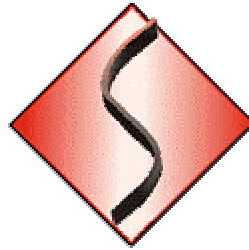


SQL TOOLS™

© 1999-2003 Perfect Sync, Inc.
All Rights Reserved



This is version 2.10 of the
SQL Tools Help File
Build ID# 071803

Copyright and Trademark Information...

SQL TOOLS™

© 1999-2003 Perfect Sync, Inc.



This document is © 1999-2003 Perfect Sync, Inc.
All Rights Reserved.

No portion of this document may be reproduced
without the written permission of Perfect Sync.

Perfect Sync and SQL Tools are trademarks of

PERFECT SYNC

**8121 Hendrie Blvd., Suite C
Huntington Woods, Michigan
(USA) 48070**

Internet: <http://www.perfectsync.com>.
EMail: support@perfectsync.com

Fax: +1 (248) 546-4888
Voice: +1 (248) 546-5300
*SQL Tools support is not available via
telephone except on an hourly-fee basis.*
See Getting Technical Help

Microsoft®, **Windows®**, **Windows 95®**, **Windows 98®**,
Windows ME®, **Windows NT®**, **Windows 2000®**, **Windows XP®**,
Visual Studio®, **Microsoft Access®**, **Microsoft Excel®**, **FoxPro®**,
SQL Server®, **Visual Basic®**, **Visual C++®**, **QBASIC®**, and **MSFlexGrid®**
are trademarks of Microsoft Corporation.

PowerBASIC®, **PB/CC®**, **PB/DLL®**, and **PB/DOS®**
are trademarks of PowerBASIC, Inc.

UltraEdit® and **UltraEdit32®**
are trademarks of IDM Computer Solutions, Inc.

InstallShield® is a trademark of InstallShield Software Corporation.

InstallBuilder® is a trademark of Wise.Solutions, Inc.

Oracle® and **SQL *Plus®** are trademarks of Oracle Corporation.

Paradox® is a trademark of Corel Corporation, and
Quattro® Pro is a trademark of Corel Corporation Limited.

dBASE® is a trademark of dBASE, Incorporated.

Delphi® is a trademark of Borland Software Corporation.

Lotus Notes® is a trademark of Lotus Development Corporation.

SQLBase® is a trademark of Centura Software Corporation.

Amazon.com® is a trademark of Amazon.com, Inc.

AltaVista® is a trademark of Digital Equipment Corporation.

Spread® is a trademark of FarPoint Technologies, Inc.

DBGrid® and **VSFlexGrid®** are trademarks of ComponentOne, LLC.

UltraGrid® and **DataTable®** are trademarks of Infragistics, Inc.

SQLBase® is a trademark of Centura Software/Gupta Technologies, LLC.

Btrieve® is a trademark of Pervasive Software

Sybase® is a trademark of Sybase, Inc.

Adobe Acrobat® is a trademark of Adobe Systems Incorporated

**All brand names and product names that are used
in this document are trade names, service marks, or
registered trademarks of their respective owners.**

Other trademarks may have been inadvertently used in this document
without recognition here. Please contact support@perfectsync.com
and omissions will be corrected in future versions of this document.

TABLE OF CONTENTS

COPYRIGHT AND TRADEMARK INFORMATION.....	2
LICENSE AGREEMENT AND RUNTIME FILE DISTRIBUTION RIGHTS.....	25
SQL TOOLS AUTHORIZATION CODES	27
TROUBLESHOOTING YOUR PROGRAMS.....	29
GETTING TECHNICAL HELP.....	31
FREQUENTLY ASKED QUESTIONS.....	33
WHAT SQL TOOLS IS AND ISN'T.....	34
WHAT'S THE DIFFERENCE BETWEEN SQL TOOLS STANDARD AND PRO?	35
WHAT WILL SQL TOOLS DO FOR MY PROGRAMS?	37
WHAT WILL I NEED TO USE SQL TOOLS?.....	38
WHAT'S THE DIFFERENCE BETWEEN SQL AND ODBC?	40
CAN I USE SQL TOOLS TO WRITE "UNIVERSAL" PROGRAMS?	41
DO ALL SQL TOOLS FEATURES WORK WITH ALL DATABASES?	42
HOW COMPLETE IS SQL TOOLS?	43
EXACTLY WHICH ODBC FEATURES ARE NOT SUPPORTED?.....	44
READY TO WRITE PROGRAMS? START HERE!.....	46
CONVENTIONS USED IN THIS DOCUMENT.....	47
VARIABLE NAMING CONVENTIONS.....	48
SIGNED AND UNSIGNED INTEGERS.....	49
INSTALLING SQL TOOLS	52
INSTALLING ODBC DRIVERS.....	54
INSTALLING ODBC DRIVERS FROM THE MICROSOFT INTERNET SITE	56
INSTALLING ODBC DRIVERS FROM A DATABASE PRODUCT	57
TERMINOLOGY DIFFERENCES.....	58
COMPLIANCE ISSUES.....	59
TWO OF EVERYTHING: THE "ABBREVIATED" AND "VERBOSE" FUNCTIONS	61

FOUR OF MANY THINGS	63
EIGHT OR MORE OF SOME THINGS	64
THE ABBREVIATIONS	66
FOUR CRITICAL STEPS FOR EVERY SQL TOOLS PROGRAM	67
SPECIAL CONSIDERATIONS FOR DLL PROGRAMMERS	70
A SQL TOOLS PRIMER.....	72
WHAT A DATABASE IS	73
SQL AND ODBC.....	74
ODBC DRIVERS, AND THE DRIVER MANAGER	75
SQL TOOLS AND ODBC DRIVERS	77
OPENING A DATABASE.....	78
USING A DSN FILE TO OPEN A DATABASE	79
USING A CONNECTION STRING TO OPEN A DATABASE.....	80
MANUAL NAVIGATION: USING SQL_OPENDB TO CREATE A DSN FILE	81
ERROR MESSAGES AFTER OPENING A DATABASE	82
DETERMINING DATABASE CAPABILITIES.....	83
TABLES, ROWS, COLUMNS, AND CELLS.....	85
TABLE METADATA.....	86
SQL DATA TYPES.....	87
SQL_CHAR	88
SQL_VARCHAR	89
SQL_LONGVARCHAR.....	90
SQL_INTEGER	91
SQL_SMALLINT	92
SQL_TINYINT	93
SQL_BIT	94
SQL_BIGINT	95
SQL_REAL	96

SQL_DOUBLE	97
SQL_FLOAT	98
SQL_NUMERIC AND SQL_DECIMAL	99
SQL_TIMESTAMP AND SQL_TYPE_TIMESTAMP	100
SQL_DATE AND SQL_TYPE_DATE.....	103
SQL_TIME AND SQL_TYPE_TIME	104
SQL_INTERVAL DATA TYPES.....	105
SQL_BINARY, SQL_VARBINARY, AND SQL_LONGVARBINARY	106
LENGTHS OF SQL_CHAR AND SQL_BINARY DATA TYPES	107
SQL_DEFAULT	108
DATASOURCE-DEPENDENT DATA TYPES	109
UNICODE DATA TYPES.....	110
SQL_WCHAR.....	112
SQL_WVARCHAR	113
SQL_WLONGVARCHAR.....	114
SQL DATA TYPE "PROPERTIES"	115
CONCISE TYPE.....	116
BUFFER SIZE	117
TRANSFER OCTET LENGTH.....	118
NUM PREC RADIX.....	119
DISPLAY SIZE	120
DECIMAL DIGITS.....	121
BASIC DATA TYPES.....	122
SQL STATEMENTS.....	124
EXECUTION OF SQL STATEMENTS	125
ASYNCHRONOUS EXECUTION OF SQL STATEMENTS.....	127
SQL STATEMENT MODE.....	128
BINDING STATEMENT INPUT PARAMETERS	130

BINDING NUMERIC PARAMETERS	132
SETTING A BOUND PARAMETER TO THE NULL VALUE	137
BINDING FIXED-LENGTH STRING/BINARY PARAMETERS.....	138
BINDING DYNAMIC STRING/BINARY PARAMETERS.....	139
BINDING LONG PARAMETER VALUES	141
ARRAYS OF BOUND PARAMETERS	144
RESULT SETS.....	145
RESULT COLUMN BINDING (BASIC).....	146
FETCHING ROWS FROM RESULT SETS (BASIC)	147
CURSORS	148
FORWARD-ONLY CURSORS	149
SCROLLABLE CURSORS	150
PROBLEMS WITH SCROLLABLE CURSORS	151
FETCHING ROWS FROM RESULT SETS (ADVANCED)	153
DETERMINING CURSOR CAPABILITIES	154
USING BOOKMARKS	155
BINDING COLUMN ZERO	157
RELATIVE FETCHES	158
RESULT COLUMN BINDING (ADVANCED).....	159
AUTOBINDING	160
OTHER BINDING ALTERNATIVES	161
PROXY BINDING	162
MANUAL BINDING AND DIRECT BINDING.....	163
DIRECT BINDING.....	164
MANUAL BINDING	165
ROW-WISE BINDING	166
ACCESSING RESULT COLUMNS	167
LONG COLUMNS.....	168

"DATA TRUNCATED" ERROR MESSAGES.....	169
POSSIBLE DRIVER RESTRICTIONS ON LONG COLUMNS.....	170
RESULT COLUMN INDICATORS.....	171
NULL VALUES	172
OTHER USES OF COLUMN INDICATORS.....	173
RESULTS FROM NON-SELECT STATEMENTS.....	174
WHY YOU CAN'T USE SQL_RESROWCOUNT FOR SELECT STATEMENTS	175
DETECTING THE END OF DATA.....	176
DETECTING "NO DATA AT ALL "	179
ERROR HANDLING IN SQL TOOLS PROGRAMS	180
ERROR CODES.....	181
USING ERROR MESSAGES INSTEAD OF ERROR CODES.....	182
IGNORING PREDICTABLE ERRORS.....	184
MISCELLANEOUS ERROR HANDLING TECHNIQUES	186
SQL TOOLS TRACE MODE	187
ODBC API TRACING.....	188
SQL TOOLS UTILITY FUNCTIONS.....	189
DATABASE INFORMATION AND ATTRIBUTES	190
STATEMENT INFORMATION AND ATTRIBUTES.....	191
ENVIRONMENT ATTRIBUTES.....	192
MANUALLY OPENING AND CLOSING DATABASES.....	193
MANUALLY OPENING AND CLOSING STATEMENTS.....	194
USING DATABASE NUMBERS AND STATEMENT NUMBERS	195
STATEMENT ZERO OPERATION	197
CACHED INFORMATION	198
INDEXES.....	199
AUTOCOLUMNS.....	200
UNIQUE COLUMNS AND PRIMARY COLUMNS	201

FOREIGN KEYS	203
TABLE PRIVILEGES AND COLUMN PRIVILEGES.....	204
COMMITTING TRANSACTIONS MANUALLY.....	205
STORED PROCEDURES.....	206
MULTIROW CURSORS	207
NAMED CURSORS.....	209
BULK OPERATIONS.....	210
USING BULK_UPDATE.....	212
USING BULK_ADD.....	213
USING BULK_FETCH	214
USING BULK_DELETE.....	215
POSITIONED UPDATES AND DELETES.....	216
USING LONG VALUES WITH BULK AND POSITIONED OPERATIONS.....	217
"CLEANING UP" AFTER A BULK OPERATION.....	219
USING SQL TOOLS WITH A GRID	220
MULTI-THREADED PROGRAMS	221
SQL HANDLES.....	225
REFERENCE GUIDE FORMAT	226
FUNCTIONAL FAMILIES.....	227
CONFIGURATION FAMILY.....	228
ENVIRONMENT FAMILY	229
USE FAMILY	230
DATABASE OPEN/CLOSE FAMILY	231
DATABASE INFO/ATTRIB FAMILY	232
TABLE INFO FAMILY.....	233
TABLE COLUMN INFO FAMILY	234
STATEMENT OPEN/CLOSE FAMILY.....	236
STATEMENT FAMILY	237

STATEMENT INFO/ATTRIB FAMILY	238
STATEMENT BINDING FAMILY	239
STORED PROCEDURE FAMILY	240
RESULT COLUMN BINDING FAMILY	241
RESULT COUNT FAMILY	242
RESULT COLUMN FAMILY	243
ERROR/TRACE FAMILY	244
UTILITY FAMILY	245
GET INFO FAMILY	246
HANDLE FAMILY	247
SQL_ASYNCERRORS	248
SQL_ASYNCSTATEMENT	249
SQL_ASYNCSTATUS	251
SQL_ASYNCSTMT	253
SQL_AUTHORIZE	257
SQL_AUTOBINDCOL	260
SQL_AUTOBINDCOLUMN	262
SQL_AUTOCOLCOUNT	264
SQL_AUTOCOLINFOSINT	265
SQL_AUTOCOLINFOSTR	267
SQL_AUTOCOLUMNCOUNT	269
SQL_AUTOCOLUMNINFOSINT	270
SQL_AUTOCOLUMNINFOSTR	271
SQL_BINARYSTR	272
SQL_BINDPARAM	274
SQL_BINDPARAMETER	276
SQL_BKMK	278
SQL_BOOKMARK	279

SQL_BULKOP	280
SQL_BULKOPERATION	281
SQL_CLOSEDATABASE	282
SQL_CLOSEDDB	283
SQL_CLOSESTATEMENT	285
SQL_CLOSESTMT	286
SQL_COLPRIVCOUNT	288
SQL_COLPRIVINFOSTR	289
SQL_COLUMNPRIVILEGECOUNT	292
SQL_COLUMNPRIVILEGEINFOSTR	293
SQL_CURNAME	294
SQL_CURRENTDB	295
SQL_CURRENTSTMT	296
SQL_CURSORNAME	297
SQL_DATABASEATTRIBSTR	298
SQL_DATABASEATTRIBUINT	299
SQL_DATABASEAUTOCOMMIT	300
SQL_DATABASEDATATYPECOUNT	301
SQL_DATABASEDATATYPEINFOSINT	302
SQL_DATABASEDATATYPEINFOSTR	303
SQL_DATABASEDATATYPENUMBER	304
SQL_DATABASEINFOSTR	306
SQL_DATABASEINFOUINT	307
SQL_DATABASEISOPEN	308
SQL_DATASOURCEADD	309
SQL_DATASOURCEADMIN	311
SQL_DATASOURCECOUNT	313
SQL_DATASOURCEINFOSTR	314

SQL_DATASOURCEMODIFY	315
SQL_DATASOURCENUMBER	320
SQL_DBATTRIBSTR.....	321
SQL_DBATTRIBUINT	323
SQL_DBAUTOCOMMIT.....	326
SQL_DBDATATYPECOUNT.....	327
SQL_DBDATATYPEINFOSINT	329
SQL_DBDATATYPEINFOSTR	333
SQL_DBDATATYPENUMBER	335
SQL_DBINFOSTR.....	336
SQL_DBINFOUINT	342
SQL_DBISOPEN	376
SQL_DIAGNOSTIC	377
SQL_DIRECTBINDCOL	381
SQL_DIRECTBINDCOLUMN	383
SQL_DRIVERCOUNT	385
SQL_DRIVERINFOSTR	387
SQL_DRIVERNUMBER	389
SQL_ENDOFDATA.....	391
SQL_ENDTRANS.....	393
SQL_ENDTRANSACTION.....	394
SQL_ENVIRONATTRIBSINT	395
SQL_EOD	397
SQL_ERRORCLEARALL	398
SQL_ERRORCLEARONE	399
SQL_ERRORCOLUMNNUMBER.....	400
SQL_ERRORCOUNT	401
SQL_ERRORDATABASENUMBER	402

SQL_ERRORFUNCNAME	403
SQL_ERRORFUNCTION	405
SQL_ERRORIGNORE	406
SQL_ERRORNATIVECODE.....	408
SQL_ERRORNUMBER	410
SQL_ERRORPENDING	411
SQL_ERRORQUICKALL	412
SQL_ERRORQUICKONE	413
SQL_ERRORSIMULATE	415
SQL_ERRORSTATEMENTNUMBER.....	416
SQL_ERRORSTR.....	417
SQL_ERRORTIME	419
SQL_ERRORTIME	421
SQL_FETCH.....	422
SQL_FETCHPOS	424
SQL_FETCHPOSITION.....	427
SQL_FETCHREL.....	429
SQL_FETCHRELATIVE.....	432
SQL_FETCHRESULT.....	434
SQL_FOREIGNKEYCOUNT	435
SQL_FOREIGNKEYINFOSINT	436
SQL_FOREIGNKEYINFOSTR	437
SQL_FORNKEYCOUNT	438
SQL_FORNKEYINFOSINT	440
SQL_FORNKEYINFOSTR	443
SQL_FUNCAVAIL	445
SQL_FUNCTIONAVAILABLE.....	448
SQL_GETAUTOCOLS	449

SQL_GETAUTOCOLUMNS	450
SQL_GETCOLPRIVS	451
SQL_GETCOLUMNPRIVILEGES	452
SQL_GETDATABASEDATATYPES.....	454
SQL_GETDATASOURCES	455
SQL_GETDBDATATYPES.....	456
SQL_GETDRIVERS	457
SQL_GETFOREIGNKEYS.....	458
SQL_GETFORNKEYS.....	460
SQL_GETINDEXES	461
SQL_GETINDXES	462
SQL_GETPRIMARYKEYS.....	463
SQL_GETPRIMKEYS	465
SQL_GETPROCCOLS	466
SQL_GETPROCEDURECOLUMNS	467
SQL_GETPROCEDURES.....	468
SQL_GETPROCS.....	469
SQL_GETTABLECOLUMNS	470
SQL_GETTABLEINFO	472
SQL_GETTABLEPRIVILEGES.....	473
SQL_GETTBLCOLS.....	475
SQL_GETTBLINFO	476
SQL_GETTBLPRIVS	479
SQL_GETUNIQCOLS.....	480
SQL_GETUNIQUECOLUMNS	481
SQL_HDATABASE	483
SQL_HDB	484
SQL_HENVIRONMENT	486

SQL_HPARENTWINDOW	487
SQL_HSTATEMENT	489
SQL_HSTMT	490
SQL_INDEXCOUNT	491
SQL_INDEXINFOSINT	492
SQL_INDEXINFOSTR	493
SQL_INDXCOUNT	494
SQL_INDYINFOSINT	495
SQL_INDYINFOSTR	497
SQL_INFOEXPORT	499
SQL_INFOIMPORT	501
SQL_INIT	503
SQL_INITIALIZE	504
SQL_ISTRING	508
SQL_LIMITTEXTLENGTH	511
SQL_LONGPARAM	513
SQL_LONGPARAMETER	514
SQL_LONGRESCOL	515
SQL_LONGRESULTCOLUMN	518
SQL_MANUALBINDCOL	519
SQL_MANUALBINDCOLUMN	521
SQL_MORERES	523
SQL_MORERESULTS	525
SQL_MSGBOX	526
SQL_MSGBOXBUTTON	528
SQL_NAMECUR	530
SQL_NAMECURSOR	532
SQL_NEWDBNUMBER AND SQL_NEWDATABASENUMBER	533

SQL_NEWSTATEMENTNUMBER	535
SQL_NEWSTMTNUMBER	536
SQL_NEXTPARAM	538
SQL_NEXTPARAMETER	539
SQL_OKAY	540
SQL_ONERRORCALL	542
SQL_OPENDATABASE.....	544
SQL_OPENDATABASE1	546
SQL_OPENDATABASE2	547
SQL_OPENDB	548
SQL_OPENSTATEMENT	553
SQL_OPENSTMT	554
SQL_OPTIONRESETALL	556
SQL_OPTIONSINT	557
SQL_OPTIONSTR	559
SQL_PARAMETERINFOUINT	561
SQL_PARAMINFOUINT	563
SQL_PRIMARYKEYCOUNT	565
SQL_PRIMARYKEYINFOSINT	566
SQL_PRIMARYKEYINFOSTR	567
SQL_PRIMKEYCOUNT	568
SQL_PRIMKEYINFOSINT	569
SQL_PRIMKEYINFOSTR	571
SQL_PROCCOLCOUNT	573
SQL_PROCCOLINFOSINT	575
SQL_PROCCOLINFOSTR	579
SQL_PROCCOUNT	582
SQL_PROCEDURECOLUMNCOUNT	583

SQL_PROCEDURECOLUMNINFOSINT	584
SQL_PROCEDURECOLUMNINFOSTR.....	586
SQL_PROCEDURECOUNT	588
SQL_PROCEDUREINFOSINT	589
SQL_PROCEDUREINFOSTR	590
SQL_PROCINFOSINT	591
SQL_PROCFINFOSTR.....	593
SQL_RESCOLBINT	595
SQL_RESCOLBUFFERPTR	597
SQL_RESCOLCOUNT	599
SQL_RESCOLDATE.....	600
SQL_RESCOLDATETIME	603
SQL_RESCOLDATETIMEPART	605
SQL_RESCOLFLOAT	607
SQL_RESCOLIND.....	610
SQL_RESCOLINDPTR.....	612
SQL_RESCOLINFOSINT	614
SQL_RESCOLINFOSTR	618
SQL_RESCOLLEN	621
SQL_RESCOLMORE	623
SQL_RESCOLNULL.....	625
SQL_RESCOLNUMBER	626
SQL_RESCOLSINT	627
SQL_RESCOLSIZE.....	629
SQL_RESCOLSTR	630
SQL_RESCOLTEXT	632
SQL_RESCOLTIME.....	636
SQL_RESCOLTYPE.....	639

SQL_RESCOLUINT	640
SQL_RESETSTATEMENTMODE	642
SQL_RESETSTMTMODE	643
SQL_RESROWCOUNT	644
SQL_RESULTCOLUMNBINT	645
SQL_RESULTCOLUMNBUFFERPTR	646
SQL_RESULTCOLUMNNCOUNT	647
SQL_RESULTCOLUMNDATE	648
SQL_RESULTCUMNDATETIME	649
SQL_RESULTCUMNDATETIMEPART	650
SQL_RESULTCOLUMNFLOAT	652
SQL_RESULTCOLUMNIND	653
SQL_RESULTCOLUMNINDPTR	654
SQL_RESULTCOLUMNINFOSINT	655
SQL_RESULTCOLUMNINFOSTR	656
SQL_RESULTCOLUMNLEN	657
SQL_RESULTCOLUMNMORE	659
SQL_RESULTCOLUMNNULL	660
SQL_RESULTCOLUMNNUMBER	661
SQL_RESULTCUMNSINT	662
SQL_RESULTCUMNSIZE	663
SQL_RESULTCOLUMNSTR	665
SQL_RESULTCUMNTEXT	666
SQL_RESULTCUMNTIME	667
SQL_RESULTCUMNTYPE	669
SQL_RESULTCUMNUINT	670
SQL_RESULTTROWCOUNT	671
SQL_SELECTFILE	673

SQL_SETDATABASEATTRIBSTR.....	679
SQL_SETDATABASEATTRIBUINT	680
SQL_SETDBATTRIBSTR.....	681
SQL_SETDBATTRIBUINT.....	683
SQL_SETENVIRONATTRIBSINT	688
SQL_SETOPTIONSINT.....	690
SQL_SETOPTIONSTR.....	691
SQL_SETPOS.....	704
SQL_SETPOSITION	707
SQL_SETSTATEMENTATTRIB	709
SQL_SETSTMTATTRIB.....	711
SQL_SHUTDOWN	716
SQL_STATE.....	717
SQL_STATEMENT.....	718
SQL_STATEMENTATTRIB	719
SQL_STATEMENTCANCEL	720
SQL_STATEMENTINFOSTR	721
SQL_STATEMENTISOPEN.....	722
SQL_STATEMENTMODE.....	723
SQL_STATEMENTNATIVESYNTAX.....	724
SQL_STATEMENTPARAMETERCOUNT	725
SQL_STMT	726
SQL_STMTATTRIB	729
SQL_STMTCANCEL.....	730
SQL_STMTINFOSTR.....	732
SQL_STMTISOPEN	734
SQL_STMTMODE	736
SQL_STMTNATIVESYNTAX	743

SQL_STMTPARAMCOUNT	745
SQL_STRINGTOTYPE.....	746
SQL_SYNCFETCHPOS	748
SQL_SYNCFETCHPOSITION.....	749
SQL_TABLECOLUMNCOUNT	751
SQL_TABLECOLUMNINFOSINT	752
SQL_TABLECOLUMNINFOSTR.....	753
SQL_TABLECOLUMNNUMBER.....	754
SQL_TABLECOUNT	755
SQL_TABLEINFOSINT	756
SQL_TABLEINFOSTR	757
SQL_TABLENUMBER	758
SQL_TABLEPRIVILEGECOUNT	759
SQL_TABLEPRIVILEGEINFOSTR.....	760
SQL_TABLESTATISTICSINT	761
SQL_TBLCOLCOUNT.....	762
SQL_TBLCOLINFOSINT	764
SQL_TBLCOLINFOSTR	768
SQL_TBLCOLNUMBER	771
SQL_TBLCOUNT	772
SQL_TBLINFOSINT.....	774
SQL_TBLINFOSTR.....	776
SQL_TBLNUMBER.....	778
SQL_TBLPRIVCOUNT.....	780
SQL_TBLPRIVINFOSTR	782
SQL_TBLSTATSINT.....	784
SQL_TEXTDATE.....	786
SQL_TEXTDATETIME	787

SQL_TEXTSTR AND SQL_TEXTSTROLE	789
SQL_TEXTTIME	791
SQL_THREAD	792
SQL_TOOLSVERSION	794
SQL_TRACE	795
SQL_TRACESINT	798
SQL_TRACESTR AND SQL_TRACESTROLE	799
SQL_UNBINDCOL	801
SQL_UNBINDCOLUMN	803
SQL_UNIQCOLCOUNT	804
SQL_UNIQCOLINFOSINT	805
SQL_UNIQCOLINFOSTR	808
SQL_UNIQUECOLUMNCOUNT	810
SQL_UNIQUECOLUMNINFOSINT	811
SQL_UNIQUECOLUMNINFOSTR.....	812
SQL_USEDDB.....	813
SQL_USEDBSTMT	814
SQL_USESTMT.....	815

APPENDIX A: SQL STATEMENT SYNTAX.....	816
BASIC SQL SYNTAX RULES.....	817
CREATE TABLE.....	821
DROP TABLE	822
INSERT INTO	823
DELETE FROM.....	824
UPDATE.....	825
SELECT.....	826
CALL.....	829
 APPENDIX B: ODBC RESERVED WORDS	 830
 APPENDIX C: ODBC SCALAR AND AGGREGATE (SET) FUNCTIONS	 832
ODBC AGGREGATE FUNCTIONS.....	833
ODBC STRING FUNCTIONS	835
ODBC NUMERIC FUNCTIONS.....	838
ODBC TIME/DATE/INTERVAL FUNCTIONS.....	840
ODBC SYSTEM FUNCTIONS.....	843
EXPLICIT DATA TYPE CONVERSION	844
 APPENDIX D: SQL TOOLS ERROR CODES.....	 845
 APPENDIX E: ODBC ERROR CODES	 849
 APPENDIX F: SQL STATES (ODBC ERROR MESSAGES).....	 851
 APPENDIX G: CONNECTION STRING SYNTAX	 864

APPENDIX H: LOGICAL TRUE AND FALSE	866
APPENDIX I: INTERNET RESOURCES	869
APPENDIX J: USING BITMASKED VALUES.....	871
APPENDIX K: SQLSETENVATTR.....	873
APPENDIX L: UPGRADING FROM SQL TOOLS VERSION 1 TO VERSION 2.....	874
APPENDIX M: USING SQL_TEST.EXE	880
APPENDIX Z: TOPICS NOT COVERED	882

A SIMPLE PROGRAM, STEP BY STEP	883
QUICK AND DIRTY: THE SQL-DUMP PROGRAM	884
SQL-DUMP VISUAL BASIC STEP 1: LINK SQL TOOLS TO YOUR PROJECT	885
SQL-DUMP VISUAL BASIC STEP 2: OPEN THE DATABASE	887
SQL-DUMP VISUAL BASIC STEP 3: TELL THE DATABASE WHICH DATA WE WANT	888
SQL-DUMP VISUAL BASIC STEP 4: RETRIEVE THE DATA.....	889
SQL-DUMP VISUAL BASIC STEP 5: DETECT THE END OF THE DATA.....	890
SQL-DUMP VISUAL BASIC STEP 6: USE THE DATA	892
SQL-DUMP VISUAL BASIC STEP 7: COMPILE AND RUN	894
SQL-DUMP VISUAL BASIC STEP 8: ADD ERROR CHECKING	895
THE POWERBASIC VERSION OF SQL-DUMP.....	897
SQL-DUMP POWERBASIC STEP 1: LINK SQL TOOLS TO YOUR PROGRAM	898
SQL-DUMP POWERBASIC STEP 2: OPEN THE DATABASE	901
SQL-DUMP POWERBASIC STEP 3: TELL THE DATABASE WHICH DATA WE WANT .	903
SQL-DUMP POWERBASIC STEP 4: RETRIEVE THE DATA	904
SQL-DUMP POWERBASIC STEP 5: DETECT THE END OF THE DATA	905
SQL-DUMP POWERBASIC STEP 6: USE THE DATA	907
SQL-DUMP POWERBASIC STEP 7: COMPILE AND RUN	908
SQL-DUMP POWERBASIC STEP 8: ADD ERROR CHECKING	909

License Agreement and Runtime File Distribution Rights

PLEASE READ THIS ENTIRE SECTION.
It describes your legal rights and obligations.

Also see Authorization Codes

SQL Tools Standard License

The SQL Tools Standard License allows you to install the SQL Tools Standard development package (the contents of the SQL Tools Installation File as originally received from Perfect Sync or an authorized distributor) on a single development computer, to use the package for software development, and to distribute the SQL Tools Standard Runtime File(s) with applications which you develop, which require the Runtime File(s) to operate properly, and which add significant functionality to the Runtime File(s).

SQL Tools Pro License

The SQL Tools Pro License allows you to install the SQL Tools Pro development package (the contents of the SQL Tools Installation File as originally received from Perfect Sync or an authorized distributor) on up to four (4) development computers, to use the package for software development, and to distribute the SQL Tools Pro Runtime File(s) with applications which you develop, which require the Runtime File(s) to operate properly, and which add significant functionality to the Runtime File(s).

IMPORTANT NOTE: Each SQL Tools Runtime File is serialized. The unique Authorization Code that is embedded in each copy of the Runtime File(s) will allow Perfect Sync to attribute unauthorized or improper distribution to the original licensee. Attempting to change the embedded Authorization Code is a violation of U.S. and international law, and the Runtime File(s) will self-deactivate or malfunction if tampering is detected. Perfect Sync cannot be held responsible for damage to databases or other files that may be caused by a SQL Tools Runtime File that has been intentionally altered. (See LIMITED WARRANTY below.)

If you have not purchased a SQL Tools Software License from Perfect Sync or an authorized distributor then you are not legally entitled to use the SQL Tools Runtime File(s) for software development or to distribute the SQL Tools Runtime File(s) in any manner whatsoever. You may be violating the law and may be subject to prosecution if you distribute this product or use it for software development. Please refer to the U.S. Copyright Act (and other applicable U.S. and international laws and treaties) for information about your legal obligations regarding the use and distribution of copyrighted and other legally protected works.

Software License

This Software License is an agreement between the Licensee ("you") and the Licensor (Perfect Sync, Inc.). By installing SQL Tools (the "software") on a computer system and/or by using the SQL Tools machine-executable files (the "Runtime Files") for software development, you agree to the following terms:

LICENSE

The software and documentation is protected by United States copyright law and international treaties. It is licensed for use on a single computer system (SQL Tools Standard License) or on four computer systems (SQL Tools Pro License). If this software is installed on a computer network, you must obtain a separate license for each network workstation (or group of four workstations) where the software can be used for software development, regardless of whether or not the software is actually used concurrently on multiple workstations.

DISTRIBUTION

Only individuals or corporations that have purchased a SQL Tools License from Perfect Sync or from an authorized distributor may reproduce and distribute the SQL Tools Runtime File(s), and then only with application(s) that 1) are written by the licensee, 2) require the Runtime File(s) to operate, and 3) add significant functionality to the Runtime File(s). In that case, and provided that your application bears your complete and legal copyright notice or the following notice (in no less than a 10pt font)...

Portions © Copyright 2001 Perfect Sync, Inc

...you may distribute the SQL Tools Runtime File(s) royalty free.

The Perfect Sync Authorization Code which is provided in human-readable form with the SQL Tools installation package is also embedded in the SQL Tools Runtime File(s) and is considered to be part of the Runtime File(s). The Authorization Code may be distributed as part of a machine-readable computer program that meets the requirements above, but it may not be distributed in human-readable form (including source code), disclosed physically, electronically, or verbally to any third party, or distributed in any other form. Disclosure or improper distribution of the Authorization Code would allow the unauthorized use of the SQL Tools Runtime File(s) by others, and is legally equivalent to the unauthorized distribution of the Runtime File(s) themselves.

No other portion of the SQL Tools package, including documentation, header files, and sample program code, may be distributed in any form except by Perfect Sync or an authorized distributor.

LIMITED WARRANTY

Perfect Sync, Inc. warrants that the physical disks (if any) and physical documentation (if any) are free of defects in workmanship and materials for a period of thirty (30) days from the date of purchase. If the disks or documentation are found to be defective within the warranty period, Perfect Sync, Inc. will replace the defective items at no cost to you. The entire liability of this warranty is limited to replacement and shall not, under any circumstances, encompass any other damages.

PERFECT SYNC, INC. SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

GOVERNING LAW

This license and limited warranty shall be construed, interpreted, and governed by the laws of the State of Michigan, in the United States of America, and any action hereunder shall be brought only in Michigan. If any provision is found invalid or unenforceable, the balance of this license and limited warranty shall remain valid and enforceable. Use, duplication, or disclosure by the U.S. Government of the computer software and documentation in this product shall be subject to the restricted rights under DFARS 52.227-7013 applicable to commercial computer software. All rights not specifically granted herein are reserved by Perfect Sync, Inc.

If you have any questions about your rights and responsibilities under this Software License, please contact Perfect Sync, Inc. 8121 Hendrie Blvd., Suite C, Huntington Woods, Michigan (USA) 48070.

You can reach us by electronic mail at support@perfectsync.com, or via fax at (248) 546-4888.

SQL Tools Authorization Codes

This is a topic that all SQL Tools programmers should read and understand thoroughly. If you have any questions about it, please contact support@perfectsync.com.

Unfortunately, not everybody is honest and not everybody obeys the law. That's the reason that our houses have locks on their doors.

We at Perfect Sync have every expectation that *you*, as a SQL Tools licensee, intend to comply with the terms of the SQL Tools License Agreement. But it would be very difficult for you to guarantee that *everybody who uses your program* will be equally honest, especially if your program is widely distributed or if it is available for download from the internet.

Like many programming tools, SQL Tools contains certain security measures that make it more difficult for people to use it illegally. Notice that we said "more difficult", not "impossible". Frankly there is no such thing as 100% security when it comes to protecting a computer program from illegal use. If a "cracker" is determined enough, and has enough time, they can bypass virtually any security system. Just as a determined thief can break into your home, office, or car.

Every SQL Tools Runtime File is serialized. That means that your copies of the Runtime Files contain a unique, embedded key number called an Authorization Code. Nobody else's SQL Tools Runtime Files have the same Authorization Code as your copies of the Runtime Files. This allows Perfect Sync to identify a SQL Tools Runtime File that is being used illegally (i.e. distributed in violation of the SQL Tools License Agreement) and to determine the identity of the original licensee.

In order to use a SQL Tools Runtime File, you must prove to the Runtime File that you know its correct Authorization Code by using the `SQL_Authorize` function. This is done so that when you distribute the SQL Tools Runtime Files *legally*, nobody else will be able to remove them from your program and use them *illegally*. They won't have the correct Authorization Number, and the Runtime Files will not function properly without it.

Protect Your Authorization Code!

Your Authorization Code must be treated as confidential information. If your Authorization Code becomes known to other people, it will allow them to use your copy of the SQL Tools Runtime File(s) illegally. YOU are legally responsible for preventing that from happening!

Using the SQL_Authorize Function

If you don't use the `SQL_Authorize` function at all, the `SQL_Init` and `SQL_Initialize` functions will refuse to work, making it impossible for your program to use SQL Tools in any way.

If you use the `SQL_Authorize` function with the Authorization Code that matches your Runtime File -- using the exact Code that was provided *with* the Runtime File -- it will work normally.

But it's not quite *that* simple...

It would be relatively easy for somebody to write a program that used the `SQL_Authorize` function to test all of the possible Authorization Codes one by one, until it found one that worked with your Runtime File. The `SQL_Authorize` function returns `SUCCESS` when it accepts an Authorization Code, so all it would take would be a simple "loop" program that stopped when the correct Code was found.

So the `SQL_Authorize` function also returns `SUCCESS` when certain other codes are used.

There are approximately 4.2 billion possible Authorization Codes. Of those, only one is the correct Code for your Runtime File, but about 64,000 "Dummy Codes" will also cause the `SQL_Authorize` function to return `SUCCESS`. This makes it much more difficult to use the `SQL_Authorize` function to determine the correct Authorization Code for a given Runtime File.

THIS IS A VERY IMPORTANT POINT: If one of the 64,000 Dummy Codes is used instead of the correct code, the `SQL_Authorize` function will return `SUCCESS`, the `SQL_Init` and `SQL_Initialize` functions will work properly, and *all other SQL Tools functions will appear to work properly*. But in reality, the SQL Tools Runtime File will purposely malfunction. At random intervals, many different SQL Tools functions will produce results that are completely or partially incorrect. For example, every so often a SQL Statement like `SELECT` might not return all of the rows that it should. Or an `UPDATE` statement might return `SUCCESS` when it actually -- purposely -- failed. Or certain values might be set to zero. This will make the SQL Tools Runtime Files seem to work properly *most* of the time, but they will be unreliable.

Don't worry, the SQL Tools Runtime Files have been tested *extremely* thoroughly to make sure that no random errors will be produced when the *correct* Authorization Code is used.

And we have taken great care to make sure that a simple typo will not result in a SQL Tools program that malfunctions unexpectedly. Among other things, the code numbers have been chosen so that accidentally mis-typing *any single digit* of a valid Authorization Code will *never* produce a Dummy Code that `SQL_Authorize` will accept. If you mis-type two of the eight digits of a valid Code there is less than a one-in-10,000 chance that you will accidentally type a Dummy code that `SQL_Authorize` will accept. If you mis-type three out of eight digits... well, you should probably take typing lessons before attempting to use SQL Tools.

With just a little bit of care when you type the Authorization Code into your program, you can rest assured that SQL Tools will work properly from that point forward.

IMPORTANT NOTE: Be sure to test the return value of the `SQL_Authorize` function to make sure that it is `SUCCESS`. This will virtually guarantee that you typed the Authorization Code correctly, and that the SQL Tools Runtime File will work properly.

Please see *Four Critical Steps For Every SQL Tools Program* and `SQL_Authorize` for more information.

