of the Kerberos APIs that are involved in the error: <pre>truss -f -o krb.out -u'a.out,libsybskrb,mech_krb5' -p pid</pre>
Additional information	Symbolic constant: <code>SRV_ESCLFUNC</code>

## Error 16370 (Deprecated)

Severity	N/A
Message text	Security control layer initialization failed.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ESCLINIT

## Error 16371

Severity	10
Message text	Security protocol error occurred.
Explanation	An error has occurred while reading a security protocol TDS token from a client application.
Action	If you can reproduce this error, run Ribo between the client application and server to examine which TDS packets are malformed or missing from the exchange.
Additional information	Symbolic constant: SRV_EBADSECPROT

## Error 16372

Severity	10
Message text	Unknown security version (%1!) was received.
Explanation	An invalid security version has been detected during the login phase security negotiation.
Action	Run Ribo between the client application and server and examine the TDS_SEC_SECSSESS token, which follows the TDS_MSG_SEC_OPAQUE token. TDS_SEC_SECSSESS consists of five parameters, first of which is the TDS security version of type TDS_INTN, which should have a value of 50.
Additional information	Symbolic constant: SRV_EBADSECVER



## Error 16373

Severity	10
Message text	Unknown security mechanism object identifier was received.
Explanation	The security mechanism requested by the client application is unknown to the server.
Action	Use a process trace utility, such as <code>truss</code> , to find supplemental errors specific to the security driver in use. For example, this command returns details about which Solaris Native Kerberos and Sybase Kerberos related APIs are involved in the error: <pre>truss -f -o krb.out -u'a.out,libsybskrb,mech_krb5' -p pid</pre>
Additional information	Symbolic constant: <code>SRV_EBADSECMECHOID</code>

## Error 16374

Severity	10
Message text	Security session expired.
Explanation	The session for your security mechanism has expired.
Action	Reinitialize your security mechanism's session.
Additional information	Symbolic constant: <code>SRV_ESESSTO</code>

## Error 16375

Severity	10
Message text	Security session credentials expired.
Explanation	The credentials for your security mechanism have expired.
Action	Obtain a fresh set of credentials. For example, when using Kerberos, you need to issue the <code>kinit</code> command to get new credentials.
Additional information	Symbolic constant: <code>SRV_ECREDITO</code>

## Error 16376 (Deprecated)

Severity	N/A
Message text	An invalid security signature was received.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBADSIGN

## Error 16377 (Deprecated)

Severity	N/A
Message text	Security decryption failed.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBADSEAL

## Error 16378

Severity	10
Message text	Security mechanism name is too long.
Explanation	The length of the security mechanism's name is greater than 32 characters (the maximum).
Action	Ensure that the security mechanism's name is the same length, or smaller than, the maximum length of 32 characters.
Additional information	Symbolic constant: SRV_ESECMECHLEN

## Error 16379

Severity	10 or 20
----------	----------

Message text	List manager operation failed.
Explanation	If this error's severity is 10, it typically indicates a problem with the secmech entry found in the <i>interfaces</i> file. If the severity is 20, it indicates that there is a problem allocating a list to store the security mechanisms.
Action	If the severity is 10, check the secmech entry in the <i>interfaces</i> file and make sure it is valid by comparing it to the secmech entry found in <i>\$SYBASE/config/objectid.dat</i> . If the severity is 20, use an application tracing utility such as truss to see if there are any resource issues (for example, insufficient memory).
Additional information	Symbolic constant: SRV_ELMAN

## Error 16380

Severity	10
Message text	A bad security message token was received.
Explanation	One of the parameters following the TDS_MSG_SEC_OPAQUE message token has encountered an error during the login phase security negotiation.
Action	Run Ribo to examine the five parameters following the TDS_MSG_SEC_OPAQUE message token: <ul style="list-style-type: none"> <li>• First parameter – TDS security version of type TDS_INTN. Value: 50.</li> <li>• Second parameter – security message of type TDS_INTN. Value: TDS_SEC_SECSCESS.</li> <li>• Third parameter – the object identifier (OID) of the security mechanism being used, and is of type TDS_VARBINARY.</li> <li>• Fourth parameter – an opaque security token of type TDS_LONGBINARY.</li> <li>• Fifth parameter – security services requested, and is of type TDS_INTN.</li> </ul>
Additional information	Symbolic constant: SRV_EBADSECTOK

## Error 16381

Severity	10
Message text	Requested security service is not supported.
Explanation	Not all of the security services are supported by all security mechanisms.
Action	Determine whether a given service is supported by the current security mechanism. Call <code>ct_config()</code> or <code>ct_con_props()</code> , setting <i>action</i> to <code>CS_SUPPORTED</code> , <i>buffer</i> to the address of a <code>CS_BOOL</code> variable, and <i>property</i> as the symbolic property constant that represents the security service. Review “Security services” on page 174.
Additional information	Symbolic constant: <code>SRV_ESECSRVNOTSUP</code>

## Error 16382

Severity	15
Message text	The configured maximum number of mutexes have already been created.
Explanation	<code>srv_createmutex()</code> has failed because the maximum number of allowed mutexes has been exceeded.
Action	Limit the number of mutexes to <code>SRV_MAX_MutexES</code> , which is the maximum number of mutexes defined in <i>oscompat.h</i> .
Additional information	Symbolic constant: <code>SRV_EMAXMUTEXES</code>

## Error 16383

Severity	20
Message text	<code>TlsAlloc()</code> failed in %1!.
Explanation	An attempt to create a thread-specific data key visible to all threads in the process (through <code>pthread_key_create()</code> or <code>TlsAlloc()</code> depending on the platform) has failed. This type of failure is usually due to lack of operating system resources.

Action	Use an application tracing utility such as truss to see the underlying operating system error.
Additional information	Symbolic constant: SRV_ETLSALLOC

## Error 16384 (Deprecated)

Severity	N/A
Message text	N/A TlsFree() failed in %!.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_ETLSFREE

## Error 16385

Severity	20
Message text	TlsSetValue() failed in %!.
Explanation	An attempt to set a thread-specific data key (through TlsSetValue() or pthread_setspecific() depending on the platform) has failed.
Action	This type of failure is usually due to insufficient operating system resources. Use an application tracing utility such as truss to see the underlying operating system error.
Additional information	Symbolic constant: SRV_ETLSSETVAL

## Error 16386

Severity	20
Message text	TlsGetValue() failed in %!.
Explanation	An attempt to get a thread-specific data key using TlsGetValue() has failed.

Action	Contact Sybase Technical Support and request an instrumented version of the Sybase libraries that use GetLastError() to report underlying operating system errors that prevent TlsGetValue() from successfully completing.
Additional information	Symbolic constant: SRV_ETLSGETVAL

## Error 16387 (Deprecated)

Severity	N/A
Message text	Either addr_trantype or addr_tranaddress length is invalid.
Explanation	N/A
Action	N/A
Additional information	Symbolic constant: SRV_EBADTRANADDRLEN

## Error 16388

Severity	10
Message text	%!!: Could not run async manage.
Explanation	Length (%!!) of server name is invalid. The <i>namelen</i> parameter passed to <i>srv_getserverbyname()</i> is invalid.
Action	Specify a <i>namelen</i> parameter that is greater than or equal to 1 but less than 255.
Additional information	Symbolic constant: SRV_ESRVNAMELEN

## Error 16389

Severity	10
Message text	Client does not support HA.
Explanation	<i>srv_send_ctlinfo()</i> has failed because the client application does not support extended high availability (HA) failover.

Action	Run Ribo between the client application and server to see if the client is at TDS level 5.0 or later, and that the CS_CAP_EXTENDEDFAILOVER bit is set to CS_TRUE.
Additional information	Symbolic constant: SRV_ENOSESSION

## Error 16390

Severity	10
Message text	Identifier starting with %! is too long.
Explanation	srv_descfmt() has failed because the client application does not support large identifiers, which are table or column names with length greater than 30 characters.
Action	Run Ribo between the client application and server to see if CS_REQ_LARGEIDENT request capability is set to FALSE or CS_NO_LARGEIDENT is set to TRUE. These settings indicate that the client application does not support large identifiers.
Additional information	Symbolic constant: SRV_ENOLARGEID

## Error 16391

Severity	10
Message text	The point to start (%!) copying characters cannot be greater than the number of characters (%2!) in the request buffer.
Explanation	srv_langcpy() has received an incorrect <i>start</i> parameter value.
Action	Correct the code to make sure the <i>start</i> value is shorter than the length of the language or RPC command.
Additional information	Symbolic constant: SRV_EBADSTART
Version in which this error is raised	12.5.1 ESD #21, 15.0, and later versions.

## Error 16392

Severity	10
Message text	The configured maximum number of message queues have already been created.
Explanation	srv_createmsgq() has been called, but the maximum number of configured message queues are already in use.
Action	Free message queues using srv_deletemsgq() or configure SRV_S_NUMMSGQUEUES to increase the number of message queues.
Additional information	Symbolic constant: SRV_EMAXMQUEUES
Version in which this error is raised	15.0 and later.

## Error 16393

Severity	10
Message text	%1!: Property - %2! is not available.
Explanation	srv_thread_props(.., CS_GET, property, ..) has been called for a property that is not available for the specified thread. Currently, this error occurs only for SRV_T_SESSIONID if the client connection does not provide a session ID. This property is available only for client threads for which the client connection either migrated or failed over.
Action	Retrieve a property value only if the thread is known to have a valid value for the specified property.
Additional information	Symbolic constant: SRV_EPROPNOTAVAIL
Version in which this error is raised	15.0 ESD #14 and later.

## Error 16394

Severity	10
Message text	%1!: Connection cannot migrate.



Explanation	<code>srv_send_ctlinfo(srvproc, SRV_CTL_MIGRATE, ..)</code> has been called for a client thread from a connection that is incapable of migration.
Action	Before attempting to migrate a connection, verify that the connection can be migrated using <code>CS_REQ_MIGRATE</code> .
Additional information	Use <code>srv_capability_info(.., CS_GET, CS_CAP_REQUEST, ..)</code> to test if the connection is capable of migration. Symbolic constant: <code>SRV_ENOTMIGRATABLE</code> .
Version in which this error is raised	15.0 ESD #14 and later.

## Error 16395

Severity	10
Message text	<code>%1!: Migration can no longer be cancelled.</code>
Explanation	A request to cancel a migration has been issued but the client application has already started the migration. This error may also occur when you start a new migration while an earlier migration is still in progress.
Action	Cancel a migration request before exiting the <code>SRV_MIGRATE_STATE</code> event handler.
Additional information	Symbolic constant: <code>SRV_EMIGNOTCANCEL</code>
Version in which this error is raised	15.0 ESD #14 and later.

## Error 16396

Severity	10
Message text	<code>Migration failed but no SRV_MIGRATE_STATE handler was installed.</code>
Explanation	A client connection has failed to migrate, and a <code>SRV_MIGRATE_STATE</code> event handler has not been installed to handle the failure.
Action	Install a <code>SRV_MIGRATE_STATE</code> handler to take the appropriate action when a connection fails to migrate.

Additional information This is an informational message. Symbolic constant: SRV\_EMIGFAILED.  
Version in which this error is raised 15.0 ESD #14 and later.

## Error 16397

Severity 10  
Message text `Property %1! can only be set on a listener thread.`  
Explanation An attempt has been made to set the specified property on another type of thread.  
Action Set the specified property only on a listener thread.  
Additional information Symbolic constant: SRV\_ENOTLISTENER  
Version in which this error is raised 15.5 and later.

## Error 16398

Severity 10  
Message text `Maximum listener count (%1!) would be exceeded.`  
Explanation An attempt has been made to start another listener when the configured number of listeners has already been reached.  
Action Configure SRV\_S\_MAXLISTENERS to increase the number of listeners.  
Additional information Symbolic constant: SRV\_EXMAXLISTEN  
Version in which this error is raised 15.5 and later.

# 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.

**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 Enterprise 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.

<b>character set</b>	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.
<b>character set conversion</b>	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 Enterprise 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.
<b>client</b>	In client/server systems, the client is the part of the system that sends requests to servers and processes the results of those requests.
<b>Client-Library</b>	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.
<b>code set</b>	See <b>character set</b> .
<b>collating sequence</b>	See <b>sort order</b> .
<b>command</b>	In Client-Library, a server request initiated by an application's call to <code>ct_command</code> , <code>ct_dynamic</code> , or <code>ct_cursor</code> and terminated by the application's call to <code>ct_send</code> .
<b>command structure</b>	(CS_COMMAND) A hidden Client-Library structure that Client-Library applications use to send commands and process results.
<b>connection structure</b>	(CS_CONNECTION) A hidden Client-Library structure that defines a client/server connection within a context.
<b>context structure</b>	(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 <code>cs_ctx_alloc</code> and <code>cs_ctx_drop</code> allocate and drop a context structure.
<b>conversion</b>	See character set conversion.
<b>CS-Library</b>	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.
<b>current row</b>	With respect to cursors, the row to which a cursor points. A fetch against a cursor retrieves the current row.
<b>cursor</b>	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.
<b>database</b>	A set of related data tables and other database objects that are organized to serve a specific purpose.  See also <b>scrollable cursor</b> .
<b>datatype</b>	A defining attribute that describes the values and operations that are legal for a variable.
<b>DB-Library</b>	Part of Open Client, a collection of routines for use in writing client applications.
<b>deadlock</b>	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 Enterprise detects deadlocks and resolves them by killing one user's process.
<b>default</b>	Describes the value, option, or behavior that Open Client and Open Server products use when none is explicitly specified.
<b>default database</b>	The database that a user gets by default when he or she logs in to a database server.
<b>default language</b>	<ol style="list-style-type: none"><li>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.</li><li>2. The language that Adaptive Server Enterprise uses for messages and prompts when a user has not explicitly chosen a language.</li></ol>
<b>dynamic SQL</b>	Allows an Embedded SQL or Client-Library application to execute SQL statements containing variables whose values are determined at runtime.
<b>error message</b>	A message that an Open Client and Open Server product issues when it detects an error condition.
<b>event</b>	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.
<b>event handler</b>	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.

<b>exposed structure</b>	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.
<b>extended transaction</b>	In Embedded SQL, a transaction composed of multiple Embedded SQL statements.
<b>FIPS</b>	An acronym for Federal Information Processing Standards. If FIPS flagging is enabled, Adaptive Server Enterprise or the Embedded SQL precompiler issue warnings when a non-standard extension to a SQL statement is encountered.
<b>gateway</b>	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.
<b>hidden structure</b>	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.
<b>host language</b>	The programming language in which an application is written.
<b>host program</b>	In Embedded SQL, the application program that contains the Embedded SQL code.
<b>host variable</b>	In Embedded SQL, a variable which enables data transfer between Adaptive Server Enterprise and the application program. See also <b>indicator variable</b> , <b>input variable</b> , <b>output variable</b> , <b>result variable</b> , and <b>status variable</b> .
<b>indicator variable</b>	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.
<b>input variable</b>	A variable that is used to pass information to a routine, a stored procedure, or Adaptive Server Enterprise.
<b>interfaces file</b>	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.

---

<b>isql script file</b>	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.
<b>key</b>	A subset of row data that uniquely identifies a row. Key data uniquely describes the <b>current row</b> in an open cursor.
<b>keyword</b>	A word or phrase that is reserved for exclusive use in Transact-SQL or Embedded SQL. Also called a reserved word.
<b>listing file</b>	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.
<b>locales file</b>	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.
<b>locale name</b>	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.
<b>locale structure</b>	(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 <code>cs_loc_alloc</code> and <code>cs_loc_drop</code> allocate and drop a locale structure.
<b>localization</b>	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.
<b>login name</b>	The name a user uses to log in to a server. An Adaptive Server Enterprise login name is valid if Adaptive Server Enterprise has an entry for that user in the system table <code>syslogins</code> .
<b>message number</b>	A number that uniquely identifies an error message.
<b>message queue</b>	In Open Server, a linked list of message pointers through which threads communicate. Threads can write messages into and read messages from the queue.
<b>multibyte character set</b>	A character set that includes characters encoded using more than one byte. EUC JIS and Shift-JIS are examples of multibyte character sets.