Troubleshooting Your Programs

Nobody's perfect. Anybody who writes computer programs is bound to make a few mistakes. Finding and correcting those mistakes -- the process of troubleshooting your program -- can be as time-consuming as writing the original program. Fortunately, SQL Tools provides several very powerful features that can make troubleshooting much easier and faster.

Surprisingly, the most common mistake that people seem to make is not even *checking* for errors! Virtually all SQL Tools functions provide a way for you to determine whether or not they worked correctly. You may need to check a function's return value to make sure that it is `SUCCESS`, or you may need to use one of the many `SQL_Error` functions that are provided. The use of these functions is covered in the section of this document that is titled Error Handling In SQL Tools Programs.

For example, many of the "problem" programs that are submitted to our Technical Support department look something like this:

```
lResult& = SQL_Init
lResult& = SQL_OpenDB("MyData.DSN")
lResult& = SQL_Stmt("SELECT * FROM MYTABLE")
DO
    lResult& = SQL_Fetch(NEXT_ROW)
    '(etc.)
```

The problem with that code is that none of the result values are being *checked*! That may be acceptable once your program is working properly, but during development and debugging your program should look more like this:

```
lResult& = SQL_OpenDB("MyData.DSN")
IF NOT SQL_Okay(lResult&) THEN
    MSGBOX "ERROR A:" + FORMAT$(lResult&)
    EXIT FUNCTION
END IF

lResult& = SQL_Stmt("SELECT * FROM MYTABLE")
IF NOT SQL_Okay(lResult&) THEN
    MSGBOX "ERROR B:" + FORMAT$(lResult&)
    EXIT FUNCTION
END IF
```

...and so on. It's very important to *check* those return values! (Actually we recommend that you leave the debugging code in place whenever possible, even when you finish a project. You never know when you will need to fix a well-hidden bug.)

SQL Tools also has the built-in ability to *automatically* display a message box whenever a runtime error is detected. It can't automatically `EXIT FUNCTION` like the code above does, and it is *only* intended for debugging purposes, but it can be a very powerful tool when you are not sure where your program is failing. For more information about this, see Miscellaneous Error Handling Techniques.

Another often-overlooked troubleshooting technique is the Trace File. SQL Tools has the ability to create a text file that can show you exactly where an error is taking place, and often, what is causing it. For more information, see the SQL Tools Trace Mode.

The great majority of the questions that are received by Perfect Sync Technical Support can

be answered almost instantly if you use the troubleshooting tools that SQL Tools provides. In fact, when we respond to most questions, we usually have to ask "*What does the SQL_ErrorQuickAll function tell you?*" or "*What does the Trace File tell you?*" When we get the answers to those questions, *then* we can begin analyzing the problem.

So if you check those things *before* contacting Technical Support, you will save yourself (and us!) a lot of time!

Getting Technical Help

To save time, please read the page titled Troubleshooting Your Programs, which contains general troubleshooting guidelines, before you contact Perfect Sync.

We have worked very hard to make sure that this document contains everything that you'll need to know about using SQL Tools. Before contacting Perfect Sync for help, please search this document for words and phrases that might be related to your question. (For example, when this document is presented as a Help File, use the Windows Help Contents, Index, and Find features.) Almost every SQL Tools topic is covered twice in this document: once in the User's Guide and once in the Reference Guide.

If you don't find an answer in this document, Perfect Sync provides free Technical Support via electronic mail to all developers who license SQL Tools. Please send all pertinent technical questions to support@perfectsync.com. Be sure to include your SQL Tools Serial Number, an email address where we can send our response, and a detailed description of the problem. If possible please include sample source code and Trace Files.

If you contact us and it turns out that the answer to your question *is* given in this document, that is probably the answer that you will receive: a polite suggestion that you read a particular section of the Help File. After all, an informal email message from our Tech Support department wouldn't be able to explain a topic nearly as thoroughly as this document. If you feel that the SQL Tools documentation does not explain a topic well enough, please cut and paste the unclear help text into your message, and ask a specific question. We'll be glad to try to clarify!

If the answer to your question is not covered in this document but does fall within the bounds that we have established, we will do our best to **1)** answer your question quickly and completely via email and **2)** add the answer to the next release of this document, so that others can benefit.

If your question is outside the bounds that we have established, we reserve the right to decline to provide an answer. For example, if a SQL statement does not produce the results that you think it should, it is probably safe to assume that SQL Tools is functioning correctly and you are not using the SQL language correctly. (SQL Tools simply submits SQL statements to the ODBC driver without modifying them, so it is virtually impossible for SQL Tools to interfere with the proper execution of a SQL statement.) We will be *very* pleased to confirm that you are using the `SQL_Stmt` function correctly, but that is where our responsibility ends: providing a reliable function and an accurate explanation of what it does.

Another general area worth mentioning is "ODBC Error Messages". Each ODBC Driver provides a set of Error Message that are *very specific* that that driver and database. SQL Tools supports well over 50 different ODBC Drivers, not to mention the many different versions of each driver that are available. If we are not familiar with a particular Error Message that your program is generating, we will direct you toward the appropriate database-specific documentation.

In the end, a SQL Tools function either works properly or it doesn't. If it *doesn't* work, we will endeavor to provide a bug fix for the SQL Tools Runtime File(s). If it *does* work, *you* are responsible for investigating the meaning of database-specific error messages or figuring out why a particular SQL statement (or other operation) does not give you the results that you expect.

As the president of PowerBASIC, Inc. is fond of saying, "When you buy a hammer it doesn't

come with instructions for building a house".

Don't get us wrong: we will be *very* glad to help you learn to use our "toolkit"! But we can't possibly provide free training in the rest of the skills that you will need to complete a project, whether it's a birdhouse or a (data) warehouse.

Perfect Sync reserves the right, at our sole discretion, to charge hourly fees for technical support that does not fall within the bounds of what we consider to be normal and reasonable. (No fees will be charged without the prior consent of the SQL Tools Licensee.)

Questions about the licensing and distribution of the SQL Tools Runtime Files and other components should be directed to sales@perfectsync.com.

Visual Basic support is available from a wide variety of internet and phone-based sources.

For PowerBASIC PB/CC and PB/DLL questions, please contact support@powerbasic.com. PowerBASIC also sponsors several excellent "peer support forums" on their "Web BBS" site at <http://www.powerbasic.com/support/forums/Ultimate.cgi>. They also provide many different support files via their ftp site.

And by the way, the Internet is an *excellent* source for general SQL and ODBC support. See Appendix I for more information.

Frequently Asked Questions

This section of this document is intended to answer basic questions like "What Is SQL Tools?" and "How Complete Is SQL Tools?".

For more technical questions and answers, you should refer to the User's Guide (a detailed, narrative-style explanation of SQL Tools) and the Reference Guide (which contains detailed descriptions of every SQL Tools function).

What is SQL Tools?

What's the difference between SQL Tools Standard and Pro?

What will SQL Tools do for my programs?

What will I need to use SQL Tools?

What's the difference between SQL and ODBC?

Can I use SQL Tools to write "universal" or "interoperative" programs?

Do all SQL Tools features work with all database?

How complete is SQL Tools?

Exactly which ODBC features are not supported?

What SQL Tools IS and ISN'T

SQL Tools is a package of developer's tools that allow programmers to add high-performance, low-overhead SQL database support to their 32-bit Windows programs. It was specifically designed to be used with Visual Basic and PowerBASIC, but it can be used with any 32-bit computer language that can use functions in standard-format 32-bit DLLs.

SQL Tools is NOT a database-design program like Microsoft Access or Oracle's SQL*Plus. In other words, SQL Tools does NOT provide a GUI environment for building databases from scratch. (It would theoretically be possible, but extremely time-consuming, to use SQL Tools and a programming language to create a full-featured database-design program like Access. But when inexpensive, highly sophisticated database design tools are readily available, why do it?)

Also, SQL Tools does NOT provide a "direct link" to a database in the same way (for example) that the BASIC language's `OPEN` statement provides direct access to a disk file. SQL Tools requires the use of ODBC drivers to provide the link between your program and the database.

What's the Difference Between SQL Tools Standard and Pro?

Basically, the SQL Tools Standard Runtime Files contain all of the functions that you will need to create programs that can read and modify SQL databases. The entire text-based, single-statement SQL language is supported; SQL Tools does not impose *any* limitations on the SQL language.

The **Standard** Runtime Files allow a program to have one or two databases open at the same time, and each database can use one or two concurrent SQL statements. (Within certain restrictions, three concurrent statements can be used by each database.)

Several basic Info functions are provided, such as API Info, Database Info, Database Attributes, Table Info, Table Column Info, and Result Column Info. (The Database Attribute function alone provides well over 200 different values.)

A powerful set of Error Handling functions is also provided, including two Trace Modes and an "ignore specified error" system.

PLEASE NOTE: All of the SQL Tools functions that are described in this document are subject to the limitations of the ODBC driver that you choose to use. SQL Tools cannot support features that are not supported by your ODBC driver. *Most late-version ODBC drivers provide most of the functions that are described in this document, but Perfect Sync cannot guarantee that every feature listed here will be available to every program. Think of it this way... Your word processor may support color printing, but if your Printer Driver doesn't support color then all you will see is black lettering on white paper. The same is true for your SQL Tools programs and your ODBC driver. SQL Tools can't do things that your driver doesn't support.*

The **Pro** Runtime Files provide all of the Standard Runtime File(s) functionality, plus they allow the use of up to 256 databases by the same program, each having up to 256 concurrent SQL statements.

The Pro Runtime Files also include a large number of advanced features, including:

- Multithreaded Operation (see note below)
- Batched SQL Statements
- Multirow (Block) Cursors
- Bound Statement Parameters, including Long values and arrays
- Stored Procedures
- Manual Commit/Rollback of Transactions
- Relative Fetches
- Bookmarks
- Bulk Operations
- Positioned Operations
- Named Cursors
- Connection Pooling
- Low level SQL/ODBC Diagnostics

PLEASE NOTE: Some SQL Tools functions may be unavailable because of limitations in your programming language. For example, Visual Basic (up to version 6.0) does not support true multi-threading, so SQL Tools cannot be used in that mode. This is a limitation of Visual Basic, not SQL Tools or your ODBC driver. We are not aware of any other limitations that are imposed by Visual Basic or by other languages, but we cannot (of course) guarantee that they do not exist.

Many different Info (catalog) functions are also included in the Pro Runtime Files, including Driver Info, Datasource Info, Data Type Info, Table Statistics, Table and Column Privilege Info, Unique Column Info, Primary Column Info, Autocolumn Info, Index Info, Foreign Key Info, Stored Procedure Info, and others.

The Pro Runtime Files also allows you to access low level functions that require ODBC Handles and memory pointers to SQL Tools data buffers.

The SQL Tools Pro Runtime Files provide virtually 100% of the functionality that is included in the ODBC 3.5, Level 2 specification.

For a very brief list of ODBC 3.5 features that are not supported by the SQL Tools Pro Runtime Files, see Unsupported Features.

For a function-by-function breakdown of the Standard and Pro Runtime Files, see Functional Families.

What Will SQL Tools Do For My Programs?

SQL Tools will allow your 32-bit Windows programs to use the worldwide-standard Structured Query Language (SQL) to read-from and write-to databases that have been created by other programs. Within certain limits (imposed by the creators of the various types of databases that SQL Tools supports) you can also create new databases.

SQL Tools Pro will also enable your programs to access many different types of information about a database, such as Table Names, Index Names, and literally hundreds of other "Catalog Info" functions.

What Will I Need To Use SQL Tools?

You'll need:

1) A computer with a 32-bit Microsoft Windows operating system, such as...

- Windows 95
- Windows 98
- Windows ME
- Windows NT4
- Windows 2000
- Windows XP

2) A 32-bit programming language such as...

- Microsoft Visual Basic version 4, 5, or 6
- PowerBASIC's PB/DLL compiler
- PowerBASIC's PB/CC "console compiler"

If you are proficient in converting BASIC-language "declaration" syntax into other languages, the SQL Tools Version 2 Runtime Files can also be used by:

- Microsoft Visual C++
- Microsoft Visual Fortran
- Borland's C++ Builder
- Borland's Delphi
- Any other 32-bit language that can call functions in standard Win32 DLLs

3) The SQL Tools development package

4) The ODBC Driver(s) for the database(s) that you want to use. Drivers for more than 50 popular databases are available from various sources, including the free MDAC package from microsoft.com.

If you want to *design* a database in a "visual" environment (as opposed to working with an existing database using a program that you write) you will also need the appropriate database management software, such as Microsoft Access, Corel Paradox, or Oracle SQL*Plus.

We also strongly suggest that you acquire reference materials related to SQL programming. While this document contains a *lot* of information, it could not possibly be all-inclusive. There are literally hundreds of SQL books available. As of this writing, <http://amazon.com> lists over 600 books with the word SQL or ODBC in the title.

In particular, we recommend that you acquire books related to 1) using SQL statement syntax that is specific to the type of database that you are using, and 2) "good practice" in database design. These are lengthy, complex topics that are well beyond the scope of this document.

There are also several Internet newsgroups dedicated to SQL and ODBC, and using the <http://www.altavista.com> search engine to find "SQL" currently yields over 8 million web pages. (See Appendix I: Internet Resources.)

Finally, if you are going to use the most advanced features that SQL Tools provides, we recommend that you download the (free) ODBC Software Developers Kit from Microsoft. The ODBC SDK Help File, when printed, is well over 1350 pages long, and it is a rich source of

low-level details. It would not be possible (or legal, from a copyright standpoint) for Perfect Sync to include that level of detail in this document.

What's the Difference Between SQL and ODBC?

SQL is a standard language for accessing databases.

ODBC is an even broader set of standards that allow programs to access many different types of databases with standard techniques. ODBC defines not only the language, but how databases should be opened and closed, standard error messages, and many, many other details.

SQL Tools is capable of "talking to" any ODBC-compliant database: Access, SQL Server, Paradox, Excel, dBASE, FoxPro, Oracle, Lotus Notes... or any other type of database for which an "ODBC Driver" is available. Even old-fashioned Flat Text Files can be used!

You can read SQL and ODBC later in this document for more details, but we recommend that you read the introductory section titled A SQL Tools Primer first.

Can I Use SQL Tools to Write "Universal" Programs?

Theoretically, yes. But some ODBC-compliant databases are extremely limited, so writing a universal or "interoperable" application that would work with *any* ODBC-compliant database would require you to write a "least common denominator" program.

For example, the absolute-bare-minimum ODBC specification requires that an ODBC-compliant database support only one type of data. Bare-minimum databases have their choice of supporting *either* a fixed-length or variable-length string, but they are not required to support both. Of course it would be possible to use that one data type to "simulate" numeric variables, TYPE structures, and strings of different types, but it probably wouldn't be worth the effort.

Fortunately, most modern ODBC-compliant databases support at least a dozen different data types, from single bits to huge BLOBs (Binary Large Objects, which can store binary images like sounds, pictures, or even entire programs).

Instead of writing truly "universal" programs, most programmers choose to write SQL Tools programs that require a certain minimum ODBC functionality, such as that provided by Microsoft Access 97.

For more information about the levels of functionality that different databases provide, see Compliance Issues.

Do All SQL Tools Features Work With All Databases?

Absolutely not! SQL Tools can only support features that are supported by a given ODBC driver. It does not, for example, simulate Oracle features for Access databases.

If you choose to, you could use SQL Tools and your programming language to simulate those features -- in fact that is a common programming practice -- but SQL Tools simply provide the "raw" functionality that makes that possible.

How Complete is SQL Tools?

SQL Tools supports virtually all of the major features in Microsoft ODBC 3.5, Level 2, which (as of this writing) is the state of the art for ODBC.

SQL Tools supports 100% of the SQL statement syntax that is supported by the ODBC driver that you use. Basically, SQL Tools allows you to access *any* ODBC-compatible database for which you have an ODBC driver, and it allows you to use virtually *all* of the functionality that is provided by the driver. (See Which ODBC Features are Not Supported?)

The SQL Tools Standard Runtime Files support up to two (2) simultaneous database connections, with up to two (2) concurrent SQL statements on each connection. (Within certain limitations, three concurrent statements can be used.) Each database can have up to 999 tables, each table and/or result set can have up to 999 columns, and block cursors up to 256 rows are supported.

The SQL Tools Pro Runtime Files support up to 256 simultaneous database connections, with up to 256 concurrent SQL statements on each connection. Each database can have up to 9,999 tables, each table and/or result set can have up to 9,999 columns, and block cursors up to 1,024 rows are supported.

Exactly Which ODBC Features are Not Supported?

SQL Tools Version 2 *does* support Asynchronous Execution of SQL Statements, but it only supports one of the two available methods. Here is what the Microsoft ODBC Software Developer Kit says about Asynchronous Execution:

"In general, applications should execute functions asynchronously only on single-threaded operating systems. On multithread operating systems, applications should execute functions on separate threads, rather than executing them asynchronously on the same thread. No functionality is lost if drivers that operate only on multithread operating systems do not support asynchronous execution."

The Windows operating system is capable of multithreading -- as are PowerBASIC and most other 32-bit Windows programming languages -- so SQL Tools does not support ODBC-style asynchronous execution. PowerBASIC and most other languages can create threads that allow SQL statements to be executed asynchronously.

Microsoft Visual Basic does not support true multi-threading, but SQL Tools Pro includes functions that allow SQL Tools *itself* to create threads which can execute asynchronous SQL statements in VB programs.

To be clear, SQL Tools Pro does not support ODBC-style asynchronous execution, but it does support thread-based asynchronous execution, exactly as recommended by Microsoft.

Descriptors

Here is what the Microsoft ODBC Software Developer Kit says about Descriptors:

"An application calling ODBC functions need not concern itself with descriptors. No database operation requires that the application gain direct access to descriptors. However, for some applications, gaining direct access to descriptors streamlines many operations. For example, direct access to descriptors provides a way to rebind column data that may be more efficient than calling"... [the SQL_ManualBindCol function]... "again."

SQL Tools supports virtually 100% of the ODBC functions that can be performed without descriptors. If you feel that your program would benefit from using them, we suggest that you consult the Microsoft ODBC Software Developer Kit for more information. SQL Tools should be completely compatible with any descriptor-API-based functions that you write, but (of course) it is not possible for us to guarantee compatibility.

(By the way, "rebinding column data" was an interesting choice for Microsoft to use as an example, because a *very* efficient alternate method -- which does not use descriptors -- is provided via "statement attributes".)

We believe that only the most complex ODBC programs would *require* the use of descriptors, and very few programs would benefit in any way from using them.

Deprecated Functions

As the ODBC specification has grown from version 1.0 to 2.0 to 3.0 to 3.5, a few functions have been "Retired In Place" along the way. An "R.I.P." or "deprecated" ODBC function is

one that has been replaced by a better, more powerful function, but is still available for older applications to use.

SQL Tools does not support deprecated functions.

Duplicate Functions

In a *very* few cases where two or more ODBC functions can be used to perform the same operation, SQL Tools does not support all of the different methods. Generally speaking, SQL Tools supports the most sophisticated method that is available.

Ready to Write Programs? Start Here!

Whether you're an experienced SQL guru or a novice, there are a few things that you *really* need to know about before writing your first SQL Tools program. We'll try to be as brief as possible, and to follow up later with more detailed information, but we strongly suggest that you read these brief sections of this document:

Conventions used in this Document

Variable Naming Conventions

Installation of SQL Tools

Installation of ODBC Drivers

Terminology Differences

Compliance Issues

Two Of Everything

Four of Many Things

Eight or More of Some Things

Four Critical Steps For Every SQL Tools Program

Conventions Used In This Document

Most of the text in this document will appear in a non-bold Arial font.

Important Warnings are shown in **bold red**. Less urgent warnings are shown in **bold dark red**.

If this document is presented in electronic (Help File) form, clickable links look like This. (The highlight color is determined by your Windows settings.)

SQL Tools function names, source code, numeric values, string values, and BASIC keywords are shown in the `Courier New` font. BASIC source code comments (remarks) are shown in `Green` because that is the way most source-code editors display them.

BASIC keywords and SQL Tools functions appear in `Mixed Case` letters with the first letter capitalized, such as `If` and `Then`. To help distinguish them from BASIC keywords, all SQL Tools functions start with the prefix `SQL_`, as in `SQL_OpenDatabase`. If this document is presented in electronic (Help File) form, SQL Tools function names will often be highlighted like `THIS` to indicate that you can click on the highlighted word to jump to the function's Reference Guide entry.

Variable names also appear in mixed-case letters, but with the first letter in lower case, like `lResult&` and `sParam$`. For information about the variable naming convention that is used, see the next page.

Constants -- words that represent *fixed* numeric values -- appear in `UPPER_CASE` letters, like `SUCCESS`, `SQL_STMT_IMMEDIATE`, and `NEXT_ROW`. (PowerBASIC users will need to add the `%` prefix to all constants in order to create PowerBASIC "equates", such as `%SUCCESS`, `%SQL_STMT_IMMEDIATE`, and `%NEXT_ROW`. Most example source code is shown without the prefix, to keep it consistent with the Microsoft ODBC, Win32, MSDN, and Visual Basic documentation.)

SQL statements like *`SELECT * FROM MYTABLE`* and individual elements of the SQL statement syntax like *`SELECT`* are shown in blue italics, to indicate that you must use SQL syntax that is compatible with the ODBC driver that you are using. (It should also be helpful in keeping the BASIC keyword `SELECT` separate from the SQL *`SELECT`* keyword.) You should think of the SQL syntax as a language that is separate from BASIC, C, or Delphi, so the blue lettering is used as a visual clue to indicate a different kind of "source code".

While Microsoft prefers that "SQL" be pronounced "Ess Cue Ell", people that actually use SQL in their work usually pronounce it like the word "sequel". This document uses the later, more popular and casual pronunciation. This is only significant, and is only mentioned here, because this document will refer to things like "a SQL database" while Microsoft documentation will say "*an* SQL database".

Variable Naming Conventions

Some programmers prefer to use explicit "type identifiers" in their variable names. An example of this would be the addition of an ampersand (&) to the end of a variable name to indicate that a variable such as `Something&` is a Long Integer. Other programmers prefer a convention called "Hungarian notation" where something is added to the *beginning* of the variable name. The Hungarian notation version of `Something&` would be `lSomething`, with the lower-case L prefix standing for Long. (Hungarian notations vary. For example, some people use `i` for Integer, others use `n`.)

For maximum readability by both groups of people, this document uses both prefixes *and* suffixes, so every variable you see will look like `lSomething&`. The following prefixes and suffixes are used in this document:

Signed Integer Variables

<code>iSomething%</code>	INTEGER
<code>lSomething&</code>	LONG Integer

Unsigned Integer Variables

<code>wSomething??</code>	WORD *
<code>dwSomething???</code>	DWORD (Double Word) *

Floating Point Variables

<code>spSomething!</code>	Single precision floating point
<code>dpSomething#</code>	Double precision floating point

String Variables

<code>sSomething\$</code>	Dynamic (variable-length) String
<code>lpzSomething</code>	Fixed-length or ASCIIZ string (no suffix defined)

* indicates a variable type that is not available in Visual Basic. See Signed and Unsigned Integers for details.

See BASIC Data Types for more information about the individual data types.

Signed and Unsigned Integers

The Windows operating system and the ODBC subsystem support several different data types that are not supported by all programming languages.

*If you are a PowerBASIC programmer you may want to skip this section because all of the data types that SQL Tools supports are also supported by PowerBASIC. Visual Basic and other programmers should definitely read this section, and decide which of the information applies to you. At the very least, you should read the **IMPORTANT NOTE** near the end of this section.*

LONG Integers and DWORD Integers

LONG Integers are supported by virtually all 32-bit programming languages, including Visual Basic. A LONG Integer is a signed integer variable that requires four (4) bytes of memory. A LONG can store whole-number (i.e. non-fractional) values between $-2,147,483,648$ and $+2,147,483,647$.

It is also possible to use four bytes of memory to store a Double Word or "DWORD" value. DWORD variables can store whole-number values from zero (0) to $+4,294,967,295$. That's exactly the same *number* of values as a LONG integer, it's just that LONGs take half of the range and use it to specify negative numbers. That's the basic difference between a "signed" and "unsigned" value.

As you probably know, "four bytes of memory" and "32 bits" are exactly the same thing. Different patterns of 1 and 0 (On and Off) represent different numbers. One of the 32 bits -- also called the Most Significant Bit or the Sign Bit -- can be interpreted as meaning either "this is a negative number" or "this is a number that is larger than $+2,147,483,647$ ". LONGs interpret the last bit one way, and DWORDs interpret it the other way. The other 31 bits are 100% identical in LONGs and DWORDs. So unless a value is negative or greater than $+2,147,483,647$, there is no difference at all between a LONG and a DWORD. LONGs and DWORDs "overlap" in the range from zero (0) to $+2,147,483,647$.

Many different Windows API and ODBC functions actually do return DWORD values. For example, all Windows and ODBC "handle" values are defined as DWORDs. So the fact that Visual Basic and certain other programming languages do not support DWORDs can be inconvenient. Fortunately, it is almost always possible to substitute a LONG variable for a DWORD variable. That is possible because both LONGs and DWORDs are integers that use four bytes of memory.

When you pass a LONG or DWORD variable to a function (such as a SQL Tools function) all the function really sees is "four bytes of memory". It has no way of knowing whether your program will interpret those four bytes as a LONG or a DWORD. So it will read a numeric value from those four bytes, or it will place a numeric value into those four bytes, and *it is up to your program* to determine how the value should be interpreted. In most cases it won't make any difference at all.

This document, and the names of several SQL Tools functions, use the abbreviations `UInt` and `SInt` for Unsigned Integer and Signed Integer. Visual Basic (and certain other languages) can use only Signed Integers (LONGs) so you may need to "offset" the Unsigned values in order to interpret them properly.

For example, let's say that you have a database table that contains the descriptions of several

computer workstations. In addition to having columns for the CPU speed, the amount of memory, and so on, you would probably want to include a column for the hard-drive size. If that column was defined as a `SQL_INTEGER` it could be used for either a LONG value or a DWORD value, depending on how the database designer decided to use it. If it was used as a LONG integer, you could store numbers up to `+2,147,483,647`, which would correspond to a hard drive size of `2.1` gigabytes. But if you tried to enter a record for a `3.0` gigabyte drive, it would be stored in the table as a *negative* number, and that would probably cause unexpected results.

Again, this is caused by a limitation in the way Visual Basic and certain other languages interpret 4-byte integer values. It is not a limitation of Windows, ODBC, or SQL Tools.

The largest value that can be stored in a LONG integer is `+2,147,483,647`. If you try to add one to that value and store `+2,147,483,648` then the value will appear to "roll over" to *negative* `2,147,483,648`. If you add one more (`+2,147,483,649`) then you will add one to that *negative* value, resulting in `-2,147,483,647`. If you keep adding one, the resulting negative value will get closer and closer to zero. When you reach `+4,294,967,295` (the largest value that a DWORD can hold) the corresponding number will be *negative one*.

You can look at the relationship between LONGs and DWORDs this way, using pseudo-code:

```
IF LONG => 0 THEN
    'values => zero are identical
    DWORD = LONG
ELSE
    'values < zero represent large positive values
    DWORD = LONG + 4,294,967,296
END IF
```

...and...

```
IF DWORD <= 2,147,483,647 THEN
    'values <= 2.1 gig are identical
    LONG = DWORD
ELSE
    'values > 2.1 gig represent negative numbers
    LONG = DWORD - 4,294,967,296
END IF
```

DWORD Bitmask Values

Fortunately, most DWORD variables are not used to store large numeric values. A much more common use of a DWORD variable is a "bitmask" value. For example, the `SQL_DBInfoUInt` function returns an Unsigned Integer value. (That's what the `UInt` in the function name means.) Many of the values that `SQL_DBInfoUInt` returns are not really "numbers", they are "bitmasks" where each *bit* has a particular meaning. So saying that the overall value is "positive or negative" has very little meaning. (See Using Bitmasked Values for more information.)

When an unsigned integer DWORD value is used as a bitmask, substituting a LONG integer variable will make no difference whatsoever. Since your program will be looking at the "bit pattern" and not the actual value, LONGs and DWORDs can be considered to be 100% identical. Your program can use LONGs and DWORDs interchangeably without worrying about side effects like "unexpected negative values".

Other Unsupported Data Types

Visual Basic programmers have the same problem with INTEGER and WORD variables that they do with LONG and DWORD variables, because Visual Basic does not support the WORD data type. Virtually all of the information above applies equally to INTEGER and WORD variables, except that they are two-byte integers, not four-byte integers. INTEGERS have a range of $-32,768$ to $+32,767$ and WORDs have a range of zero (0) to $+65,535$.

Installing SQL Tools

IMPORTANT INFORMATION!

You must perform these steps before
using Graphics Tools for the first time!

SQL Tools is provided as a single-file "Installation Program" which takes care of unpacking all of the necessary disk files. Simply execute the installation program, and it will walk you through all of the various installation choices that you will need to make, such as the name of the directory where SQL Tools will be installed.

The default directory is `\SQLTOOLS`, and the rest of this section will assume that you used the default. If you choose a different directory, simply substitute that directory's name wherever you see `\SQLTOOLS` below. *We do recommend that you use the default `\SQLTOOLS\` directory because the sample programs that are provided with SQL Tools are hard-coded for that directory name. They can be changed easily enough, but if you use `\SQLTOOLS\` the sample programs can usually be compiled and run without any modifications.*

After you have installed SQL Tools, we suggest that you use the Windows Explorer program to examine the files that were placed in the `\SQLTOOLS\` directory. A variety of sample programs, blank databases, and other files are provided. Some of them are provided as self-extracting files that are *not* automatically unpacked. You should also look at your system's Start Menu, where you will find several SQL Tools components listed under the heading "Perfect Sync Development Tools".

In order for your programs to be able to *use* the SQL Tools DLL, they will need to be able to find it. In most cases it will be necessary for you to place a second copy of the `SQLT_STD.DLL` or `SQLT_PRO.DLL` file (depending on the version of SQL Tools that you are using) somewhere on your computer's hard drive. We recommend that you leave the original copy in the `\SQLTOOLS\` directory, to serve as a backup. Copy the file, don't move it.

The *ideal* location for the `SQLT_STD.DLL` or `SQLT_PRO.DLL` file is *the same directory as the executable program that you are developing*, but keep in mind that you may be developing database programs in more than one directory. If that is the case, you may choose to place the SQL Tools DLL in your Windows System Directory or in another directory that is in your System Path. On Windows 95/98/ME systems, the System directory is usually `C:\Windows\System`. On NT4/2000/XP systems it is usually `C:\WinNT\System32`. (On some versions of Windows, the System directory is hidden by default so you may need to change your Windows Explorer settings in order to see it. Typically, you can change Tools > Folder Options > View to "Show Hidden Folders".)

If you write a program using SQL Tools, and when you run it you see a Windows message box that says something like "The dynamic link library `SQLT_STD.DLL` could not be found in the specified path" it means that Windows was unable to link SQL Tools to your program's EXE. This almost *always* means that the `SQLT_STD.DLL` or `SQLT_PRO.DLL` file needs to be copied to a location where your program can find it.

That's all there is to it! Once you have copied the SQL Tools DLL to a place where your program can find it, SQL Tools is installed and ready for use.

BUT...

Before you can begin using SQL Tools to write programs, you may need to install one or more ODBC drivers.

Installing ODBC Drivers

A driver is a special kind of software program that becomes part of the Windows operating system and allows other programs to access a particular capability. For example, a certain Printer Driver might allow Windows to use a certain brand of printer, and a certain Mouse Driver might allow Windows to use a certain brand of mouse or trackball.

An ODBC Driver is a piece of software that allows your computer to access certain types of databases almost as if they were "devices" like printers and mice. Just as every major printer manufacturer has its own drivers, every major type of database has its own drivers. So there is a Microsoft Access ODBC driver, an Oracle ODBC driver, a dBASE ODBC driver, and so on.

ODBC (Open Database Connectivity) is a Microsoft standard that allows programs to access different database formats through a standard interface. It is possible for an ODBC-compliant program (like SQL Tools) to access virtually any ODBC-compliant database. An ODBC driver is the software that makes that possible.

As a software developer, you may need to address two different issues:

- 1) ODBC drivers that you can install and use on your own computer, and
- 2) ODBC drivers that you can legally distribute with your applications.

It is not enough for your development computer to have an ODBC driver. In order for a SQL Tools application on *any* computer to access an ODBC database, you must first install the appropriate ODBC driver on *that* computer. ODBC drivers are available for virtually every major database format, but not *all* computers are pre-configured with ODBC drivers.

There are four basic ways to obtain and install ODBC drivers...

The Windows Installation CD

All versions of Microsoft Windows 98 SE, Windows NT4, Windows ME, Windows 2000, and Windows XP include a standard package of ODBC drivers called "MDAC", or Microsoft Data Access Components. However, depending on the Windows version, the ODBC drivers may or may not be part of the default installation. If they are not already installed, you will need to re-insert the Windows installation CD and install the ODBC drivers that are provided on the disk.

It is important to note that the versions of MDAC/ODBC that are supplied with the various versions of Windows are not all the same, and that they contain different drivers and driver *versions*. You should research the driver(s) that you want to use and make sure that they are included with all versions of Windows. Microsoft Access drivers are fairly standard, for example, but the "Jet" drivers have been dropped from more recent versions of MDAC.

Note also that Windows 95 and Windows 98 "classic" do not include ODBC drivers on the Windows installation CDs. Fortunately those versions of Windows are increasingly uncommon, because they are no longer available for sale from Microsoft. ODBC drivers are, however, *compatible* with Windows 95 and 98 systems and can be installed on those systems using the other methods described below.

The Internet

Microsoft and many other vendors provide current version of their ODBC drivers on their internet sites. There is sometimes a fee that is charged for drivers, but under certain circumstances you may be able to download, install, and redistribute these drivers at no cost. See Installing ODBC Drivers from the Microsoft Internet Site.

The Database Product's CD

The necessary ODBC drivers are almost always included *with* database products such as Microsoft Access, and they can be installed by using the product's installation/update CD. See Installing ODBC Drivers from a Database Product.

Software Installation Programs

Many programmers use "installation programs" such as InstallShield to distribute their applications, and many of those programs have the ability to automatically install ODBC drivers when your application is installed. Unfortunately, as of this writing, the version of InstallShield that comes with Visual Studio does not have that ability. (We used InstallShield as a well-known example of an installation program, not as an example of a program that can install ODBC drivers.) We have been told that Installshield *Express*, Wise Installbuilder, and several other programs can install ODBC drivers, but we do not have first-hand experience with those products.

Installing ODBC Drivers from the Microsoft Internet Site

Microsoft provides ODBC drivers for many different Microsoft and non-Microsoft databases, including Access, SQL Server, Excel, FoxPro, dBASE, Paradox, and Oracle, plus the Microsoft Text Driver for flat files.

The Microsoft package is called MDAC, which stands for Microsoft Data Access Components. At the time of this writing, the name of the downloadable ODBC driver file was MDAC_TYP.EXE, and it could be downloaded from <http://msdn.microsoft.com/library/default.asp?url=/downloads/list/dataaccess.asp>. Please note that the file name and location are subject to change, so this information may be out of date. If you have trouble finding it, we suggest that you visit <http://microsoft.com>, and use their Search feature to find MDAC_TYP.EXE or simply MDAC.

Frankly, MDAC is notorious for being difficult to manage, but nearly all of the problems are related to the ADO and OLE DB portions of MDAC. Specifically, some applications require that certain versions of the ADO and OLE DB drivers be used, so when you install a version of MDAC that will allow one program to work, it may break another program. But since *SQL Tools does not use the ADO or OLE DB drivers*, we have experienced very, very few problems related to MDAC installation or "version problems".

SQL Tools should work well with almost any version of MDAC, so a good rule of thumb is "if MDAC has already been installed on a system, leave it alone or you might break somebody else's software". If MDAC has *not* been installed on a system, you should review the release notes that are provided on the Microsoft web site to determine which version is best for you. Some versions of MDAC include ODBC drivers with known bugs, but they are relatively rare, relatively minor, and surprisingly well documented on the Microsoft web site.

Distributing the MDAC Package

IMPORTANT NOTE: Perfect Sync disclaims all liability for information and/or opinions provided in this document regarding your legal rights under any Microsoft License Agreement. You should consider consulting a qualified attorney before making any decisions that could potentially place you in violation of a Microsoft License Agreement and/or international copyright law.

As we understand it, you may, under certain circumstances, legally re-distribute the MDAC package as a means of distributing ODBC drivers with your applications. You should read the terms of the Microsoft End User License Agreement (EULA) to find out whether or not you qualify.

At the time of this writing, that document can be found by searching Microsoft's web site for "MDAC EULA".

Installing ODBC Drivers from a Database Product

To describe the general process of installing ODBC Drivers, we will walk through the specific steps that are involved in installing a very popular group of Microsoft ODBC drivers, which are included with the Microsoft Office bundle. Specifically, these instructions were written using a copy of Microsoft Office 97 Professional as a guide.

1) Before beginning, run the `\SQLTOOLS\MicrosoftODBC\ODBCAD32.EXE` program that is included with SQL Tools. You can use that program's ODBC Drivers tab to find out which drivers are already installed on a computer. (Windows NT, 2000, and XP users already have a copy of this program in their Windows Control Panel, labeled ODBC.)

2) Locate your Microsoft Office installation disk(s).

3) Locate and run the `SETUP.EXE` program.

4) Select Add/Remove Components.

5) A list of items with checkboxes will appear. Be careful not to *accidentally* change any of the checkboxes. If you do, we suggest that you exit from the Setup program and start over.

6) Look at the list of items and single-click on Data Access (*not* Microsoft Access), then click the Change Option button.

7) Single-click on the Database Drivers item, then click the Change Option button.

8) Double-click on the various drivers, to change the status of their checkmarks. A black checkmark indicates that a driver *will* be installed. We suggest that you install all of the available drivers, so that you won't have to repeat this process later. During our tests we were able to install drivers for Microsoft Access, Microsoft FoxPro, Microsoft Excel, Microsoft SQL Server, and dBASE, plus the Microsoft Text and HTML Driver.

9) Click the various Ok and/or Continue buttons to complete the installation process.

10) Run the `ODBCAD32.EXE` program again, and look at the list of drivers on the ODBC Drivers tab. You should see all of the original drivers, plus those that you just installed.

SQL Tools programs can now access databases that are supported by the ODBC drivers that are installed on your system.

You will need to repeat this process, or another process that installs ODBC drivers, on every computer on which a SQL Tools program will be run.

Terminology Differences

SQL terminology, as defined by the evolution of the SQL language, and BASIC terminology, as defined by the evolution of the BASIC language, are not identical.

For example, ODBC defines a data type (a type of variable) called a `SQL_INTEGER`. It has a range of roughly plus and minus 2.1 billion. BASIC also has a variable type called `INTEGER`, but it has a much smaller range: plus or minus 32,767.

The BASIC data type that has the same numeric range as a `SQL_INTEGER` is called a `LONG INTEGER`, and the SQL data type that has the same range as a BASIC `LONG INTEGER` is called a `SQL_SMALLINT`, so there's bound to be some confusion when you begin mixing SQL and BASIC.