<b>mutex</b>	A mutual exclusion semaphore. This is a logical object that an Open Server application uses to ensure exclusive access to a shared object.
<b>null</b>	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.
<b>Open Server</b>	A Sybase product that provides tools and interfaces for creating custom servers.
<b>Open Server application</b>	A custom server constructed with Open Server.
<b>output variable</b>	In Embedded SQL, a variable that passes data from a stored procedure to an application program.
<b>parameter</b>	<ol style="list-style-type: none"><li>1. A variable that is used to pass data to and retrieve data from a routine.</li><li>2. An argument to a stored procedure.</li></ol>
<b>passthrough mode</b>	<p>A state of being pertaining to gateway applications.</p> <p>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.</p>
<b>property</b>	A named value stored in a structure. Context, connection, thread, and command structures have properties. A structure's properties determine how it behaves.
<b>query</b>	<ol style="list-style-type: none"><li>1. A data retrieval request; usually a select statement.</li><li>2. Any SQL statement that manipulates data.</li></ol>
<b>registered procedure</b>	In Open Server, a collection of C statements stored under a name. Open Server-supplied registered procedures are called <b>system registered procedures</b> .
<b>remote procedure call</b>	<ol style="list-style-type: none"><li>1. One of two ways in which a client application can execute an Adaptive Server Enterprise stored procedure. (The other is with a Transact-SQL execute statement.) A Client-Library application initiates a remote procedure call command by calling <code>ct_command</code>. A DB-Library application initiates a remote procedure call command by calling <code>dbrcpinit</code>.</li><li>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.</li><li>3. A <b>stored procedure</b> executed on a different server from the server to which the user is connected.</li></ol>



---

<b>result variable</b>	In Embedded SQL, a variable that receives the results of a select or fetch statement.
<b>scrollable cursor</b>	Allows a current cursor position to be set anywhere in a result set. See also <b>cursor</b> .
<b>server</b>	In client/server systems, the part of the system that processes client requests and returns results to clients.
<b>Server-Library</b>	A collection of routines for use in writing Open Server applications.
<b>sort order</b>	Used to determine the order in which character data is sorted. Also called collating sequence.
<b>sqlca</b>	<ol style="list-style-type: none"><li>1. In an Embedded SQL application, a SQLCA is a structure that provides a communication path between Adaptive Server Enterprise and the application program. After executing each SQL statement, Adaptive Server Enterprise stores return codes in the SQLCA.</li><li>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.</li></ol>
<b>sqlcode</b>	<ol style="list-style-type: none"><li>1. In an Embedded SQL application, a SQLCODE is a structure that provides a communication path between Adaptive Server Enterprise and the application program. After executing each SQL statement, Adaptive Server Enterprise stores return codes in the SQLCODE. A SQLCODE can exist independently or as a variable within a SQLCA structure.</li><li>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.</li></ol>
<b>statement</b>	In Transact-SQL or Embedded SQL, an instruction that begins with a keyword. The keyword names the basic operation or command to be performed.
<b>status variable</b>	In Embedded SQL, a variable that receives the return status value of a stored procedure, thereby indicating the procedure's success or failure.
<b>stored procedure</b>	In Adaptive Server Enterprise, a collection of SQL statements and optional control-of-flow statements stored under a name. Adaptive Server Enterprise-supplied stored procedures are called <b>system procedures</b> .
<b>System Administrator</b>	The user in charge of server system administration, including creating user accounts, assigning permissions, and creating new databases. On Adaptive Server Enterprise, the System Administrator's login name is "sa."

<b>system descriptor</b>	In Embedded SQL, an area of memory that holds a description of variables used in Dynamic SQL statements.
<b>system procedures</b>	Stored procedures that Adaptive Server Enterprise 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.
<b>system registered procedures</b>	Internal registered procedures that Open Server supplies for registered procedure notification and status monitoring.
<b>target file</b>	In Embedded SQL, one of the three files the precompiler can generate. A target file is similar to the original input file, except that all SQL statements are converted to Client-Library function calls.
<b>TDS</b>	(Tabular Data Stream) An application-level protocol that Sybase clients and servers use to communicate. It describes commands and results.
<b>thread</b>	A path of execution through Open Server application and library code and the path's associated stack space, state information, and event handlers.
<b>Transact-SQL</b>	An enhanced version of the database language SQL. Applications can use Transact-SQL to communicate with Adaptive Server Enterprise.
<b>transaction</b>	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.
<b>transaction mode</b>	The manner in which Adaptive Server Enterprise manages transactions. Adaptive Server Enterprise supports two transaction modes: Transact-SQL mode (also called "unchained transactions") and ANSI mode (also called "chained transactions").
<b>user name</b>	See <b>login name</b> .

# Index

## A

- ad hoc negotiations 123
- aggregates
  - compute rows 227
- allocating
  - memory 221
- allocating memory 143
- ANSI compliance, updates and deletes 126
- application name 306
- application-defined login handshake 123, 154
- arithmetic exceptions 126
- ASCII character format 159
- asynchronous events 285
- attentions 153
  - checking for with `srv_thread_props` 21
  - coding recommendations for 21
  - and interrupt level 20
  - and the `SRV_ATTENTION` event handler 20
- authentication of client 123

## B

- binary datatypes 26, 29, 207
- binding
  - variables 236
- binding data 139
- bit datatype 26, 29, 208
- bit masks
  - `CS_BROWSEDESC` structure 53
  - `CS_DATAFMT` status value 57
  - `CS_KEY` 76
- bitmasks
  - capabilities 36
- boundary datatype 26, 29, 216
- browse mode 52
  - and the `CS_BROWSEDESC` structure 23
  - returning browse mode results to a client 23
  - steps to support 23

- building an Open Server application 6, 16
- bulk
  - copy requests 95
  - data transfer 152, 159
  - insert 152
- byte ordering 122
  - retrieving scheme through `srv_thread_props` 152
- bytes
  - copying 241

## C

- call stack, threads 263
- callback handlers
  - errors 60
  - installing for a thread 119
  - registered procedures 169
- callbacks
  - installing 244
  - security session 195, 198, 201
  - timeslice 149
- capabilities 122, 248, 255
  - ad hoc retrieval of 37
  - bit masks 36
  - and the capability macros 36
  - changing default values through `srv_props` 30
  - client connection 249
  - and the `CS_CAP_TYPE` structure 36
  - default 30
  - explicit negotiation of 35
  - list of default values for 31, 35
  - macros 36
  - negotiating one at a time 35
  - negotiating with pre-10.0 clients 37
  - negotiation 24
  - Request Capabilities table 24, 27
  - Response Capabilities table 29, 30
  - TDS version 32, 37
  - transparent negotiation of 30

## Index

- uses of 24
- certificates
  - SSL 182
- chained transactions 126
- challenge/response 154
- channel binding 156, 175
- character data representation 152, 159
- character datatypes 26, 29, 209, 210
- character set 101, 122, 142
  - changing 107
  - notification of change 155
  - processing client request to change 107
  - renegotiating 124
  - returning information about 108
- chunks 62
  - messages 38
- client
  - definition of 2
  - login information 292
  - types of clients 2
- client command errors
  - and the CS\_SERVERMSG structure 38
  - sending through srv\_sendinfo 38, 39
- client login request 161
- client logout 152
- client requests 122, 147
- client threads 113, 164
- client/server
  - architecture 1, 2
- Client-Library
  - context properties 142
  - retrieving client version through srv\_thread\_props 152
- clock rate 150
- close, cursor command 64
- collating sequence 101
- columns
  - original names 53
- Common Libraries 59
- common name validation
  - SDC environment 182
- compute rows 223, 231
  - and aggregates 227
  - sending to client 234
- concurrency 115, 119
- connect handler. See SRV\_CONNECT event handler 216
- connection attributes. See Capabilities 24
- Connection migration 40
- context properties
  - and cs\_config 142
  - and ct\_config 142
  - definition of 142
  - and srv\_props 142
- context structure. See CS\_CONTEXT structure 216
- context switching 115
- coroutine scheduling. See Non-preemptive scheduling 115
- credentials 174
  - delegated 157
  - timeout 156
- cryptographic signature 157
- CS\_ABSOLUTE fetch type 68
- CS\_ACK dynamic operation 278
- CS\_ALL\_CAPS argument 36
- CS\_BIGDATETIME datatype 205, 210, 212
- CS\_BIGINT datatype 213
- CS\_BIGTIME datatype 205, 210, 212
- CS\_BINARY datatype 205, 207
- CS\_BIT datatype 205, 209
- CS\_BOUNDARY\_TYPE value 216
- CS\_BROWSEDESC structure 40, 53
- cs\_calc routine 207
- CS\_CANBENULL value 57, 241
- CS\_CANCEL\_ATTN argument 21
- CS\_CAP\_REQUEST argument 35
- CS\_CAP\_RESPONSE capabilities 250
  - CS\_RES\_NOXNLMETADATA 30
- CS\_CAP\_TYPE structure 36
- CS\_CHAR datatype 205, 209
- CS\_CLR\_CAPMASK macro 36
- cs\_cmp routine 207
- cs\_config command 60, 104, 106, 142
- CS\_CONNECTION structure 133
- CS\_CONTEXT structure 7, 60, 104, 105, 142
- cs\_convert command 104
  - CS\_DATAFMT structure 54
- cs\_convert routine 207
- cs\_ctx\_alloc command 106
- CS\_CUR\_RELOCKS\_ONCLOSE 78
- CS\_CURSOR\_CLOSE command 70, 75
- CS\_CURSOR\_DECLARE command 67, 70, 73
- CS\_CURSOR\_DELETE command 68, 70, 75
- CS\_CURSOR\_FETCH command 67, 70, 74

- CS\_CURSOR\_INFO command 67, 71, 73
- CS\_CURSOR\_OPEN value 71, 74
- CS\_CURSOR\_UPDATE command 68, 71, 75
- CS\_CURSTAT\_CLOSED value 69
- CS\_CURSTAT\_DEALLOC value 69
- CS\_CURSTAT\_DECLARED value 69
- CS\_CURSTAT\_OPEN value 69, 78
- CS\_CURSTAT\_RDONLY value 69
- CS\_CURSTAT\_ROWCNT value 69, 78
- CS\_CURSTAT\_UPDATABLE value 69
- CS\_DATA\_LBIN capability 208
- CS\_DATA\_LCHAR capability 209
- CS\_DATAampfmt structure 275
- CS\_DATAFMT structure 53, 57, 139
- CS\_DATE datatype 205, 210
- CS\_DATETIME datatype 205, 210
- CS\_DATETIME4 datatype 205, 210, 211
- CS\_DEALLOC dynamic operation 278
- CS\_DEALLOC value 88
- CS\_DECIMAL datatype 206, 214
- CS\_DEF\_PREC value 56, 214
- CS\_DEF\_SCALE value 56, 214
- CS\_DESCIN value 57, 85
- CS\_DESCOUT value 57, 86
- CS\_DESCRIBE\_INPUT dynamic operation 279
- CS\_DESCRIBE\_INPUT value 85
- CS\_DESCRIBE\_OUTPUT dynamic operation 279
- CS\_DESCRIBE\_OUTPUT value 86
- cs\_dt\_crack routine 207, 211
- cs\_dt\_info routine 207
- CS\_EXEC\_IMMEDIATE dynamic operation 278
- CS\_EXEC\_IMMEDIATE value 87
- CS\_EXECUTE dynamic operation 278
- CS\_EXECUTE value 87
- CS\_EXPRESSION argument 53
- CS\_FIRST fetch type 68
- CS\_FIRST\_CHUNK argument 39, 62
- CS\_FLOAT datatype 206, 213
- CS\_FMT\_NULLTERM argument 56
- CS\_FMT\_PADBLANK argument 56
- CS\_FMT\_PADNULL argument 56
- CS\_FMT\_UNUSED argument 56
- CS\_FOR\_UPDATE value 76
- CS\_GOODDATA value 224, 238
- CS\_HASEED bit 40, 62
- CS\_HIDDEN value 57
- CS\_IMAGE datatype 207, 216
- CS\_IMAGE\_TYPE value 58
- CS\_IMAGELOCATOR datatype 206
- CS\_INPUTVALUE value 57
- CS\_INT datatype 206, 213
- CS\_IODATA value 58
- CS\_IODESC structure 57, 59, 202
- CS\_KEY value 57, 76
- CS\_LANG\_CMD value 109
- CS\_LAST fetch type 68
- CS\_LAST\_CHUNK argument 39, 62
- CS\_LC\_ALL value 104
- cs\_loc\_alloc command 104, 106
- cs\_loc\_drop command 104, 106
- CS\_LOC\_PROP value 104, 106
- cs\_locale command 103, 104, 106
- CS\_LOCALE structure 57, 153
- CS\_LOCATOR datatype 215
- CS\_LOGININFO structure 133, 294
- CS\_LONGBINARY datatype 205, 208
- CS\_LONGCHAR datatype 205, 209
- CS\_MAX\_MSG argument 38
- CS\_MAX\_PREC value 56, 214
- CS\_MAX\_SCALE value 56, 214
- CS\_MIN\_PREC value 56, 214
- CS\_MIN\_SCALE value 56, 214
- CS\_MONEY datatype 206, 215
- CS\_MONEY4 datatype 206, 215
- CS\_NEXT fetch type 68
- CS\_NOAPICLK value 144
- CS\_NODEFAULT value 57
- CS\_NULLDATA value 238
- CS\_NUMERIC datatype 206, 213
- CS\_OP\_AVG operator type 228
- CS\_OP\_COUNT operator type 228
- CS\_OP\_MAX operator type 228
- CS\_OP\_MIN operator type 228
- CS\_OP\_SUM operator type 228
- CS\_OPT\_ANSINULL server option 126
- CS\_OPT\_ANSIPERM server option 126
- CS\_OPT\_ARITHABORT server option 126
- CS\_OPT\_ARITHIGNORE server option 126
- CS\_OPT\_AUTHOFF server option 126
- CS\_OPT\_AUTHON server option 126
- CS\_OPT\_CHAINXACTS server option 126
- CS\_OPT\_CURCLOSEONXACT server option 126

## Index

- CS\_OPT\_CURREAD server option 126
- CS\_OPT\_CURWRITE server option 127
- CS\_OPT\_DATEFIRST server option 127
- CS\_OPT\_DATEFORMAT server option 127
- CS\_OPT\_FIPSFLAG server option 127
- CS\_OPT\_FORCEPLAN server option 127
- CS\_OPT\_FORMATONLY server option 127
- CS\_OPT\_GETDATA server option 127
- CS\_OPT\_IDENTITYOFF server option 127
- CS\_OPT\_IDENTITYON server option 127
- CS\_OPT\_ISOLATION server option 127
- CS\_OPT\_LEVEL1 value 127
- CS\_OPT\_NOCOUNT server option 125, 127
- CS\_OPT\_NOEXEC server option 128
- CS\_OPT\_PARSEONLY server option 128
- CS\_OPT\_QUOTED\_IDENT server option 128
- CS\_OPT\_RESTREES server option 128
- CS\_OPT\_ROWCOUNT server option 128
- CS\_OPT\_SHOWPLAN server option 128
- CS\_OPT\_STATS\_IO server option 128
- CS\_OPT\_STATS\_TIME server option 128
- CS\_OPT\_STR\_RTRUNC server option 128
- CS\_OPT\_TEXTSIZE server option 129
- CS\_OPT\_TRUNCIGNORE server option 129
- CS\_PASSTHRU\_MORE value 134
- CS\_PREPARE dynamic operation 279
- CS\_PREPARE value 85
- CS\_PREV fetch type 68
- CS\_REAL datatype 206, 213
- CS\_RELATIVE fetch type 68
- CS\_RENAMED argument 53
- CS\_REQ\_MIGRATE 42
- CS\_REQUEST capabilities 32
- CS\_RES\_NOTDSDEBUG capability 30
- CS\_RESPONSE capabilities 34
- CS\_RESPONSE\_CAP argument 35
- CS\_RETURN value 57
- CS\_SECSSESSION\_CB value 195
- CS\_SENSITIVITY\_TYPE value 216
- CS\_SERVERMSG structure 38, 60, 62
  - CS\_HASEED bit 40
- CS\_SET\_CAPMASK macro 36
- CS\_SMALLINT datatype 206, 213
- CS\_SRC\_VALUE argument 56
- CS\_SYB\_CHARSET value 106
- CS\_TEXT datatype 207, 216
  - CS\_TEXT\_TYPE value 58
- CS\_TEXTLOCATOR datatype 206
- CS\_TIME datatype 205, 210
- CS\_TIMESTAMP value 57
- CS\_TINYINT datatype 206, 213
- CS\_TST\_CAPMASK macro 36
- CS\_UBIGINT datatype 213
- CS\_UINT datatype 213
- CS\_UNICHAR datatype 205, 210
- CS\_UNITEXT datatype 216
- CS\_UNITEXTLOCATOR datatype 206
- CS\_UPDATABLE value 57
- CS\_UPDATECOL value 57
- CS\_USER\_MAX\_MSGID value 80
- CS\_USER\_MSGID value 80
- CS\_USMALLINT datatype 213
- CS\_VARBINARY datatype 205, 208
- CS\_VARCHAR datatype 205, 207, 209
- CS\_VERSION\_KEY value 57
- CS\_XML datatype 205
- CS-Library 59, 60
  - context properties 142
  - definition of 6, 59
  - error messages 103, 104
  - errors 60, 142
- ct\_cancel command 95
- ct\_capability command 36
- ct\_close command 96
- ct\_command command 21, 80, 96, 109
- ct\_connect command 36, 95
- ct\_cursor command 64
- ct\_exit command 96
- ct\_getlogininfo command 133
- ct\_recvpassthru command 134
- ct\_send command 96
- ct\_sendpassthru command 134
- ct\_setlogininfo 133
- curcmd field, SRV\_CURDESC structure 69, 78
- curid field, SRV\_CURDESC structure 72
- cursor commands 137
- cursor handler. See SRV\_CURSOR event handler 216
- cursors 27, 62, 76
  - benefits of using 63
  - CS\_DATAFMT structure 57
  - definition of 63
  - fetch types 68