Unfortunately, the ODBC standard also uses the word `LONG`, to refer to a variable-length string that is more than a certain length. For example, a `SQL_LONGVARCHAR` variable (SQL Long Variable-length Character string) is a string that can be more than 256 characters long.

To help keep things straight, this document will usually refer to Data Types with either a `SQL_` or `BAS_` prefix, but you'll still have to be careful. Some SQL Tools functions return strings that contain the SQL terminology, such as the string "INTEGER" that is returned by the various SQL Tools "Data Type Info" functions. These strings are defined by the ODBC driver that you use, and you are required (by the ODBC specification) to use those strings under certain circumstances, so SQL Tools can't really change them. And of course your BASIC compiler will not recognize `SQL_INTEGER` or `BAS_LONG`. You must use the keyword `LONG` in the appropriate places in your source code, such as `Dim` statements.

Compliance Issues

SQL Tools is based on something called the ODBC Standard. Specifically, it is based on ODBC Version 3.x, Level 2. ODBC is a very complex set of standards that was designed (by Microsoft) to provide a common set of commands and techniques that databases of all types could use.

The root of the Compliance Issue is that Microsoft doesn't control all of the databases in the world (Oracle, dBASE, Paradox, and so on), so the ODBC 3.x Level 2 "standard" isn't perfect. And actually, some of Microsoft's *own* products do not fully support ODBC 3.x Level 2.

It is important to remember that not all software which uses "the ODBC standard" supports 100% of the ODBC 3.x Level 2 features. SQL Tools *does* support ODBC 3.x Level 2, but *only if you use an ODBC driver that supports that level of compliance.*

Think of it this way... Your word processor may support color printing, but if your Printer Driver doesn't support color then all you will see is black lettering on white paper. The same is true for your SQL Tools programs and your ODBC driver. SQL Tools can't do things that your driver doesn't support.

If you are writing a program that will always be used with a single type of database, such as Microsoft Access, ODBC compliance really isn't much of a problem. SQL Tools provides alternate methods of performing many basic tasks, and it is possible to write programs that accomplish *many* things even if the ODBC driver doesn't support them directly.

But if you are writing a program that will be used with more than one database -- such as a program that could use either Access *or* Oracle -- you will have to be much more diligent with your testing and debugging. It would be very easy to write a program that works perfectly with Oracle but fails when used with Access, because the Access ODBC driver does not support all of the functions that the Oracle ODBC driver does.

There are two "dimensions" of ODBC compliance with which you may need to be concerned.

The first dimension is the "ODBC Version Number" that a driver supports, which will usually be 1.0, 2.0, or 3.0.

The second dimension is the "Level" of ODBC functionality that a driver supports. The levels are called "Core", "Level 1", and "Level 2".

Each individual ODBC *function* has been assigned a version number and a level number. The version number refers to the first version of ODBC in which the function became available. The level number is an approximation of the level of "sophistication" that the function represents.

In order to claim that it supports ODBC 2.0, a driver must support 100% of the ODBC 2.0 Core functionality. It may or may not also support ODBC 2.0 Level 1 and/or Level 2 functions, in any combination.

In order to claim that it supports ODBC 3.0, a driver must support 100% of the ODBC 3.0 Core functionality. It may or may not also support ODBC 3.x Level 1 and/or Level 2 functions, in any combination.

It is important to note that ODBC 3.x Core functionality is *not* the same thing as ODBC 2.0

Core functionality. In fact, all ODBC 2.0 Level 1 functionality was re-defined as Core functionality in ODBC 3.0.

To help clarify some of these issues, let's look at a specific example. The Microsoft Access 97 ODBC driver reports that it supports an ODBC version of 2.5. If you examine the driver's capabilities in detail, you will find that it supports 100% of the ODBC 2.0 Core *and* Level 1 functionality, plus many Level 2 functions. So they call it "2.5".

Access 97 does not, however, support a feature called "Foreign Keys" which were introduced all the way back in ODBC 1.0. Foreign Keys have always been considered to be a Level 2 feature, so support is not required.

On the other hand, Access 97 does *not* support a Level 2 feature called "Parameter Options", but it doesn't really matter. Parameter Options are an ODBC 1.0 Level 2 feature that has been "deprecated". That means that support is no longer required, because the function has been replaced by a new function. In this case, the Access 97 driver also supports the new function.

Finally, you need to keep in mind that you won't *need* certain ODBC functions. Of the twelve ODBC Level 2 functions that Access 97 does not support, five are related to something called "descriptors", which most programs never need to use.

For more information about ODBC compliance issues, we suggest that you consult the Microsoft ODBC Software Developer Kit.

Two Of Everything: The "Abbreviated" and "Verbose" Functions

When you look at the list of SQL Tools functions, you'll probably notice that there are two of just about everything. Closer examination will reveal that there are four of many things, and eight or more of others. Here's why...

SQL Tools is capable of handling extremely complex programs. In fact, SQL Tools Pro could theoretically be used to write a program that uses 256 different databases at the same time, and where each database has 256 SQL statements that are active, all at the same time. (A much more likely scenario would be a program that uses several databases with one active statement at a time, or one database with many active statements, but anything is possible.)

But *most* of the time, *most* programs will use a single database and a single statement at a time.

Here is an example of "two of everything"...

One of the most commonly used SQL Tools functions is called `SQL_Statement`. It is used to execute SQL statements, to tell a database what to do. To use the `SQL_Statement` function, you need to specify a Database Number (from 1-256), a Statement Number (from 1-256), a parameter like `SQL_STMT_PREPARE` or `SQL_STMT_EXECUTE`, and a string that contains the SQL statement to be prepared or executed.

Since *most* of the time you will be dealing with Database #1 and Statement #1, it can be very tedious to type `1, 1` at the beginning of every single function's parameter list, so SQL Tools provides a complete set of "abbreviated" functions that use default values for the database number and statement number.

If a SQL Tools function name contains the word "Database", "Statement", "Table", "Column", or "Result" it is a *verbose* function that requires you to specify a Database number and/or a Statement Number.

On the other hand, if a SQL Tools function name contains the abbreviation "DB", "Stmt", "Tbl", "Col", or "Res" it is an *abbreviated* function that does not allow the Database Number and Statement Number to be specified as parameters. (Please note that certain words like "Info" are never spelled out in function names and do not indicate an abbreviated function.)

Here is a specific example of a verbose function...

```
SQL_Statement 1, 1, SQL_STMT_EXECUTE, "SELECT * FROM MYTABLE"
```

And here is the abbreviated function that would perform *precisely* the same operation...

```
SQL_Stmt SQL_STMT_EXECUTE, "SELECT * FROM MYTABLE"
```

The `SQL_Statement` and `SQL_Stmt` functions are called "twins".

If you are writing a program that uses one database at a time, with one statement at a time, we recommend that you use the abbreviated functions. It will save you a lot of typing, and it will keep you from making errors.

If you are writing a more complex program, you have a choice: **1)** Use the verbose functions for everything, or **2)** use the `SQL_UseDB` and `SQL_UseStmt` functions to specify which

database and statement you want the abbreviated functions to handle.

For example, a program could use *Database 1, Statement 3* followed by *Database 2, Statement 9* in this way...

```
SQL_Statement 1,3, SQL_STMT_EXECUTE, "SELECT * FROM MYTABLE"  
SQL_Statement 2,9, SQL_STMT_EXECUTE, "SELECT * FROM YOURTABLE"
```

...or it could do this...

```
SQL_UseDB 1  
SQL_UseStmt 3  
SQL_Stmt SQL_STMT_EXECUTE, "SELECT * FROM MYTABLE"  
  
SQL_UseDB 2  
SQL_UseStmt 9  
SQL_Stmt SQL_STMT_EXECUTE, "SELECT * FROM YOURTABLE"
```

If you often switch the default Database number and Statement number at the same time, you can also use this function...

```
SQL_UseDBStmt 2,9
```

...to change both at once.

The advantage of using the `SQL_Use` functions is that they are "sticky". In other words, once you use `SQL_UseDB 2`, all of the abbreviated functions will continue to use Database 2 until you use `SQL_UseDB` again to change the default. That way, you can use the `SQL_Use` functions to specify a database or statement, and then perform a large number of abbreviated functions.

It is also possible to mix the verbose and abbreviated functions. For example if a program did 90% of its work with one database and 10% with a handful of others, you could use the abbreviated functions to handle Database 1, Statement 1, and use the verbose functions for the other 10%. The use of verbose functions does not affect the setting of the `SQL_Use` functions.

If you are writing a *multi-threaded* application which uses more than one database or statement at a time, we strongly recommend that you use the verbose functions for everything. The `SQL_Use` functions affect *all threads at once* so it is not possible, for example, for one thread to use `SQL_UseDB 1` and another to use `SQL_UseDB 2`. Whenever a `SQL_Use` function is used, it affects all abbreviated functions *in all threads*.

Four of Many Things

In addition to providing verbose and abbreviated versions of almost every SQL Tools function, many different functions are provided in both String and Numeric versions.

For example, the various SQL Tools "Info" functions are used to obtain information about databases. The `SQL_TableInfo` family of functions returns information about Tables, and this information can be either numeric or string, depending on the type of information that you are interested in. You might use the `SQL_TableInfoStr` (Info String) function to get the name of a table, like `MYTABLE`, and you might use the `SQL_TableInfoSInt` (Info Signed Integer) function to get the number of columns that a table has, like `20`.

In some case the String and Numeric functions will both be useful. If you use the `SQL_DatabaseInfoStr` function to obtain the ODBC version that a certain database supports, it might return the string `"02.50"`. If you use the `SQL_DatabaseInfoUInt` (Unsigned Integer) function to obtain the same information, it would return `2`. If your program is only interested in the major ODBC version number (2 or 3), that would be enough. (If you're wondering why `SQL_DatabaseInfoUInt` doesn't return `2.5` in this example, remember that `UInt` stands for Unsigned *Integer*.)

So the bottom line is that if you look at a family of functions such as "Table Info", you will see verbose and abbreviated versions, plus String and Numeric versions...

<code>SQL_TableInfoStr</code>	(verbose function, returns a string)
<code>SQL_TlbInfoStr</code>	(abbreviated function, returns a string)
<code>SQL_TableInfoSInt</code>	(verbose, signed integer)
<code>SQL_TblInfoSInt</code>	(abbreviated, signed integer)

That's four different functions that all perform the same basic operation in different ways. The function that you choose will depend on the program that you are writing.

Eight or More of Some Things

Beyond verbose vs. abbreviated functions and string vs. numeric functions, some SQL Tools functions come in *many* different forms. In particular, when you are accessing the columns of a result set (i.e. the results of a SQL statement), there are many different ways to access the data. Since ODBC databases can store many different kinds of data, SQL Tools must be able to return the data to your program in many different forms.

(For information about the various `BAS_` constants, see BASIC Data Types.)

`SQL_ResultColumnStr`

Returns String values. Also, numeric values that are accessed with the `SQL_ResultColumnStr` function are returned as a "binary image" of the numeric data that is compatible with the BASIC `MKI$, CVI, MKL$, CVL`, (etc.) functions. For example, if a LONG Integer value was accessed with `SQL_ResultColumnStr`, it would return a 4-byte binary-coded string that corresponded to the memory image of a LONG integer.

`SQL_ResultColumnSInt`

Signed Integer, in the `BAS_LONG` range. See Signed and Unsigned Integers.

`SQL_ResultColumnUInt`

Unsigned Integer, in the `BAS_DWORD` range. See Signed and Unsigned Integers.

`SQL_ResultColumnBInt`

Big Integer. A human-readable string that represents a number that can be extremely large, usually (but not always) in the `BAS_QUAD` range.

`SQL_ResultColumnFloat`

Floating Point, in the `BAS_DOUBLE` range.

`SQL_ResultColumnDate`

`SQL_ResultColumnTime`

`SQL_ResultColumnDateTime`

Date and time values, in string (`BAS_STRING`) form.

`SQL_ResultColumnText`

Returns a string that can represent string *or* numeric information in human-readable form. Numeric columns are converted to human-readable text in a manner similar to the BASIC `FORMAT$` function.

And then there are several `SQL_ResultColumn` functions that tell you things *about* the data...

`SQL_ResultColumnNull`

Numeric value (logical true/false) that indicates whether or not a column contains a null value.

`SQL_ResultColumnMore`

A numeric value (logical true/false) that indicates whether or not there is more of a Long column that needs to be read.

`SQL_ResultColumnInd`

Numeric "Indicator" value that tells you different things about the status of the column, such as whether or not it contains a null value, how long a string value is, and so on.

`SQL_ResultColumnBufferPtr`

A pointer to the memory buffer where a result column's value is stored.

`SQL_ResultColumnIndPtr`

A pointer to the memory buffer where a result column's Indicator is stored

`SQL_ResultColumnLen`

The length of a string-type result column. (Similar to the BASIC `LEN` function.)

`SQL_ResultColumnSize`

The length of a result column's buffer. (Similar to the BASIC or C `SIZEOF` function.)

`SQL_ResultColumnType`

The SQL Data Type of a column: `SQL_INTEGER`, `SQL_VARCHAR`, and so on.

`SQL_ResultColumnCount`

The number of columns that were returned by a SQL statement.

`SQL_ResultColumnNumber`

The column number that corresponds to a column name.

The Abbreviations

SQL Tools function names make extensive use of abbreviations, in order to reduce the amount of typing that you'll have to do.

Async	Asynchronous
Attrib	Attribute
BInt	Big Integer
Bkmk	Bookmark
Col	Column
Cur	Cursor
DB	Database
Forn	Foreign
Ind	Indicator
Indx	Index
Info	Information
Param	Parameter
Prim	Primary
Proc	Procedure
Priv	Privilege
Rel	Relative
Res	Result
SInt	Signed Integer
Stat	Statistic
Stmt	Statement
Tbl	Table
UInt	Unsigned Integer
Uniq	Unique

Four Critical Steps For Every SQL Tools Program

STEP 1: Tell Your Compiler That You Are Using SQL Tools

Overview: Add the appropriate header file (.INC, .BAS, .H, etc.) to your program.

Details: Keep in mind that if you installed SQL Tools in a directory other than the default, when following these directions you will need to replace \SQLTOOLS\ with the name of the directory where your SQL Tools files are stored.

Visual Basic: Add the SQL Tools "module" file `modSQLToolsStd.BAS` or `modSQLToolsPro.BAS` to your program, depending on which version of SQL Tools (Standard or Pro) you are using. The exact technique you use will vary, depending on the version of Visual Basic that you are using. For example, if you are using VB5 or VB6, open your project, select Project > Add Module, select the "Existing" tab, navigate to the \SQLTOOLS\ directory, and select the appropriate `mod*.BAS` file.

PowerBASIC: At the very beginning of your program, preferably before any other executable code (such as SUBs and FUNCTIONS) you should include the line:

```
$INCLUDE "\SQLTOOLS\SQLT_STD.INC"
```

If you are using SQL Tools Pro you should use `SQLT_PRO.INC` instead of `SQLT_STD.INC`.

(Recent versions of PowerBASIC recognize `$INCLUDE` and `#INCLUDE` as meaning the same thing, and future versions may require `#INCLUDE`.)

Other Programming Languages: Contact support@perfectsync.com to find out whether or not a SQL Tools header file is available for your language. If not...

Every language has its own unique methods for 1) defining the values of constants and 2) declaring functions that are located in external DLLs. The SQL Tools DLLs use 100% standard Win32 conventions, so it should be relatively simple for you to convert the `SQLT_PRO.INC` file to your language. (If you do, please send a copy of the finished file to support@perfectsync.com so that we can share it with others!)

STEP 2: Authorize the SQL Tools Runtime File(s)

Overview: Execute the `SQL_Authorize` function with the appropriate Authorization Code.

Details: At some point in your code *before any other SQL Tools functions are used*, add this line of code:

```
SQL_Authorize &h.....
```

...where the eight dots represent the eight-digit SQL Tools Authorization Code that was provided with your SQL Tools installation file. (The `&h` prefix tells your BASIC compiler that the Authorization Code is a hexadecimal number.) For more information about the codes themselves, see Authorization Codes.

Visual Basic: The best place to put the `SQL_Authorize` line is usually in your

main form's `Form Load` event handler. But that placement is not critical, as long as `SQL_Authorize` is the *very first* SQL Tools function that is used by your program.

PowerBASIC: We recommend that you place the `SQL_Authorize` line at the very beginning of your `PBMain`, `WinMain`, or `Main` function, before any other executable code.

Other Programming Languages: You will probably be required to use a different prefix (such as `0x`) to tell your compiler that the Authorization Code is a hexadecimal number. As with Visual Basic and PowerBASIC, the only important "rule" is that `SQL_Authorize` must be executed before any other SQL Tools functions are used.

STEP 3: Initialize SQL Tools

Overview: Execute the `SQL_Init` or `SQL_Initialize` function.

Details: Either the `SQL_Init` or the `SQL_Initialize` function must be executed after `SQL_Authorize` (see above) but before you use *any other SQL Tools functions*. If you attempt to use any other SQL Tools function before `SQL_Init` or `SQL_Initialize`, your program will certainly fail, and it may even generate an Application Error.

`SQL_Init` initializes SQL Tools using default values that will work well for most programs. If you are writing a complex program you may need to use the `SQL_Initialize` function instead of the simpler `SQL_Init`. See `SQL_Initialize` for details.

Visual Basic *and* PowerBASIC: Add this line of code immediately after the `SQL_Authorize` line that was added in Step 2.

```
SQL_Init
```

Other Programming Languages: Your program must execute the `SQL_Init` or `SQL_Initialize` function after `SQL_Authorize` but *before any other SQL Tools functions are executed*.

SQL Tools is now ready for use.

Your code goes here. Your program may use all of the other SQL Tools functions here, between Steps 3 and 4.

After your program has finished using SQL Tools...

STEP 4: Shutting Down SQL Tools

Overview: As late as possible in your program, execute the `SQL_Shutdown` function.

Details: You must execute the `SQL_Shutdown` function at some point in your program *after* it is finished using all other SQL Tools functions. It is also very important that you choose a location that will be executed *reliably*. If your program is allowed to close without executing `SQL_Shutdown`, database connections may be "orphaned". This will reduce the number of connections that are available to your program and to other programs in the future. If this happens too many times, it may

be necessary for you to shut down and restart the database server before a new connection can be made.

Visual Basic: Your main form's `Form Unload` event handler is *usually* a good place for `SQL_Shutdown`. But keep in mind that if the Visual Basic `End` statement is executed by your program, the `Form Unload` event will not be generated. *If your program uses `End`, your program should execute `SQL_Shutdown` before using `End`!* Please consult the Visual Basic documentation for more information about `End`. Also keep in mind that if (during development) your program is stopped with the Visual Basic IDE's "End" toolbar button, the `Form Unload` event will not be generated. If possible, this should be avoided. (**Tip:** Add a temporary "Close SQL Tools" button to your main form, and add `SQL_Shutdown` to the button's `Click` event. Then get in the habit of clicking that button before you click the `End` button on the toolbar.)

PowerBASIC: The ideal location for the `SQL_Shutdown` function is the very end of your `WinMain`, `PBMain`, or `Main` function, just *before* the `END FUNCTION` line. **IMPORTANT NOTE:** If your program's `WinMain`, `PBMain`, or `Main` function contains any `EXIT FUNCTION` lines, you must also add `SQL_Shutdown` to those locations. **Or...**

In order to simplify the `SQL_Authorize`, `SQL_Init` and `SQL_Shutdown` process, we recommend that you use an extremely small "wrapper" function, like this:

```
FUNCTION PBMain AS LONG
    SQL_Authorize &h.....
    SQL_Init
    PBMain = MainProgram
    SQL_Shutdown
END FUNCTION

FUNCTION MainProgram AS LONG
    'your code goes here
END FUNCTION
```

...and confine your own code to the `MainProgram` function. That way, no matter what happens in your code (short of an Application Error or GPF) the `SQL_Shutdown` function is guaranteed to execute.

Other Programming Languages: It is very important for you to identify a location for the `SQL_Shutdown` function that will ensure that it executes even if your program terminates abnormally. It is not (usually) possible to protect against Application Errors and GFPs but all other types of exits should be covered, including normal and "break" exits.

IMPORTANT NOTE: If you are creating a DLL (as opposed to an EXE) that will use SQL Tools, be sure to read *Special Considerations for DLL Programmers*.

Tip: The `\SQLTOOLS\SAMPLES` directory contains Visual Basic and PowerBASIC "skeletons" for simple SQL Tools programs. If you always use a copy of the appropriate skeleton program as a starting point for your SQL Tools programs, you won't have to worry about any of the critical steps that are described above. (Be careful to use a *copy*, and to avoid editing the skeleton files themselves.)

Special Considerations for DLL Programmers

If you are writing normal "EXE" programs you can skip this section. It deals with issues that are only pertinent if you are creating DLLs that use SQL Tools.

Because of the way Microsoft Windows and ODBC work, it is not possible to use any of the following SQL Tools functions in the `DLL_PROCESS_DETACH` section of a DLL's `DLLEntryPoint` or `LibMain` function:

```
SQL_Shutdown  
SQL_CloseDB (and SQL_CloseDatabase)  
SQL_CloseStmt (and SQL_CloseStatement)
```

Failure to follow this rule will always result in **1)** program lock-ups, **2)** Application Errors, **3)** `OleMainThreadWndName` errors, **4)** memory-related problems (such as memory leaks) on the computer that is running the SQL Tools program, **5)** similar memory-related problems on the database server computer, and **6)** database connections that are never closed, possibly resulting in your database server running out of available connections. Other serious problems are also possible.

Because the `SQL_Shutdown` function is the most commonly used, and because it uses the other two functions (`SQL_CloseDB` and `SQL_CloseStmt`) internally, the rest of this discussion will focus on `SQL_Shutdown`.

It is *very* important for the `SQL_Shutdown` function to be used *before* your program's main executable program (EXE) closes. The `SQL_Shutdown` function must be called directly from your program's EXE, just before it exits. An acceptable alternative is for your EXE to call a function in your DLL, which then calls `SQL_Shutdown`. It is not acceptable for an EXE program to close without using `SQL_Shutdown` or to rely on `DLL_PROCESS_DETACH` in a DLL.

Please note that this `DLL_PROCESS_DETACH` restriction is not a defect in SQL Tools. It is, according to Microsoft, an intentional design detail of Windows and ODBC. For confirmation of this fact you can visit the <http://microsoft.com> web site and read Microsoft Knowledge Base article number Q177135. It is titled "Do not Call ODBC Within `DLL_PROCESS_DETACH` Case", and it contains the phrase "This behavior is by design".

Technical Details, If You're Interested...

When an executable program starts up, Windows checks to see whether or not the program uses any DLLs. In the case of this example, it would determine that *your* DLL is required, and your DLL would be automatically loaded into memory. Then Windows would detect that your DLL uses SQL Tools, and the `SQLT_STD.DLL` or `SQLT_PRO.DLL` library would be automatically loaded by Windows. Finally, Windows would see that SQL Tools uses a standard Windows file called `ODBC32.DLL`, and that library would be loaded into memory.

At that point, your EXE and all of the DLLs can use each other's functions. This is a perfectly normal relationship between Windows modules (EXEs and DLLs), and it works very well.

When your EXE program closes, all of the DLLs are automatically unloaded from memory.

As they unload, their `DLL_PROCESS_DETACH` code is automatically executed so that they can perform "cleanup" and "closedown" operations. This would *normally* be a good place to put the `SQL_Shutdown` function.

Unfortunately, when certain functions in the Microsoft `ODBC32.DLL` library are used, the DLL actually creates one or more new threads of execution. (If you're not familiar with threads, that means that the `ODBC32.DLL` library actually runs a separate, invisible program that is "attached" to the main program.)

For example, when your program uses the `SQL_OpenDB` function to open a database, SQL Tools uses (among other things) an ODBC function called `SQLDriverConnect`, which is located in the `ODBC32.DLL` library. The Microsoft `SQLDriverConnect` function then launches a new, invisible sub-program to "manage" the database connection for you. (If you're using Windows NT, 2000, or XP you can use the Windows Task Manager to detect these new threads. Use View/Select/ThreadCount.)

When your executable program (EXE) closes, Windows automatically and *instantly* shuts down all of the threads, before it tells the DLLs to unload. That's the way threads work. So if a DLL then tries to use the `SQL_Shutdown` function when it unloads, *the database "manager" thread will no longer be available*, and your program will crash or hang. And if your program was connected to a database, the connection would not be closed. The only way to close the connection may be to restart the database server.

According to Microsoft Developer Support, there is no known solution or "workaround" for this problem, other than *requiring* executable programs to close all database connections before they exit.

A SQL Tools Primer

The following sections of this document are intended to summarize all of the terms and concepts that are used in SQL Tools programming.

Each term or concept is presented in its own section, "encyclopedia" style, but instead of being ordered alphabetically the topics are arranged so that they progress from basic to complex, to allow them to be read sequentially in a "tutorial" style. If this document is presented in electronic form (such as a Help File), you can use the >> button at the top of the screen to advance from page to page.

If you want to look up a certain word or phrase, use the Index and Table of Contents that are provided with this document. When it is presented as a Help File you can also use the Find feature to locate every instance of a word or phrase.

For a complete tutorial, use the >> button at the top of the Help screen to read the following pages in order.

What a Database Is

Broadly speaking, a database is a collection of information in one form or another.

Of course, that's not a very useful description for a programmer. To explain it better, we'll first need to define a few terms.

Modern computer databases always contain one or more "tables". Tables will be described in more detail shortly. For now, you should picture them as "spreadsheets" or "grids", with rows and columns containing words and/or numbers.

Databases also contain a wide variety of other structures that are necessary for the maintenance of the tables, but the database's *data* -- the useful information that is stored in the database -- is contained in the tables.

When most programmers visualize a database, they see one or more tables.

SQL and ODBC

Over the years, many different types of databases have been designed and used.

As time went by, a standard language called SQL gradually evolved. SQL stands for Structured Query Language (some people say Standard Query Language), and it is simply a standard way of "talking" to a database. If you use a SQL command like *UPDATE*, SQL databases understand what that means.

You can think of SQL as a computer language, much like BASIC. Since you can't really write programs in 100% SQL, it may be better to think of it as a sub-language that can be added to a computer language like BASIC.

If you know how to use the SQL language, you can (at least theoretically) write a program that can interface with a "SQL compliant" database, i.e. a database that understands the SQL language. To be clear, when somebody says that a database is SQL *compliant*, it means that the database *complies* with the SQL rules.

Of course, not all SQL compliant databases are created equal. As with BASIC, there are several different variations or "dialects" of SQL. All SQL databases understand the core commands, but each database has its own extensions and quirks. Some understand more complex commands than others, some can handle hundreds of simultaneous users, and so on. Not only that, but the SQL commands themselves are only *part* of most programs. For example, each type of database requires that you "connect to it" in a different way.

So Microsoft designed and published an even broader standard called ODBC. ODBC stands for Open DataBase Connectivity, which simply means that it is an attempt to create an "open", standard way of doing *everything*. SQL is certainly a large part of ODBC, but ODBC takes the additional steps of specifying how you connect to a database, standard error messages, and on and on.

If a database is "ODBC compliant", that means that virtually everything is standardized. As with SQL, you can write a program that can interface with a compliant database, but more than that, you can (at least theoretically) write a single program that can interface with *any* ODBC compliant database.

If SQL is a computer language, you can think of ODBC as an operating system much like Windows, except (of course) that it runs *inside* Windows. "Subsystem" is probably a better way to visualize ODBC.

To summarize, ODBC is a subsystem of Windows. If you use SQL Tools, SQL is a sub-language of BASIC.

ODBC Drivers, and the Driver Manager

A driver is a special kind of software program that effectively becomes part of the Windows operating system and allows other applications (such as your programs) to access a particular capability.

ODBC is a Microsoft standard that allows programs to access different database formats through a standard interface. It is possible for an ODBC-compliant program (like SQL Tools) to access virtually any ODBC-compliant database.

An ODBC driver is a piece of software that allows your computer to use ODBC capabilities.

All of the ODBC drivers that are installed on your computer are "managed" by another piece of software, called the ODBC Driver Manager.

You can visualize it this way...



In order to talk to the database, your program simply tells SQL Tools to do something via the first down-arrow.

SQL Tools then communicates with the ODBC Driver Manager, which talks to the specific ODBC Driver that is used for a particular type of database, and then the ODBC Driver talks to the actual database. All of that takes place in a small fraction of a second. The database then processes the request, replies "okay, I did that", and sends the message back up the chain (via the up-arrows) to SQL Tools.

And then SQL Tools passes the response back to your program.

You, as a programmer, really only have to deal with two parts of the chain: SQL Tools (of course) and the ODBC driver. Don't worry: SQL Tools provides 100% of the functions that you will need to work with a database. But an ODBC driver has to be *installed* on your computer before SQL Tools can do its job.

If you want to use a Microsoft Access database, you'll need to install the Microsoft Access ODBC Driver on your computer. If you want to use an Oracle database, you'll need to install an Oracle driver. You can think of the ODBC drivers as "translators" which allow your SQL Tools programs to work with different ODBC compliant databases.

It is possible to install many different ODBC drivers on the same computer. Microsoft provides a program called the ODBC Database Administrator to handle the management of ODBC drivers. If your computer is running Windows NT, 2000, or XP, you already have a copy of the ODBC Administrator. Look in your Control Panel, and double-click the ODBC icon. If you're running Windows 95, 98, or ME, you can use the copy of the Administrator

program that is supplied with SQL Tools. Look in the \SQLTOOLS\MicrosoftODBC directory.

SQL Tools and ODBC Drivers

Because SQL Tools relies on ODBC drivers for all database operations, it is not possible for SQL Tools to support a feature, even if it is supposed to be part of the SQL or ODBC standard, if the ODBC driver that you are using does not support that feature.

For example, if you are using the Microsoft Access ODBC driver, SQL Tools will not provide advanced Oracle-style functionality for Access databases. SQL Tools will give you access to the features that the Microsoft Access ODBC driver provides, but it does not attempt to "simulate" advanced features that are not provided by a given driver.

It is possible for programs to do this -- in fact it is a common programming technique -- but SQL Tools and your programming language simply provide the "raw" functions that would allow you to write programs that simulate advanced features.

ALL OF THE SQL TOOLS SPECIFICATIONS THAT ARE LISTED IN THIS DOCUMENT ARE SUBJECT TO LIMITATIONS BY THE ODBC DRIVERS THAT YOU CHOOSE TO USE.

IF YOU NEED ADVANCED DATABASE FUNCTIONALITY THAT IS NOT PROVIDED BY A GIVEN ODBC DRIVER, YOU MUST EITHER WRITE THE FEATURE-SIMULATION CODE YOURSELF OR UPGRADE YOUR PROGRAM TO A DIFFERENT ODBC DRIVER.

Opening a Database

Once the appropriate ODBC driver has been installed, your SQL Tools program will be able to use the types of databases that the driver supports.

The first runtime step in using a database is establishing communication between your program and the database. Other books that you may read will probably refer to a "database connection", but SQL Tools uses the term "open" because that term is very familiar to most BASIC programmers. When you open a database, you tell SQL Tools to prepare it for use.

Three different methods can be used to specify how SQL Tools should open a database.

- 1)** Specify a DSN file name
- 2)** Specify a Connection String
- 3)** Manual "Navigation"

All three methods use the same SQL Tools functions, called `SQL_OpenDatabase` or `SQL_OpenDB`.

Using a DSN File to Open a Database

Example: `SQL_OpenDB "MYDATA.DSN"`

A DSN or "Datasource Name" file is *not* a database. It is a text file that contains information *about* a database, such as where it is located, the ODBC driver that is required to access it, and so on. A valid DSN file contains all of the information that is needed to open a database.

DSN files can be created in several different ways.

- 1) If you search your hard drive you may find that some DSN files already exist on your system. Many programs that use ODBC drivers also use DSN files. If you specify a *partial* DSN file name such as `*.DSN`, the `SQL_OpenDB` function will display a standard Open File dialog box that will allow you to browse for a DSN file.
- 2) If you know the format of a DSN file for a particular type of database, you can hand-edit a DSN file, or create one from scratch using a text editor. (This is usually not necessary.)
- 3) You can use the Microsoft ODBC Datasource Administrator program, which is included with SQL Tools, to create DSN files.
- 4) The Manual Navigation method can also be used to create DSN files.

See Appendix G for information about the DSN File keywords `DSN`, `FILEDSN`, `DRIVER`, `UID`, `PWD`, and `SAVEFILE`.

Using a Connection String to Open a Database

Example: `SQL_OpenDB "DSN=SYS1;UID=JOHN;PWD=HELLO"`

Like a DSN file, a valid Connection String contains all of the information that is necessary to connect to a database. In fact, if you create a text file that contains a connection string, and give it a name with the `.DSN` extension, you have created a DSN file.

Connection strings can be very complex. For example, here is the connection string that is used to open the sample database called "Book Collection" that is provided with Microsoft Access 97.

```
DBQ=C:\WINNT\Profiles\xxx\Personal\Book Collection.mdb;  
DefaultDir=C:\WINNT\Profiles\xxx\Personal; Driver={Microsoft Access  
Driver (*.mdb)}; DriverId=25; FIL=MS Access; ImplicitCommitSync=Yes;  
MaxBufferSize=512; MaxScanRows=8; PageTimeout=5; SafeTransactions=0;  
Threads=3; UID=admin; UserCommitSync=Yes
```

Each type of database requires a different kind of connection string. We suggest that you begin by using a DSN file to open a database, and then examine the contents of the DSN file to learn about the various options. See Appendix G for information about the connection string keywords `DSN`, `FILEDSN`, `DRIVER`, `UID`, `PWD`, and `SAVEFILE`.

Manual Navigation: Using the SQL_OpenDB Function to Create a DSN File

Example: `SQL_OpenDB ""`

If you use an empty string with the `SQL_OpenDB` function, it will display a series of dialog boxes that will allow you to "navigate" to a connection, save a DSN file, and then select the DSN file. In the future, your programs can simply specify the new DSN file instead of repeating the "navigation" process.

The `SQL_OpenDB` dialog boxes are actually provided by a Microsoft subprogram that is very similar to certain parts of the Microsoft ODBC Database Administrator. The subprogram includes its own Windows Help File, which explains how to use the dialog boxes.

Error Messages After Opening a Database

It is very common for the `SQL_OpenDB` function to return an Error Code of `SUCCESS_WITH_INFO` and to generate an Error Message that says...

The driver doesn't support the version of ODBC behavior that the application requested.

That message means that your program specified ODBC 3.x behavior (via the `SQL_Initialize` function), and that you have opened a database that does not support ODBC 3.x behavior. Most ODBC drivers can *emulate* at least some 3.x behavior, so it is not a good idea to use a different `ODBCVersion` value with `SQL_Initialize`. If you do that, the `SUCCESS_WITH_INFO` message will no longer be generated but you will not be able to use certain ODBC functions such as Bookmarks.

You should not be concerned by the "doesn't support..." Error Message. An Error Code of `SUCCESS_WITH_INFO` means that the `SQL_OpenDB` function *was successful*, and that the ODBC Driver Manager simply wanted to alert you to the fact that the ODBC Driver that you are using does not support ODBC 3.x behavior.

We suggest that you have your program check and clear the SQL Tools Error Stack (see) before using `SQL_OpenDB`, and then check it again after `SQL_OpenDB`. If the only message in the stack is a `SUCCESS_WITH_INFO` message, you can safely ignore it.

Another technique for ignoring errors is covered under Ignoring Predictable Errors.

For more information, see Error Codes.

Determining Database Capabilities

Once you have opened a database, you *may* need to determine what capabilities the database has. This is particularly important **1)** during development and **2)** at runtime if your program may use different databases at different times.

SQL Tools provides a wide variety of functions that can provide literally *hundreds* of pieces of information about a database.

For example, if your program relies on an advanced feature like Table Privileges you may need to use the `SQL_FuncAvail` function to determine whether or not a database supports them.

A "generic" way to determine a database's capabilities is to use the `SQL_DBInfoStr` and `SQL_DBInfoUInt` functions to obtain database "version" information. The following values are of particular interest...

```
PRINT SQL_DBInfoStr(DB_DM_VER)
PRINT SQL_DBInfoStr(DB_ODBC_VER)
PRINT SQL_DBInfoStr(DB_DRIVER_NAME)
PRINT SQL_DBInfoStr(DB_DRIVER_VER)
PRINT SQL_DBInfoStr(DB_DRIVER_ODBC_VER)
PRINT SQL_DBInfoStr(DB_DBMS_NAME)
PRINT SQL_DBInfoStr(DB_DBMS_VER)
```

When used with a Microsoft Access 97 test database, those functions returned the following values:

```
03.00.2301.0000
03.00.0000
odbcjt32.dll
03.50.3428.00
02.50
ACCESS
3.5 Jet
```

The 03.00.2301.0000 value is the version number of the ODBC Driver Manager that is being used. The 03.00.0000 indicates that the Driver Manager supports ODBC 3.0.

"odbcjt32.dll" is the actual file name of the ODBC driver that is being used, and the driver's version number is 03.50.3428.00. Note that the next value is 02.50 which is the ODBC version that the driver *supports*. That is *not* the same thing as the driver version number. (In fact the major version numbers in this example are different.)

The DBMS program that was used to create the database was ACCESS, and the Access version was 3.5 Jet.

If your ODBC driver supports them, you can also use the following functions to determine the "level" of ODBC that the driver supports (Core, Level 1, or Level 2), and the level of "SQL Conformance" that the driver supports.

```
PRINT SQL_DBInfoUInt(DB_ODBC_INTERFACE_CONFORMANCE)
PRINT SQL_DBInfoUInt(DB_SQL_CONFORMANCE)
```

As you can see, even determining a database's "version" can be a fairly complex task.

Fortunately, the `SQL_DBInfoStr` and `SQL_DBInfoUInt` functions can also provide very specific answers to very specific questions, such as *"does the database support Outer Joins"* or *"what is the maximum length of a column name"*, or *"does the database support a `ILockType` value of `LOCK_EXCLUSIVE` when the `SQL_SetPos` function is used with a keyset-driven MultiRow cursor?"*

We suggest that you take a few minutes to review the types of values that can be obtained from the `SQL_DBInfoStr` and `SQL_DBInfoUInt` functions. They are *extremely* powerful tools.

Tables, Rows, Columns, and Cells

Once you have opened a database, you can access everything that is inside it. You may remember from the beginning of this primer that a database was loosely defined as "one or more tables".

A table can be visualized as a two-dimensional grid, with rows and columns. Here is a very simple Address Book table, with columns for Name, Address, and City, and with four rows for four different people:

John Q. Public	123 Main Street	Anytown
Jane Doe	456 First Blvd.	Janestown
Bob Smith	789 Second Ave.	Buffalo
Mary Jones	321 Deebee Row	Jonesville

The *columns* of a table always contain uniform data. That is to say, the different columns can contain different types of data, but the data in any given column is always of the same type. For example, the Name column in the Address Book table contains names and nothing but names.

The *rows* of a table always contain data that is related in another way. In the Address Book table, one row represents one person. In another type of table, such as a Book Collection table, one row might represent one book.