- fetching rows 26
- handling cursor requests 72, 76
- ID 65
- and key data 76
- server option 126
- and the SRV\_CURDESC structure 65, 76
- and the SRV\_CURSOR event handler 72
- srv\_cursor\_props 259, 262
- types of cursor commands 64
- update columns 76
- update text 71
- updates 67, 76
- curstatus field
  - SRV\_CURDESC structure 69

## D

- data
  - confidentiality 156
  - describing, binding, transferring of 139
  - integrity 157
  - origination 156
- datastream messages. See Messages 80
- datatype Summary table 204, 207
- datatypes 207
  - See also Types 204
  - CS\_BIGDATETIME 210
  - CS\_BIGTIME 210
  - CS\_DATE 210
  - CS\_DATETIME 210
  - CS\_DATETIME4 210
  - CS\_TIME 210
  - response capabilities 30
  - routines that manipulate 207
- dates
  - order of parts 127
- datetime datatypes 26, 29, 210
  - conversion to 8-byte 152
- datetime formats 101
- datetime types
  - CS\_DATE 210
  - CS\_TIME 210
- dbcancel command 20
- deallocate, cursor command 64
- debugging 247, 265
- decimal datatype 26, 29, 56
- declare, cursor command 64
- default event handlers 94
- deferred event
  - queue size 144
- delegated credentials 157
- delete, cursor command 64
- deletes 127
- describing
  - columns and parameters 272
- describing data 139
- detection of message replay 157
- directory drivers 82
- directory service provider 144
- directory services 81, 83
- disconnect handler. See SRV\_DISCONNECT event handler 216
- disconnects
  - handling of 21
- distributed service providers 174
- double quotes, identifiers 128
- DSLISEN environment variable 306
- dump/load 152
- dynamic listeners 88
  - configuration 88
  - properties 89
  - starting 89
  - thread properties 89
  - thread type 89
- dynamic SQL 27, 83, 88
  - benefits of using 84
  - commands 137
  - CS\_DATAFMT structure 57
  - cursors 63
  - responding to client Dynamic SQL commands 84
  - srv\_dynamic 275
    - and the SRV\_DYNAMIC event handler 84
    - and the srv\_dynamic routine 84
    - uses for 84
- dynamic SQL handler. See SRV\_DYNAMIC event handler 216

## E

- EBCDIC character format 159

## Index

- encryption 156
    - key 123
    - passwords 154
  - environment changes 280
  - environment variables 281
  - error handler 60, 90, 144
  - error handlers
    - environment variable changes 282
    - installation of 8, 16
  - error messages 38
    - sending to a client 38
    - troubleshooting 481
  - errors 38, 60, 88, 92
    - See also client command errors 38
    - column-level information 39
    - CS-Library 60
    - extended data 39
    - local language messages 102, 104
    - numbers 92
    - severity of 91
    - types of 90
  - event handlers
    - coding custom handlers 94
    - default 94
    - default versus custom 94
    - definition of 93
    - interrupt level 20
    - messages 81
    - srv\_capability 37
    - srv\_handle 303
  - event queue 113
  - event-driven threads 112
  - events 92, 99
    - attention 20
    - cursor 64, 72
    - definition of 93
    - disconnects 21
    - dynamic SQL 84
    - handling 8
    - list of standard 94, 99
    - message 80
    - notifications 27
    - programmer-defined 99
    - srv\_event 282
  - execute statement 63
  - explicit negotiation 24, 122
  - extended error data 39, 40
    - definition of 39
    - sending to a client 39
- ## F
- fatal errors 92
  - fetch types 68
  - fetching rows 26, 64
  - file descriptor
    - endpoint 152
  - first day of week 127
  - floating point datatype 26, 29
    - conversion to 8-byte 152
    - representation 153
  - floating point representation 122, 161
  - free, C routine 288
  - freeing memory 145, 287
- ## G
- gateway applications 100, 101, 103, 105, 125, 131
    - attentions 21
    - direct security sessions 189, 195
    - separate security sessions 189, 190
    - srv\_getloginfo 294
- ## H
- hidden columns
    - CS\_DATAFMT structure 57
  - host machine, of client 153
- ## I
- I/O channel
    - threads 153
  - I/O descriptor structure 57
  - identifiers 128
  - identity columns 127
  - image data 57
    - transferring 159



image datatype 26, 201  
     srv\_get\_data 289  
     srv\_get\_text 290  
 in-band attentions 26  
 information, cursor command 64  
 informational errors 91  
 initialization  
     setting properties during 143  
     summary of steps in 143  
 inserts 127  
 installing  
     error handlers 90  
     event handlers 303  
 integer types 26, 29, 213  
 integrity service 157  
 interfaces file 188  
     directory services 83  
     looking up server name in 148  
     specifying name of through srv\_props 145  
 intermediary applications 100  
 internal I/O statistics 128  
 international support. See localization 101  
 interrupt level  
     and attentions 20  
     Server-Library calls permitted at 20  
 interrupts 20, 95, 121, 144  
 is NULL 126  
 isbrowse structure element 53

## J

joins 127

## K

keys 76

## L

language  
     calls 109  
     commands 137  
     datastream 141

    requests 96  
 language and character set 101  
     changing 106  
 language handler. See SRV\_LANGUAGE event handler  
     216  
 Large Object (LOB) locator datatype 215  
 libtcl.cfg file 82  
 listening address 82  
 local language 122  
 localization 101, 109, 142  
     creating localized connections 105  
     and the CS\_LOCALE structure 102, 103  
     of a CS\_CONTEXT structure 106  
     of an Open Server application 102, 104  
     properties related to 108  
     returning localization information to clients 108  
     and sp\_serverinfo 108  
     supporting localized clients 102, 106  
 localized clients 101, 103  
 locking 117  
 log file 91, 105, 118  
     configuring size of through srv\_props 145  
     maximum size 145  
     name 145  
     specifying through srv\_props 145  
     truncation at startup 149  
 login negotiations 121  
 login requests 161  
 logout, by client 152

## M

macros  
     capabilities 36  
 malloc C routine 223  
 maximum rows 128  
 memory  
     allocating 143, 147, 221  
     freeing routines, specifying through srv\_props 145  
     moving bytes 241  
     reallocation routines, specifying through srv\_props  
         147  
     setting to zero 243  
     srv\_free 287  
 message event 80

- message handler. See SRV\_MSG event handler 216
- message queues
  - activity 151
  - configuring number of through `srv_props` 147
  - creating 253
  - definition of 118
  - deleting 268
  - object IDs 297
  - `srv_getmsgq` 294
  - `srv_getobjname` 300
- message replay 157
- messages 27, 137
  - chunking 38, 62
  - data parameters 240
  - definition of 80
  - error 38, 481
  - and event handlers 81
  - ID 80
  - number available 146
  - numbers 61
  - receiving 80
  - retrieving from client 80
  - severity 61
  - text length 38
  - types of in Open Server 110
- money datatype 27, 29, 215
  - conversion to 8-byte 152
- multithread programming 111, 121
  - and callback handlers 119, 120
  - definition of thread 111
  - and message queues 118
  - and mutexes 117
  - overview of 16
  - special programming considerations 120, 121
  - and `srv_setpri` 118
  - thread scheduling 115, 117
  - tools and techniques for 117, 120
  - types of threads 112, 115
- mutexes
  - configuring number of through `srv_props` 147
  - creating 255
  - definition of 117
  - deleting 270
  - object IDs 297
  - `srv_getobjname` 300
- mutual authentication 157

## N

- naming services 81, 83
- national language 122, 142, 149, 158
  - notification of change 155
  - renegotiating 124
- negotiated behavior 121, 124
- negotiated login
  - commands 137
  - retrieving client request for through `srv_thread_props` 154
- negotiated packet size 155
- negotiating
  - capabilities 24
  - in the SRV\_CONNECT event handler 122
  - TDS protocol level 132, 136
  - transparently 30
  - via options commands or language commands 123
- Net-Library
  - providing network services 6
- net-Library tracing file
  - specifying through `srv_props` 146
- network authentication 157
- network connections
  - configuring number of through `srv_props` 146
- network I/O buffer
  - configuring size of through `srv_props` 146
- non-client events 93
- non-client threads 257
- non-preemptive scheduling
  - definition of 115
  - specifying with `srv_props` 147
- non-standard SQL 127
- notification
  - registered procedures 165
- notification procedures 166
- nullable bit datatype 26
- nulls 126

## O

- Open Server
  - header files 6
  - position in client/server architecture 3
- Open Server application
  - a simple program 8, 10

- auxiliary 4
- contrasted with Adaptive Server Enterprise 3
- definition of 2
- gateway 5
- initializing 8
- stand-alone 4
- open, cursor command 64
- operating system errors 92
- options 124, 129
  - default values for 125, 129
  - description of 125, 129
  - setting and retrieving 125
- oserror.h header file 91
- ospublic.h header file 118
- out-of-band attentions 26

## P

- packet size 155
- padding 56
- parameter data 136
- parameters
  - retrieving from a client 137
  - return parameters 16
  - RPC 172
- parse resolution trees 128
- pass-through mode 101, 129, 134
  - gateway 129
  - gateway with direct security session 189, 195
  - negotiating the TDS level in 133, 136
  - routines used in 133
- passthrough mode 155
- password
  - retrieving clientxd5 s via srv\_thread\_props 155
- platform capabilities 249, 253
- platform-dependent services 248
- precision
  - decimal datatype 56, 214
- preemptive thread scheduling 115, 116
  - definition of 115
  - specifying through srv\_props 147
- preemptive threads 112
  - scheduling 248
- prepared statement 83
- preparing

- statements 278
- principals 184
- priority levels 116, 155
- process ID
  - client 153
- processing parameter and row data 136
- programmer-defined events 93, 99
- protocol capabilities 249, 253
- providers, directory services 82

## Q

- query
  - information 127
  - processing behavior 124
  - syntax 128

## R

- real-time data 117
- receiving messages 80
- registered procedures
  - benefits of 165
  - contrasted with remote procedure calls 165
  - definition of 14, 164
  - executing 167
  - maintaining lists of 167
  - steps to register 166
  - using callback handlers with 169
- registering
  - with a directory 82, 144
- remote passwords 155
  - retrieving through srv\_thread\_props 155
- remote passwords, retrieving through srv\_thread\_props 155
- remote procedure calls 27, 98, 137, 171, 172
  - CS\_DATAFMT structure 57
  - definition of 171
  - processing of 171
- remote servers 100
  - passwords 155, 163
  - retrieving name of through srv\_thread\_props 156
  - security sessions 175
- renegotiating client/server behavior 123

## Index

- request Capabilities table 25, 27
  - requests
    - dynamic SQL 84
  - response Capabilities table 29, 30
  - responses 147
  - results
    - order returned in 16
    - overview of 14
    - processing 15
  - retrieving parameter data 136
  - return parameters 136
    - processing 138
    - processing in a SRV\_LANGUAGE event handler 140
  - returning parameters 136, 138
    - language datastream 141
  - returning rows 138
  - row data 136
  - rows
    - affected 127
    - maximum 128
    - processing 23, 138
  - RPC. See Remote procedure calls 171
  - run queues 116
- S**
- scale
    - decimal datatype 56, 214
  - scheduling threads 115, 117
  - secure connections 123
    - negotiating with client to establish 123
  - security datatypes 215
  - security labels 123, 126, 154
  - security levels 122
    - negotiation of 122
  - security mechanisms 174
    - interfaces file 188
    - local name 157
    - local names 184
  - security services 174, 201
    - thread properties 156
  - security session callback 195, 198, 201
  - security sessions
    - gateway applications 189, 190
    - simple Open Server application 189, 190
    - timeout 157
  - select query option 127
  - select statements 128
  - sending
    - messages to client 81
    - row data 136
  - sensitivity datatype 27, 29, 216
  - server error messages 60
    - troubleshooting 481
  - server name
    - specifying through `srv_props` 148
  - server properties
    - definition of 143
  - Server-Library
    - context properties 142
  - server-Library
    - version 150
  - servers
    - types of servers 2
  - service threads 99, 112, 114, 164
  - set command 124
  - severity of errors 91
  - shared disk cluster environment
    - certificate 182
  - signals (UNIX) 286
  - significant byte 152
  - SIGTRAP signal 266
  - site handler 112, 164
    - configuring number of through `srv_props` 147
    - login request 161
    - subchannel login 161
  - sleeping threads 116
  - sort order 101, 108, 142
    - returning information about 108
  - `sp_ps` 168, 461, 464
  - `sp_regcreate` 464
  - `sp_regdrop` 471
  - `sp_reglst` 472
  - `sp_regnowatch` 472, 473
  - `sp_regwatch` 473, 475
  - `sp_regwatchlist` 475
  - `sp_serverinfo` 108, 475, 476
    - responding to `sp_serverinfo` requests 108
  - `sp_terminate` 168, 476, 477
  - `sp_who` 168, 477, 479
  - SQL queries 109

- srv\_alloc 221, 223
- srv\_alt\_bind 223, 227, 231, 235
- srv\_alt\_descampfmt 227, 231, 235
- srv\_alt\_header 227, 231, 232, 235
- srv\_alt\_xferdata 227, 231, 234, 236
- SRV\_APPDEFINED value 154
- SRV\_ATTENTION event 20, 95, 121, 144
- SRV\_ATTENTION event handler 20, 22, 144
  - calling to handle client disconnect 22
- SRV\_BIG\_ENDIAN value 152
- srv\_bind 136, 140, 236, 238
  - CS\_DATAFMT structure 54
- srv\_bmove 241, 243
- SRV\_BULK event 93, 95, 159, 202
- SRV\_BULKLOAD value 159
- srv\_bzero 243, 244
- SRV\_C\_DEBUG capability 248
- SRV\_C\_DEFAULTPRI constant 118
- SRV\_C\_EXIT callback type 245
- SRV\_C\_EXIT capability 248
- SRV\_C\_EXIT state transition 120
- SRV\_C\_LOWPRIORITY constant 118
- SRV\_C\_MAXPRIORITY constant 118
- SRV\_C\_MQUEUE value 300
- SRV\_C\_Mutex value 300
- SRV\_C\_PREEMPT capability 248
- SRV\_C\_PROCEXEC callback type 245
- SRV\_C\_PROCEXEC state transition 120
- SRV\_C\_RESUME callback type 245
- SRV\_C\_RESUME capability 248
- SRV\_C\_RESUME state transition 120
- SRV\_C\_SELECT capability 248
- SRV\_C\_SUSPEND callback handler 119
- SRV\_C\_SUSPEND callback type 245
- SRV\_C\_SUSPEND capability 248
- SRV\_C\_SUSPEND state transition 120
- SRV\_C\_TIMESLICE callback type 245
- SRV\_C\_TIMESLICE capability 248
- SRV\_C\_TIMESLICE state transition 120
- srv\_callback 244, 247
  - in multithread programming 119, 120
- srv\_capability 116, 247, 248
- srv\_capability\_info 24, 35, 36, 249, 250
  - event handlers 37
- SRV\_CHALLENGE value 154
- SRV\_CHAR\_ASCII value 159
- SRV\_CHAR\_EBCDIC value 159
- SRV\_CHAR\_UNKNOWN value 159
- SRV\_CLEAROPTION value 125
- SRV\_CLIENT login type 161
- SRV\_CONNECT event 93, 95, 114
- SRV\_CONNECT event handler 31, 35, 37, 122, 123, 137, 158, 190
  - passthrough mode 133
  - security sessions 190, 195
  - srv\_getloginfo 294
- SRV\_CONTINUE return value 247
- srv\_createmsgq 118, 253, 255
- srv\_createmutex 257
- srv\_createproc 257, 259
- SRV\_CTL\_MIGRATE 42
- SRV\_CUR\_ASKSTATUS value 71
- SRV\_CUR\_DEALLOC value 70, 79
- SRV\_CUR\_DYNAMIC value 70
- SRV\_CUR\_HASARGS value 71
- SRV\_CUR\_INFORMSTATUS value 71
- SRV\_CUR\_RDONLY value 70
- SRV\_CUR\_SETROWS value 71
- SRV\_CUR\_UNUSED value 70, 71, 79
- SRV\_CUR\_UPDATABLE value 70, 79
- SRV\_CURDATA type of data 237
- SRV\_CURDATA value 137
- SRV\_CURDESC structure 65, 69, 261
  - curcmd field 69, 78
  - curid field 72
  - curstatus field 69
- SRV\_CURSOR event 72, 95
- SRV\_CURSOR event handler 64, 72, 137, 261
- srv\_cursor\_props 65, 73, 259, 262, 263
- srv\_dbg\_stack 262, 263, 265
- srv\_dbg\_switch 265, 266
- SRV\_DEBUG return value 247
- srv\_define\_event 99, 266, 268
- srv\_deletemsgq 118, 268, 270
- srv\_deletemutex 270, 272
- srv\_descampfmt 272, 275
- srv\_descfmt 30, 136, 139
  - CS\_DATAFMT structure 54
  - SRV\_CURDATA argument 73
  - SRV\_UPCOLDATA argument 73
- SRV\_DISCONNECT event 93, 96, 144, 284, 286
  - fatal errors 92