If you're an experienced BASIC programmer, you may be familiar with the terms "record" and "field". In the world of SQL databases, records correspond to rows, and fields correspond to columns.

Another term for this type of data structure is a "relation", because the columns in a given row contain "related" data. (A name is related to an address, and so on.) That's where the term "relational database" comes from: it's any database that uses relations (tables, rows and columns) as its most basic data structure.

Each "box" in a table is called a *cell*. A cell contains the data for one column in one row. A less formal (and more common) term for a cell would be a "column value", which usually implies that one row is being discussed.

Adding new columns to a table is *usually* a "design time" operation. For example, if you were designing the Address Book table you would probably want to add columns for State, Country, Zip Code, Phone Number, and so on. You might also want to create separate columns for First Name, Last Name, and Middle Initial. The choices are virtually endless, and they will depend on the type of table that you are designing. But in the end, your database *columns* will be usually be part of your program's design, and will rarely be changed.

Adding new rows to a table, on the other hand, is a very common runtime operation. For example, adding new people to the Address Book table is something that would happen all the time.

Other common operations include deleting rows and updating rows.

Table Metadata

"Metadata" is a fancy word for "behind-the-scenes information".

For example, here is our simple Address Book table again:

John Q. Public	123 Main Street	Anytown
Jane Doe	456 First Blvd.	Janestown
Bob Smith	789 Second Ave.	Buffalo
Mary Jones	321 Deebee Row	Jonesville

The table itself is useful all by itself -- it contains information -- but in order to be efficient the database must also contain metadata about the table.

For example, consider this diagram:

<i>FullName</i>	<i>StreetAddress</i>	<i>City</i>
<i>String</i>	<i>String</i>	<i>String</i>
<i>20 char max</i>	<i>40 char max</i>	<i>20 char max</i>
John Q. Public	123 Main Street	Anytown
Jane Doe	456 First Blvd.	Janestown
Bob Smith	789 Second Ave.	Buffalo
Mary Jones	321 Deebee Row	Jonesville

The column labels *FullName*, *StreetAddress* and *City* are not part of the "data grid", and they do not count as rows. Neither do the "data type" descriptions. All of those things are metadata that *describe* the columns.

In Win32 programming, the word "property" is often used to refer to metadata. For example, it might be said that the second column's "column name property" is *StreetAddress*.

A wide variety of metadata is provided by modern databases. Each column has a name, a Data Type, a Width, and many other properties. You can picture a table as a grid that is "surrounded" by metadata of many different types.

So far we have described column metadata (like column names), but you should keep in mind that tables -- and even databases themselves -- have metadata too. Each table has a name and a type (such as TABLE, SYSTEM TABLE, or VIEW), among other properties. And databases have metadata values such as the name of the disk file that contains the database.

Here is a better definition of a database than we were able to give earlier:

A database consists of one or more tables and all of their metadata.

SQL Data Types

One of the most important kinds of metadata is the Column Data Type. Every column of every table must have a Data Type assigned to it, so that the database (and your program) will know how to deal with the column.

The following pages list of all of the basic SQL Data Types.

SQL_CHAR
SQL_VARCHAR
SQL_LONGVARCHAR
SQL_INTEGER
SQL_SMALLINT
SQL_TINYINT
SQL_BIT
SQL_BIGINT
SQL_REAL
SQL_DOUBLE
SQL_FLOAT
SQL_NUMERIC
SQL_DECIMAL
SQL_TIMESTAMP
SQL_DATE
SQL_TIME
SQL_INTERVAL
SQL_BINARY
SQL_VARBINARY
SQL_LONGVARBINARY

SQL Unicode Data Types

SQL_wCHAR
SQL_wVARCHAR
SQL_wLONGVARCHAR

You should also become familiar with Datasource-Dependent Data Types.

SQL_CHAR

A fixed-length string. This is the oldest and most basic SQL data type, but it is used by relatively few modern databases because it wastes storage space in the database and in runtime memory.

The length of this data type is specified, on a column-by-column basis, when a database is designed. It is most appropriate for something like a MiddleInitial or SocialSecurityNumber column, where the length of the data is fixed and is known ahead of time. But even something as seemingly-standard as a telephone number -- which might contain a Country Code or an Extension -- might not work well as a fixed-length string.

Many databases do not support SQL_CHAR values which are longer than 254 characters. (The legal range of string lengths is usually 0–255, but one character is often reserved for a CHR\$(0) string terminator.)

The Display Size and Octet Length of a SQL_CHAR value depend on the length that was specified when the database was designed. (The Octet Length property does not include the byte that is required for the string's null terminator.)

SQL_VARCHAR

A variable-length string. The maximum length of each `SQL_VARCHAR` column is specified when a database is designed.

This is probably the most common type of column in most modern databases. It is appropriate for a wide variety of uses, such as Name, Address, and City columns where the length of the data can vary greatly but will not exceed a certain "reasonable" value, such as 255 characters. In fact, most databases limit `SQL_VARCHAR` values to a maximum of 255 characters, and many do not support more than 254 characters. (The legal range of string lengths is usually 0–255, but one character is often reserved for a `CHR$(0)` string terminator.)

`SQL_VARCHAR` columns are more efficient than `SQL_CHAR` columns because they do not waste space in the database if the contents of a column do not fill the available column length.

The Display Size and Octet Length of a `SQL_VARCHAR` value depend on the maximum length that was specified when the database was designed. (The Octet Length property *does not* include the byte that is required for the string's null terminator.)

SQL_LONGVARCHAR

A "long" variable-length string. The definition of "long" depends on what you're doing. In most cases, ODBC considers strings that are *potentially* over 255 characters to be "long".

The maximum length of a SQL_LONGVARCHAR column is defined by the ODBC driver that is used. A common maximum length is 1,073,741,824 characters (1 gigabyte).

The most common use of this data type is a "memo" field that allows the user to enter strings of virtually any length.

The Display Size and Octet Length of a SQL_LONGVARCHAR value depend on the maximum length that was specified when the database was designed. (The Octet Length property does not include the byte that is required for the string's null terminator.)

SQL_INTEGER

A 32-bit integer value, stored in binary form. It can be interpreted as a Signed Integer in the BAS_LONG range of $-2,147,483,648$ to $+2,147,483,647$, or an Unsigned Integer in the BAS_DWORD range of zero (0) to $+4,294,967,295$.

This is the most common and most efficient type of numeric data for your programs to *process*, but it may or may not be the most efficiently-stored and retrieved data type. Your results will vary, depending on the type of database that you choose, and you may get better, faster results with a different integer data type.

The Display Size for a SQL_INTEGER value is 10 if the value is unsigned, or 11 if it is signed.

The Octet Length for a SQL_INTEGER value is 4.

SQL_SMALLINT

A 16-bit integer value, stored in binary form. It can be interpreted as a Signed Integer in the `BAS_INTEGER` range of `-32,768` to `+32,767` or an Unsigned Integer in the `BAS_WORD` range of zero (0) to `+65,535`.

The Display Size for a `SQL_SMALLINT` value is 5 if the value is unsigned, or 6 if it is signed.

The Octet Length for a `SQL_SMALLINT` value is 2.

SQL_TINYINT

An 8-bit integer value, stored in binary form. It can be interpreted as a Signed Integer in the range -128 to $+127$ or an Unsigned Integer in the `BAS_BYTE` range zero (0) to $+255$. (Visual Basic and PowerBASIC do not directly support 8-bit Signed Integers, but they can be stored in `BAS_INTEGER` variables since that data type has a range of $-32,768$ to $+32,767$.)

The Display Size for a `SQL_TINYINT` value is 3 if the value is unsigned, or 4 if it is signed.

The Octet Length for a `SQL_TINYINT` value is 1.

SQL_BIT

A one-bit integer value, stored in binary form. A `SQL_BIT` column can be interpreted as having...

- 1)** a value of zero (0) or +1, if you use the SQL Tools `SQL_ResultColumnUInt` function to return the column's value as an unsigned integer, or
- 2)** a value of zero (0) or --1, if you use `SQL_ResultColumnSInt` to return the value as a Signed Integer.

Most BASIC programmers should probably use 0 and --1. See Appendix H: Logical True and False for more details.

In most databases, `SQL_BIT` columns are actually stored as larger data structures, so they can provide extremely efficient storage for True/False values. Generally speaking, adding one `SQL_BIT` column to a table adds a certain amount of overhead, and then a fixed number of additional `SQL_BIT` columns (often 7 or 15) can be added to the same table with little or no additional overhead.

Visual Basic and PowerBASIC do not directly support the `BAS_BIT` data type, but they can be stored in other types of `BAS_` variables such as `BAS_LONG` and `BAS_INTEGER`. (Using a signed `BAS_` data type allows the storage of 0, +1, or --1.)

The Display Size for a `SQL_BIT` value is always considered to be 1, because the meanings of +1 and --1 are identical.

The Octet Length for a `SQL_BIT` value is 1.

SQL_BIGINT

A 64-bit integer value (or larger), stored in string form. Since many computer languages do not yet support 64-bit math, all ODBC drivers return these values as strings.

Visual Basic does not support `BAS_QUAD` integers but it does support other data types that may or may not be usable, depending on the actual values that are stored in the database.

Signed `SQL_BIGINT` values up to plus-or-minus 9.22×10^{18} are supported by PowerBASIC's `QUAD` (`BAS_QUAD`) data type, as well as the PowerBASIC `Val`, `Str$`, and `Format$` functions, among others. PowerBASIC does not currently support 64-bit Unsigned Integers. Fortunately, neither do most databases, so it is *usually* safe to use the PowerBASIC `VAL` function to convert a `SQL_BIGINT` string value into a `BAS_QUAD` value.

The Display Size for a `SQL_BIGINT` value is always 20, regardless of whether the value is signed or unsigned.

The Octet Length for a `SQL_BIGINT` value is the length of the string that would be required to hold the character (i.e. text) representation of the data.

SQL_REAL

A single precision floating-point numeric value in the range of plus-or-minus 8.43×10^{-37} to 3.37×10^{38} .

This SQL Data Type corresponds directly to the `BAS_SINGLE` Data Type.

The Display Size for a `SQL_REAL` value is always 14.

The Octet Length for a `SQL_REAL` value is 4.

SQL_DOUBLE

A double precision floating-point numeric value in the range of plus-or-minus 4.19×10^{-307} to 1.67×10^{308} .

This SQL Data Type corresponds directly to the `BAS_DOUBLE` Data Type.

The Display Size for a `SQL_DOUBLE` value is always 24.

The Octet Length for a `SQL_DOUBLE` value is 8.

SQL_FLOAT

A floating-point numeric value, the range and precision of which can be specified while you are designing a database.

Because it is user-defined, this Data Type does not correspond directly to any Visual Basic or PowerBASIC Data Type. It may therefore require special handling. See `SQL_ResColFloat`, with particular attention to the section titled "Important Notes about SQL_Float Data".

The Display Size for a `SQL_FLOAT` value is always 24.

The Octet Length for a `SQL_FLOAT` value is 8.

SQL_NUMERIC and SQL_DECIMAL

These are *numeric* data types that are returned to your program in *string* form. For example, the value "2000.0153" could be stored in a SQL_NUMERIC or SQL_DECIMAL column.

The precision and scale (the total number of digits, and the number of digits to the right of the decimal point) are specified when a database is designed. The usual notation is DECIMAL(X,Y) where X and Y are integer values. Most database do not support SQL_NUMERIC or SQL_DECIMAL values with a total of more than 15 digits.

Unlike floating-point columns, these data types provide a way to store *precise* non-integer values. (As you may know, floating-point variables can only store approximate values, within a given precision.)

With some ODBC drivers, SQL_NUMERIC and SQL_DECIMAL columns perform this unusual feat by storing the column's value as two separate integer values of pre-determined length, and creating a string when the value is requested by your program. For example, a database could store "2000.0153" as the integer numbers 2000 and 153. Since the database would know how many digits there are before and after the decimal point, it could re-create the string "2000.0153".

Some programmers have used SQL_NUMERIC and SQL_DECIMAL columns to store "fractional" data. For example, the string value "1.3" could be used to represent the fraction $1/3$, which cannot be precisely stored in a floating point variable -- or in any other type of variable for that matter -- because it has an infinite number of digits after the decimal point and therefore requires a variable of infinite precision.

The Display Size for a SQL_NUMERIC or SQL_DECIMAL value is the precision of the column plus 2. For example, the display size of a DECIMAL(10,3) column would be 12.

The Octet Length for a SQL_NUMERIC or SQL_DECIMAL value is the length of the string that would be required to hold the character (i.e. text) representation of the data..

SQL_TIMESTAMP and SQL_TYPE_TIMESTAMP

ODBC Version 2 requires the use of `SQL_TIMESTAMP`.

ODBC Version 3 requires the use of `SQL_TYPE_TIMESTAMP`.

You must use the appropriate timestamp data type, depending on the capabilities of the ODBC driver that you are using and, if you use `SQL_Initialize` instead of `SQL_Init`, the value of the `IODBCVersion` parameter. (`SQL_Init` automatically specifies ODBC Version 3.)

A timestamp column stores one Date value and one Time value in a standard 16-byte (128-bit) binary format. It is also frequently used to store either a date *or* a time.

The SQL Tools Declaration Files (`SQLT_STD.INC`, `SQLT_PRO.INC`, `modSQLToolsPro.BAS`, `modSQLToolsPro.BAS`, etc.) contain the data structure that is returned by a `SQL_TIMESTAMP` or `SQL_TYPE_TIMESTAMP` column. The structure looks like this:

```
Type TIMESTAMP_STRUCT
    Year      As Integer
    Month     As Integer
    Day       As Integer
    Hour      As Integer
    Minute    As Integer
    Second    As Integer
    Fraction  As Long
End Type
```

(The data types such as `INTEGER` will vary from file to file, because of differences in the way Visual Basic and PowerBASIC handle certain data types, but the results will be exactly the same.)

You can access a `SQL_TIMESTAMP` or `SQL_TYPE_TIMESTAMP` value in several different ways. The easiest way is to use the `SQL_ResColDateTime`, `SQL_ResColDate`, or `SQL_ResColTime` function to read the column data and produce a formatted string. The `SQL_ResColText` function can also be used. If you are using the SQL Tools Pro Runtime Files you can also use the `SQL_ResColDateTimePart` function. For an alternate (and even more flexible) method of accessing timestamp columns, see **Advanced Techniques** below.

You have to be careful when using timestamp columns because they can *appear* to contain invalid information. For example, instead of restricting `Seconds` values from zero to fifty-nine (0-59) as you might expect, the ODBC specification allows values from zero to sixty-one (0-61) in order to allow times involving one or two "leap seconds".

The `Fraction` element of a `SQL_TIMESTAMP` column is a `BAS_DWORD` that can hold values from 0 to 4,294,967,295, but the largest legal value is 999,999,999 so a `BAS_LONG` value can also be used. The maximum resolution of a timestamp is therefore one one-billionth of one second, or one nanosecond. In practice, few databases (or ODBC drivers) actually support this level of precision. (Most notably, SQL Server only supports resolutions of approximately 1/300 of a second.)

The Display Size for a timestamp value is 19 if fractional seconds are not included, or 20 plus

the number of fractional-seconds digits after the decimal point.

The Octet Length for a timestamp value is 16 (the size of the `TIMESTAMP_STRUCT` structure).

Advanced Techniques

Another way to access a timestamp column would be to use the User Defined Type `TIMESTAMP_STRUCT` more directly. The "raw" contents of the structure can be obtained by using the `SQL_ResColStr` function, which returns a string value. That string would then be placed directly into the `SQL_TIMESTAMP` structure using a technique that is appropriate to the programming language you are using. (Exactly the same technique can be used for `SQL_DATE` and `SQL_TIME` columns if the appropriate structures are used instead of `TIMESTAMP_STRUCT`.)

PowerBASIC provides a very easy way to perform this operation, so we will discuss that first. Visual Basic users should also read the PowerBASIC explanation, because it establishes some basic concepts.

If column 10 of a result set contained a timestamp, you could do this with PowerBASIC:

```
Dim DateTime As TIMESTAMP_STRUCT  
  
LSet DateTime = SQL_ResColStr(10)
```

The `SQL_ResColStr` function will return a string, and the PowerBASIC `LSet` operation will place the contents of the string directly into the `TIMESTAMP_STRUCT`. Then your program could access the various elements of `DateTime` by using the UDT's elements. For example, you might access `DateTime.Month` or `DateTime.Seconds`.

IMPORTANT NOTE: If the timestamp column is nullable the `SQL_ResColStr` function can return an empty string. If that happens, the PowerBASIC `LSet` function will fill the structure with space characters, resulting in invalid date-time values like 8224-8224-8224 @ 8224:8224:8224. So code like this should be used if the date-time column is nullable:

```
Dim DateTime As TIMESTAMP_STRUCT  
  
If SQL_ResColNul(10) Then  
    'The column contains a null value  
    LSet DateTime = String$(16,0)  
Else  
    'The column contains a date-time  
    LSet DateTime = SQL_ResColStr(10)  
End If
```

Visual Basic

Visual Basic's `LSet` function will generate a "Type Mismatch" error if you try to use the PowerBASIC code shown above. With Visual Basic, it is necessary to do it this way:

```

Dim DateTime As TIMESTAMP_STRUCT
Dim sString$

If SQL_ResColNul(10) Then
    'The column contains a null value.
    sString$ = String$(Len(DateTime),0)
Else
    'The column contains a date-time.
    sString$ = SQL_ResColStr(10)
End If

'Place the string into the structure...
SQL_StringToType sString$, VarPtr(DateTime), Len(DateTime)

Then your program can use the various elements of DateTime, such as DateTime.Year,
DateTime.Hour, and so on.

See SQL_StringToType.

```

SQL_DATE and SQL_TYPE_DATE

ODBC Version 2 requires the use of `SQL_DATE`.

ODBC Version 3 requires the use of `SQL_TYPE_DATE`.

You must use the appropriate date data type, depending on the capabilities of the ODBC driver that you are using and, if you use `SQL_Initialize` instead of `SQL_Init`, the value of the `IODBCVersion` parameter. (`SQL_Init` automatically specifies ODBC Version 3.)

A `SQL_DATE` or `SQL_TYPE_DATE` column is similar to a `SQL_TIMESTAMP` column, except that it contains a 6-byte `DATE_STRUCT` structure that represents a date *only*. The elements of a date structure are...

```
Type DATE_STRUCT
      Year  As Integer
      Month As Integer
      Day   As Integer
End Type
```

(The data types such as `INTEGER` will vary from file to file, because of differences in the way Visual Basic and PowerBASIC handle certain data types, but the results will be exactly the same.)

The `SQL_ResColDate` function can be used to read and format a `SQL_DATE` column. The `SQL_ResColText` function can also be used. If you are using the SQL Tools Pro Runtime Files you can also use the `SQL_ResColDateTimePart` function.

For an even more flexible way to use a date column, see `SQL_TIMESTAMP` and read the section titled "Advanced Techniques". The same techniques can be used with date columns, but you should use a `DATE_STRUCT` structure instead of a `TIMESTAMP_STRUCT`.

The Display Size for a date value is always 10

The Octet Length for a date value is 6 (the size of the `DATE_STRUCT` structure).

SQL_TIME and SQL_TYPE_TIME

ODBC Version 2 requires the use of `SQL_TIME`.

ODBC Version 3 requires the use of `SQL_TYPE_TIME`.

You must use the appropriate time data type, depending on the capabilities of the ODBC driver that you are using and, if you use `SQL_Initialize` instead of `SQL_Init`, the value of the `IODBCVersion` parameter. (`SQL_Init` automatically specifies ODBC Version 3.)

A `SQL_TIME` or `SQL_TYPE_TIME` column is similar to a `SQL_TIMESTAMP` column, except that it contains a 6-byte `TIME_STRUCT` structure that represents a time *only*. The elements of a time structure are...

```
Type TIME_STRUCT
      Hour   As Integer
      Minute As Integer
      Second As Integer
End Type
```

(The data types such as `INTEGER` will vary from file to file, because of differences in the way Visual Basic and PowerBASIC handle certain data types, but the results will be exactly the same.)

Note that the "fractional seconds" element that is part of a `SQL_TIMESTAMP` column is *not* part of a `SQL_TIME` column. A `SQL_TIMESTAMP` column therefore contains more information than a `SQL_DATE` plus a `SQL_TIME` column.

You can read and format the contents of a time column by using the `SQL_ResColTime` function. The `SQL_ResColText` function can also be used. If you are using the SQL Tools Pro Runtime Files you can also use the `SQL_ResColDateTimePart` function.

For an even more flexible way to use a time column, see `SQL_TIMESTAMP` and read the section titled "Advanced Techniques". The same techniques can be used with time columns, but you should use a `TIME_STRUCT` structure instead of a `TIMESTAMP_STRUCT`.

The Display Size for a time value is 8 if fractional seconds are not included, or 9 plus the number of digits after the decimal point.

The Octet Length for a time value is 6 (the size of the `TIME_STRUCT` structure).

SQL_INTERVAL Data Types

SQL_INTERVAL columns are used to store the *difference* between two dates and/or times.

Many different SQL_INTERVAL column types are listed in the SQL Tools Declaration Files (SQLT_STD.INC, modSQLToolsStd.BAS, etc.), such as DAY_TO_HOUR (the number of days/hours between two date/times), YEAR_TO_MONTH (the number of years/months between two dates), and YEAR (the number of years between two dates).

SQL_INTERVAL columns can be somewhat difficult to use because they are defined differently by the ODBC 2.0 and ODBC 3.x specifications. You will probably notice that the SQL Tools Declaration Files contain two complete sets of numbers, one for the ODBC 2.0 data-type ID numbers and one for the ODBC 3.x numbers. If you know ahead of time that a column contains a SQL_INTERVAL this is not usually a problem, but if you are using SQL Tools Info functions to examine an unfamiliar database, it can be very confusing.

The SQL Tools Declaration Files contain the User Defined Type structures that you will need for SQL_INTERVAL columns. They consist of an 8-byte Year-Month UDT and a 20-byte Day-Second UDT, which are combined into a 20-byte SQL_INTERVAL structure via a UNION statement. If you are using Visual Basic (which does not support UNION) we have provided individual TYPE definitions for each interval type.

SQL_INTERVAL columns are always 26-byte structures, regardless of the type of interval being measured. Your program must access the appropriate structures and elements, based on the type of interval. Because of their complexity (and relative rarity) SQL Tools does not provide functions that interpret or format SQL_INTERVAL structures. You should perform this task with BASIC code.

For a complete description of SQL_INTERVAL columns, we suggest that you consult either the Microsoft ODBC Software Developer Kit or another comprehensive ODBC reference.

The Microsoft ODBC Reference lists the Octet Length of a SQL_INTERVAL value as 34.

SQL_BINARY, SQL_VARBINARY, and SQL_LONGVARBINARY

These data types are virtually identical to `SQL_CHAR`, `SQL_VARCHAR`, and `SQL_LONGVARCHAR`, except that they are intended for "binary" data instead of "string" data. (Some books refer to `SQL_LONGVARBINARY` as `SQL_LONGBINARY`.)

String data traditionally consists of characters that humans can read (A–Z, a–z, 0–9, !@#%&*, etc.) plus a few control characters like Carriage Return and Line Feed. In practice, any ASCII character *with the exception of the string-termination character* `CHR$(0)` can be stored in a `CHAR` column.

Binary columns can store all 256 ASCII characters, including `CHR$(0)`.

Binary columns are remarkably versatile. They can be used to store sounds, pictures, and even entire programs.

A common term for a `SQL_LONGVARBINARY` column is a BLOB, which stands for Binary Large Object. It is just that: a "blob" of binary data, containing virtually anything that you can imagine.

Many different database programs use binary columns for their own internal purposes. For example, when you store a "Form" or a "Report" in a Microsoft Access database, it is actually stored in a special kind of table called a SYSTEM TABLE, in a `SQL_LONGVARBINARY` column.

Binary columns can be used to store User Defined Types, entire numeric arrays, data structures that you design yourself, or just about anything else.

As with `SQL_CHAR` columns, `SQL_BINARY` columns have a *fixed* length that is defined when a database is designed. The usual maximum length is 255 bytes.

As with `SQL_VARCHAR` columns, `SQL_VARBINARY` columns are *variable*-length, with a maximum length (usually 256 bytes) that is defined when a database is defined.

As with `SQL_LONGVARCHAR` columns, the maximum size for a `SQL_LONGVARBINARY` column is defined by the ODBC driver that is used. A common maximum length is 1 gigabyte.

The Display Size and Octet Length of all of the `SQL_BINARY` data types depend on the maximum length that was specified when the database was designed.

Lengths of SQL_CHAR and SQL_BINARY Data Types

The lengths of the various `SQL_CHAR` and `SQL_BINARY` data types (including all of the `VAR` and `LONGVAR` permutations) can vary from database to database and from ODBC driver to ODBC driver.

SQL Tools uses default maximum lengths for these columns that work well with most databases and drivers, but it may be necessary for you to change the defaults to work better in your particular circumstances. The default sizes for all `SQL_CHAR` and `SQL_BINARY` columns can be changed with the `SQL_SetOptionSInt` function.

`SQL_CHAR`, `SQL_VARCHAR`, `SQL_BINARY`, and `SQL_VARBINARY` columns all default to a maximum of 256 bytes. If your databases and ODBC drivers use lengths of 256 characters *or less*, you will not need to change the SQL Tools defaults unless you are trying to optimize an application to use the absolute minimum amount of memory that is possible. (This can be accomplished even more efficiently by manually binding result columns.)

`SQL_LONGVARCHAR`, `SQL_WLONGVARCHAR`, and `SQL_LONGVARBINARY` columns often contain data that is longer than 256 bytes, but SQL Tools uses a default buffer size of 256 to allow you to "preview" the first portions of these columns using the usual `SQL_ResultColumn` functions. It may be desirable for you to change the default to a larger value, but you should be careful not to use values that are too large or your program will use great quantities of memory. The appropriate way to access most `SQL_LONGVARCHAR`, `SQL_WLONGVARCHAR`, and `SQL_LONGVARBINARY` columns is to use the default 256-byte buffer to preview the data (or to unbind LONG columns to disable the preview buffer), and then to use the `SQL_LongResultColumn` function to obtain the actual contents of the column. The `SQL_LongResultColumn` function uses a default "chunk size" of 32k bytes, and can be used repeatedly to obtain data that is longer than 32k. For more information, see Long Columns.

SQL_DEFAULT

This data type can *sometimes* be used when you do not know which SQL Data Type you should use for a function. It means "use the native data type of the column, as defined by the database itself".

The Microsoft ODBC Software Developer Kit both **1)** recommends *against* using this value and **2)** *requires* that it be used under certain circumstances.

Datasource-Dependent Data Types

A Datasource-dependent Data Type is a data type that **1)** is supported by a particular database and **2)** is *completely described* by the database. Database-Specific Data Types are always *based on* the standard SQL Data Types but they are not identical to them. Think of a SQL Data Type as a general description, and a Datasource-dependent Data Type as a complete description.

For example, if a database supports the `SQL_CHAR` data type, it must specify a maximum string length. If it supports a `SQL_DECIMAL` or `SQL_NUMERIC` data type, it must specify the number of digits that will be used after the decimal point. If it supports the `SQL_INTEGER` data type, the length and decimal-digits are pre-defined, but the database must assign a name to the data type (like `INTEGER` or `LONG`) so that certain SQL statements (such as `CREATE TABLE`) can use the names. Every single SQL Data Type requires *some* parameters to be defined.

Many different databases also support "variations" on the standard SQL Data Types.

For example, it is very common for a database to support a data type called `COUNTER`. This is usually a `SQL_INTEGER` column that is not allowed to contain Null values and that is auto-incrementing. That means that the database itself is responsible for inserting unique, usually sequential values into the column, as a means for providing unique row identifiers.

That same database may *also* support a data type called `INTEGER`, which might be a `SQL_INTEGER` column that is nullable and is not auto-incrementing, or non-nullable and non-incrementing... the exact definition will depend on the database.

Another common Datasource-dependent Data Type is `MONEY`, which would (presumably) describe how the database handles monetary values. It might be a `SQL_DECIMAL` value, or a `SQL_INTEGER` value that is used to store "cents" and multiplied times a certain factor to obtain "dollars and cents", or it might be a string value, or a floating-point value... It depends on what *the database designer* decided to use for a "money" column.

Unicode Data Types

For information about specific Unicode Data Types, see `SQL_WCHAR`, `SQL_WVARCHAR`, and `SQL_WLONGVARCHAR`.

There are two basic types of strings in the modern Windows world: ANSI strings and Unicode strings.

(The terms ANSI and ASCII have slightly different meanings, but for the purposes of this discussion you can consider them to be identical. The same is true for the terms Unicode, "wide characters", and "multi-byte characters".)

In an ANSI string, each character is represented by a single byte of data. That is the reason that there are exactly 256 different ASCII characters: a byte (8 bits) can only represent 256 different values, from 0-255. An ANSI string has a length that is the same as its character count. For example, the three-character string "SQL" requires three bytes of storage (memory, disk space, etc.).

In a Unicode string, each character is represented by *two* bytes (one *word*) of data. That means that the Unicode character set contains 65,536 different characters, from 0 to 65,535. The Unicode character set is intended to replace the ASCII character set, so that characters from many different languages can be displayed. If you have a Windows NT, 2000, or XP computer, use the *Start > Programs > Accessories > Character Map* program to select a font like Arial. If you use the *Subset* option, you will see many different pages of up to 256 characters each.

Since each Unicode character requires a word instead of a byte, that means that the representation of the string "SQL" requires *six* bytes of storage. Unicode strings are always twice as long as ANSI strings with the same content.

And that means that if a database contains a Unicode column (`SQL_WCHAR`, `SQL_WVARCHAR`, or `SQL_WLONGVARCHAR`, where the *W* stands for Wide), you must double the amount of storage space that your program provides.

Most database columns that contain strings are not Unicode columns.

The Unicode data types were introduced in the ODBC 3.5 standard. In fact, Unicode support is the most significant difference between ODBC 3.0 and 3.5.

Visual Basic and Unicode

In a Visual Basic program, *all* strings are Unicode strings. When your VB program uses a SQL Tools function that has a string parameter or return value, virtually all of the necessary ANSI-to-Unicode and Unicode-to-ANSI conversions are performed automatically by Visual Basic. (Actually, that is true for *all* DLLs, not just SQL Tools. Visual Basic assumes that all DLLs use ANSI strings, and it performs the necessary Unicode conversions 100% automatically.)

If a database *contains* Unicode data, however, Visual Basic will "double convert" the string and it will be necessary for your program to perform an extra step. (Again, *most* databases do not contain Unicode data, so this is not usually a problem for VB programmers.)

For example, let's say that a database column contains a "normal" ANSI string. When you

use a function like `SQL_ResColStr`, SQL Tools will return an ANSI string. Visual Basic will automatically convert that string into Unicode, and your VB program will never know the difference. Everything will be handled automatically.

But if a database column contains a Unicode string, SQL Tools will *return* a Unicode string to your program. (In other words, SQL Tools will return the appropriate string type, but VB won't recognize it.) Visual Basic will incorrectly assume that it is an ANSI string and it will perform its convert-to-Unicode operation. This will result in a string that contains alternating characters and `CHR$(0)`, and your program will not be able to use the string properly.

The solution to this problem is to un-convert the incorrectly-converted string like this...

```
sResult$ = SQL_ResColStr(1)

sResult$ = StrConv(sResult$, vbFromUnicode)
```

This is somewhat counter-intuitive -- in order to obtain a Unicode string you must use the `vbFromUnicode` constant -- but that is the most reliable way to transfer Unicode strings from a database to a Visual Basic program.

PowerBASIC and Unicode

PowerBASIC does not currently support Unicode strings directly. However, a "normal" PowerBASIC string variable can be used to *store* a Unicode string. You can add them together and manipulate them in many other ways, as long as you remember to work with *pairs* of bytes. For example, to obtain the left-most *character* of a Unicode string you would need to do this:

```
sFirstChar$ = LEFT$(sUnicode$, 2)
```

One more time... *Most databases use ANSI strings, not Unicode strings*, so this is rarely a problem for PowerBASIC users. If you do encounter Unicode strings in a database, we recommend that you investigate the `MultiByteToWideChar` and `WideCharToMultiByte` API functions that are provided by Windows.

SQL_wCHAR

A fixed-length Unicode string. This data type is very similar to `SQL_CHAR`, except that it is a Unicode data type. (The `w` stands for Wide Characters, which is another term for Unicode.)

The length of this data type is specified, on a column-by-column basis, when a database is designed. It is most appropriate for something like a `MiddleInitial` or `SocialSecurityNumber` column, where the length of the data is fixed and is known ahead of time. But even something as seemingly-standard as a telephone number -- which might contain a Country Code or an Extension -- might not work well as a fixed-length string.

Many databases do not support `SQL_wCHAR` values which are longer than 254 characters. (The legal range of string lengths is usually 0–255, but one character is often reserved for a `CHR$(0)` string terminator.)

The Display Size and Octet Length of a `SQL_CHAR` value depend on the length that was specified when the database was designed. (The Octet Length property does not include the byte that is required for the string's null terminator.)

IMPORTANT NOTE: You must always keep in mind that Unicode strings require *two* bytes per character. So a `SQL_wCHAR` column with 10 characters would require 20 bytes, not 10.

SQL_wVARCHAR

A variable-length Unicode string. This data type is very similar to `SQL_VARCHAR`, except that it is a Unicode data type. (The `w` stands for Wide Characters, which is another term for Unicode.)

The maximum length of each `SQL_wVARCHAR` column is specified when a database is designed.

This data type is appropriate for a wide variety of uses, such as Name, Address, and City columns where the length of the data can vary greatly but will not exceed a certain "reasonable" value, such as 255 characters. In fact, most databases limit `SQL_wVARCHAR` values to a maximum of 255 characters, and many do not support more than 254 characters. (The legal range of string lengths is usually 0–255, but one character is often reserved for a `CHR$(0)` string terminator.)

`SQL_wVARCHAR` columns are more efficient than `SQL_wCHAR` columns because they do not waste space in the database if the contents of a column do not fill the available column length.

The Display Size and Octet Length of a `SQL_wVARCHAR` value depend on the maximum length that was specified when the database was designed. (The Octet Length property *does* not include the byte that is required for the string's null terminator.)

IMPORTANT NOTE: You must always keep in mind that Unicode strings require *two* bytes per character. So a `SQL_wVARCHAR` column with a maximum length of 10 characters would require 20 bytes, not 10.

SQL_WLONGVARCHAR

A "long" variable-length Unicode string. This data type is very similar to `SQL_LONGVARCHAR`, except that it is a Unicode data type. (The `w` stands for Wide Characters, which is another term for Unicode.)

The definition of "long" depends on what you're doing. In most cases, ODBC considers strings that are *potentially* over 255 characters to be "long".

The maximum length of a `SQL_WLONGVARCHAR` column is defined by the ODBC driver that is used. A common maximum length is 1,073,741,824 characters (1 gigabyte).

The most common use of this data type is a "memo" field that allows the user to enter strings of virtually any length.

The Display Size and Octet Length of a `SQL_WLONGVARCHAR` value depend on the maximum length that was specified when the database was designed. (The Octet Length property does not include the byte that is required for the string's null terminator.)

IMPORTANT NOTE: You must always keep in mind that Unicode strings require *two* bytes per character. So a `SQL_WLONGVARCHAR` column with a maximum length of 1000 characters would require 2000 bytes, not 1000.

SQL Data Type "Properties"

Each SQL Data Type has a set of "properties". Some properties (such as the buffer size that is required to hold a `SQL_INTEGER` value) never change. Other properties (such as the prefix that must be used for literal `SQL_VARCHAR` value) can be defined differently by various ODBC drivers. And some properties (such as the length of a `SQL_CHAR` column) are defined when a database is designed.

Most of the time you won't have to worry about a data type's properties. After all, by the time your SQL Tools program opens a database, nearly all of the properties have been determined and there's nothing you can do about them. But sometimes you will need to find out the value of a data type property. Some ODBC functions (and therefore some SQL Tools functions) require that certain properties be used as parameters when the functions are used.

You can use the `SQL_DBDataTypeCount` function to find out how many different data types a database supports, and then you can use the `SQL_DBDataTypeInfoStr` and `SQL_DBDataTypeInfoSInt` functions to obtain a data type's properties.

The various types of columns (table columns, result columns, Stored Procedure columns, AutoColumns, etc.) all have specific lists of data type properties that they use, but there are six common properties with which you should become familiar:

Concise Type

Buffer Size

Transfer Octet Size

Num Prec Radix

Display Size

Decimal Digits (Precision)

Concise Type

This data type property is usually the data type *itself*, such as `SQL_CHAR` or `SQL_INTEGER`.

The only time that "concise" data types get complicated is when date-times are involved. For example, a particular value might have a concise data type of `SQL_INTERVAL`, and a "Date-Time Subcode" of `ODBC3_INTERVAL_YEAR` to describe the Interval in more detail.

If a Data Type is not "concise" then something like `ODBC3_INTERVAL_YEAR` will be specified, and `SQL_INTERVAL` will be *implied*.

Buffer Size

This data type property is the length of the memory buffer that is required to hold a value.

Generally speaking, the buffer size is defined by the type of `BAS_` variable that you use for a value.

Also see Transfer Octet Size.

Transfer Octet Length

In some cases you will need to be concerned with the "octet length" of a data type, which is the buffer size that *would* be used for a value if the `SQL_DEFAULT` data type was used.

"*Octet*" refers to a byte, which (of course) has 8 bits. The "transfer octet length" is the number of 8-*bit* blocks of memory that are required for a value.

Also see Buffer Size.

Num Prec Radix

Frankly, "Num Prec Radix" is an obscure term that is not defined very well by the Microsoft ODBC Software Developer Kit.

This data type property is very important because it determines how two *other* properties are interpreted.

This property will always have a numeric value of ten (10), two (2), or zero (0).

If this value is 10, the Display Size and Decimal Digits properties refer to the number of *digits* that are allowed for the value.

For example, a `SQL_DECIMAL(12,5)` column would return a `NUM_PREC_RADIX` value of **10**, a `DISPLAY_SIZE` value of 12, and a `DECIMAL_DIGITS` value of 5.

A `SQL_FLOAT` column, on the other hand, could return a `NUM_PREC_RADIX` value of **10**, a `DISPLAY_SIZE` value of 15, and a `DECIMAL_DIGITS` value of zero (0).

If this value is 2, the Display Size and Decimal Digits properties refer to the number of *bits* that are allowed for the value.

For example, a `SQL_FLOAT` column could have a `NUM_PREC_RADIX` value of **2**, a `COLUMN_SIZE` value of 53, and a `DECIMAL_DIGITS` value of zero (0). (Or it could have the values shown for a Num Prec Radix of **10**, just above.)

If this value is zero (0), `NUM_PREC_RADIX` is not applicable to the data type.

Display Size

The exact meaning of the Display Size property is dependent on the value of the Num Prec Radix property.