## Index

- SRV\_DISCONNECT event handler 21, 99, 152
- SRV\_DS\_PROVIDER property 82, 186
- SRV\_DYN\_values 276
- srv\_dynamic 84, 275, 279
- SRV\_DYNAMIC event 96, 278
- SRV\_DYNAMIC event handler 84, 137
- SRV\_DYNAMICDATA type of data 237
- SRV\_DYNDATA value 85, 86
- SRV\_ENCRYPT value 154
- SRV\_ENO\_OS\_ERR value 92
- srv\_envchange 279, 280
- SRV\_QUEUED event type 266
- SRV\_ERRORDATA argument 40
- SRV\_ERRORDATA type of data 237
- srv\_event 93, 96, 98, 99, 268, 280, 284
- srv\_event\_deferred 21, 99, 285, 287
- SRV\_FATAL\_PROCESS error severity 92
- SRV\_FATAL\_SERVER error severity 92
- SRV\_FLT\_ floating point formats 161
- srv\_free 223, 287, 288
- srv\_freeserveraddrs 288
- srv\_get\_text 202, 288, 292
- srv\_getloginfo 31, 133, 292, 294
- srv\_getmsgq 116, 118, 294, 297
- srv\_getobjid 297, 299
- srv\_getobjname 255, 257, 299, 302
- SRV\_GETOPTION value 125
- srv\_getserverbyname 302
- srv\_handle 94, 302, 306
- SRV\_HASPARAMS value 80, 81
- SRV\_I\_DELETED value 296
- SRV\_I\_INTERRUPTED value 296
- SRV\_I\_NOEXIST value 298
- SRV\_I\_PASSTHRU\_MORE value 134
- SRV\_I\_UNKNOWN value 296, 298
- SRV\_I\_WOULDWAIT value 296
- SRV\_IMAGELOAD value 159
- SRV\_INFO error severity 91
- srv\_init 306, 308
  - and directory services 82
- SRV\_KEYDATA type of data 237
- srv\_langcpy 110, 308, 310
- SRV\_LANGDATA type 237
- srv\_langlen 110, 310, 312
- SRV\_LANGUAGE event 96, 109
- SRV\_LANGUAGE event handler 99, 109, 137, 138
  - option requests 124
  - renegotiating behavior 123
- SRV\_LISTENER\_POSTBIND event 89, 97, 430
- SRV\_LISTENER\_PREBIND event 88, 97, 430
- SRV\_LITTLE\_ENDIAN value 152
- srv\_lockmutex 116, 312, 314
- srv\_log 91, 314, 317
- SRV\_M\_NOWAIT value 295
- SRV\_M\_READ\_ONLY value 295
- SRV\_M\_WAIT value 295
- SRV\_M\_WAKE\_INTR 20
- srv\_mask 317, 318
- SRV\_MAXRESMSG message ID 80
- SRV\_MIG\_STATE enumerated type 46
- SRV\_MIGRATE\_RESUME 44
- SRV\_MIGRATE\_RESUME event 97
- SRV\_MIGRATE\_STATE 45
- SRV\_MINRESMSG message ID 80
- srv\_msg 80, 81, 318, 320
- SRV\_MSG event 80, 98
- SRV\_MSG event handler 94, 137
- SRV\_MSGDATA type of data 237, 241
- SRV\_NEGDATA type of data 237
- srv\_negotiate 123, 322, 329
- SRV\_NOPARAMS value 81
- srv\_numparams 137, 172, 329, 331
- SRV\_OPTION event 97, 98, 125
- SRV\_OPTION event handler
  - renegotiating behavior 123
- srv\_options 125, 331, 337
- srv\_orderby 337
- srv\_poll (UNIX only) 339, 341
- SRV\_POLL capability 248
- SRV\_PROC structure 95
- SRV\_PROCLIST structure 168
- srv\_props 20, 142, 341, 343
- srv\_putmsgq 118, 348, 350
- srv\_realloc 350, 352
- srv\_recvpassthru 134, 352, 354
- srv\_regcreate 167, 354, 356
- srv\_regdefine 167, 356, 360
- srv\_regdrop 167, 360, 362
- srv\_regexec 167, 362, 363
- srv\_reginit 167, 363, 366
- srv\_reglist 167, 366, 367
- srv\_reglistfree 168, 367, 369

- srv\_regnowatch 167, 369, 371
- srv\_regparam 167, 371, 374
- srv\_regwatch 167, 374, 377
- srv\_regwatchlist 167, 377, 379
- SRV\_ROWDATA type of data 237
- SRV\_ROWDATA value 137
- SRV\_RPC event 98, 171
  - and registered procedures 164
- SRV\_RPC event handler 137, 165, 171
  - trapping errors 172
- SRV\_RPCDATA type of data 237
- srv\_rpcdb 171, 379, 380
- srv\_rpcname 171, 380, 383
- srv\_rpcnumber 171, 383, 384
- srv\_rpcoptions 384, 386
- srv\_rpcowner 171, 386, 387
- srv\_run 98, 387, 389
- SRV\_S\_ALLOCFUNC property 143
- SRV\_S\_APICLK property 144
- SRV\_S\_ATTNRASON property 144
- SRV\_S\_CURTHREAD property 144
- SRV\_S\_DEFQUEUESIZE property 144
- SRV\_S\_DISCONNECT property 22, 144
- SRV\_S\_DS\_REGISTER property 82, 144
- SRV\_S\_DSPROVIDER property 144
- SRV\_S\_ERRHANDLE property 90, 144
- SRV\_S\_FREEFUNC property 145
- SRV\_S\_IFILE property 145
- SRV\_S\_INHIBIT property 169
- SRV\_S\_INHIBIT return value 247
- SRV\_S\_LOGFILE property 145
- SRV\_S\_LOGSIZE property 91, 145
- SRV\_S\_MAXLISTENERS property 89, 145
- SRV\_S\_MSGPOOL property 146
- SRV\_S\_NETBUFSIZE property 146
- SRV\_S\_NETTRACEFILE property 146
- SRV\_S\_NUMCONNECTIONS property 146
- SRV\_S\_NUMLISTENERS property 89, 146
- SRV\_S\_NUMMSGQUEUES property 147
- SRV\_S\_NUMMUTEXES property 147
- SRV\_S\_NUMREMBUF property 147
- SRV\_S\_NUMREMSITES property 147
- SRV\_S\_NUMTHREADS property 147
- SRV\_S\_NUMUSEREVENTS property 147
- SRV\_S\_PREEMPT property 116, 147
- SRV\_S\_REALLOCFUNC property 147
- SRV\_S\_REQUEST\_CAP property 147
- SRV\_S\_RESPONSE\_CAP property 147
- SRV\_S\_RETPARAMS property 148
- SRV\_S\_RETPARMS property 148
- SRV\_S\_SEC\_PRINCIPAL property 148, 184
- SRV\_S\_SERVERNAME property 148
- SRV\_S\_STACKSIZE property 149, 430
- SRV\_S\_TCP\_RCVBUF 344
- SRV\_S\_TCP\_SNDBUF property 344
- SRV\_S\_TDSVERSION property 149, 150, 163
- SRV\_S\_TIMESLICE property 149
- SRV\_S\_TRACEFLAG property 149, 150
- SRV\_S\_TRUNCATELOG property 149
- SRV\_S\_USERSRVLANG property 149
- SRV\_S\_USESRVLANG property 102, 108, 158
- SRV\_S\_VERSION property 150
- SRV\_S\_VIRTCLKRATE property 150
- SRV\_S\_VIRTIMER property 150
- SRV\_SECLABEL value 154
- srv\_select (UNIX only) 389, 392
- srv\_send\_ctlinfo 393
- srv\_send\_data 394
- srv\_send\_text 203, 392, 401
- srv\_senddone 401, 405
- srv\_sendinfo 38, 405, 409
- srv\_sendpassthru 134, 409, 411
- srv\_sendstatus 411, 413
- SRV\_SERVER structure 308
- srv\_setcolotype 413, 414
- srv\_setcontrol 414, 417
- srv\_setlogininfo 31, 417, 419
- SRV\_SETOPTION value 125
- srv\_setpri 419, 421
  - in multithread programming 118
- srv\_signal (UNIX only) 421, 422
- SRV\_SITEHANDLER login type 161
- srv\_sleep 114, 116, 424, 427
- srv\_spawn 427, 430
- SRV\_START event 93, 98
- SRV\_START handler 94
- SRV\_STOP event 93, 96, 98, 284, 287
  - fatal errors 92
  - SRV\_SERVER structure 308
- SRV\_SUBCHANNEL login type 161
- srv\_symbol 430, 431
- SRV\_T\_APPLNAME property 151