The Display Size property *usually* refers to the number of *characters* that a user interface would have to display in order to show the entire column in character form. For example, an unsigned `SQL_TINYINT` column (which can contain values from 0 to 255) would have a display size of 3, because the maximum width required to display a value is 3 characters, for "255". A signed `SQL_TINYINT` column, on the other hand, can contain values between --128 and +127, so it would have a display size of 4 because it might be necessary to display the value "-128".

Do not confuse the display size value with the size of the memory buffer that is required to store a column's value. For example, a `SQL_TINYINT` column requires a one-byte buffer, but requires three or four characters to *display*.

Decimal Digits

The exact meaning of the Decimal Digits property is dependent on the value of the Num Prec Radix property.

"Decimal Digits" or "Precision" *usually* refers to the maximum number of digits that a value can have to the right of the decimal point.

For `SQL_NUMERIC` and `SQL_DECIMAL` values, this is the `Y` number in the `DECIMAL(X, Y)` notation.

For date-time data values, Decimal Digits refers to the number of digits in the *fractional-seconds* portion of the value.

For all other data types (including floating point types) the Decimal Digits value is considered to be zero (0).

BASIC Data Types

In addition to becoming familiar with the SQL Data Types, you should review all of the BASIC data types shown below. This document provides only a brief summary of each BASIC data type; for complete information, please refer to your BASIC language's documentation.

Each data type below has a BUFFER SIZE notation. This indicates the amount of memory that the data type requires for the storage of one variable of that data type. The BUFFER SIZE value is used by many different SQL Tools functions.

BAS_ASCIIIZ

This is a PowerBASIC ASCIIIZ string or a Visual Basic Fixed-Length string. (ASCIIIZ is the more "generic" term because it is also used by C and Delphi, so that is what SQL Tools calls it.) A `BAS_ASCIIIZ` is always a string with a pre-defined *maximum* length, with a null-terminator (`CHR$(0)`) that marks the end of the string's *current* value. BUFFER SIZE: Depends on the `Dim` statement that is used to create the variable. For example, to create a 100-byte string in Visual Basic you would use `Dim sMyString As String * 100`. In PowerBASIC you would use `Dim lpzMyString As ASCIIIZ * 100`. The default buffer size for a `BAS_ASCIIIZ` value is 256 bytes, which equals 255 characters of data plus one (1) byte for the null terminator.

BAS_STRING

This is the BASIC "dynamic string" data type. It is very similar to a `SQL_VARCHAR`, except that it is not null-terminated. BUFFER SIZE: Depends on the longest string that can be stored in the variable, which is a function of the `SQL_VARCHAR` value. For example, a `SQL_VARCHAR` column has a default maximum length of 256 characters. Therefore a `BAS_STRING` variable would usually use a 256-byte buffer, even though a PowerBASIC or Visual Basic dynamic string can be up to 2 gigabytes in length. (See Long Columns for more information.)

Please note that the `BAS_STRING` data type can be used to store binary data, such as those found in `SQL_BINARY` columns. Unlike `SQL_CHAR` values, `BAS_STRING` values are allowed to contain `CHR$(0)`.

BAS_LONG

This is the BASIC LONG INTEGER data type, which can hold a signed numeric value between `-2,147,483,648` and `+2,147,483,647`. This is the most efficient 32-bit BASIC data type. It corresponds to a *signed* `SQL_INTEGER` value. BUFFER SIZE: 4.

BAS_DWORD

This is the PowerBASIC DWORD data type, which can hold an unsigned numeric value between 0 and `+4,294,967,295`. It corresponds to an *unsigned* `SQL_INTEGER` value. (Visual Basic does not support unsigned integers. See Signed and Unsigned Integers for more information.) BUFFER SIZE: 4.

BAS_INTEGER

This is the BASIC INTEGER data type, which can hold a signed numeric value between $-32,768$ and $+32,767$. It corresponds to a *signed* SQL_SMALLINT value. (Please see Terminology Differences for a discussion about avoiding confusion between BAS_INTEGER and SQL_INTEGER, which are *not* the same thing.)
BUFFER SIZE: 2.

BAS_WORD

This is the PowerBASIC WORD data type, which can hold an *unsigned* numeric value between 0 and $+65,535$. It corresponds to an *unsigned* SQL_SMALLINT value. (Visual Basic does not support unsigned integers. See Signed and Unsigned Integers for more information.)
BUFFER SIZE: 2.

BAS_BYTE

This is the BASIC BYTE data type, which can hold an *unsigned* numeric value between 0 and $+255$. It corresponds to an *unsigned* SQL_TINYINT value. BUFFER SIZE: 1.

BAS_QUAD

This is the PowerBASIC QUAD Integer data type, which can hold a signed numeric value between plus-and-minus 9.22×10^{18} . Visual Basic does not support QUAD Integers. BUFFER SIZE: 8

BAS_SINGLE

This is the BASIC SINGLE data type, which can hold a signed *floating-point* value in the SQL_REAL range. BUFFER SIZE: 4.

BAS_DOUBLE

This is the BASIC DOUBLE data type, which can hold a signed *floating-point* value in the SQL_DOUBLE range. BUFFER SIZE: 8.

Please note that there are several places where the BAS_ and SQL_ data types do not overlap. For example, PowerBASIC supports the QUAD data type, but there is no corresponding *numeric* SQL data type. (SQL_BIGINT comes close, but it is a *string* data type, and it can be either signed or unsigned.) And SQL supports both unsigned SQL_TINYINT values (0 to $+255$) and signed SQL_TINYINT values (-128 to $+127$), while Visual BASIC and PowerBASIC support only unsigned BAS_BYTE values.

Please also note that BASIC *can* be used to construct User Defined Type data structures that can be used for virtually *any* SQL data type. For example, the SQL Tools Declaration Files (SQLT_STD.INC, modSQLToolsPro.BAS, etc.) contain BASIC User Defined Types that can be used for SQL_TIMESTAMP values.

And you could easily convert a BAS_BYTE value into a *signed* number in the range -128 to $+127$ by using a BAS_LONG variable (which is signed) and subtracting 128 from any value over 127.

SQL Statements

A complete discussion of SQL Statements is well beyond the scope of this document. Entire books have been written on the topic! This basic information, and the information in Appendix A, is provided as background material only. You should acquire reference materials related to the particular "flavor" of SQL that your ODBC driver accepts.

A SQL Statement is a "command" that you send to a SQL database, to tell it to do something.

For example, in order to read data from a database you would use the `SELECT` statement. If a database contains a table called `MYTABLE`, and if you want the database to give you all of the rows and all of the columns, you would use the SQL statement...

```
SELECT * FROM MYTABLE.
```

Here are some examples of commonly used SQL statements:

```
SELECT -- retrieves one or more rows from a database
```

```
UPDATE -- changes the values in one or more rows
```

```
INSERT -- adds one or more rows
```

```
DELETE -- deletes one or more rows
```

For more information, see Appendix A: SQL Statement Syntax.

Execution of SQL Statements

The processing of most SQL statements is basically an "interpreted" operation. The ODBC driver must analyze a string that contains a SQL statement and then "compile" the statement into an executable form. This first step is called "preparation" and is roughly equivalent to the steps that are taken by a BASIC interpreter like QBASIC to convert source code into executable code at run time. The actual "execution" of a SQL statement is a separate process.

SQL statements are either prepared or executed, or both, by using the `SQL_Stmt` function. (It is also possible to execute SQL statements "asynchronously" by using the `SQL_AsyncStmt` function, which is very similar to `SQL_Stmt`.)

Here is the basic syntax for `SQL_Stmt`:

```
SQL_Stmt lOperation&, sStatement$
```

The `sStatement`\$ variable represents a SQL statement such as...

```
SELECT * FROM MYTABLE.
```

The parameter `lOperation`& should always be one of the following constants:

`SQL_STMT_PREPARE` tells the `SQL_Stmt` function to prepare the SQL statement in `sStatement`\$ but not to execute it. The alias `PREP` is also recognized.

`SQL_STMT_EXECUTE` tells the `SQL_Stmt` function to execute a SQL statement that was previously prepared. The alias `EXEC` is also recognized.

`SQL_STMT_IMMEDIATE` tells the `SQL_Stmt` function to prepare and then immediately execute a SQL statement, as if it were a one-step process. The alias `IMMED` is also recognized, as are `SQL_STMT_EXECDIRECT` and `DIRECT`, which are based on the original ODBC terminology.

Most programs will use `SQL_STMT_IMMEDIATE` most of the time.

The major advantage of using `SQL_STMT_PREPARE` and `SQL_STMT_EXECUTE` as separate steps is that it allows statement input parameters to be bound to the statement between the two steps. A SQL statement can be prepared once, bound to one or more parameter variables, and then executed many times with different parameter values. If a SQL statement is to be executed repeatedly with different parameter values it is much more efficient to use this two-step process than to use `SQL_STMT_IMMEDIATE` to prepare/execute the statement strings over and over.

If you use the `SQL_STMT_PREPARE` or `SQL_STMT_IMMEDIATE` option, the `sStatement`\$ parameter must contain a valid SQL statement.

If you use the `SQL_STMT_EXECUTE` option, the `sStatement`\$ string is optional. If you use an empty string for `sStatement`\$, SQL Tools will assume that you mean "execute the statement that was just prepared". If you have not previously prepared a statement, an error message will be generated. If you *do* pass a `sStatement`\$ string to the `SQL_Stmt` function when the `SQL_STMT_EXECUTE` option is used, SQL Tools will check to make sure that it is the same statement string that was previously prepared. If you are writing complex programs with

many different statements that can be prepared and executed, this is a good double-check to make sure that your program is executing the statement that you think it is. If the strings do not match, an error message will be generated.

Also see Asynchronous Execution of SQL Statements.

Asynchronous Execution of SQL Statements

Most programs use the `SQL_Stmt` or `SQL_Statement` function to prepare and/or execute SQL statements. When that is done, your program "pauses" until the SQL statement generates a result.

That's not *usually* a problem but, depending on your program, it is not always desirable. For example, most GUI-style programs need to continuously update their screens, and because it can take seconds, minutes, or even *hours* for some SQL statements to finish, you may wish to execute a SQL statement "asynchronously". That term means "in the background, while my main program continues to run". Asynchronous execution can allow your program to do many different things while waiting, such as checking to see if the user has clicked a Cancel button, and/or displaying a "WORKING... PLEASE WAIT" animation.

The SQL Tools "async execution" functions are intended primarily for use in Visual Basic programs (because VB does not support multi-threading) but they can also be used with PowerBASIC and other languages. Generally speaking, if all you want to do is execute a SQL statement asynchronously, the SQL Tools async functions are easier to use than PowerBASIC's `THREAD` functions.

See `SQL_AsyncStmt` for a complete discussion of asynchronous SQL statements.

Also see `SQL_AsyncStatus`, `SQL_AsyncErrors`, and `SQL_StmtCancel`.

For a discussion of another (more complex) method of performing asynchronous database operations, see Multi-Threaded Programs.

SQL Statement Mode

If you want to get the most from a SQL statement, there's more to it than just using the `SQL_Stmt` function to tell the database what to do. SQL Tools provides a wide variety of options that you can use to tell a database *how* you want it to execute SQL statements.

The `SQL_StmtMode` function is used to "set up" a SQL statement even before you use `SQL_Stmt` to `SQL_STMT_PREPARE` or `SQL_STMT_EXECUTE` it. Actually, that's a very important distinction:

The `SQL_StmtMode` function can only be used to change the way future SQL statements will be prepared and/or executed. If you make a mistake and use the `SQL_StmtMode` function after you use `SQL_Stmt`, an error message will be generated and the new setting won't take effect until the next time you use `SQL_Stmt`. Note that it is not possible to `SQL_STMT_PREPARE` a statement, then change the Statement Mode, and then `SQL_STMT_EXECUTE` the statement. All mode changes must be made before a SQL statement is used for the first time.

If you have already executed a SQL statement and need to change the statement mode for future statements, you should use the `SQL_CloseStmt` function to explicitly close the old statement, and then use `SQL_StmtMode` to change the mode before you use `SQL_Stmt` again. (Closing statements is not usually a necessary step when you're using SQL Tools, because it is handled automatically.)

One of the more common uses for `SQL_StmtMode` is to tell the ODBC driver the maximum number of rows of data that you want it to return. For example, executing a simple statement like `SELECT * FROM MYTABLE` can overwhelm a database system or a network. If the table is very large, a *huge* volume of data can be returned by a SQL statement such as that one, and in some cases it can overload your server and/or network.

So you could use the `SQL_StmtMode` function like this...

```
SQL_OpenDB "MYDATA.DSN"
```

```
SQL_StmtMode STMT_ATTR_MAX_RESULT_ROWS, 10
```

```
SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM MYTABLE"
```

... and the ODBC driver would only return 10 rows of data to your program, even if the SQL statement would normally return thousands or millions of rows.

Other common uses of `SQL_StmtMode` include setting the `STMT_ATTR_QUERY_TIMEOUT` value to limit the amount of time that the driver will spend processing a request, and changing various aspects of the way the driver "scrolls" through result sets, to make certain types of requests more efficient.

You should keep in mind that while `SQL_StmtMode` settings are "sticky" -- once they are set, future statements will use them until the value is changed again -- but they are not "global". *Statement mode changes do not apply to all Statement Numbers.* Here is an example of what we mean...

Let's say that your program is using the "normal" settings of Database 1, Statement 1 (see) and you use `SQL_StmtMode` to change the `STMT_ATTR_MAX_RESULT_ROWS` value to 10.

Future SQL statements that use Database 1, Statement 1 will use the new setting, but statements that use different Database and/or Statement number will not see the new setting. They will use the default settings.

To reiterate this important point, the `SQL_StmtMode` function sets the statement mode for each Connection Number and Statement Number *individually*.

This may seem like it complicates things unnecessarily, but it allows you to set up different statements to perform differently. For example, you might want Statement Number 1 to always return all rows, and Statement Number 2 to only return a single row of data, regardless of the request.

See `SQL_StmtMode` for a complete list of all of the Statement Mode options.

Binding Statement Input Parameters

Please note: This is probably the most complex single topic in this document. It is recommended for advanced SQL Tools users only. If you are just learning to use SQL Tools, we suggest that you read the first few paragraphs of this section, to introduce the basic concepts surrounding bound statement input parameters, and then skip ahead to the next major section of this document (Result Sets).

You can think of a "bound statement input parameter" as a *variable* that is embedded directly into a SQL statement. A "placeholder" in a SQL statement is linked directly to a BASIC variable that your program provides, and by changing the value of the variable you can effectively change the SQL statement.

They can be somewhat difficult to set up, especially at first, but there are many advantages to using bound statement input parameters:

- 1) Bound statement input parameters make it possible to use `SQL_Stmt (SQL_STMT_PREPARE)` to prepare a statement once, and then, after binding the parameter(s), you can use `SQL_Stmt (SQL_STMT_EXECUTE)` many different times, with different parameter values. This is usually much faster than re-building a string-based SQL statement and using `SQL_Stmt (SQL_STMT_IMMEDIATE)` over and over.
- 2) Bound statement input parameters can make it much easier for your program to "construct" a SQL statement at runtime. Rather than writing complicated text-parsing routines to create a string-based statement on the fly (i.e. with certain values that are determined at runtime), you can hard-code the "static" parts of the statement and use bound parameters and BASIC variables to change the statement's values.
- 3) Bound statement input parameters are faster and more efficient than text-based parameters, especially if a parameter's value is numeric or binary. If you include a numeric parameter in a string-based SQL statement, your program must first use `STR$` or `FORMAT$` to convert the number into a string, and then the ODBC driver must locate the part of the SQL statement that represents the number, and convert it back into a numeric value so that it can be used. And all of that takes time. Providing numeric values via numeric variables is better, faster, and more efficient.
- 4) Stored Procedures can contain parameters that *must* be bound before the procedure can be executed.
- 5) Bound parameters are very useful when you want to include a value in a SQL statement that is difficult to express in "text" form. For example, it's easy to do this...

```
SELECT * FROM ADDRESSBOOK WHERE ZIPCODE = 48070
```

...but how would you create a SQL statement that used `WHERE` to search for a complex binary value in a column called `BINARYIMAGE`? You could manually type in an extremely long "literal" value, and risk making a mistake, or you could use a bound statement input parameter.

(Please note: There are actually three different types of bound statement parameters, but "output parameters" and "input-output parameters" are only used by Stored Procedures. For the sake of simplicity, the rest of this discussion will refer to "bound statement input parameters" simply as "bound parameters".)

Most simple SQL statements are executed with the `SQL_Stmt (SQL_STMT_IMMEDIATE)`

function, which automatically performs two different steps. **1)** The statement is "prepared", i.e. converted from a string like `SELECT * FROM MYTABLE` into an executable program, and then **2)** the program is executed. (For more information about this process, see `SQL_Stmt`.)

It is also possible to use the `SQL_Stmt` function twice, to perform the `SQL_STMT_PREPARE` and `SQL_STMT_EXECUTE` steps one at a time, and to perform parameter binding operations in between the steps.

The SQL statement "parameter placeholder" is the `?` (question mark) character. For example:

```
SELECT CITY FROM ADDRESSBOOK WHERE ZIPCODE = ?
```

If you attempt to execute that statement with `SQL_Stmt(SQL_STMT_IMMEDIATE)` you will receive an ODBC Error Message with "parameter" in it, such as "Wrong number of parameters". The exact error message will vary from driver to driver.

However, if you use `SQL_Stmt(SQL_STMT_PREPARE)` to simply *prepare* that same statement, without executing it, no error will be generated.

And if you use the `SQL_StmtParamCount` function *after* the statement has been prepared, it will return a value of one (1), to indicate that the driver detected one parameter placeholder. That means that the ODBC driver will not execute the statement until you have provided a value for that placeholder.

Bound parameters are *not* allowed in the column-list that is to be returned by a `SELECT` statement, like this...

```
SELECT ? FROM ADDRESSBOOK
```

...and you may *not* use bound parameters for both of the operands of a comparison, like this...

```
SELECT CITY FROM ADDRESSBOOK WHERE ? = ?
```

And finally, bound parameters cannot usually be used in statements that change a table's design, like `CREATE TABLE` and `DROP TABLE`. If you stick to using bound parameters in `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements, you shouldn't have any problems. (Some ODBC drivers *may* allow bound parameters to be used in other types of statements.)

Because the process of binding a statement parameter requires information that is fairly complex, and because the ODBC "Info" functions are relatively slow, SQL Tools does not provide an "autobinding" function for parameters (as it does for result columns). After all, the primary advantage of bound parameters is *speed*, and if SQL Tools used ODBC Info functions to look up all of the required information at runtime, the speed advantage would be greatly diminished.

The process of manually binding a statement parameter is fairly complex, but it is surprisingly similar to the process of manually binding a column of a result set. If you are not already familiar with that process, it would be a good idea for you to pause here to review Manual Column Binding, and to experiment with the manual binding of result columns. The rest of this discussion will assume that you are familiar with the basic concepts of memory buffers, Indicators, and the general process of binding. You should also be familiar with the various BASIC Data Types and SQL Data Types, and the various properties that SQL Data can have, such as "decimal digits" and "display sizes".

Binding Numeric Parameters

The first step in binding any statement parameter is to determine the parameter number.

It's easy: parameters are always numbered starting with one (1). In other words, the first question mark in a SQL statement is parameter number one, the second is parameter number two, and so on.

```
SELECT CITY FROM ADDRESSBOOK WHERE ZIPCODE = ?
```

Our example statement only uses one bound parameter, so we will be using the number 1 for the Parameter Number.

Next we have to make another relatively simple decision. Is the parameter an Input Parameter, an Output Parameter, or an Input-Output Parameter? Since we are trying to send a value *to* the SQL statement, this is clearly an Input Parameter. It provides *input* to the SQL statement. (Only Stored Procedures use the other two types of bound parameters. For more information, see Stored Procedures.)

The next step in manually binding a statement parameter is to figure out which SQL Data Type the placeholder represents. It wouldn't make much sense to use a value like "Smith" or "January 1, 2000" in the example above, because our imaginary ZIPCODE column is a numeric column that would never contain string or date values.

If you're not sure which SQL Data Type to use for a parameter, you can use two different SQL Tools functions to determine the appropriate type. The first function requires that you write a little more code than the second, but it always works. The second function is somewhat easier, but *it is not supported by all ODBC drivers*.

Method 1: SQL_TblColInfoSInt

Assuming that a database containing a table called ADDRESSBOOK is already open, and that it contains a column called ZIPCODE...

```
'get the Table Number for ADDRESSBOOK:
lTblNo& = SQL_TblNumber("ADDRESSBOOK")

'get the Column Number for ZIPCODE:
lColNo& = SQL_TblColNumber(lTableNumber&,"ZIPCODE")

'get the data type of the ADDRESSBOOK/ZIPCODE column:
lDataType& = SQL_TblColInfoSInt(lTblNo&,lColNo&,TBLCOL_DATA_TYPE)
```

While you're at it, you're going to be needing three other pieces of information about the ZIPCODE column.

```
lDigits& = SQL_TblColInfoSInt(lTblNo&,lColNo&,TBLCOL_DECIMAL_DIGITS)

lBuffLen& = SQL_TblColInfoSInt(lTblNo&,lColNo&,TBLCOL_BUFFER_LENGTH)

lSize& = SQL_TblColInfoSInt(lTblNo&,lColNo&,TBLCOL_DISPLAY_SIZE)
```

Method 2: SQL_ParamInfoUInt

You can determine whether or not this method will work by examining the result of this function:

```
lResult& = SQL_FuncAvail(API_SQLDESCRIBEPARAM)
```

If it returns False (0), then your ODBC driver does *not* support it and you *cannot* use method 2. If it returns True (-1), you can use Method 2.

If your ODBC driver supports it, you can use the following code to obtain the necessary values for parameter number 1:

```
lDataType& = SQL_ParamInfoUInt(1,PARAM_DATA_TYPE)
```

```
lSize& = SQL_ParamInfoUInt(1,PARAM_SIZE)
```

```
lDigits& = SQL_ParamInfoUInt(1,PARAM_DIGITS)
```

If you use this method, you should determine the lBuffLen& value by consulting this document and/or your BASIC documentation, to determine the length of the buffer that is required for an lDataType& column. (More about this in a moment.)

Both Methods

We should emphasize that we are writing "test code" here, to obtain some numeric values that will be necessary for the final program. You would (probably) not actually use the Method 1 or Method 2 code above in your *finished* program.

For this example, let's assume that the lDataType& value that is returned by the code above is four (4). According to the SQL Tools Declaration Files (SQLT_STD.INC, modSQLToolsPro.BAS, etc.) that corresponds to a SQL Data Type of SQL_INTEGER, which makes sense for a numeric column. (If the data type didn't seem to make sense, we would re-check our test code to make sure we were obtaining the correct value.)

And let's say that the lDispSize& value is ten (10). That simply means that a text column that is ten characters wide would be required to display the largest possible SQL_INTEGER value that the ZIPCODE column can hold. Ten is a perfectly normal "display size" for a SQL_INTEGER column, even though a real Zip Code would never require ten columns to display.

The lDigits& value would be zero (0), because a SQL_INTEGER column is not a floating point column, so there are zero "digits to the right of the decimal point".

Finally, the lBuffLen& value would almost certainly be four (4), because all SQL_INTEGER columns require a four-byte buffer. (See BASIC Data Types for more information.)

When you become familiar with the process of binding statement parameters, you will often be able to make educated guesses about these values, and skip the test-code step.

Choosing a Variable Type

The next step in binding the `ZIPCODE` parameter is to decide which type of BASIC variable you want to use to represent the value.

You can always safely choose a numeric variable type, but if you are going to use a `BAS_ASCII` fixed-length string or a `BAS_STRING` dynamic string (\$) variable, make sure that you read this *entire* section. Some very important warnings regarding strings are included near the end.

The best choice would be a `BAS_` data type that corresponds to `SQL_INTEGER`, which would be `BAS_LONG` if the value was a signed integer, or `BAS_DWORD` if it was unsigned. The U.S. Postal Service has not yet begun assigning "signed zip codes" (as in "my zip code is negative 48070"), so `BAS_DWORD` would seem to be the logical choice. But actually, you have some leeway when choosing the `BAS_` data type. Since the `BAS_LONG` data type is the most efficient BASIC data type, and since the largest Zip Code value is well within the positive range of `BAS_LONG` variables, we're going to use `BAS_LONG`.

Actually, you have a *lot* of options when choosing a `BAS_` variable type for parameter binding. As a matter of fact, if you follow the special instructions below you could even use a *string* variable. Most ODBC drivers automatically perform "reasonable" data-type conversions, so binding an `ASCII` string variable that contained "90210" would be basically the same thing as binding a numeric parameter that contained the value 90210. The ODBC driver will, of course, take a split-second to perform the string-to-numeric conversion, and it may be faster for your program to use the BASIC `VAL` function to convert a string into a numeric value, but the choice is yours. The data-type conversions that are considered "reasonable" vary from driver to driver, but most conversions are supported by most drivers. If you try to do something "unreasonable" like using a `SQL_TIMESTAMP` to represent a `SQL_DOUBLE` floating-point value, it will be rejected by the driver and an Error Message will be generated.

Again, for this example we've chosen a `BAS_LONG` variable for the `ZIPCODE` parameter.

The final step in getting ready to bind the `ZIPCODE` parameter is to create the buffers for the data and the Indicator. If you have reviewed Manual Result Column Binding, you should be familiar with creating buffers, and with Indicators.

We are going to use two `BAS_LONG` variables, one for the data (the actual Zip Code) and one for the parameter's Indicator. We'll call the first one `lZipCode`, and the second `lZCInd` (short for Zip Code Indicator).

Putting It All Together

Now that we have accumulated all of the information we need, we can construct the source code that we need to bind the `ZIPCODE` parameter. The following line uses constants and the variable names from the test code above to make it easier to read, but you could also use the literal numeric values that correspond to the constants and variables. And of course you can make up your own variable names.

```

lResult& = SQL_BindParam(1, _
                        SQL_PARAM_INPUT, _
                        BAS_LONG, _
                        SQL_INTEGER, _
                        lDispSize&, _
                        lDigits&, _
                        VarPtr(lZipCode&), _
                        4, _
                        lZCInd&)

```

Let's review those values one by one. The first "1" means that we are binding parameter number 1. `SQL_PARAM_INPUT` means that parameter number 1 is an Input Parameter. `BAS_LONG` means that we are going to use a BASIC LONG INTEGER for the parameter data. `SQL_INTEGER` means that we determined that the `ZIPCODE` column contains a `SQL_INTEGER` value, and the `lDispSize&` and `lDigits&` values are appropriate for a `SQL_INTEGER` column.

IMPORTANT NOTE: The next parameter must be `VarPtr(something)` because the `SQL_BindParam` function requires a *memory pointer* to the first byte of the data buffer. Remember: the third-to-last parameter of `SQL_BindParam` is called `lPointerToBuffer&` and, just as with Manual Column Binding, you must provide a value from the BASIC `VarPtr` function. (Visual Basic has an *undocumented* `VarPtr` function. See MSDN article Q199824.)

VERY IMPORTANT NOTE: If you are using a `BAS_STRING` dynamic string (\$) variable for the parameter's buffer, you should read [Binding Dynamic String/Binary Parameters](#) and then use `StrPtr` instead of `VarPtr`.

VERY IMPORTANT NOTE: Some versions of PowerBASIC have restrictions against using `VARPTR` with register variables. Unless you are certain that this will not be a problem, we recommend the use of `#REGISTER NONE` to disable the automatic use of register variables in PowerBASIC programs that require the use of the `VARPTR` function.

The second-to-last parameter is "4" because all `BAS_LONG` variables required 4 bytes of memory. For information, see BASIC data types.

Finally, just as with Manual Result Column Binding, the `lZCInd&` variable is always a `BAS_LONG` variable that is passed "normally". Do *not* use `VarPtr` or `StrPtr`.

That's it. (That's a lot of parameters, but it wasn't *really* that hard, was it?)

When that source code is executed, it will bind the ? placeholder in the SQL statement to the `lZipCode&` and `lZCInd&` variables.

Sample Program

```
'(Open the database here.)

'prepare the SQL statement that contains the "?" marker:
sStmt$ = "SELECT CITY FROM ADDRESSBOOK WHERE ZIPCODE = ?"
SQL_Stmt(SQL_STMT_PREPARE,sStmt$)

'bind the parameter:
lResult& = SQL_BindParam(1, _
                        SQL_PARAM_INPUT, _
                        BAS_LONG, _
                        SQL_INTEGER, _
                        lDispSize&, _
                        lDigits&, _
                        VarPtr(lZipCode&), _
                        4, _
                        lZCInd&)

'set the parameter value
lZipCode& = 48070

'set the Indicator value
lZCInd&    = SQL_NUMERIC_DATA

SQL_Stmt(SQL_STMT_EXECUTE,sStmt$)

SQL_Fetch NEXT_ROW

'(Use the result set here.)
```

Of course, the best thing about a bound parameter is that you can `SQL_STMT_EXECUTE` the statement many times without using the time-consuming `SQL_STMT_PREPARE` step again, like this...

```
lZCInd&    = SQL_NUMERIC_DATA

For lZipCode& = 48070 To 48079
    SQL_Stmt(SQL_STMT_EXECUTE,sStmt$)
    '(fetch and use the result set here)
Next
```


Setting a Bound Parameter to the Null Value

If you want to set a bound parameter to the Null value, you must assign the value `SQL_NULL_DATA` (negative one) to the Indicator variable, *not* to the data variable. In the example above, doing this:

```
lZipCode& = 0
lZCInd&   = SQL_NULL_DATA

SQL_Stmt (SQL_STMT_EXECUTE, sStmt$)

'(fetch and use the result set)
```

...would do the same thing as executing the following SQL statement:

```
SELECT CITY FROM ADDRESSBOOK WHERE ZIPCODE = NULL
```

It's always a good idea to set both the data value and the Indicator value at the same time, to avoid (for example) accidentally leaving an Indicator variable set to `SQL_NULL_DATA` instead of `SQL_NUMERIC_DATA`.

Binding Fixed-Length String/Binary Parameters

If you are going to use a `BAS_ASCII` variable for bound string parameters you must set the value of the Indicator variable to equal the number of *characters* in the current string value (instead of using `SQL_NUMERIC_DATA` or `SQL_NULL_DATA`).

For example, if you had bound an `ASCII` variable called `lpzZipCode` to the example statement above, you would be required to do this:

```
Visual Basic: Dim lpzZipCode As String * 5
PowerBASIC:  Dim lpzZipCode As ASCII * 5
```

```
lpzZipCode = "48070"
lzcInd&    = 5
```

If the values that are being assigned to a bound parameter are not always the same length, you can use the BASIC `LEN` function to obtain a value for the Indicator. For example...

```
Visual Basic: Dim lpzLastName As String * 32
PowerBASIC:  Dim lpzLastName As ASCII * 32
```

```
lpzLastName = "Smith"
lzcInd&     = Len(lpzLastName)
```

REMEMBER: You must always set the Indicator to the appropriate "length" value if the parameter is either a string or a binary parameter (`SQL_CHAR`, `SQL_VARCHAR`, `SQL_LONGVARCHAR`, `SQL_WCHAR`, `SQL_WVARCHAR`, `SQL_WLONGVARCHAR`, `SQL_BINARY`, `SQL_VARBINARY`, or `SQL_LONGVARBINARY`. Otherwise, the Indicator value should be set to `SQL_NUMERIC_DATA`.)

Binding Dynamic String/Binary Parameters

If you are going to use a BASIC "dynamic string" (`BAS_STRING`) variable for a bound string parameter, there is one additional factor that you *must* consider. Failure to heed these warnings will result in Application Errors.

First, in order to obtain a memory pointer to a dynamic string variable, you must use the BASIC `StrPtr` function instead of `VarPtr`. For more information about `StrPtr`, please consult your BASIC documentation. (Visual Basic has an *undocumented* `StrPtr` function. See MSDN article Q199824. It is important to note that Visual Basic's `StrPtr` function provides a pointer to a Unicode string, so you will usually need to use the Visual Basic `StrConv(vbFromUnicode)` function to convert the string to the ANSI format, and then use `StrPtr` to obtain a pointer to the *converted* string.)

Second, whenever you assign a value to a dynamic string, *the variable's data is moved to a new location in memory*. That means that any `StrPtr` information that you give to the `SQL_BindParam` function *will become invalid* every time you assign a new value to the string.

There are two basic solutions to this problem.

1) Use `SQL_BindParam` to re-bind the parameter every time you change the value of the string.

...or...

2) Rather than assigning a new value like this...

```
sLastName$ = "Smith"
```

...which would cause the string variable to be relocated in memory, always do this instead...

```
LSet sLastName$ = "Smith"
```

Using the BASIC `LSet` function to change a string's value does *not* require it to be moved to a new memory location, so the `StrPtr` memory-pointer value will remain valid. If you decide to use `LSet`, you must remember to *start out* with a string that is filled with spaces, bind the parameter, and *then* use `LSet` to insert the values. If you don't "initialize" the string -- and make sure that it is long enough to hold the *longest* parameter string that you intend to use -- then the `LSet` function will truncate the string. For example, if you start out with `sLastName$ = "Doe"` and then use `LSet sLastName$ = "Smith"` you will end up with "Smi". The `LSet` function can't change the *length* of the initial string. See your BASIC documentation for more information about `LSet`.

To bind a parameter to a dynamic string you should create a dynamic string (\$) variable and fill it with a "dummy" string that is long enough to hold the longest value that you'll be using, and then bind the parameter.

```

Dim sLastName$
Dim sTemp$

SQL_Stmt (SQL_STMT_PREPARE, (etc))

'"size" the buffer...
sLastName$ = Space$(32)

SQL_BindParam(sLastname$, etc.)

'To make things easier, we'll use a
'"working" variable...
sTemp$ = "Smith"

'set the parameter's value...
Mid$(sLastName$,1) = sTemp$

'set the parameter's Indicator value...
lLastNameIndicator& = Len(sTemp$)

SQL_Stmt (SQL_STMT_EXECUTE, " ")

```

Binding Long Parameter Values

If you need to bind a parameter that requires a very large buffer (typically over 32k bytes), it is possible to send the parameter's value to the SQL statement "in pieces" without actually creating a buffer.

First, as always, you should use `SQL_Stmt(SQL_STMT_PREPARE)` to prepare a SQL statement that contains a `?` in the appropriate location.

Then you should bind the parameter normally, with one important exception. Create an Indicator buffer, but *do not create a data buffer*. Instead of providing a `VarPtr` or `StrPtr` value for the `lPointerToBuffer&` parameter, you should pass *the parameter number*. (To be clear, both the `lParameterNumber&` and `lPointerToBuffer&` parameters must have the *same* value.)

Next, instead of placing the length of the data into the Indicator variable, you must use one of two special values. To determine which special value you need to use, use this test code:

```
sResult$ = SQL_DBInfoStr(DB_NEED_LONG_DATA_LEN)
```

If `sResult$` does *not* contain "Y" you should use the special Indicator value `SQL_LONG_DATA`.

If `sResult$` *does* contain "Y" then you must use an Indicator value that is created by the following equation:

```
Indicator = 0 - (DataLength + 100)
```

In other words, add 100 to the length of the Long data, and make the value negative. If the Long column's data is 8000 bytes long, the special Indicator value that you must use would be `--8100`.

Note: Once you have determined whether or not "Y" is returned by a certain ODBC driver for a certain database, you do not need to repeat the `DB_NEED_LONG_DATA_LEN` test. You can assume that the answer will always be the same, and remove the test code.

Then you should use this code, as you normally would...

```
lResult& = SQL_Stmt(SQL_STMT_EXECUTE, " ")
```

Instead of executing the prepared statement, however, the `SQL_Stmt` function will return immediately and the value of `lResult&` will be `SQL_NEED_DATA` (value 99).

Then you should use the `SQL_NextParam` function like this...

```
lResult& = SQL_NextParam
```

...to find out the parameter number of the parameter that needs data. In this simple example, the return value of `SQL_NextParam` will be one (1), because the one-and-only parameter needs data. *Even if you know that a parameter needs data*, you must use the `SQL_NextParam` function after `SQL_Stmt` to tell SQL Tools "here comes the data for the next parameter".

Then you should use the `SQL_LongParam` function to send the Long value and an Indicator value to the parameter.

For example, if the long value that you want to send to the parameter is contained in the variable `sLongValue$`, you should use this code:

```
SQL_LongParam sLongValue$, LEN(sLongValue$)
```

(Keep in mind that the `SQL_LongParam` function automatically sends data to the parameter with the number that was returned by the `SQL_NextParam` function.)

If you want to send a Null value to a Long parameter, use...

```
SQL_LongParam( " ", SQL_NULL_DATA)
```

You can use `SQL_LongParam` repeatedly, to send the data in "chunks", if that is convenient. For example, if the Long parameter value was stored in two different variables called `sLong1$` and `sLong2$`, you would use this code...

```
SQL_LongParam(sLong1$, LEN(sLong1$)  
SQL_LongParam(sLong2$, LEN(sLong2$))
```

...and SQL Tools would automatically add together *all* of the strings that you submit in this way.

When you are done sending the Long value to the parameter, use the `SQL_NextParam` function again. This does two things: **1)** it tells SQL Tools that you are done sending data for that parameter, and **2)** it returns a value that indicates whether or not more columns need data. If there is another Long column that needs data, the column number will be returned by the `SQL_NextParam` function. If not, zero (0) will be returned.

You must use the `SQL_NextParam` function even if you know that there are no more parameters that need data. If you don't, SQL Tools won't know that you are finished sending data and it will generate an Error Message.

When you have provided data for all of the Long columns that need it, `SQL_NextParam` will return zero (0) or a negative Error Code number, to indicate that you are ready to proceed.

After you have told SQL Tools that all of the Long data has been sent, the ODBC driver will build a result set. Keep in mind that this often-time-consuming operation is usually performed by the `SQL_Stmt` function, but in this case your program will appear to pause when `SQL_NextParam` is used for the final time.

Under normal circumstances, the `SQL_NextParam` function will return `SUCCESS` (zero). It can also return all of the Error Codes that can be returned by `SQL_Stmt`, if an error is detected. Unfortunately, one of those Error Codes is `SUCCESS_WITH_INFO` (value 1), and this Error Code can be confused with "parameter 1 needs data". (This difficult-to-handle situation is caused by the ODBC driver, not by SQL Tools.) Fortunately this is a rare occurrence.

Keep in mind that, even if the SQL statement was a `SELECT` statement, no result set was generated by the `SQL_Stmt` function because it did not have the data that it needed to do so. That means that the SQL Tools AutoAutoBind feature was not able to automatically bind the columns of your result set. So, if the statement that contained a Long parameter was a

SELECT statement, the last "unusual" step that you must perform when using Long parameters is this...

```
SQL_AutoBindCol ALL_COLS
```

Then you can use `SQL_Fetch` to begin retrieving and using the results of the SQL statement.

Arrays of Bound Parameters

For information about even more advanced Parameter Binding techniques, see `SQL_SetStmtAttrib(STMT_ATTR_PARAMSET_SIZE)`.

Result Sets

When a *SELECT* statement is used to retrieve rows from a database, something called a "result set" is created. You can think of a result set as a new, temporary table. Your programs can never actually access a database table directly; they can only access result sets.

For example, if you use the SQL statement...

```
SELECT * FROM ADDRESSBOOK
```

...that would tell the database to create a new, temporary table that contains all of the rows and columns from a table called ADDRESSBOOK. And if you use.

```
SELECT NAME, CITY FROM ADDRESSBOOK
```

...a new, temporary table -- a "result set" -- would be created that contains all of the rows from ADDRESSBOOK, but only the NAME and CITY columns. If you used...

```
SELECT NAME, CITY FROM ADDRESSBOOK WHERE ZIPCODE < 50000
```

...the result set would contain the NAME and CITY columns, but only the rows from ADDRESSBOOK where the ZIPCODE column had a value less than 50000. If you add...

```
SELECT NAME, CITY FROM ADDRESSBOOK WHERE ZIPCODE < 50000 AND NAME <>  
'SMITH' ORDER BY ZIPCODE
```

...you would get a somewhat different result set.

The SQL syntax that you use will depend on **1)** what you are trying to accomplish and **2)** the syntax that is supported by the ODBC driver that you are using.

Result Column Binding (Basic)

When a SQL `SELECT` statement is executed, a result set is produced. If the result set contains one or more rows (i.e. if it did not return "no data") then a process called "column binding" must take place.

Each column of the result set must be "bound" to your program. Your program can't access columns that haven't been bound.

(Okay, technically you don't have to bind *all* of the columns of a result set if the result set contains some columns that you want to ignore. But that's a sign of sloppy SQL programming. You should design your SQL statements so that they only return columns that you need. Returning columns that you don't need wastes database resources, server processing time, and network bandwidth.)

(And yes, if you skipped ahead in this document you know that there is a special kind of column called a Long Column that doesn't have to be bound in order to be used. For now, pretend that you don't know that.)

Like we said, each column of a result set must be "bound" to your program. Your program can't access columns that haven't been bound. Binding is a complex, error-prone process. If it is not performed correctly your program is very likely to generate an Application Error.

Fortunately, SQL Tools can handle 100% of the binding process for you.

If you use the `SQL_Stmt(SQL_STMT_EXECUTE)` or `SQL_STMT_IMMEDIATE` option, SQL Tools will automatically bind all of the columns in your SQL statement's result set, so that your program can access the resulting data.

It is also possible to use the SQL Tools `SQL_ManualBindColumn` function to bind result columns to memory buffers that your program manages, but we do not recommend that you use manual binding unless it is very important to squeeze every last *drop* of performance out of your program. The SQL Tools AutoBinding process is very efficient, but in some cases using manual binding can help an application run slightly faster. See Manual Column Binding for more information.

Fetching Rows from Result Sets (Basic)

"Fetch" is the SQL term for "get a row of data from a result set."

Once your program has used the `SQL_Stmt` function (and possibly `SQL_StmtMode`) to tell the database which data it should give you, two things will happen automatically: **1)** The ODBC driver will construct a result set and **2)** SQL Tools will automatically bind all of the columns in the result set. After those things have been done, your program can access the data in the result set.

The most common way to access a result set is row-by-row. It is also possible to access several rows at a time, but for now we are going to concentrate on the basics. (For more information, see MultiRow Cursors.)

The `SQL_Fetch` function can be used in several different ways, but not all methods are supported by all ODBC drivers. The most common method (by far) is...

```
SQL_Fetch NEXT_ROW
```

... which is roughly equivalent to performing a `LINE INPUT` operation. It fetches the next row of data from the result set.

If you have not yet fetched a row from a result set, `NEXT_ROW` has the same effect as `FIRST_ROW`. If all of the rows of a result set have already been fetched, using `SQL_Fetch` does not return any data and the value of the `SQL_EOD` (End Of Data) function is set to Logical True. More about this later.

Cursors

A result set's "cursor" can be visualized in much the same way that a cursor operates in a word processor. The little blinking "marker" doesn't really exist in a word processing document, it simply shows the location where the next operation (such as typing a letter) will take place.

ODBC cursors mark the location where the next `SQL_Fetch` operation will be performed.

For now, this discussion will concentrate on single-row cursors.

Forward-Only Cursors

All ODBC drivers support "forward-only" cursors, which allow the `SQL_Fetch NEXT_ROW` function to get a row of data. Forward-only cursors (naturally enough) only allow the cursor to move forward. If your program needs to go back and re-read a row of data, the only way to do it is to re-execute the SQL statement and move forward from the beginning again.

While very limited, forward-only cursors are *very* fast, and when your program simply needs to read a result set from beginning to end, forward-only operation is usually sufficient.

More complex programs, however, may need more complex cursor movement. By default, SQL Tools uses something called a "Static Scrollable Cursor" which allows more complex cursor control.

Scrollable Cursors

A scrollable cursor is a cursor that can "scroll" forward and backward through a result set.
(Compare Forward-Only Cursors.)

Problems with Scrollable Cursors

This long, complex section of this document does not apply to programs that "own" a database. If the database that your program accesses is never accessed by other programs which can change the database while your program is accessing it, and if your program only uses one SQL statement at a time, you can probably skip this section.

Unfortunately, the ability to scroll can add greatly to the complexity of a Windows program.

For example, let's say that your program is accessing a database that can also be accessed by another program at the same time. You execute a SQL statement that returns a result set, and begin reading the rows. Then, when you're halfway through, the other program changes several rows that your program has already read.

If your program scrolls back to re-read rows that it has already read, should the result set reflect the changes that were made by the other program or should it contain the same data as before? What if the other program *deleted* a row of data that is included in your result set? Or added one?

The first issue that you must consider is called a "value change". What happens if another program (or another "concurrent" SQL statement in your *own* program) changes the values in a row?

The second issue is called a "membership change", and it relates to rows that are added and/or deleted. What happens when a row that is a "member" of your result set is deleted by another program? And should the result set include rows that are added by another program *after* a SQL statement has been executed by your program?

The third major issue is called an "order change". If your program has used the SQL statement `ORDER BY` clause to read a result set in alphabetical order, what happens if another program either **1)** adds/deletes rows or **2)** changes a value that affects the order of the result set, like changing a row from "Apple" to "Zebra".

To help address these issues, ODBC drivers can provide three different types of scrollable cursors: Static, Dynamic, and Keyset Driven.

Static Cursors provide result sets that appear to be static. In other words, once a result set has been created by the ODBC driver it is treated like a "snapshot" and is not allowed to change. For that reason, static cursors may not always reflect the real-time status of a database. But static cursors are a type of *scrollable* cursor that is supported by virtually every ODBC driver, so in some cases you may be forced to choose between a forward-only cursor and a static cursor.

Dynamic Cursors always reflect all of the changes that are made in a database, in real time. They are usually slower and can be much more difficult to manage than static cursors, but they have obvious advantages in applications such as real-time displays.

Keyset Driven Cursors are a combination of static and dynamic capabilities. The rows of keyset-driven cursors always contain current data. If another program changes the data in a row, your program *will* see the change. But the *order* and *membership* of the result set do not change. This is accomplished (by the ODBC driver) by creating a "keyset" which keeps track of the result set, and allows the ODBC driver to "manage" the results that it gives to your program.

Some ODBC drivers support a fourth type of scrollable cursor, called a **Mixed Cursor**. If a result set is too large for the driver to be able to create a reasonable-sized keyset (based on available memory, etc.) the driver will automatically limit the size of the keyset. If scrolling is performed within the keyset the result set will appear to be keyset-driven, but if scrolling is performed outside that range the result set will be dynamic.

Fetching Rows from Result Sets (Advanced)

If your ODBC driver supports scrollable cursors (most do), you can also use the following `SQL_Fetch` options...

```
SQL_Fetch  FIRST_ROW  
SQL_Fetch  LAST_ROW  
SQL_Fetch  PREV_ROW
```

...to fetch the first, last, or previous row (i.e. the row of the result set that is located before the most-recently-fetched row).

You can also use a positive numeric value like...

```
SQL_Fetch  113
```

... to specify that you want a specific row from a result set. This is called an "absolute" fetch. Again, this capability is not supported by *all* ODBC drivers.

See Determining Cursor Capabilities.

Determining Cursor Capabilities

The easiest way to determine whether or not your ODBC driver supports a particular type of cursor scrolling (such as absolute fetching) is to simply try it. If an Error Message is generated, your ODBC driver probably does not support the type of scrolling that you are attempting to perform.

Cursor capabilities can also be determined programmatically, by using the `SQL_DBInfoUInt` function. In particular, a database's ability to provide different types of scrollable cursors (static, dynamic, and/or keyset driven) can be determined by using this code...

```
lResult& = SQL_DBInfoUInt(DB_type_CURSOR_ATTRIBUTES1)
```

...where *type* is the type of cursor that is being used (STATIC, DYNAMIC, etc.)

For more information, see `SQL_DBInfoUInt`.

Also see Using Bitmasked Values.

Using Bookmarks

ODBC Bookmarks are used to identify a row in a result set, so that your program can easily return to that row and re-fetch it at a later time. Bookmarks can also be used by the `SQL_BulkOp` function to perform "bulk operations" such as `BULK_UPDATE` and `BULK_DELETE`.

Not all ODBC drivers support bookmarks, and some driver support them only when ODBC 3.x behavior is specified with the `SQL_Initialize` function. You can determine whether or not your driver supports bookmarks by using the `SQL_DBInfoUInt (DB_type_CURSOR_ATTRIBUTES1)` function, where *type* is the type of cursor being used (`STATIC`, `DYNAMIC`, etc), and examining the `SQL_CA1_BOOKMARK` bit. (See Using Bitmasked Values.)

If your driver supports them, you must activate the bookmark feature *before* a SQL statement is executed, by using this code:

```
SQL_StmtMode STMT_ATTR_USE_BOOKMARKS, BMARKS_VARIABLE
```

If you re using an ODBC 2.0 driver and have used the value 2 for the *IODBCVersion*& parameter of `SQL_Initialize`, you should use this code instead:

```
SQL_StmtMode STMT_ATTR_USE_BOOKMARKS, BMARKS_ON
```

(The old-style fixed-length ODBC 2.0 bookmarks are not supported by ODBC 3.x drivers. ODBC 3.x programs must use `BMARKS_VARIABLE` to specify the new variable-length bookmarks, instead of using the old `BMARKS_ON` value.)

Once bookmarks have been activated for a statement, the statement will automatically produce bookmarks that your program can use.

The ODBC term "bookmark" really isn't as descriptive as it might be. When you place a bookmark in a printed book, you insert something *into* a book, to mark your place. ODBC bookmarks do not change the database in any way -- nothing is "inserted" -- but they accomplish much the same thing. Your program actually asks the database for a bookmark (via the `SQL_Bkmk` function), and the ODBC driver returns a specially-formatted string to your program. If you later give the string back to the database (via `SQL_FetchRel`) it will re-fetch the row that corresponds to the bookmark.

A better term than "bookmark" might have been "row address", but because it is so widely used by SQL programmers, we have maintained the ODBC terminology in SQL Tools. But you should always think of a bookmark as *a string that identifies a row in a result set*. It's more like "writing down a page number" than "using a bookmark".

Your program can save as many bookmarks as you like, by storing different bookmark strings. Some ODBC drivers use "universal" bookmarks that can be used by any statement, i.e. a bookmark that is obtained by one SQL statement can be used by another statement. Other ODBC drivers provide bookmarks that are only valid until you execute another SQL statement. You can determine how your driver handles bookmarks by using the `SQL_DBInfoUInt (DB_BOOKMARK_PERSISTENCE)` function.

Incidentally, the format of a bookmark string is understood by the driver that produces it, but it is not normally possible for your program to make sense of them. Depending on how the ODBC driver works, a bookmark can be as simple as a `BAS_DWORD` value that specifies an

offset in a data file, or a string that contains a Primary Key, or it can be an extremely complex binary string.

To obtain a bookmark for the most-recently-fetched row in a result set, use the `SQL_Bkmk` function, and save the function's return value in a string variable.

To return to that row, use the `SQL_FetchRel` function with the bookmark string and an "offset" value of zero (0). (`SQL_FetchRel` stands for Fetch Relative.)

To return to a row that is after the bookmarked row, use the same bookmark string and a positive offset value, to indicate the number of rows after the bookmarked row that you want the fetch to take place. For example, using an offset of 1 would fetch the first row after the bookmarked row.

To return to a row that is before the bookmarked row, use the same bookmark string and a negative offset value, to indicate the number of rows before the bookmarked row that you want the fetch to take place. For example, using an offset of -1 would fetch the row just before the bookmarked row.

If you use an offset value that causes the fetch to take place before the first row or after the last row of the result set, the `SQL_FetchRel` function's return value will be `SQL_NO_DATA`, and the `SQL_EOD` (End Of Data) function will return Logical True until a valid row is fetched.

Binding Column Zero

When you activate bookmarks, it becomes possible for your result sets to include a Column Number Zero (0), which contain bookmark values.

IMPORTANT NOTE: *The `SQL_AutoBindCol(ALL_COLS)` function does not automatically bind column 0.* Unless your program makes *extensive* use of bookmarks, you should not normally bind the bookmark column of a result set. Bookmark columns can be quite long, so binding them usually results in a performance penalty. As noted above, other functions can be used to obtain bookmark values without binding column zero.

If your program makes *extensive* use of bookmarks, you might want to consider binding column zero. You can use the `SQL_AutoBindCol(0)` function to perform this function. Then you can use the `SQL_ResColStr(0)` function to obtain strings that are compatible with the `SQL_FetchRel` function.

Relative Fetches

A Relative Fetch is used to fetch a row number *relative* to the current row number. For example, if a statement's cursor was positioned at row 10 and you performed a "+2" relative fetch, row 12 would be fetched. If you then performed a "-4" fetch, row 8 would be fetched. Relative fetches are always performed relative to *the current cursor position at the time of the fetch*.

Not all ODBC drivers support relative fetches. To find out whether or not yours does, you can use the `SQL_DBInfoUInt(DB_type_CURSOR_ATTRIBUTES1)` function, where *type* is the type of cursor that is being used (STATIC, DYNAMIC, etc.), and examining the `SQL_CA1_RELATIVE` bit. (See Using Bitmasked Values.)

If your ODBC driver supports relative fetches, you can use the `SQL_FetchRel` function to perform them. You can use the *IOffset* parameter to specify how far, forward or backward, you want to "jump".

If you use an offset value that causes the fetch to take place before the first row or after the last row of the result set, the `SQL_FetchRel` function's return value will be `SQL_NO_DATA`, and the `SQL_EOD` (End Of Data) function will return Logical True.

Result Column Binding (Advanced)

Whenever a *SELECT* statement is executed by the `SQL_Stmt` function, SQL Tools automatically "binds" all of the columns of the result set to "data buffers" and "Indicator buffers" which are then managed by SQL Tools.

These buffers are actually small blocks of computer memory. SQL Tools creates them and then tells the ODBC driver where they are located, and the ODBC driver places data and Indicator values into the buffers whenever a `SQL_Fetch` or `SQL_FetchRel` operation is performed.

When you use a `SQL_ResCol` function (`SQL_ResColStr`, `SQL_ResColSInt`, `SQL_ResColNull`, etc.), SQL Tools gets the values from the memory buffers and passes them to your program in a useful form.

AutoBinding

Other Binding Alternatives

AutoBinding

The default SQL Tools mode is called "AutoAutoBinding". That means that SQL Tools *automatically autobinds* all of the columns in every result set, so that you don't have to worry about the process. (See Result Column Binding for background information.)

You can disable the AutoAutoBind mode by using...

```
SQL_SetOptionSInt OPT_AUTOAUTO_BIND, 0
```

If you do that, SQL Tools will no longer automatically autobind the results of every *SELECT* statement.

You can then "manually autobind" the results of a SQL statement by using the `SQL_AutoBindCol` function immediately after you use the `SQL_Stmt` function. If you were to use `SQL_AutoBindCol(ALL_COLS)` after every `SQL_Stmt` that contains a *SELECT*, it would accomplish the same thing as using the AutoAutoBind mode.

Other Binding Alternatives

As your SQL Tools programs get more sophisticated, there may be circumstances when you want to "manually bind" one or more columns of a result set.

Manually bound columns can be accessed *slightly* faster than autobound columns, so if speed is a very high priority for your program, you want to might try it. You may also choose to manually bind one or more columns if you need to handle the data in a column in a way that the standard `SQL_ResCol` functions do not allow, or in a more efficient way. (For example, you could bind a column directly to a PowerBASIC `UNION` structure in order to access the data in a more flexible way than the `SQL_ResCol` functions provide.) Finally, some programmers simply *prefer* to handle the binding process themselves.

Please note that all of the various SQL Tools binding alternatives can be used at the same time, within the same result set, with no limitations except that each column may only *end up* with one type of binding. In other words you are free to use the `SQL_AutoBindCol` function to bind all of the columns in a result set, and then use the other binding functions (see below) to re-bind columns as needed. Any time that you use a function to re-bind a column, the previous binding is lost.

You can even have columns that are not bound at all, by using the `SQL_UnbindCol` function. The SQL Tools column-binding functions are very flexible.

Proxy Binding

Before we discuss the different methods of manually binding a column of a result set, you should be aware of a process that we call Proxy Binding. Proxy Binding is when you use AutoBinding on all of the columns of a result set, but then, instead of using the `SQL_ResCol` functions to access the values, you use `SQL_ResColBufferPtr` and `SQL_ResColIndPtr` functions to obtain *pointers* to some of the data buffers and Indicators. If you are comfortable with using memory pointers -- and frankly you had better be, if you are considering manual binding -- then Proxy Binding may be an excellent, flexible, high-performance alternative to manual binding.

You should refer to the Reference Guide entries for `SQL_ResColBufferPtr`, `SQL_ResColIndPtr`, `SQL_ResColSize`, and `SQL_ResColLen` for more information about how those functions can allow you to perform Proxy Binding.

Manual Binding and Direct Binding

First, a couple of very important warnings about manual result column binding:

IMPORTANT NOTE: If you manually bind a column of a result set to a buffer that your program manages, and then your program fails to maintain the buffer correctly, Application Errors are very likely.

IMPORTANT NOTE: If you choose to manually bind a column of a result set, you will effectively be disabling the `SQL_ResCol` functions for that column, and your program is *completely* responsible for obtaining values from the column's buffers. If you attempt to use a `SQL_ResCol` function with a column that has not been Autobound, an Error Message will be generated.

As you probably know, every column of every result set requires two buffers: one for the data, and one for the Indicator value. SQL Tools provides two alternative methods of column binding: **1)** Direct Binding, where your program manages the data buffer but SQL Tools continues to manage the Indicator, and **2)** Manual Binding, where your program manages both the data buffer and the Indicator buffer.

An Indicator buffer is much less complex than a data buffer, so most programs will be able to use Direct Binding. After all, there are only so many different things that you can do with an Indicator value. Manual Binding, the most complex binding process, should be reserved for programs that must squeeze the last *drop* of performance out of a system.

Direct Binding

Direct Binding is a process where your program manages the data buffer for a column of a result set, but SQL Tools manages the column's Indicator buffer.

The `SQL_DirectBindCol` function is used to Direct Bind a column of a result set. The parameters of the function allow you to specify the size and location of a memory buffer that your program has created. (You must create the buffer *before* you can use the `SQL_DirectBindCol` function.)

Whenever possible, you should create a memory buffer that will not move. You can do this with a PowerBASIC `ASCIIIZ`, numeric, or UDT variable, with a Visual Basic fixed-length string, or with an array of numeric variables.

It is also possible to use a BASIC dynamic string variable (a `$` variable) and a function like `String$` to create a memory buffer, but if you use this method you must be very careful to *never assign a value to the string* after it has been bound to a result column. Whenever you assign a new value to a dynamic string, it is automatically *relocated* in memory. If you tell SQL Tools that a string-based buffer is available at a certain location, and then the string moves because you assign a new value to it, the buffer location will be invalid and the ODBC driver will cause an Application Error the next time that `SQL_Fetch` or `SQL_FetchRel` is used.

So you should always use the `Mid$` or `LSet` function to change the value of a dynamic string buffer, never an "equal sign" type of assignment.

If you *must* assign a value to a dynamic string that is used for a column data buffer, your program should re-bind the column *before* `SQL_Fetch` or `SQL_FetchRel` is used again.

Manual Binding

Manual Binding is just like Direct Binding, except that your program *also* provides a buffer for the column's Indicator.

The `SQL_ManualBindCol` function is used to Manually Bind a column of a result set. The parameters of the function allow you to specify the size and location of a memory buffer that your program has created for the column data, and the location of a memory buffer that your program has created for the column Indicator. (You must create both buffers *before* you can use the `SQL_ManualBindCol` function.) It is not necessary to specify an Indicator-buffer *length*, because a four-byte `BAS_LONG` buffer is always used.

The same "buffer movement" warnings apply to Manual Binding. (If you haven't read them already, see Direct Binding above, for details.)

Row-Wise Binding

In almost all cases, your SQL Tools programs will use column-wise binding. That means that each of the *columns* in a result set is bound to *individual* data buffers and Indicator buffers.

If you use row-wise binding, each *row* of a result set is bound to a *single* buffer that contains all of the data and all of the Indicators. Row-wise binding is often used with MultiRow cursors, to create complex (and efficient) memory structures.

Row-wise binding requires that you use Manual Binding to bind a row of a result set to a single, large buffer. The structure of the buffer is determined by the columns that the result set contains. For example, if a result set contains two SQL_CHAR columns that are each 10 bytes long, the row-wise buffer would consist of 10 bytes for the first column, plus 4 bytes for the first column's Indicator, plus 10 bytes for the second column, plus another 4-byte Indicator. That's a total of 28 contiguous bytes. As you can imagine, a more complex (and realistic) result set could produce a very complex buffer structure.

For more information about row-wise binding, see
`SQL_SetStmtAttrib(STMT_ATTR_ROW_BIND_TYPE)`.

We also suggest that you consult the Microsoft ODBC Software Developer Kit, which contains additional information about row-wise binding.

Accessing Result Columns

When you tell SQL Tools to fetch a row from a result set, it automatically loads the row's data from the database into memory buffers that are (usually) hidden from your program. You can then access the data by using SQL Tools functions which return the appropriate kinds of values. For example, if column 1 of a result set contained a string value, you would probably use something like this...

```
PRINT SQL_ResultColumnStr(1)
```

...or, if column 19 contained a signed integer value (a BAS_LONG value) you might use...

```
PRINT SQL_ResultColumnSInt(19)
```

SQL Tools provides functions that can be used to access all of the various SQL Data Types.

SQL_ResColStr	(String columns)
SQL_ResColSInt	(Signed Integer columns)
SQL_ResColUInt	(Unsigned Integer columns)
SQL_ResColBInt	(Big Integer columns)
SQL_ResColFloat	(Floating Point columns)
SQL_ResColDate	(Date columns)
SQL_ResColTime	(Time columns)
SQL_ResColDateTime	(Date-Time columns)
SQL_ResColText	(all types of columns)

Long Columns

When SQL Tools binds a column of a result set, it is required (by the ODBC driver) to set aside some of your computer's memory -- a "memory buffer" -- for the column's data. The buffer must be long enough to hold the maximum amount of data that the column can possibly return. For example, if a table contains a `SQL_VARCHAR` column that is allowed to be up to 256 characters long, SQL Tools must create a 256-character buffer just in case the data fills the entire column.

That process works well most of the time, but it creates a serious problem for the "Long" column types `SQL_LONGVARCHAR`, `SQL_WLONGVARCHAR`, and `SQL_LONGVARBINARY`. Those data types (which we will call `SQL_LONGVAR` columns) can be up to 1 gigabyte in length, so setting aside a full-size memory buffer -- or even a hard-disk buffer -- becomes impractical. Especially if a result set contains more than one `SQL_LONGVAR` column.

For this reason, SQL Tools purposely binds `SQL_LONGVAR` columns to buffers that are too small. By default, SQL Tools uses 256-byte buffers for all long columns. This allows you to use the normal `SQL_ResultColumn` functions to "preview" the first 256 characters of a memo field (for example), but it does not allow access to the rest of the column's data. For that, we use a pair of special functions.

The `SQL_LongResCol` function is used to access `SQL_LONGVARCHAR`, `SQL_WLONGVARCHAR`, and `SQL_LONGVARBINARY` columns. It returns a string that contains a *piece* of a long column, so that you can handle it in manageable "chunks".

(Technically, `SQL_LongResCol` can be used to obtain results from *any* type of column. For example, you might use it if your program only needed the contents of a certain column under such rare circumstances that it wasn't worth binding the column. But the most common use -- by far -- is accessing `SQL_LONGVAR` columns.)

The default chunk size is 32k bytes (32,768 bytes). Each time you use `SQL_LongResCol`, it will return another 32k chunk of data. If there is still more data to be read, the `SQL_ResColMore` function will return a Logical True value. You can use `SQL_LongResCol` repeatedly, until the `SQL_ResColMore` function finally returns False (zero) to tell you that you have read all of the data.

"Data Truncated" Error Messages

When `SQL_Fetch` or `SQL_FetchRel` is used to retrieve a row of data that contains column data that is too long to fit in the memory buffer to which the result column was bound, a `SUCCESS_WITH_INFO` Error Message will be generated to warn your program that the buffer contains partial data. The wording of the message will vary from ODBC driver to ODBC driver, but it will usually contain the word "truncated".

It is also fairly common for a `SUCCESS_WITH_INFO` message containing the words "Error In Row" to be generated.

In some cases (such as with Microsoft Access) both messages will be generated. If more than one column is truncated, several error messages may be generated for each `SQL_Fetch` or `SQL_FetchRel` operation.

If you know ahead of time that a column contains data that is too long to fit in a buffer, and if your program does not need to "preview" the first block of data, you can use the `SQL_UnbindCol` function *before* you use the `SQL_Fetch` function for the first time, to unbind the column. You can then use the `SQL_LongResCol` function to access the data in the long column. It is only necessary to unbind a column once; you do not have to unbind it before every `SQL_Fetch` operation (for example, in a `DO/LOOP` structure).

Also see Ignoring Predictable Errors.

Possible Driver Restrictions on Long Columns

If your program has to deal with long columns, there are several restrictions that you *may* need to keep in mind.

These restrictions are imposed by some ODBC drivers, and if they are imposed they cannot be bypassed. To determine your ODBC driver's exact restrictions, use the `SQL_DBInfoUInt (DB_GETDATA_EXTENSIONS)` function.

- 1)** You may be *required* to unbind long columns before you can access them with `SQL_LongResCol`. (See `SQL_UnbindCol` for more information.)
- 2)** It may only be possible to access long columns that have column numbers that are higher than the highest-numbered *bound* column, so it may be necessary to use `SQL_UnbindCol` to unbind columns other than long columns. (You can usually get around this problem by designing your SQL statements to produce result sets where all of the long columns are located at the end of the row.)
- 3)** It may be necessary for your program to access long columns in ascending numeric order, i.e. you may have to get all of the long data from column 10 before you can get it from column 11. Or you may be able to get part of the data from column 10 and then get some data from column 11, but you may then be unable to return to column 10.
- 4)** It may be impossible to get data from long columns if Multirow cursors are being used.

Again, these restrictions *may or may not* be imposed by your ODBC driver.

Result Column Indicators

Whenever a database gives you data from a column of a result set, it actually provides two different types of information.

The first type is the actual column data, in string or numeric form.

The second type is called the *Column Indicator*, and it is a separate value that tells you *about* the column data.

Many, many SQL programs have been written without using Indicators, but other programs can't do without them.

The most common use of an Indicator is the detection of "null columns", which are described in the following section. Then we'll discuss the other uses of column Indicators.

Null Values

Let's say that you are creating a "family tree" database that lists all of the members of your family, both living and dead. You might create a table with columns called `FirstName`, `LastName`, `MiddleName`, `MotherName`, `FatherName`, `BirthDate`, and `DeathDate`. You would, of course, use your family's records to enter one row of data for each person.

And since you are a careful database designer, you would create a database that uses the appropriate SQL Data Types for all of the columns. You would probably use `SQL_VARCHAR` (variable length string) for everything except the last two, which would be `SQL_DATE` or `SQL_TIMESTAMP` columns.

What do you enter when it comes time to type in your own `DeathDate`? You haven't died yet, so there is no logical value to enter. You could make up a "magic number" like `01/01/9999` to indicate that you are still alive, but that would require your program to understand that special value. If somebody used another program to view your database, they would have to figure out that `01/01/9999` means "still alive".

SQL databases provide a special value for cases like this. It is called a Null Value, and it can be used to signify "no data". (As you'll see, it can also be used to signify other things.)

Here's another example: All of your family records list your great-great-great-grand-uncle as "John Smith". What do you enter into the `MiddleName` field? If you enter " " (an empty string), that could be interpreted to mean either "this person's middle name is unknown" or "this person did not *have* a middle name". The Null value can be used to distinguish between those two conditions without resorting to "magic" values like "???". In this case you could define " " to mean "no middle name" and a null value to mean "unknown". Or vice versa.

Finally, let's reconsider your Uncle John. His death certificate turns out to be missing, so you don't know his `DeathDate`. Since he's a great-great-great-whatever it is fairly unlikely that he's still alive, so you probably need to redefine a Null value in the `DeathDate` columns as "unknown" instead of "still alive". Then perhaps you'd add a true/false column (a `SQL_BIT`) called `IsAlive`.

In the end, the meaning of a null value is up to the database designer. It does not have a predefined, "universal" meaning, but it *is* another tool that you can use when dealing with unusual circumstances.

Clearly, the efficient design of a database with things like `DeathDate` columns can be a very complex undertaking (sorry about the pun), but the SQL Null value can make things a little easier.

The `SQL_ResultColumnNull` function returns a true or false value that is based on the column Indicator.

Other Uses of Column Indicators

As we described above, the most common use of a column Indicator is the detection of null values. The `SQL_ResultColumnNull` function returns a true or false value that is based on the column Indicator.

However, the column Indicator itself is actually a numeric value, not a simple true/false value.

If the Indicator has a zero or positive value, that indicates the length of a string. For example, if a result column contained the string "John Smith", the result column Indicator would contain the number 10. If the result column contained an empty string ("") the Indicator value would be zero (0).

A value of negative one (-1) corresponds to a null value.

A value of negative four (-4) corresponds to "length unknown" for certain types of Long columns.

Other negative numbers have special meanings too, but they do not normally apply to SQL Tools programs.

Various SQL Tools functions such as `SQL_ResColMore` and `SQL_ResColNull` interpret the column Indicator values for you, so you don't usually need to be concerned about the actual numeric values, but you should be aware that the column Indicator values exist.

Results from non-SELECT Statements

`SELECT` statements return result sets, i.e. temporary tables that contain rows of data that your program can read.

Other SQL statements such as `UPDATE` do not return result sets. They simply return the number of rows that were affected by the statement.

The `SQL_ResRowCount` function can be used immediately after a `SQL_Stmt` function, to obtain the number of rows that were affected by the statement.

VERY IMPORTANT NOTE: The `SQL_ResRowCount` function should *not* be used to determine the number of rows that were returned by a `SELECT` statement. See next page for more details.

Why You CAN'T Use SQL_ResRowCount for SELECT Statements

Some ODBC drivers return a value for `SQL_ResRowCount` for *all* statements, including `SELECT` statements. In that case, `SQL_ResRowCount` can theoretically be used to obtain the number of rows in a result set that is produced by a `SELECT` statement. *But not all ODBC drivers provide this information, and those that do are not always reliable.*

The Microsoft Access ODBC Driver, for example, returns a value of `--1` (negative one) if you attempt to obtain the number of rows that were created by a `SELECT` statement.

Unless you are writing a program that **1)** will only be used with a database that you know returns a value for `SQL_ResRowCount`, and **2)** will be the *only* program that accesses a database, you should avoid using that function to determine the number of rows in a result set.

There is a very good reason for this limitation: an ODBC driver can not always *know* how many rows a result set contains. For example, let's say that you execute a SQL statement that returns all of the rows in a table where a certain column contains a zero. If you were to use code like this...

```
For lRow& = 1 To SQL_ResRowCount
    SQL_Fetch NEXT_ROW
    'process a row
Next
```

...what would happen if another program that was accessing the database at the same time *changed* a row so that the column no longer contained a zero? The number of rows in the result set could change, and your loop would fail.

Consider, too, that even if the `SQL_ResRowCount` value that a driver provides is updated in real time, code like this cannot be 100% reliable...

```
Do
    lRow& = lRow& + 1
    If lRow& > SQL_ResRowCount Then
        Exit Do
    End If
    SQL_Fetch NEXT_ROW
    'process a row
Loop
```

If the `SQL_ResRowCount` value is updated in real time (in other words, if the ODBC driver re-counts the rows in the result every time you use the function), it is still possible for a row to be deleted during the split second between that elapses between the `SQL_ResRowCount` and `SQL_Fetch` lines.

For these and other reasons, you should always use a read-to-end-of-data strategy to read all of the rows in a result set.

The only reliable way to detect the End Of Data condition is to attempt to read a row with `SQL_Fetch`, and then check the `SQL_EOD` function to find out whether or not it worked.

Detecting the End Of Data

Since not all ODBC drivers return a reliable value for `SQL_ResRowCount`, it is not practical to read all of the rows of a result set using code that looks like this...

```
For lRow& = 1 To SQL_ResRowCount
    SQL_Fetch NEXT_ROW
Next
```

Many ODBC drivers, including the Microsoft Access driver, return a value of negative one (-1) for `SQL_ResultRowCount` when it is used with `SELECT` statements, so the code above would result in *no* rows being fetched. In fact, *most ODBC drivers cannot tell your program how many rows there are in a result set*, using any technique other than fetching them and counting them, one by one. See [Why You CAN'T Use SQL_ResRowCount for SELECT Statements](#) for a more complete discussion of this topic.

The *only* reliable technique for reading all of the rows in a result set is this:

```
Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD Then Exit Do
    'process one row of data
Loop
```

(Please note that this loop does not include error handling, which is covered below.)

The `SQL_EOD` function is conceptually similar to the BASIC `Eof` (End Of File) function. `SQL_EOD` stands for End Of Data, and the function returns a Logical True (-1) value if the most recent `SQL_Fetch` or `SQL_FetchRel` operation failed because the end of data was reached.

It is important to note that "end of data" can also mean "beginning of data" if you are using fetch operations that can move the cursor backward, such as `SQL_Fetch PREV_ROW` or `SQL_FetchRelative` with a negative offset. In those cases, an "end of data" condition can also mean "the fetch operation failed because you have reached the *start* of the result set". However, since it is much more common to fetch in a forward direction, this discussion will focus on the *end-of-data* condition.

There are some very important differences between `Eof` and `SQL_EOD`. For example, the following BASIC code could be used to read a file from beginning to end:

```
Open "FILENAME.EXT" For Input As #1
Do
    If Eof(1) Then Exit Do
    Line Input #1,
    'process a line of data here
Loop
Close #1
```

But that same code would *not* work properly if SQL Tools functions were simply substituted for the BASIC functions. For example:


```

SQL_OpenDB "MYDATA.DSN"
SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM MYTABLE"
Do
    If SQL_EOD Then Exit Do
    SQL_Fetch NEXT_ROW
    'process a row of data here
Loop
SQL_CloseDB

```

Because the `SQL_EOD` function cannot detect that a `SQL_Fetch` or `SQL_FetchRel` operation is *about* to fail (the way `Eof` can), the fetch function *would* fail when it tried to read a row of data after the last row had already been reached. The program would then attempt to process a row of non-existent data.

The correct way to use SQL Tools functions to read an entire result set is this:

```

SQL_OpenDB "MYDATA.DSN"
SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM MYTABLE"
Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD Then Exit Do
    'process a row of data here
Loop
SQL_CloseDB

```

Note that the `SQL_EOD` function is located immediately *after* the `SQL_Fetch`, so that when `SQL_Fetch` fails and the End Of Data condition is detected, your program can respond correctly. (By the way, this lack of a "look ahead" capability *is a limitation of all ODBC drivers*. It is not a limitation imposed by SQL Tools. ODBC databases simply do not "know" that the last row of data has been read until it fails to read a new row. See Why You CAN'T Use `SQL_ResRowCount` for SELECT Statements for a more complete discussion of this topic.)

When you are writing a read-until-end-of-data loop, there is another factor that you should take into account. Consider the following BASIC code:

```

Open "FILENAME.EXT" For Input As #1
Do
    If Eof(1) Then Exit Do
    Line Input #1,
    'process a line of data
Loop
Close #1

```

What happens if there is a hard drive error that keeps the `Line Input` from working correctly? *The loop will run forever.*

You should check for the same types of errors when you are attempting to read a result set, like this:

```

SQL_OpenDB "MYDATA.DSN"
SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM MYTABLE"
Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD Then Exit Do
    If SQL_ErrorPending Then
        'Check the error type,
        'and exit if necessary.
    End If
    'process a row of data here
Loop
SQL_CloseDB

```

Instead of checking the value of `SQL_ErrorPending`, you could also check the return value of the `SQL_Fetch` function for `SUCCESS`, like this:

```

SQL_OpenDB "MYDATA.DSN"
SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM MYTABLE"
Do
    If SQL_Fetch(NEXT_ROW) <> SUCCESS Then
        Exit Do
    End If
    'process a row of data here
Loop
SQL_CloseDB

```

For a more complete discussion of this topic, see [Error Handling](#).

Detecting "No Data At All"

If your program uses a SQL statement that may not produce *any* results (an *empty* result set), you can use the method described in Detecting The End Of Data to find out whether or not any rows were returned, or you can use a method that is usually faster.

The `SQL_ResColCount` function (not `SQL_ResRowCount`) returns the number of *columns* in a result set. You usually know this number ahead of time -- after all, you designed the SQL statement and specified which rows should be returned -- but if `SQL_ResColCount` returns a zero (0) value, that means that the SQL statement did not return any columns, and that means that it did not return any rows.

Error Handling in SQL Tools Programs

In DOS programs, runtime errors are often handled with an `On Error GoTo` function. When a runtime error is detected, the program's flow is interrupted and a special error-handling function is executed. Then, if the program is able to recover from the error, a `Resume` statement tells the program to jump back to the point where it was interrupted.

In Windows programs, it is much more common to use an error handling strategy called `On Error Resume Next`. If an error occurs, the program automatically skips the offending line and goes directly to the next line of code. If errors are possible in a section of code, programs routinely check the `Err` system variable, and if a nonzero value is found, they handle the error.

(A complete discussion of good programming and error-handling technique is well beyond the scope of this document. This information is provided as background for this User's Guide.)

ODBC drivers take the `Err` concept a couple of steps further, and SQL Tools expands your error-handling options even beyond that.

See Error Codes and Error Messages.

Error Codes

Almost all ODBC functions, and some SQL Tools function, produce Error Codes as their return values. The most common Error Code is actually `SUCCESS`, which means "no errors". `SUCCESS` has a value of zero, and nonzero values usually mean that an error was detected.

Usually.

An Error Code of `SUCCESS_WITH_INFO` (which has a numeric value of 1) means that an ODBC driver *was* able to perform a certain function, but there's something that it thinks you need to know about the operation. For example, a `SQL_Fetch` operation might return `SUCCESS_WITH_INFO` if the fetch worked but one or more columns contained data that was too long to fit in the buffers that were provided. That *may or may not* be a problem for your particular program, so the driver says "success" but also provides "info".