## Index

- SRV\_T\_BULKTYPE property 152, 159
- SRV\_T\_BYTEORDER property 152
- SRV\_T\_CHARTYPE property 159
- SRV\_T\_CHARYPE property 152
- SRV\_T\_CIPHER\_SUITE property 152
- SRV\_T\_CLIB property 152
- SRV\_T\_CLIBVERS property 152
- SRV\_T\_CLIENLOGOUT property 152
- SRV\_T\_CONVERTSHORT property 152
- SRV\_T\_DUMPLOAD property 152
- SRV\_T\_ENDPOINT property 152
- SRV\_T\_EVENT property 153, 160
- SRV\_T\_EVENTDATA property 153
- SRV\_T\_FLTTYPER property 153, 161
- SRV\_T\_FULLPASSTHRU property 153
- SRV\_T\_GOTATTENTION property 21, 153
- SRV\_T\_HOSTNAME property 153
- SRV\_T\_HOSTPROCID property 153
- SRV\_T\_IODEAD property 153
- SRV\_T\_LISTENADDR property 89, 153
- SRV\_T\_LOCALE property 153
- SRV\_T\_LOCALID property 89, 154
- SRV\_T\_LOGINTYPE property 154, 161
- SRV\_T\_MACHINE property 154
- SRV\_T\_MIGRATE\_STATE property 154
- SRV\_T\_MIGRATED 47
- SRV\_T\_MIGRATED property 154
- SRV\_T\_NEGLOGIN property 154
- SRV\_T\_NOTIFYCHARSET property 155
- SRV\_T\_NOTIFYDB property 155
- SRV\_T\_NOTIFYLANG property 155
- SRV\_T\_NUMRMTPWDS property 155
- SRV\_T\_PACKETSIZE property 155
- SRV\_T\_PASSTHRU property 155
- SRV\_T\_PRIORITY property 155
- SRV\_T\_PWD property 155
- SRV\_T\_REMOTEADDR property 89, 155
- SRV\_T\_RETPARAMS property 155
- SRV\_T\_RMTPWD structure 163
- SRV\_T\_RMTPWDS property 156, 163
- SRV\_T\_RMTSERVER property 156
- SRV\_T\_ROWSENT property 156
- SRV\_T\_SEC\_CHANBIND property 156
- SRV\_T\_SEC\_CONFIDENTIALITY property 156
- SRV\_T\_SEC\_CREDTIMEOUT property 156
- SRV\_T\_SEC\_DATAORIGIN property 156
- SRV\_T\_SEC\_DELEGATION property 156
- SRV\_T\_SEC\_DELEGCREC property 157
- SRV\_T\_SEC\_DETECTREPLAY property 157
- SRV\_T\_SEC\_DETECTSEQ property 157
- SRV\_T\_SEC\_INTEGRITY property 157
- SRV\_T\_SEC\_MECHANISM property 157
- SRV\_T\_SEC\_MUTUALAUTH property 157
- SRV\_T\_SEC\_NETWORKAUTH property 157
- SRV\_T\_SEC\_SESSTIMEOUT property 157
- SRV\_T\_SESSIONID 47
- SRV\_T\_SESSIONID property 158
- SRV\_T\_SPID property 158
- SRV\_T\_SSL\_VERSION property 158
- SRV\_T\_STACKLEFT property 158
- SRV\_T\_TDSVERSION property 158
- SRV\_T\_TYPE property 158, 164
- SRV\_T\_USER property 158
- SRV\_T\_USERDATA property 158
- SRV\_T\_USESRVLANG property 102, 108, 158
- SRV\_T\_USTATE property 158
- srv\_tabcolname 434, 436
  - calling to return browse mode results 23
- srv\_tabname 436, 438
  - calling to return browse mode results 23
- SRV\_TCLIENT thread type 164
- SRV\_TDS\_values 150, 164
- srv\_termproc 259, 438, 439
- srv\_text\_info 58, 202, 440, 442
- SRV\_TEXTLOAD value 159
- srv\_thread\_props 142, 151, 442, 444
- srv\_thread\_props property 20
- SRV\_TIMESLICE configuration parameter 120
- SRV\_TLISTENER thread type 89, 164
- SRV\_TR\_ATTEN value 151
- SRV\_TR\_DEFQUEUE value 151
- SRV\_TR\_EVENT value 151
- SRV\_TR\_MSGQ value 151
- SRV\_TR\_NETDRIVER value 151
- SRV\_TR\_NETREQ value 151
- SRV\_TR\_NETWAKE value 151
- SRV\_TR\_TDSDATA value 151
- SRV\_TR\_TDSHDR value 151
- SRV\_TSERVICE thread type 164
- SRV\_TSITE thread type 164
- SRV\_TSUBPROC thread type 164
- srv\_ucwakeup 20, 449



- SRV\_UNITEXTLOAD value 159
  - srv\_unlockmutex 452
  - SRV\_URGDISCONNECT event 93, 99, 284, 286
  - srv\_version 452, 454
  - SRV\_VIRTCLKRATE configuration parameter 120
  - SRV\_VIRTTIMER configuration parameter 120
  - srv\_wakeup 20, 114, 449, 454, 456
  - srv\_xferdata 136, 140, 240, 456, 458
  - srv\_yield 114, 116, 458, 460
  - SSL
    - certificates 182
    - SDC 182
  - stack size
    - threads 149
  - stack space
    - determining through srv\_thread\_props 158
  - standard events 93
  - start handler. See SRV\_START handler 216
  - state transition handler. See Callbacks 244
  - state transitions
    - for srv\_callback 120
  - status values
    - returning to client 16
  - suspended threads 116
  - switching
    - thread contexts 265
  - system registered procedures
    - definition of 168
    - mapping to Server-Library routines 168
- T**
- Tabular Data Stream protocol. See TDS 3
  - TCL
    - Net-Lib driver requests 151
    - wakeup requests 151
  - TDS
    - definition of 3
    - pass-through mode 101
    - protocol level 122
    - retrieving and setting client threadxd5 s version via
      - srv\_thread\_props 158
    - specifying initial version value for through
      - srv\_props 149
  - TDS packets
    - header information 151
    - pass-through mode 131
  - TDS version 150
    - and capabilities 37
    - legal values 150
    - negotiation 163
  - text and image 201, 204
  - text and image data
    - retrieving from a client 201, 204
    - sending to a client 202
  - text and image datatypes 216
  - text datatype 27, 29, 57, 201, 215, 216
    - srv\_get\_text 289, 290
    - text pointer 201
    - text timestamp 201
    - transferring 159
  - @@textsize global variable 129
  - third-party security 174
  - threads
    - See also multithread programming 111
    - call stack 263
    - communication 118
    - configuring number available, through srv\_props
      - 147
    - current state 158
    - definition 111
    - IDs 158
    - login records 161
    - messages 111
    - non-client 257
    - preemptive 112
    - properties 151, 163
    - stack size 149
    - state transitions 244
    - switching contexts 265
    - Thread Properties table 151, 158
    - types 112, 164
    - types, retrieving clientxd5 s via srv\_thread\_props
      - 158
  - time slice callback 149
  - trace flags
    - summary of Open Server trace flags 151
  - tracing 149, 151
  - transaction isolation 127
  - transferring data 140
  - transparent negotiation 30, 122

## *Index*

capabilities 24  
types 204, 216

## **U**

unchained transactions 126  
unitext datatype  
    srv\_get\_data 289  
updates 127  
    cursors 63, 64, 67, 76  
use db command 155  
user authorizations 126  
user events  
    defining 266  
    number 147  
user name  
    retrieving clientxds via srv\_thread\_props 158  
user-defined events 284, 287

## **V**

variable-length binary datatype 26  
    long 26  
version string 150  
virtual timer 150

## **W**

weeks, first day 127  
writetext stream 202

## **X**

XML datatype 210