If an Error Code is detected by your program, SQL Tools gives you a wide variety of functions that allow you to examine the errors that are reported by ODBC drivers and by SQL Tools itself. In the case of the `SQL_Fetch` error described above, examining the return value of the `SQL_ErrorText` function would reveal a string that said something like...

```
[Microsoft][ODBC Microsoft Access 97 Driver]Data truncated
```

`SUCCESS_WITH_INFO` basically means "the operation that you requested was performed, and your program can continue running, but you may need to address a problem."

The third-most common return code (after `SUCCESS` and `SUCCESS_WITH_INFO`) is almost certainly `SQL_ERROR`. If a SQL Tools function returns this error code it means that something serious occurred and the ODBC driver reported an error. Examining the `SQL_ErrorText` function after a `SQL_ERROR` might reveal a message that says that the network connection to your database has failed, or that you do not have the access rights that are required to perform the operation that you requested. Hundreds of different messages are possible. See Appendix E: ODBC Error Codes and Appendix F: SQL States for more details.

In addition to "passing along" error messages from the ODBC driver, SQL Tools itself can also generate several different error codes. For example, if you use an illegal value with a SQL Tools function it will usually return `ERROR_BAD_PARAM_VALUE`.

A complete list of SQL Tools Error Codes is provided in Appendix D.

Please note that some SQL Tools functions do not return error codes. For example, the various SQL Tools functions that return string values cannot (of course) also return numeric error codes. And many functions that return numeric values do not return error codes, in order to simplify their use. For instance, a function like `SQL_TableCount`, which returns the total number of tables in a database, does not return error codes because things like `SUCCESS_WITH_INFO` (value 1) would be easily confused with "this database contains 1 table".

All SQL Tools functions *do*, however, generate Error Messages even if they do not return Error Codes.

Using Error Messages Instead of Error Codes

Some programmers prefer to ignore the return values of functions and rely on other techniques instead. This is conceptually similar to using `ERR`. Programs periodically check a certain function (`SQL_ErrorPending`) to find out whether or not any errors have occurred since the last time it was checked.

Another interesting aspect of ODBC error handling is that more than one error message can be generated for a single error. Returning to the `SQL_Fetch` "Data Truncated" example above, let's assume that three different columns contained data that was too long for their buffers. In that case the `SQL_Fetch` function would return `SUCCESS_WITH_INFO`, and three different error messages would be produced: In fact, the Microsoft Access 97 ODBC driver produces an additional error message in this case, and the actual error messages, in order, would look like this:

```
[Microsoft][ODBC Microsoft Access 97 Driver]Error in row  
[Microsoft][ODBC Microsoft Access 97 Driver]Data truncated  
[Microsoft][ODBC Microsoft Access 97 Driver]Data truncated  
[Microsoft][ODBC Microsoft Access 97 Driver]Data truncated
```

The first message means "there was at least one error in the row that was fetched" and then the three other messages provide details. All of that from a single use of `SQL_Fetch`!

SQL Tools maintains a "stack" of up to 64 error messages at a time. (That number can be adjusted, but 64 is the default value and it works well for most programs.) If more than 64 errors build up, the oldest ones are discarded as new ones are added.

Because **1)** not all functions return Error Codes and **2)** many different functions can return multiple errors but an Error Code can only indicate a single error, many programmers ignore the Error Codes that functions provide as return values, and rely instead on the Error Messages from the stack.

The best, most flexible strategy is to use a combination of both Error Codes and Error Messages.

The `SQL_ErrorPending` function returns a Logical True value (-1) if there are *any* errors currently in the stack.

The `SQL_ErrorCount` function returns a number from 0 to 64, indicating the current error count.

The "bottom" error on the stack -- the *oldest* error -- can be examined with functions like `SQL_ErrorText`, `SQL_ErrorNumber`, `SQL_ErrorStatementNumber`, and `SQL_ErrorFuncName`. (See The Error/Trace Family of functions for more details.) After you have found out everything that you need to know about an error, you can use the `SQL_ErrorClearOne` function to remove it from the stack. Then you can use those same functions to examine the next error in the stack (if any).

An alternate method, instead of using different functions to examine different aspects of an error, is to use the `SQL_ErrorQuickOne` function to obtain a string that contains everything SQL Tools knows about an error. The error is *automatically* cleared from the stack when

`SQL_ErrorQuickOne` is used.

As noted above, not all SQL Tools functions return Error Codes such as `SUCCESS_WITH_INFO`, but *all* of them add an error message to the stack whenever an error is detected. This is the primary reason that some programmers prefer to ignore the return value of most SQL Tools functions and rely on `SQL_ErrorPending` and/or `SQL_ErrorCount` to alert them that errors have been detected.

Ignoring Predictable Errors

You will probably encounter some "predictable" errors while you are writing programs with SQL Tools. For example, if you use the `SQL_Init` function (and thereby use the default `IODBCVersion` value of 3), and if your program opens a database (such as a Microsoft Access 97 database) that does not support all of the features that ODBC 3.x provides, you will almost certainly receive the following Error Message:

```
[Microsoft][ODBC Driver Manager] The driver doesn't support the
version of ODBC behavior that the application requested.
```

That message, among others, can be very annoying because once you have run your program *once* and have seen the message, you probably won't want to see it *every time* you run the program.

You can use the `SQL_ErrorClearOne` function to get rid of the Error Message after it happens, or you can use the `SQL_ErrorIgnore` function to tell SQL Tools "please do not report that error to this program".

Once you have identified the SQL State value that accompanies an Error Message, you can use the `SQL_ErrorIgnore` function. For example, all `SUCCESS_WITH_INFO` messages (such as the "doesn't support the version..." message above) use the SQL State value 01000. So you could do this at the very beginning of your program...

```
SQL_ErrorIgnore 1, ALL_STMTs, "01000"
```

...to tell SQL Tools not to report any errors with the SQL State 01000 that occur with Database Number 1 (the first parameter), regardless of the statement number (`ALL_STMTs`).

If you wanted to ignore all Error Messages with SQL State 12345 for Database 1, Statement 1 *only*, you could do this:

```
SQL_ErrorIgnore 1, 1, "12345"
```

If you specify a Statement Number like that, however, you must remember that certain operations like `SQL_OpenDB` are not associated with a particular Statement Number, so an Error Message with SQL State 12345 would *not* be ignored if it was generated by the `SQL_OpenDB` function.

If you want to ignore more than one SQL State at a time, use a comma-delimited list of five-character SQL State code, like this:

```
SQL_ErrorIgnore 1, 1, "12345,54321,98765"
```

Every time you use the `SQL_ErrorIgnore` function to specify a list of SQL States that should be ignored, it *replaces* the old list.

If you want to add a SQL State to an *existing* list, you can take advantage of the fact that the `SQL_ErrorIgnore` function returns a string value that corresponds to the list *before* any changes were made. So doing this...

```
SQL_ErrorIgnore 1, 1, SQL_ErrorIgnore(1,1,"") + ",12345"
```


...would add "12345" to the existing list.

See the `SQL_ErrorIgnore` function for more information.

Miscellaneous Error Handling Techniques

Whenever your program ends, if there are any errors in the error stack that have not been cleared by your program, SQL Tools can automatically use functions called `SQL_ErrorQuickAll` and `SQL_MsgBox` to display a standard Windows message box. If you activate this feature by using this code...

```
SQL_SetOptionSInt OPT_EXIT_CHECK, TRUE
```

...and you see a message box when your program ends, you probably have some troubleshooting to do. The message box can be disabled again when your program is ready for distribution, and/or it can be customized to include your program's name and icon. See `SQL_SetOptionSInt(OPT_ERROR_MSGBOXTYPE)` for more details.

SQL Tools can also display a message box every time an error is detected. You can activate this feature during program development by adding this code...

```
SQL_SetOptionSInt(OPT_ERROR_MSGBOXTYPE, lMsgBoxType&)
```

...to your program, where *lMsgBoxType&* is a constant that tells SQL Tools the types of buttons that the message box should have. For more details, see `SQL_SetOptionSInt`.

SQL Tools also provides a function that is conceptually similar to `On Error GoTo`. You can "register" one of your program's functions with SQL Tools in such a way that when an error is detected, SQL Tools will call *your* error-handling function. For complete information see `SQL_OnErrorCall`.

Finally, SQL Tools provides several different minor variations on the techniques described above, plus a "diagnostic" feature (`SQL_Diagnostic`) that allows your program to ask an ODBC driver for more information about an error.

SQL Tools Trace Mode

To make troubleshooting easier, SQL Tools includes a "trace mode" feature. You can activate the trace mode by using the line...

```
SQL_Trace TRACE_ON
```

...and turn it off by using...

```
SQL_Trace TRACE_OFF
```

You can also trace a single SQL Tools function by using...

```
SQL_Trace TRACE_SINGLE
```

...immediately before the function that you want to trace.

During the time that tracing is enabled, SQL Tools will create a trace file that contains the name of every SQL Tools function that your program uses, including the values of every parameter that is passed to the functions, all errors that are detected, and the return values that are produced by the functions.

WARNING: Because it involves the creation of a large text file, the use of the SQL Tools Trace Mode can *greatly* slow down a program. One of our very small test programs took 7.26 seconds to execute when the Trace Mode was turned on, but less than 0.05 seconds with tracing turned off. And the slowdown can be made *much* worse if the ODBC Trace Mode is used at the same time, or if an existing Trace File is being appended (which is the default behavior). Instead of activating the Trace Mode at the very beginning of your program, we suggest that you attempt to isolate a small section of code that is likely to be causing a problem, and turn the Trace Mode on then off again as quickly as possible.

For more information, see `SQL_Trace`.

ODBC API Tracing

The ODBC Driver Manager (the part of Windows that manages all of your ODBC drivers) also contains a tracing function. It is limited to the "API level", i.e. it logs all of the Windows ODBC API functions that SQL Tools uses when you use a SQL Tools function.

You can use the `SQL_SetDBAttrUInt` function to turn the API trace mode on and off by using `SQL_ATTR_TRACE`.

The `SQL_SetDBAttrStr` function can be used to change the name of the trace file, by using `SQL_ATTR_TRACEFILE`.

The resulting log file can be interesting if you want to know more about the ODBC API and how SQL Tools performs various functions, but it is usually of little value during troubleshooting. We suggest that you use the SQL Tools trace mode instead, because it usually supplies more pertinent information.

WARNING: Because it involves the creation of a large text file, the use of the ODBC Trace Mode can *greatly* slow down a program. One of our very small test programs took 40.50 seconds to execute when the ODBC Trace Mode was turned on, but less than 0.05 seconds with ODBC tracing turned off. And the slowdown can be made worse if the SQL Tools Trace Mode is used at the same time, or if an existing Trace File is being appended (which is the default behavior). Instead of activating the ODBC Trace Mode at the very beginning of your program, we suggest that you attempt to isolate a small section of code that is likely to be causing a problem, and turn the ODBC Trace Mode on then off again as quickly as possible.

SQL Tools Utility Functions

SQL Tools contains many different functions that can be used to simplify and enhance your programs.

`SQL_IString` is a "string interpreter" function that can allow you to embed hard-to-type characters in your strings, such as quotation marks, carriage returns, and tabs.

`SQL_TextStr` and `SQL_TextStrOLE` are functions that can convert any string into a displayable string. Characters that cannot normally be displayed are converted into things like `[CHR$(0)]` so that they are visible when printed or otherwise displayed.

`SQL_BinaryStr` can covert the `[CHR$(0)]` notation back into binary form.

`SQL_LimitTextLength` automatically shortens strings that are over a certain length, and adds `". . ."` to the end to indicate that they have been shortened.

`SQL_TextDateTime`, `SQL_TextDate`, and `SQL_TextTime` can be used to convert `SQL_TIMESTAMP`, `SQL_DATE`, and `SQL_TIME` data structures into a date/time in simple text form.

`SQL_SelectFile` can be used to display a standard Windows "Open File" dialog box.

`SQL_MsgBox` can be used to display a standard Windows Message Box, with several different options including custom icons and six different combinations of buttons.

`SQL_MsgBoxButton` can be used to determine which button was selected the last time that the `SQL_MsgBox` function was used.

`SQL_Okay` is a function that returns a Logical True value if the parameter that is passed to it is *either* `SUCCESS` or `SUCCESS_WITH_INFO`.

`SQL_StringToType` is a function that can be used to assign the value of a string to a User Defined Type.

And finally, `SQL_ToolsVersion` can be used to determine which version of the SQL Tools DLL is installed on a computer.

Database Information and Attributes

When you use the `SQL_OpenDB` function to open a database, certain "database attribute" and "database information" values become available to your program.

Generally speaking, an "information" value is fixed, and can't be changed by your program. An "attribute" (with a few exceptions and limitations) is a value that you can change programmatically.

The `SQL_DBInfoStr` and `SQL_DBInfoUInt` functions can be used to obtain nearly 200 different types of information about an open database, in string and numeric form.

It would be difficult to over-emphasize the importance of the two `SQL_DBInfo` functions. We strongly recommend that you take the time to familiarize yourself with these *powerful* functions.

The `SQL_DBAttribStr` and `SQL_DBAttribUInt` functions can be used to obtain many different attribute values, and the `SQL_SetDBAttribStr` and `SQL_SetDBAttribUInt` functions can be used to set new attribute values.

Statement Information and Attributes

When you use the `SQL_Stmt` function to execute a statement, certain "statement attribute" and "statement information" values become available to your program.

Generally speaking, an "information" value is fixed, and can't be changed by your program. An "attribute" (with a few exceptions and limitations) is a value that you can change programmatically.

The `SQL_StmtInfoStr` function can be used to obtain different types of information about an open statement, in string form. (ODBC drivers do not provide numeric information about statements, so no `SQL_StmtInfoSInt` function is provided by SQL Tools.)

The `SQL_StmtAttrib` function can be used to obtain several different statement attribute values, and *under certain circumstances* the `SQL_SetStmtAttrib` function can be used to set new attribute values. Most of the time, however, your programs should use the `SQL_StmtMode` function to *pre-set* statement attributes, before a SQL statement is opened. Setting the attributes of an already-open statement can be very difficult.

Environment Attributes

After the `SQL_Initialize` function has been used to "start up" SQL Tools, you can use the `SQL_EnvironAttribSInt` function to obtain certain values that are related to the ODBC "environment", i.e. values which affect *all* databases. This function can be used even before you open a database.

It is also possible to use the `SQL_SetEnvironAttribSInt` function to change the environment, but in most cases your program will *pre-set* the environment values with the `SQL_Initialize` function, and then leave them alone.

Manually Opening and Closing Databases

Normally, your program will use the `SQL_OpenDB` function to open a database before it attempts to use the `SQL_Stmt` function to execute a SQL statement.

If you attempt to use the `SQL_Stmt` function before you have used `SQL_OpenDB` to open a database, the `SQL_Stmt` function will automatically call the `SQL_OpenDB` function for you. An empty string will be used for the `sConnectionString` parameter, to allow the user to "navigate" to a database. This is rarely necessary, however, since most SQL statements only have meaning in the context of a database connection. In other words, you are unlikely to need to execute a SQL statement like `SELECT * FROM MYTABLE` unless your program has already opened a database that contains a table called MYTABLE. The auto-open feature is primarily provided as a programmer convenience, for those times that you are writing quick-and-dirty test programs.

The Database AutoOpen feature can be disabled by using the `SQL_SetOptionSInt(OPT_AUTOOPEN_DB, 0)` function.

Normally, if your program is finished using one database and uses the `SQL_OpenDB` function to open a different database (using the same database number), SQL Tools will automatically close the first database for you. The Database AutoClose feature can be disabled by using the `SQL_SetOptionSInt(OPT_AUTOCLOSE_DB, 0)` function.

If your program frequently opens and closes databases, you might want to consider disabling the Database AutoOpen and AutoClose functions, and perform these operations manually. (This can make bugs easier to find.) If you do, and if your program attempts to **1)** use the `SQL_Stmt` function before using `SQL_OpenDB`, or **2)** use `SQL_OpenDB` when a database number is already in use (i.e. without first using `SQL_CloseDB`), an Error Message will be generated.

Manually Opening and Closing Statements

Normally, SQL Tools takes care of opening and closing statements for you. All you have to do is use the `SQL_Stmt` function, and SQL Tools will automatically open the statement and execute it for you. And if you use `SQL_Stmt` again, SQL Tools will automatically close the first statement and open the second.

In some cases you may wish to disable the Statement AutoOpen and AutoClose options. You can do this by using the `SQL_SetOptionSInt(OPT_AUTOOPEN_DB, 0)` and `SQL_SetOptionSInt(OPT_AUTOCLOSE_DB, 0)` functions.

If you disable these options, your program is responsible for using the `SQL_OpenStmt` function before it uses `SQL_Stmt`, and for using `SQL_CloseStmt` before using `SQL_Stmt` for a second (or subsequent) time. If you fail to perform these operations in the correct order, an Error Message will be generated.

Using Database Numbers and Statement Numbers

Most small- and medium-sized programs will only use one database, and most will only execute one SQL statement at a time. There will probably be times, however, when you will need to **1)** use two or more databases at the same time, or **2)** use two or more SQL statements at the same time (on a single database), or **3)** use multiple databases *and* multiple statements.

The word "concurrent" is used to describe the condition "two or more at the same time". The most common use of the term is "concurrent statements", which is when your program has two or more statements open at the same time, using a single database.

SQL Tools Standard is limited to two (2) concurrent databases, each with a maximum of two concurrent statements. (Within certain restrictions, three concurrent statements can be used.)

SQL Tools Pro can theoretically handle up to 256 concurrent databases and 256 concurrent statements *per* database, all at the same time. It is very unlikely, however, that your program will ever be that large and complex. It is far more likely that you would need to access a large number of databases, using one statement each, or you will need to use a large number of concurrent statements on a single database.

VERY IMPORTANT NOTE: Not all ODBC Drivers support an unlimited number of concurrent databases and/or statements. Some are limited to a single database and a single statement. *If an ODBC driver can only support a certain number, SQL Tools is also limited to that number.*

The `SQL_Initialize` function is used to tell SQL Tools how many concurrent databases and statements your program will use. If you use the `SQL_Init` function, it will use the values "2" and "2", so that your program can use two concurrent databases, each with two concurrent statements. You should be careful not to specify unnecessarily-large values, so that SQL Tools does not reserve large blocks of memory for no reason.

When you use a SQL Tools function, you may specify any Database number between one (1) and the `IMaxDatabaseNumber` value that you specify with `SQL_Initialize`, or, if you use `SQL_Init`, between one (1) and the default `IMaxDatabaseNumber` value of two (2).

You may use any Statement number between one (1) and the `IMaxStatementNumber` value that you specify with `SQL_Initialize`, or between one (1) and the default `IMaxStatementNumber` value of two (2) if you use `SQL_Init`.

VERY IMPORTANT NOTE: If you are building an Interoperable Application (i.e. a application that needs to work with more than one ODBC driver) you should not assume that all ODBC drivers will allow you to use concurrent databases and statements. Some drivers allow only one active statement per database "connection", so you may be required to incur the additional overhead of using `SQL_OpenDB` to open the same database twice, so that you can use a statement on each of the connections. Some ODBC drivers, unfortunately, do not allow multiple connections to the same database. If you are faced with an ODBC driver that does not allow these things, you should carefully review Statement Zero Operation.

If your program only needs to use a single database and statement, you should use the "abbreviated" SQL Tools functions, which do not allow you to specify a database number or statement number as a parameter. In fact, if your program uses a small number of databases and/or statements -- perhaps two or three of each -- you may still prefer to use the

abbreviated functions.

But if your program uses more than a few concurrent databases or concurrent statements, you will probably benefit from using the "verbose" functions, which allow you to specify a database number and an statements number for every function that you use.

VERY IMPORTANT NOTE: If you are writing a *multithreaded* application, it will be nearly impossible for you to use the abbreviated functions. The `SQL_UseDB` and `SQL_UseStmt` functions (which are used to tell the abbreviated functions which database and statement numbers to use) affect *all threads at the same time*, so it is not usually practical to use them in multi-threaded applications.

Statement Zero Operation

If the ODBC driver that you are using will only allow your program to use one statement at a time (see Using Database Numbers and Statement Numbers), you will probably still be able to write a program that will do what you need it to do. You will simply be required to execute SQL statements sequentially, and while this can be less efficient than concurrent statements, it can be done.

There is an added complication, however, if your program uses any of the SQL Tools "Info" functions. Nearly all of the Info functions (also known as ODBC Catalog Functions) require SQL Tools to execute "behind the scenes" SQL statements to retrieve the information that you request. That means that if you attempt to use an Info function while a SQL statement is open, the ODBC driver won't let SQL Tools open the statement it needs in order to get the requested information.

Normally, your program should always use statement numbers between one (1) and the *lMaxStatementNumber* value that you specified with the *SQL_Initialize* function. SQL Tools uses "statement zero" for all of its Info functions, so it will never conflict with a statement that you are using.

If your ODBC driver can only handle one concurrent statement, you have three basic alternatives:

- 1)** Do not use any Info functions. If you are working with a database of known design, this is usually not difficult.
- 2)** Pre-load all of the Info functions that your program is going to need, by using the appropriate *SQL_Get* functions. These functions execute a SQL "Info" statement and cache the information, so that your program can access the information later *without* needing to execute a SQL statement. If the information is subject to change while your program is running -- such as when columns are added to a database -- this may not be a practical technique. See *Cached Information* for more details.
- 3)** Make sure that your program alternates between using open statements and Info functions, so that it does not attempt to use an Info statement while a SQL statement is open.

To make the third alternative easier to implement, SQL Tools provides a special option called Statement Zero Operation. Under normal circumstances, to avoid conflict with your statements, SQL Tools automatically uses statement number *zero* for all Info functions. That way, your program is free to use statement numbers one (1) and above, without being concerned about conflicts.

If you use statement number zero for your program's SQL statements (by using *SQL_UseStmt 0*, for example), the SQL Tools Statement AutoOpen and AutoClose features will make sure that statement zero is opened and closed properly, no matter which functions you use. You must keep in mind that *once you use an Info function, any open SQL statement will be automatically closed*, and you must write your program accordingly.

If you disable the Statement AutoOpen and AutoClose function (see *SQL_SetOptionSInt(OPT_AUTOOPEN_STMT)*), then executing an Info function while a statement is open will generate a *ERROR_STMT_NOT_CLOSED* Error Message, and the Info function will return a zero value or an empty string.

Cached Information

In order to obtain values for most of the "Info" functions (`SQL_DBInfo`, `SQL_TblInfo`, `SQL_TblColumnInfo`, etc.), it is necessary for SQL Tools to execute a special kind of "behind the scenes" SQL statement. It does so by automatically using statement number zero (see Statement Zero Operation), so that it does not interfere with your program's SQL statements.

When your program uses a SQL Tools Info function to request one piece of information, such as a column name, the ODBC driver automatically supplies a large amount of related information. In fact it returns *all* of the table's column names, their data types, their size, their precision... and on and on. The same thing is true for nearly all of the Info functions.

SQL Tools automatically caches all of that extra information (i.e. it stores it internally) so that if your program requests another column's name, it can give you an immediate answer from the cache instead of re-executing the SQL statement.

This technique works very well under most circumstances, and allows SQL Tools to provide *fast*, accurate information upon request. You will usually find that the first use of a given Info function is relatively slow, while the cache is being loaded, and that subsequent, related Info requests are very fast. For example, after you have used the `SQL_TblColumnInfoStr` function to obtain a column's name, you'll find that the `SQL_TblColumnInfoStr`, `SQL_TblColumnInfoInt`, and `SQL_TblColCount` functions will all return values *for the same table* very quickly.

There is a potential problem with this system, however, and (of course) a solution that your program may need to implement.

Nearly all Info values are static. For example, a table's column names do not usually change while your program is running. But if your program uses a SQL statement to add a column to a table, or to delete a column, the information in the SQL Tools cache will become out-of-date. If you attempt to use a SQL Tools Info function to obtain information about the *new* column, SQL Tools will return incorrect information. The same thing can happen if another program adds, deletes, or changes a column while your program is running.

The solution is to use the appropriate `SQL_Get` function to "refresh" the information in the SQL Tools cache. The `SQL_Get` functions force SQL Tools to re-read Info values and re-initialize the cached values. For example, if you add a column to a table you could use the `SQL_GetTblInfo` function to force SQL Tools to re-read the Table Information.

If your program is "mission critical" and there would be serious consequences if incorrect information was returned by a SQL Tools Info function, you should probably add the appropriate `SQL_Get` function to your program before *every use of an Info function*. This will greatly slow down the use of the Info functions, and is still not a *guarantee* of accurate information. For example, it is possible (albeit unlikely) that another program could add a column to a table in the split-second between the time that your program requests and uses the information.

Also see `SQL_InfoExport` and `SQL_InfoImport`.

Indexes

An Index is a structure that is maintained by a database, in order to speed up access to columns that have been indexed.

If your database maintains an Index for a particular column, it will be able to *find* values in that column much more quickly. However, it will take slightly longer to *update* an indexed column, because both the row and the index must be changed. It is therefore usually not a good idea to index every column in a database. (Not only that, but adding indexes tends to make databases significantly *larger*.)

To find out whether or not an ODBC driver supports indexes, you should use the following code:

```
If SQL_FuncAvail(API_STATISTICS) Then
    'THE DRIVER SUPPORTS INDEXES
End If
```

(The reason that ODBC drivers require the API_STATISTICS constant to be used is obscure and unimportant.)

If a driver does support indexes, you can use the following three functions to obtain information about them:

SQL_IdxCount returns the number of columns that have indexes.

SQL_IdxInfoStr and SQL_IdxInfoSInt can be used to obtain information about the indexes, such as the column names and data types.

Example code:

```
'
'Display the names of the indexes for table #3.
For lIndex& = 1 To SQL_IdxCount(3)
    Print SQL_IdxInfoStr(3, lIndex&, INDEX_COLUMN_NAME)
Next
```

AutoColumns

An AutoColumn is a column which is automatically updated when any value in the row is updated. (An AutoColumn is sometimes called a "Special" column. Another type of Special Column is the Unique Column.)

For example, many databases have a column called COUNTER. It is usually a SQL_INTEGER column that is not allowed to have a Null value, and the database automatically inserts a unique value into the column whenever the row is changed. It usually adds a predefined value (like 1) to the last-used value, to make sure that the same value is never used twice.

AutoColumns do not always contain *unique* values. Another common AutoColumn is often called LASTUPDATE, and it contains the date or date/time that the row was last changed. If the row is changed, the database automatically puts a new value in the LASTUPDATE column, so two or more rows could theoretically have exactly the same LASTUPDATE value.

If your ODBC driver supports AutoColumns, you can use these three SQL Tools functions to obtain information about them:

SQL_AutoColCount returns the number of AutoColumns that a table has.

SQL_AutoColInfoStr and SQL_AutoColInfoSInt can be used to obtain information about the AutoColumns.

Example code:

```
If SQL_FuncAvail(API_SPECIALCOLUMNS) Then
    'THE DATABASE SUPPORTS SPECIAL COLUMNS
End If

'print AutoColumn names...
For lCol& = 1 To SQL_AutoColCount
    Print SQL_AutoColInfoStr(lCol&,AUTOCOL_NAME)
Next
```


Unique Columns and Primary Columns

According to the Microsoft ODBC Software Developer Kit...

A Primary Key is a *"column or columns that uniquely identifies a row in a table"*, and...

A Unique Column is the *"optimal column or set of columns that, by retrieving values from the column or columns, allows any row in the specified table to be uniquely identified."*

As you can see, they are very similar. Both of those definitions are part of the ODBC 1.0 specification, but Primary Keys are only supported by ODBC drivers that support Level 2 functionality. In other words, nearly all ODBC drivers will allow you to use the `SQL_UniqCol` functions to obtain a list of Unique Columns, but only the more sophisticated ODBC drivers which support Level 2 will allow you to use the `SQL_PrimKey` functions to obtain a list of Primary Keys. (For example, Microsoft Access 97 does not support the `SQL_PrimKey` functions.)

For that reason, the rest of this discussion will focus on Unique Columns.

Another common name for a Unique Column is a "Special" column. (Another type of Special Column is the AutoColumn.)

The correct use of Unique Columns is critical to most non-`SELECT` SQL statements. For example, when an `UPDATE` statement is used to change a row's data, you will usually use the `WHERE` clause to specify which rows should be changed. Unique Columns provide a method of specifying which rows should be changed, without risking the possibility that other rows will be updated accidentally.

Well-designed tables almost always contain Unique Columns. For example, many tables contain a `COUNTER` column, which is automatically assigned a unique numeric value whenever a row is added or updated. (The database usually takes the last-used value and adds one, to make sure that the same value is never used twice.) A `COUNTER` column could be used to uniquely identify a row of a table without any possibility of error, because no two rows can ever have the same value. Your program could use an `UPDATE . . . WHERE` statement with complete confidence that only one row would be affected.

In some cases, two or more rows are combined to create a unique key or "Row ID". For example, if you were designing a database that contained one (and only one) row for each day of the year, you might have a `MONTH` column and a `DAY` column. You would have 31 different rows of data with the value `JANUARY` in the `MONTH` column (one row for each day in January), and you would have 12 different rows of data with `"1"` in the `DAY` column (one for each month), but you would only have one row with `JANUARY` and `"1"`. Those two columns could be "added together" to make a unique key, in which case the table would be said to have two Unique Columns, i.e. two columns that are used together to create a unique key.

SQL Tools provides three functions that allow you to determine which Unique Columns a table contains.

`SQL_UniqColCount` tells you how many Unique Columns are used to create a RowID. If this value is one (1), then a single column is sufficient to make sure that a row is identified.

`SQL_UniqColInfoStr` and `SQL_UniqColInfoSInt` provide information about the Unique Columns.

Also see [Cached Information](#).

Example code:

```
If SQL_FuncAvail(API_SPECIALCOLUMNS) Then
    'THE DATABASE SUPPORTS SPECIAL COLUMNS
End If

For lCol& = 1 To SQL_UniqColCount
    Print SQL_UniqColInfoStr(lCol&,UNIQCOL_COLUMN_NAME)
Next
```

Foreign Keys

A Foreign Key is a column (or a set of columns) in one table which matches the Primary Key in another table. *Generally* speaking, ODBC drivers that do not support Primary Keys (such as the Microsoft Access 97 driver) do not support Foreign Keys either.

If your ODBC driver supports Foreign Keys, SQL Tools provides three functions that can return information about them.

The `SQL_FornKeyCount` function returns the number of Foreign Keys that a table has.

The `SQL_FornKeyInfoStr` and `SQL_FornKeyInfoSInt` functions provide information about the Foreign Keys.

Also see [Cached Information](#).

Table Privileges and Column Privileges

A Privilege is an "access right" that is granted to a user, called the Grantee, by another user, called the Grantor. There are two basic kinds of Privileges: Table Privileges and Column Privileges. An ODBC driver may support one, both, or neither type. (For instance, the Microsoft Access 97 driver does not support either type of privilege.)

If Table Privileges have been specified for a certain table like `PAYROLL`, a certain user may have a "SELECT" privilege (the right to use the `SELECT` statement to retrieve data from the table) but not an "UPDATE" privilege (the right to change the values in the table). Other users might not have any rights to access the `PAYROLL` table in any way.

If Column Privileges have been specified for a certain column of the `PAYROLL` table, like `ANNUALSALARY`, a certain user may have a "SELECT" privilege (the right to use the `SELECT` statement to retrieve data from the column) but not an "UPDATE" privilege (the right to change the values in the column). Other users might not have any rights to access the `ANNUALSALARY` column in any way.

If your ODBC driver supports Privileges, SQL Tools provides functions that can return information about them.

The `SQL_TblPrivCount` and `SQL_ColPrivCount` functions return the number of privileges that a table has.

The `SQL_TblPrivInfoStr` and `SQL_ColPrivInfoStr` functions provide information about the privileges. (All Privilege information is provided in string form, so there are no corresponding `SInt` functions.)

Also see [Cached Information](#).

Example code:

```
'Display the privileges for Table #7...
```

```
For lPriv& = 1 To SQL_TblPrivCount
    Print SQL_TblPrivInfoStr(7, lPriv&, TBLPRIV_GRANTEE);
    Print " has the right to ";
    Print SQL_TblPrivInfoStr(7, lPriv&, TBLPRIV_PRIVILEGE);
    Print " table number 7."
Next
```

Committing Transactions Manually

Normally, every SQL statement that your program executes is "committed" immediately. In other words, database changes (if any) are made as soon as you execute the statement with the `SQL_Stmt` function.

But there may be times when you want to be more cautious than that. Many ODBC drivers support a "manual commit" mode, which allows your program to execute a SQL statement, examine the results (such as the value of the `SQL_ResRowCount` function), and then issue either **1**) a "commit" command that tells the database to make the changes permanent, or **2**) a "rollback" command that tells the database to return to the condition that it was in before the statement was executed.

You can determine whether or not a database can use the manual-commit mode by examining the results of the `SQL_DBInfoUIInt(DB_TXN_CAPABLE)` function.

If your ODBC driver allows it, you can activate the manual-commit mode with the `SQL_DBAutoCommit` function. (The manual-commit mode is often called the "transaction mode", because each SQL statement is treated as an individual transaction.)

Activate the manual-commit mode (i.e. disable the auto-commit mode) with this line of code *after* a database has been opened:

```
SQL_DBAutoCommit 0
```

After that, every time your program executes a SQL statement that can change the database, your program is responsible for using the `SQL_EndTrans` function like this:

```
SQL_EndTrans TRANS_COMMIT
```

...to commit the transaction, or...

```
SQL_EndTrans TRANS_ROLLBACK
```

...to tell the database *not* to make the changes.

It is not necessary to use the `SQL_EndTrans` function unless you have turned off the AutoCommit mode, or when you are using *SELECT* statements (which cannot change a database).

WARNING: If you do not use the `SQL_EndTrans` function to specify how a transaction should be completed, the default action is *not defined* by the ODBC specification. The transaction may or may not be automatically committed, so you should *always* use `SQL_EndTrans` to terminate a transaction.

It is also possible to re-activate the AutoCommit mode by using this code:

```
SQL_DBAutoCommit 1
```

Stored Procedures

As you probably know, the execution of a SQL statement is actually a two-step procedure. First the ODBC driver must "prepare" the statement, and convert it from a plain-language string into an executable program. Then it must "execute" the program. Even when you use the `SQL_Stmt (SQL_STMT_IMMEDIATE)` function, the ODBC driver breaks the process down into those two steps.

That means that SQL statements are treated as an "interpreted" language, and they cannot be executed as quickly as a "fully compiled" language would allow.

Fortunately, SQL Tools allows you to use something called a Stored Procedure to reduce or eliminate this problem. A Stored Procedure is actually a *pre-compiled* SQL statement that is stored *in* the database itself. Since the "preparation" step is performed long before your program is run, Stored Procedures can be executed much more quickly than string-based SQL statements.

IMPORTANT NOTE: The ODBC specification does not provide functions for creating and saving Stored Procedures, so SQL Tools is (of course) unable to provide those functions. It is *usually* possible to create and save a Stored Procedure by executing a SQL statement, but you should consult the documentation that was provided with your ODBC driver or your DBMS program (Microsoft Access, SQL*Plus, etc.) for specific instructions.

For example, if your program will need to use the following statement...

```
SELECT MYCOLUMN FROM MYTABLE WHERE YOURCOLUMN = 10
```

... you could pre-compile and save the statement as a Stored Procedure.

Stored Procedures are allowed to have bound parameters, so it is not necessary for the *entire* SQL statement to be pre-written and stored in the database. For example, you could compile and save the following Stored Procedure...

```
SELECT MYCOLUMN FROM MYTABLE WHERE YOURCOLUMN = ?
```

...and then insert the **?** value at the last minute, with the `SQL_BindParam` function.

The `SQL_ProcCount` function can be used to obtain the number of procedures that are stored in a database, and the `SQL_ProcInfoStr` and `SQL_ProcInfoSInt` functions can be used to obtain information like the procedure's name, the bound parameters that it requires (if any), and the result set that it will produce.

Once you have that information, you can (if necessary) use the `SQL_BindParam` function to bind the parameters of the procedure. Then you can use the `SQL_Stmt` function to execute the procedure.

Stored Procedures produce result sets that are exactly like those produced by string-based SQL statements, so you can use the entire range of `SQL_ResCol` functions to access the results.

MultiRow Cursors

A *MultiRow Cursor* (also called a *Block Cursor* or a *Row Array*) is a cursor that contains more than one row of a result set.

The current group of rows in a MultiRow cursor is called a "rowset". A rowset is a subset of a result set.

MultiRow cursors are useful for things like "data bound grid" displays and "spreadsheet" displays, where several rows of data can be displayed and edited on the screen, all at the same time. They can also be used for bulk operations and positioned operations.

When your program needs to handle more than one row at a time, it can retrieve and store the values for multiple rows internally, or, if your ODBC driver supports them, you can use a MultiRow Cursor.

When the `SQL_Fetch` or `SQL_FetchRel` function is used with a normal *single-row* cursor, the ODBC driver retrieves one row of data and places the various column and Indicator values into memory buffers. Each data buffer must be large enough to hold the longest value that a column can contain, and each Indicator buffer must be four bytes long.

When the `SQL_Fetch` or `SQL_FetchRel` function is used with a *MultiRow* cursor, instead of a single row of data, the ODBC driver retrieves two or more rows of data and places all of the column and Indicator values into extra-long memory buffers. Each data buffer must be large enough to hold the longest value that a column can contain *times* the number of rows in the cursor. Each Indicator buffer must be large enough to hold the number of rows *times* four bytes.

These extra-large buffers are often called "buffer arrays", because their memory structure resembles arrays of fixed-length data.

So if a certain result column can return a 256 byte string and you are using a 32-row cursor, the memory buffer for the column data would have to be 8192 (256 times 32) bytes long. The column value for the first row in the rowset would be stored in the first 256 bytes of the buffer, the column value for the second row would be stored in the next 256 bytes, and so on.

And since Indicator values are 4-byte `BAS_LONG` values, the Indicator buffer for each result column would have to be 128 (4 times 32) bytes long.

MultiRow data buffers are usually created using the same techniques that are used for manual result column binding. In fact, you must use the `SQL_ManualBindCol` function to bind each buffer array and Indicator array to a column of a result set.

MultiRow Indicator buffers are usually created with `BAS_LONG` arrays, so that the individual Indicator values can be accessed easily.

MultiRow cursors can be very complex, and they are usually accessed via direct-from-memory techniques, so SQL Tools does not *directly* support them, i.e. it does not provide ready-to-use functions (like `SQL_AutoBindCol`, which is used for single-row cursors) that can be used to bind MultiRow cursors. SQL Tools does, however, give you access to 100% of the tools that you will need to create a MultiRow cursor, no matter how complex it is.

Before you attempt to create a MultiRow cursor, you should familiarize yourself with two relatively complex topics:

1) You should experiment with using Manual Result Column Binding with a normal, single-row cursor. This will familiarize you with the techniques that are required for creating, binding, and maintaining data buffers and Indicator buffers. (It is not enough to practice with Proxy Binding and Direct Binding, which are less complex than Manual Binding.) You may also need to experiment with *retrieving* data and Indicator values from manually-bound columns, which cannot be accessed with the normal `SQL_ResCol` functions.

2) You should then review the six `SQL_SetStmtAttrib` functions that are related to MultiRow cursors. The attributes that you will need to use all start with `STMT_ATTRIB_ROW`.

We also recommend that you study the Microsoft ODBC Software Developer Kit, which contains extensive information about MultiRow cursors.

Named Cursors

Named Cursors are used only in "positioned" update and delete statements. For example, if you execute a SQL statement and position the statement's cursor on a certain row, you can then execute another statement that uses the first statement's cursor position as a parameter. (The second statement would have to use a *different* Statement Number so that the first statement won't be automatically closed. Both statements must be open at the same time.) The second SQL statement would look something like this:

```
UPDATE table-name ...WHERE CURRENT OF cursor-name
```

...where *cursor-name* is the name of the first statement's cursor.

Whenever you prepare or execute a SQL statement that creates a cursor, the ODBC driver automatically gives it a name. The automatically-assigned name will always start with the string "SQL_CUR", and it will be less than 18 characters long.

You can obtain the name of an open cursor by using the `SQL_CurName` function, and you can assign a new name with the `SQL_NameCur` function.

All of the cursor names that are used with a database must be unique, i.e. no two open cursors may have the same name.

Cursor names may not exceed 18 characters in length, and may not contain any special characters (as defined by the `SQL_DBAtribUInt(DB_SPECIAL_CHARACTERS)` function).

ODBC 3.x+ drivers always treat *quoted* cursor names in a case-sensitive manner, and quoted names can contain characters that would not normally be permitted, such as blanks and reserved words.

Bulk Operations

"Bulk Operations" (the `SQL_BulkOp` function) can be used to perform the following operations on a table that has been accessed with a `SELECT` statement:

- 1) Fetch one or more rows that are identified by bookmarks.
- 2) Update one or more rows that are identified by bookmarks.
- 3) Delete one or more rows that are identified by bookmarks.
- 4) Add new rows.

IMPORTANT NOTE: Not all ODBC drivers support bulk operations. In fact, according to the Microsoft ODBC Software Developer Kit, bulk operations are "not widely supported". For that reason, and because bulk operations can be *very* complex, they are not covered in great detail in this document. This document provides an overview of bulk operations that should be sufficient to get you started, but for details and advanced techniques you should refer to the Microsoft ODBC SDK.

To determine which bulk operations a driver supports (if any), you can use the `SQL_DBInfoUInt (SQL_type_CURSOR_ATTRIBUTES1)` function, where *type* is the type of cursor that is being used (STATIC, DYNAMIC, etc.).

All bulk operations use MultiRow cursors. If you are not familiar with multi-row or "block" cursors, you should read about MultiRow Cursors for background information before reading this section.

Bulk operations also use Bookmarks, so you should also be familiar with that topic before reading this section.

How Bulk Operations Work

Visualize a memory structure that you have used to create a ten-row MultiRow Cursor for a SQL statement. Each column of the result set would be bound to a data buffer and an Indicator buffer, each of which is really a "buffer array" that is large enough to hold ten rows of data (or ten Indicators) for a given column.

Nearly all bulk operations are based on bookmarks, so you should also picture a data buffer array and Indicator array for column zero.

If you execute a `SELECT` statement, manually bind the result columns to the buffer arrays, and use the `SQL_Fetch` function, the buffer arrays will be automatically filled with data. The buffer arrays will then contain the column data for ten rows of the result set (assuming that the SQL statement generated ten or more rows).

If your program was to then *change* the values in the data and Indicator *buffers*, and then use the `SQL_BulkOp (BULK_UPDATE)` function, the ODBC driver would *change the database* to reflect the new values. (Keep in mind that this was done without using an `UPDATE` statement.)

If you were to use the `SQL_BulkOp (BULK_DELETE)` function, all of the rows in the database that correspond to the rows in the current rowset would be deleted. (Note that this was done without using a `DELETE` statement.)

And if you were to use the `SQL_BulkOp (BULK_FETCH)` function after you had used

BULK_UPDATE or BULK_DELETE, the buffer arrays would be "refreshed" with data from the table, so that you could confirm that the operation worked. (You do not have to use *SELECT* again to refresh the rowset.)

Finally, you can create a MultiRow Cursor buffer structure and fill it with new values, and then use `SQL_BulkOp(BULK_ADD)` to add new rows to a table without using an *INSERT* statement.

A more sophisticated method of using bulk operations would be to create buffer arrays that are larger than you actually need for the original rowset. For example, if the rowset was 32 rows long you might create buffers that are large enough for 64 rows. `SQL_Fetch` would be used to load values into the buffers for 32 rows, and then your program could copy data and Indicator values *for selected rows* into the buffer space for the other 32 rows. For instance, if the first 32 rows were being displayed to a user, a row's column and Indicator values might be copied when the user "tagged" a row by double-clicking on it. Then, when the user clicked a "Delete All Tagged Rows" button, your program would re-bind the columns of the result set to the sections of the buffers that contain the *selected* rows, and then use the `SQL_BulkOp(BULK_DELETE)` function to delete them.

Using BULK_UPDATE

It is important to remember that when the `SQL_BulkOp(BULK_UPDATE)` function is used, *all* of the data in the currently-bound data and Indicator buffers will be transferred to the database.

That means that if your result set contains a Long column that is bound to a narrow "preview" buffer, only the data that is currently in the buffer will be sent to the database. So there is a very good chance that the Long column value *in the database* will be truncated.

To avoid this problem, you can set a column's Indicator to the special value `SQL_IGNORE` (in all of the rows of the rowset), which tells the `SQL_BulkOp` function not to update a column's value.

Using BULK_ADD

In order to use any of the `SQL_BulkOp` functions, you must first execute a SQL statement that creates a result set. That means that, even if you don't care about the current contents of a database, in order to use `SQL_BulkOp(BULK_ADD)` you must first execute a SQL `SELECT` statement.

It is therefore usually more efficient to use an `INSERT` statement to add rows to a database than it is to use `SELECT` and `BULK_ADD`.

If you decide to use `BULK_ADD` you must execute a SQL statement to create a *non-empty* result set (i.e. a result set that contains at least one row), but you are not required to use `SQL_Fetch` to actually *retrieve* any of the rows.

VERY IMPORTANT NOTE: If you do decide to use `BULK_ADD`, you *must* create data and Indicator buffer arrays for the bookmark column, and you must use them to bind result column zero. You do not have to provide *values* for column zero -- the ODBC driver will automatically fill them in when it inserts the rows into the table -- but if you fail to create and bind bookmark buffers the rows may be added to the table without bookmarks, which can cause the table to be corrupted.

Using BULK_FETCH

It can be dangerous to use `BULK_FETCH` to create a rowset that is then modified and used for `SQL_UPDATE`. For example, if your result set contains a Long column that is bound to a narrow "preview" buffer, only a small amount of the data that the column actually contains will be placed in the buffer by `SQL_BulkOp(BULK_FETCH)`. That means that when `BULK_UPDATE` is used and the data in the buffer is sent to the database, there is a very good chance that the Long column value *in the database* will be truncated.

To avoid this problem, you can set a column's Indicator (in all of the rows of the rowset) to the special value `SQL_IGNORE`, which tells the `SQL_BulkOp` function not to update a column's value.

Using BULK_DELETE

You must, of course, use this function with caution. It can cause large numbers of rows to be deleted. You should treat `SQL_BulkOp(BULK_DELETE)` with the same respect that is given to the SQL *DELETE* statement.

Positioned Updates and Deletes

Positioned updates and positioned deletes are performed with the `SQL_SetPos` function. They are very similar to bulk operations, but with some added complexities.

1) Unlike bulk operations, positioned operations can optionally be performed on a *single* row of a MultiRow Cursor.

2) Positioned "add row" operations are not supported. You must use `SQL_BulkOp(BULK_ADD)`.

3) Row locking is supported.

Not all ODBC drivers support positioned operations. To determine which operations a driver supports (if any), you can use the `SQL_DBInfoUInt(SQL_type_CURSOR_ATTRIBUTES1)` function, where *type* is the type of cursor that is being used (STATIC, DYNAMIC, etc.).

For more information, see `SQL_SetPos` or consult the Microsoft ODBC Software Developer Kit.

Using Long Values with Bulk and Positioned Operations

Long values can be sent to the `SQL_BulkOp` and `SQL_SetPos` functions in "chunks" by using the `SQL_LongParam` function.

The Long columns of a result set are not *usually* bound to data buffers, but this process requires you to manually bind Long columns in an unusual way.

First, create data and Indicator buffers for all of the *non*-Long columns.

You should then create a data buffer for each Long column that is large enough to hold a 4-byte `BAS_LONG` value for each row in the rowset. For example, if you are using a 32-row rowset, you would need a 128-byte (4 times 32) buffer.

You should also create a "normal" Indicator buffer for the Long column, just as you would if it was not a Long column.

Bind all of the non-Long columns, then bind the Long column(s) to the data buffer(s) and Indicator buffer(s) by using the `SQL_ManualBindCol` function. (If you are using MultiRow Cursors you should already be familiar with this process.)

Next you must set the Indicator value for each Long column *in every row of the rowset* to a special value. To determine which special value you must use, examine the results of the `SQL_DBInfoStr(SQL_NEED_LONG_DATA_LEN)` function. If it does *not* return "Y" you should simply set every Long column's Indicator value to `SQL_LONG_DATA`. If it does return "Y", you must set each Long column's Indicator value to the number that is given by the following formula:

```
Indicator = 0 - (DataLength + 100)
```

In other words, add 100 to the length of the Long data, and make the value negative. If the Long data for a certain column in a certain row is 9000 bytes long, the special Indicator value that you must use for that column in that row would be `--9100`.

Note: Once you have determined whether or not "Y" is returned by a certain ODBC driver for a certain database, you do not need to repeat the `DB_NEED_LONG_DATA_LEN` test. You can assume that the answer will always be the same, and remove the test code.

Then you must set the value of the *data* buffer for each Long column *in every row of the rowset* to a different (i.e. *unique*) value. For a simple example, let's assume that there is only one Long column in the rowset. Creating a different value for the Long column in each row is easy: you would simply use the row number. The value of the `BAS_LONG` data buffer for the Long column in row 1 would be 1, the value of the data buffer in the Long column in row 2 would be 2, and so on.

It gets a little more complicated if you have more than one Long column per row. Each Long cell (each Long column in each row) must be given a *unique* value. You can create the unique value in any way that you want to, as long as your program can "decode" it later. For example, you might use the number 1001 for row 1, column 1, and the value 1002 for row 1, column 2, and the value 2001 for row 2, column 1, and so on. As long as every Long column in every row is given a *unique* value, you can use whatever numbering system that you like. *No two cells may have the same value.*

Now that the Indicator buffers and data buffers for all of the Long columns in all of the rows have all been set to the required values (so that the ODBC driver will know that the columns contain Long data), we can continue with the processing of the rowset...

When the `SQL_BulkOp` or `SQL_SetPos` function is used, if there are any Long columns that need data, the function's return value will be `SQL_NEED_DATA` instead of `SUCCESS`.

1) Your program must then use the `SQL_NextParam` function to obtain the unique number that identifies the row/column that needs data. *You must use the `SQL_NextParam` function even if you know that only one Long column needs data.* The number that will be returned by this function will be the "unique" value that you chose above, and it indicates that the ODBC driver is ready to receive the data for a certain cell.

2) Your program should then use the `SQL_LongParam` function one or more times, to send data for the Long column in the row that is identified by the unique value. To send a value that is stored in a string variable called `sLongData$`, use this code:

```
SQL_LongParam(sLongData$, LEN(sLongData$))
```

If you want to send a Null value to a Long column, use...

```
SQL_LongParam(" ", SQL_NULL_DATA)
```

You can use `SQL_LongParam` repeatedly, to send the data in "chunks", if that is convenient. For example, if the Long parameter value was stored in two different variables called `sLong1$` and `sLong2$`, you would use this code...

```
SQL_LongParam(sLong1$, LEN(sLong1$))
SQL_LongParam(sLong2$, LEN(sLong2$))
```

...and SQL Tools would automatically add together *all* of the strings that you submit in this way.

3) When you are done sending the first Long value, your program must use the `SQL_NextParam` function again, A) to tell SQL Tools that you are done sending Long data for the cell, and B) to get the unique number of the *next* Long column (if any) that needs data.

4) If there are more Long columns that need data, the `SQL_NextParam` function's return value will indicate the cell's "unique" number, and you must use `SQL_LongParam` to send the appropriate value.

5) When `SQL_NextParam` finally returns `SUCCESS` (or an Error Code) the data-sending process is finished, and the `SQL_BulkOp` or `SQL_SetPos` operation will be automatically completed using the Long data that you supplied.

6) If an error occurs *before* `SQL_NextParam` returns `SUCCESS` and your program is unable to recover from the error, it should always use the `SQL_StmtCancel` function to make sure that the `SQL_BulkOp` or `SQL_SetPos` operation is aborted correctly.

7) If your program is going to use `SQL_Fetch` or `SQL_FetchRel` again *after* sending Long data in this way, *you must remember to use the `SQL_UnbindCol` function to unbind the Long column(s)*, or an Error Message will be generated by the fetch operation when the ODBC driver tries to use the 4-byte buffer for a Long value.

"Cleaning Up" After a Bulk Operation

After the `SQL_BulkOp` function has been used, the MultiRow Cursor's position is "undefined". That means that the ODBC driver momentarily "gets lost", and your program must use `SQL_Fetch` or `SQL_FetchRel` to set the cursor position after every `SQL_BulkOp` operation.

IMPORTANT NOTE: You may *not* use `SQL_FetchRel` to perform a Relative Fetch *without a bookmark* after a `SQL_BulkOp` operation. You may use `SQL_Fetch`, and you may use `SQL_FetchRel` with a bookmark string, but you may not use `SQL_FetchRel` to perform a fetch that is relative to the current cursor location, because the current cursor location is undefined.

Using SQL Tools with a Grid

A grid is a user-interface element (i.e. a "visual" part of a program) that resembles a spreadsheet. Grids have rows and columns, so they are a natural way to display the contents of a result set.

Visual Basic version 5.0 includes grid controls called MSFlexGrid and DBGrid, and VB 6.0 includes the MSFlexGrid and MSHFlexGrid controls. Several third-party grids such as FarPoint Spread, Infragistics UltraGrid and DataTable, and Apex DBGrid Pro and VSFlexGrid Pro are also available. Source code for many different grid controls can also be found on the internet.

PowerBASIC programmers often use the FarPoint Spread grid, because **1)** it is available as a DLL and **2)** FarPoint will provide registered users with a PowerBASIC declaration file. Other DLL-based grids are also available.

A "data bound grid" is a one that is connected *directly* to a database. The database and the grid control are linked -- "bound" together -- in a way that allows a result set to directly affect the display, and sometimes vice versa. *SQL Tools is not intended for use as a data source that is bound to a grid control.* That doesn't mean, however, that you can't display a SQL Tools result set in a grid control.

The Visual Basic MSFlexGrid control can be used to display virtually any type of data. VB 5.0 also includes a sample program called UnBndGrid that demonstrates how the DBGrid control can be used to display the contents of a string array.

Virtually all third-party grids can be used in an "unbound" mode as well.

Some third-party grid controls can also be used in a "virtual mode" where your program, not the grid, stores the strings and numbers that the grid displays. (In fact, that is the most *efficient* way to use most grids.) When the grid needs to display a row, it performs a "callback" operation to ask your program for the necessary data. This can be a very good way to display a result set -- particularly a *large* result set -- because it allows your program to manage the data. For example, your program could fetch just the rows that are needed to display the current "page" of the grid, and fetch other rows only as necessary, when the user scrolls the grid. While this is usually slower than fetching all of the rows and storing them in an array, it requires *far* less memory. If the result set is extremely large, you may have no choice but to use the virtual mode. (Tip: It's a good strategy to "cache" extra rows when using the virtual mode. For example, your program could store the current page, plus the pages just above and below the current page. If the user scrolls the display down one page, the grid could instantly display the necessary rows. Then, while the user is looking at those rows, your program could fetch the next "cache" page.)

Multi-Threaded Programs

There are two different ways to use SQL Tools in a "multi-threaded" mode. The first is "asynchronous execution" which is covered in the section of this document called Asynchronous Execution of SQL Statements. The second method is true Windows multi-threading, which is covered in this section.

Generally speaking, when a Windows program is run it creates one "thread of execution". This thread is usually where 100% of the program's operations take place. It is possible, however, for one Windows program to execute two (or more) different threads, each performing its own operations. A program's threads all *share* global-scope variables. It's like having two different functions in the same program executing at the same time, one managing the user interface (for example) and another performing some other activity "in the background".

PowerBASIC programmers can use the `THREAD` functions (`THREAD CREATE`, etc.) to create and manage threads. Many other programming languages such as C and Delphi also support threads using different syntax.

Visual Basic, however, does not support true multi-threading. If you need to perform multi-threaded operations in a VB program, you must use the "async" functions described in Asynchronous Execution of SQL Statements.

IMPORTANT NOTE: SQL Tools can, but not all ODBC drivers can handle multithreaded operation. If your ODBC driver is not "thread safe" you should not attempt to create a multithreaded program.

IMPORTANT NOTE: If you use the PowerBASIC `THREAD` functions (or the equivalent functions in another language) we strongly recommend that you become *very* familiar with them before reading this section of this document. Using threads can be very complicated! However, if you use the SQL Tools "async" functions, most of the common problems can be easily avoided.

For the purposes of this discussion, we need to define some terms. When this document refers to a "primary thread", it is referring to your program's *original* thread of execution, i.e. the thread that is automatically launched when your program is executed. A "secondary thread", on the other hand, is any thread that is launched with a PowerBASIC `THREAD CREATE` statement or a comparable function in another language. Keep in mind that your program can have *many* secondary threads running at the same time. "Secondary" does not necessarily imply "two".

Multithreading can introduce a number of programming complexities. For example, consider the following code:

```
SQL_ErrorClearAll

SQL_OpenDB " *.DSN"

If SQL_ErrorPending Then
    'an error occurred during SQL_OpenDB process
End If
```

First, the `SQL_ErrorClearAll` function is used to remove any error messages that might be in the Error Stack. Then, after the `SQL_OpenDB` function has been used to open a database,

the value of the `SQL_ErrorPending` function is checked, to find out whether or not any error messages have been added to the stack. Presumably, if `SQL_ErrorPending` returns a Logical True value at that point, the `SQL_OpenDB` function must have generated an error message.

In a *single*-threaded program that is a reasonable assumption, but if two or more threads are running at the same time, another thread may have added an error message to the Error Stack *while the `SQL_OpenDB` function was executing*. An error that is in the stack *may* have nothing to do with the `SQL_OpenDB` function. Or it *may* have come from the `SQL_OpenDB` function.

As you can see, using multiple threads can greatly complicate a program. Fortunately, SQL Tools contains a function called `SQL_Thread` which was specifically designed to make things easier. While it is theoretically possible to use SQL Tools in a multithreaded program without using `SQL_Thread`, we do not recommend it.

At the very beginning of your program, right after the `SQL_Init` function, you should use the `SQL_Thread` function to tell SQL Tools how many different threads you expect to use. In this example, we'll anticipate using four (4) different threads:

```
SQL_Thread THREAD_MAX, 4
```

That line tells SQL Tools "get ready for up to four threads that use SQL Tools functions". (If your program creates threads dynamically and you're not sure how many threads it will use at one time, don't worry. You can use `SQL_Thread THREAD_MAX` to increase or decrease the value later.)

IMPORTANT NOTE: The `SQL_Thread THREAD_MAX` function can be used only in your program's *primary* thread. It can not be used in a thread that is launched with `THREAD CREATE`. We suggest that you add it to your `WinMain`, `Main`, or `PBMain` function immediately following `SQL_Init` or `SQL_Initialize`. If you attempt to use `THREAD_MAX` in a secondary thread, an Error Message will be generated.

The next step is to set up your `THREAD CREATE` statement and launch a second thread of execution. For details, see your programming language documentation. For this example, we will assume that you have launched a thread that executes a function called `MyThread`.

The *very first* executable line of `FUNCTION MyThread` should look like this:

```
SQL_Thread THREAD_START, 1
```

That line assigns the number one (1) to the thread. You can use any number between one and the `MAX_THREAD` value that you chose, but *each thread that you create must use a different thread number*. (To be clear, you *can* start thread number one, allow it to finish, and then start *another* thread using the number one. But no two threads can use the same thread number *at the same time*.)

The *very last* executable line of `FUNCTION MyThread` should look like this:

```
SQL_Thread THREAD_STOP, 1
```

If `FUNCTION MyThread` contains any `EXIT FUNCTION` statements, you must also use `SQL_Thread THREAD_STOP, 1` immediately before *every possible exit point* from the thread.

TIP: If you use a "wrapper" function, like this...

```
Function MyThread(ByVal lParam As Long) As Long
    SQL_Thread THREAD_START,1
    Function = ThreadFunc(lParam)
    SQL_Thread THREAD_STOP,1
End Function
```

...and you then place all of your code in the ThreadFunc function, you won't have to worry about Exit Function. In this example, no matter what happens in your code, when ThreadFunc ends, the SQL_Thread THREAD_STOP,1 function will be executed properly. (The names MyThread and ThreadFunc are only examples. You can use any names that you like.)

IMPORTANT NOTE: While the SQL_Thread THREAD_MAX function can be used only in your program's primary thread, SQL_Thread THREAD_START and THREAD_STOP can be used *only* in *secondary* threads. They can not be used in your program's primary thread. The primary thread is handled automatically by SQL Tools.

The SQL_Thread THREAD_START,1 function tells SQL Tools "a new thread has been launched, and it is called thread number 1". SQL Tools then creates an Error Stack for the thread, so that when you use the various SQL_Error functions (SQL_ErrorPending, SQL_ErrorText, SQL_ErrorQuickOne, etc.) they will provide information *only* about errors that were produced by that thread.

If you use one of the SQL_Error functions in your primary thread it will return information about errors that were generated by the primary thread, and if you use one of the SQL_Error functions in thread number 1, it will return information about errors that were generated by thread number 1.

You are then free to use THREAD_CREATE to launch additional secondary threads. As long as each new thread uses THREAD_START with a different thread number, each thread will have its own error stack.

The use of THREAD_START also tells SQL Tools to track the value of the SQL_MsgBoxButton function for each thread individually. For example, if the SQL_MsgBox function is used in one thread and somebody selects the Ok button, only that thread's SQL_MsgBoxButton function will be affected.

Functions to AVOID in Multithreaded Programs

There are a few SQL Tools functions that are very difficult to use in a multithreaded program. In particular, the SQL_New functions (SQL_NewDBNumber, etc.) should be avoided because if two threads happen to use it at exactly the same time, they will both receive the same return value. For example, consider this code:

```
'Get an unused statement number for database number 1:
lStmtNumber& = SQL_NewStatementNumber(1)

'Open a statement using that statement number:
SQL_OpenStatement 1, lStmtNumber&
```

In a *single*-threaded program that will work perfectly. But in a multithreaded application, it would be possible for the SQL_NewStatementNumber function to return a value like 2, but

for Statement Number 2 to be used by another thread *a split-second later*, so the `SQL_OpenStatement` function could fail.

In multithreaded programs, it is usually best to **1)** use hard-coded database and statement numbers or **2)** use database and/or statement numbers that are based on the thread number. For example, thread zero (the primary thread) might always use statement number 1, and thread number 1 might always use statement number 2, and so on.

Cached Information Functions

Multithreaded programs can also have trouble with the various cached information functions that SQL Tools provides.

If your program requests a piece of information (such as a column name or the number of tables in a database), SQL Tools first checks its internal cache of information. If the cache does not contain the requested value, SQL Tools automatically uses one of the `SQL_Get` functions to get the information from your ODBC driver. The information is then returned to your program *and* it is stored in the cache, in case similar information is requested in the future. (See *Cached Information* for more details about this process.)

The use of cached information greatly speeds up your program's access to certain types of information, but it can sometimes cause problems for a multithreaded program. Imagine that your program's primary thread has just used the `SQL_TblCount` function for the first time. SQL Tools checks the cache and finds that the necessary information is not there, so it begins the process of filling the cache. This process can take up to several seconds. While it is working, imagine that your program's second thread calls another Table Information function, such as `SQL_TableInfoStr`. SQL Tools checks the cache and finds that the necessary information is not there, so the second thread *also* begins the process of filling the cache. In most cases this will simply result in a small amount of wasted processor time, but it is possible for the two processes to "collide" and to return incorrect values.

To avoid potentially serious problems, we suggest that your primary thread use the appropriate `SQL_Get` functions to fill the various Info caches before it launches any threads.

See the `SQL_Thread` function for more information about multithreaded programs.

Also see *Asynchronous Execution of SQL Statements*.

SQL Handles

Perhaps the most "advanced" uses of SQL Tools require the use of ODBC Handles. The `SQL_hEnvironment`, `SQL_hDatabase`, and `SQL_hStatement` functions can be used to obtain the actual handle values that SQL Tools uses to interact with the ODBC driver.

WARNING: SQL Tools supports virtually 100% of the functions that ODBC provides. If an ODBC feature is not supported by SQL Tools, there is probably a very good reason for it, and you should consider whether or not you *really* need to use the feature.

For example, while SQL Tools *does* support thread-based asynchronous execution of SQL statements, it does not support ODBC-based Asynchronous Execution. According to the Microsoft ODBC Software Developer Kit, *"In general, applications should execute functions asynchronously only on single-threaded operating systems. On multithread operating systems,"* [such as Windows] *"applications should execute functions on separate threads, rather than executing them asynchronously on the same thread. No functionality is lost if drivers that operate only on multithread operating systems do not support asynchronous execution."* If you attempt to add support for this feature to SQL Tools, you will probably find that most of the Info function will fail to work properly, and you will have to manually add support for those functions as well.

After all of that, you're probably asking yourself "so why are the `SQL_h` handle functions even *provided* by SQL Tools?" The primary reason is something called "descriptors". Here is what the ODBC SDK has to say about them: *"An application calling ODBC functions need not concern itself with descriptors. No database operation requires that the application gain direct access to descriptors. However, for some applications, gaining direct access to descriptors streamlines many operations. For example, direct access to descriptors provides a way to rebind column data that may be more efficient than calling SQLBindCol again."*

The various SQL Tools "handle" functions are provided so that you can use Descriptors if you need them.

Reference Guide Format

Each SQL Tools function has a page in the Reference Guide that looks something like this:

SQL_FunctionName

Summary

A brief description of the function's purpose.

Twin

The verbose or abbreviated function (see) that performs the same purpose as this function.

Family

The "functional family" to which the function belongs.

Availability

Either "Standard and Pro" or **"SQL Tools Pro Only"** (see)

Warning

Critical warnings will be shown here in **RED**. Important but less-than-urgent warnings are shown in **DARK RED**.

Syntax

The basic syntax that you must use in your source code.

Parameters

A list of the parameters that you must pass to the function, and their basic purposes.

Return Values

The numeric or string values that can be returned by the function.

Remarks

A detailed discussion of the function.

Diagnostics

The Error Codes and Error Messages that the function can generate.

Example

A brief BASIC source code example.

Driver Issues

This section is reserved for known issues with various drivers (such as driver bugs), and for other *specific* warnings. This section will not say things like *"This function is only supported by SQL Tools if your ODBC driver supports it"*, because that statement is true for virtually *all* SQL Tools functions.

Speed Issues

A discussion of speed- and performance-related issues, such as the optimum way to use a function.

See Also

A brief list of related topics.

Functional Families

Each SQL Tools function has been assigned to a "family", to make it easier to find related functions. Each page of the Reference Guide lists the function's family, so you can easily look up related functions.

Here is an alphabetical list of all of the SQL Tools Functional Families. (If you read the following pages in order, the Families will be presented in an order that naturally leads from one to the next.)

- Configuration Family
- Database Info/Attrib Family
- Database Open/Close Family
- Environment Family
- Error/Trace Family
- Get Info Family
- Handle Family
- Result Column Binding Family
- Result Column Family
- Result Count Family
- Statement Binding Family
- Statement Family
- Statement Info/Attrib Family
- Statement Open/Close Family
- Stored Procedure Family
- Table Column Info Family
- Table Info Family
- Use Family
- Utility Family

Configuration Family

SQL Tools Initialization and Shutdown functions, plus functions that allow you to set and get various "option" values, which are used to configure SQL Tools.

Program startup and shutdown:

```
SQL_Authorize
SQL_Initialize,    SQL_Init
SQL_Shutdown
```

SQL Tools Options:

```
SQL_OptionSInt,    SQL_OptionStr
SQL_SetOptionSInt, SQL_SetOptionStr
SQL_OptionResetAll
```

Save/Load Info:

[SQL Tools Pro Only...](#)

```
SQL_InfoExport
SQL_InfoImport
```

Thread startup and shutdown:

[SQL Tools Pro Only...](#)

```
SQL_Thread
```

Environment Family

Functions for setting and getting attributes and information about the overall ODBC environment in which your program operates. These values include the ODBC version, the names of the various ODBC drivers and datasources that are available to your program, and information about things like "connection pooling", which affect all of the databases in the environment.

ODBC Environment Attributes:

```
SQL_SetEnvironAttribSInt  
SQL_EnvironAttribSInt
```

Available ODBC Drivers:

[SQL Tools Pro Only...](#)

```
SQL_DriverCount  
SQL_DriverInfoStr  
SQL_DriverNumber
```

Available ODBC Datasources:

[SQL Tools Pro Only...](#)

```
SQL_DataSourceAdd  
SQL_DataSourceAdmin  
SQL_DataSourceCount  
SQL_DataSourceInfoStr  
SQL_DataSourceModify  
SQL_DataSourceNumber
```

Use Family

Function that allow you to set and get the Current Database and Current Statement numbers, which are used by all of the SQL Tools "abbreviated" functions.

Setting:

```
SQL_UseDB  
SQL_UseStmt  
SQL_UseDBStmt
```

Getting:

```
SQL_CurrentDB  
SQL_CurrentStmt
```

Database Open/Close Family

Functions related to the opening and closing of Databases.

SQL_NewDatabaseNumber,	SQL_NewDBNumber
SQL_OpenDatabase,	SQL_OpenDB
SQL_OpenDatabase1,	SQL_OpenDatabase2
SQL_DatabaseIsOpen,	SQL_DBIsOpen
SQL_CloseDatabase,	SQL_CloseDB

Database Info/Attrib Family

Functions that allow you to obtain various Database Attribute and Information values, and to set Database Attribute values. (Generally speaking, SQL Tools "Attribute" settings can be changed, and "Information" settings cannot be changed.)

General Database Information:

SQL_DatabaseInfoStr,	SQL_DBInfoStr
SQL_DatabaseInfoUInt,	SQL_DBInfoUInt

Information about a database's basic ODBC capabilities:

SQL_FunctionAvailable,	SQL_FuncAvail
------------------------	---------------

Database Attributes:

SQL_DatabaseAttribStr,	SQL_DBAtribStr
SQL_DatabaseAttribUInt,	SQL_DBAtribUInt

[Most sub-functions are limited to SQL Tools Pro Only...](#)

SQL_SetDatabaseAttribUInt,	SQL_SetDBAtribUInt
----------------------------	--------------------

[SQL Tools Pro Only...](#)

SQL_SetDatabaseAttribStr,	SQL_SetDBAtribStr
---------------------------	-------------------

Information about the Data Types that are supported by a database:

[SQL Tools Pro Only...](#)

SQL_DatabaseDataTypeCount,	SQL_DBDataTypeCount
SQL_DatabaseDataTypeInfoSInt,	SQL_DBDataTypeInfoSInt
SQL_DatabaseDataTypeInfoStr,	SQL_DBDataTypeInfoStr
SQL_DatabaseDataTypeNumber,	SQL_DBDataTypeNumber

Database Transaction Mode:

[SQL Tools Pro Only...](#)

SQL_DatabaseAutoCommit,	SQL_DBAutoCommit
SQL_EndTransaction,	SQL_EndTrans

Table Info Family

Functions that allow you to obtain information about the tables in a database, such as the number of tables, their names, their Table Types, and any remarks that the table's creator included in the database.

General Table Information:

SQL_TableCount,	SQL_TblCount
SQL_TableInfoSInt,	SQL_TblInfoSInt
SQL_TableInfoStr,	SQL_TblInfoStr
SQL_TableNumber,	SQL_TblNumber

Table Statistics:

[SQL Tools Pro Only...](#)

SQL_TableStatisticSInt,	SQL_TblStatSInt
-------------------------	-----------------

Table Privileges:

[SQL Tools Pro Only...](#)

SQL_TablePrivilegeCount,	SQL_TblPrivCount
SQL_TablePrivilegeInfoStr,	SQL_TblPrivInfoStr

Table Column Info Family

Functions that allow you to obtain information about the columns in a table, such as how many columns there are, their names and types, and whether or not they are nullable.

(For functions related to *Result* Columns, see the Result Column family.)

General Table Column Information:

SQL_TableColumnCount,	SQL_TblColCount
SQL_TableColumnInfoSInt,	SQL_TblColInfoStr
SQL_TableColumnInfoStr,	SQL_TblColInfoSInt
SQL_TableColumnNumber,	SQL_TblColNumber

Column Privileges:

[SQL Tools Pro Only...](#)

SQL_ColumnPrivilegeCount,	SQL_ColPrivCount
SQL_ColumnPrivilegeInfoStr,	SQL_ColPrivInfoStr

Unique Columns:

[SQL Tools Pro Only...](#)

SQL_UniqueColumnCount,	SQL_UniqColCount
SQL_UniqueColumnInfoStr,	SQL_UniqColInfoStr
SQL_UniqueColumnInfoSInt,	SQL_UniqColInfoSInt

AutoColumns:

[SQL Tools Pro Only...](#)

SQL_AutoColumnCount,	SQL_AutoColCount
SQL_AutoColumnInfoStr,	SQL_AutoColInfoStr
SQL_AutoColumnInfoSInt,	SQL_AutoColInfoSInt

Columns which are indexed:

[SQL Tools Pro Only...](#)

SQL_IndexCount,	SQL_IndxCnt
SQL_IndexInfoStr,	SQL_IdxInfoStr
SQL_IndexInfoSInt,	SQL_IdxInfoSInt

Columns that are used as Primary Keys:

[SQL Tools Pro Only...](#)

SQL_PrimaryKeyCount,	SQL_PrimKeyCount
SQL_PrimaryKeyInfoStr,	SQL_PrimKeyInfoStr
SQL_PrimaryKeyInfoSInt,	SQL_PrimKeyInfoSInt

Columns in other tables that are linked to this table:

SQL Tools Pro Only...

SQL_ForeignKeyCount,
SQL_ForeignKeyInfoStr,
SQL_ForeignKeyInfoSInt,

SQL_FornKeyCount
SQL_FornKeyInfoStr
SQL_FornKeyInfoSInt

Statement Open/Close Family

Functions related to the opening and closing of Statements. (SQL Tools handles most statement open/close operations automatically. These functions allow you to take control of the process, for special circumstances.)

`SQL_NewStatementNumber, SQL_NewStmtNumber`

`SQL_OpenStatement, SQL_OpenStmt`

`SQL_StatementIsOpen, SQL_StmtIsOpen`

`SQL_CloseStatement, SQL_CloseStmt`

Statement Family

Functions related to SQL statements.

SQL_Statement,	SQL_Stmt
----------------	----------

SQL_FetchResult,	SQL_Fetch
------------------	-----------

SQL_EndOfData,	SQL_EOD
----------------	---------

SQL Tools Pro Only...

SQL_FetchRelative,	SQL_FetchRel
--------------------	--------------

SQL_AsyncStatement	SQL_AsyncStmt
SQL_AsyncStatus	

SQL_Bookmark,	SQL_Bkmk
---------------	----------

SQL_StatementCancel,	SQL_StmtCancel
----------------------	----------------

SQL_MoreResults,	SQL_MoreRes
------------------	-------------

SQL_BulkOperation,	SQL_BulkOp
--------------------	------------

SQL_SetPosition,	SQL_SetPos
------------------	------------

SQL_FetchPosition	SQL_FetchPos
-------------------	--------------

SQL_SyncFetchPosition	SQL_SyncFetchPos
-----------------------	------------------

Statement Info/Attrib Family

Functions that allow you to obtain SQL statement Attribute and Information values, and to set statement Attributes. (Generally speaking, SQL Tools "Attribute" settings can be changed, and "Information" settings cannot be changed.)

General Information about a statement:

<code>SQL_StatementInfoStr,</code>	<code>SQL_StmtInfoStr</code>
<code>SQL_StatementNativeSyntax,</code>	<code>SQL_StmtNativeSyntax</code>

Statement Attributes:

<code>SQL_StatementMode,</code>	<code>SQL_StmtMode</code>
<code>SQL_ResetStatementMode,</code>	<code>SQL_ResetStmtMode</code>
<code>SQL_StatementAttrib,</code>	<code>SQL_StmtAttrib</code>

[SQL Tools Pro Only...](#)

<code>SQL_SetStatementAttrib,</code>	<code>SQL_SetStmtAttrib</code>
--------------------------------------	--------------------------------

Named Cursors:

[SQL Tools Pro Only...](#)

<code>SQL_NameCursor,</code>	<code>SQL_NameCur</code>
<code>SQL_CursorName,</code>	<code>SQL_CurName</code>

Statement Binding Family

Functions related to the Bound Parameters of SQL statements:

[SQL Tools Pro Only...](#)

SQL_StatementParameterCount,	SQL_StmtParamCount
SQL_ParameterInfoUInt,	SQL_ParamInfoUInt

SQL_BindParameter,	SQL_BindParam
--------------------	---------------

SQL_NextParameter,	SQL_NextParam
SQL_LongParameter,	SQL_LongParam

Stored Procedure Family

Functions related to Stored Procedures, which are pre-compiled SQL Statements that are stored in a database:

[SQL Tools Pro Only...](#)

<code>SQL_ProcedureCount,</code>	<code>SQL_ProcCount</code>
<code>SQL_ProcedureInfoStr,</code>	<code>SQL_ProcInfoStr</code>
<code>SQL_ProcedureInfoSInt,</code>	<code>SQL_ProcInfoSInt</code>

Information about the parameters that a Stored Procedure requires, and the result columns that it produces:

[SQL Tools Pro Only...](#)

<code>SQL_ProcedureColumnCount,</code>	<code>SQL_ProcColCount</code>
<code>SQL_ProcedureColumnInfoStr,</code>	<code>SQL_ProcColInfoStr</code>
<code>SQL_ProcedureColumnInfoSInt,</code>	<code>SQL_ProcColInfoSInt</code>

Result Column Binding Family

Functions related to the binding of result columns. (This family is rarely used because of the SQL Tools "Autobind" feature, which handles most binding operations automatically.)

<code>SQL_AutoBindColumn,</code>	<code>SQL_AutoBindCol</code>
<code>SQL_ManualBindColumn,</code>	<code>SQL_ManualBindCol</code>
<code>SQL_UnbindColumn,</code>	<code>SQL_UnbindCol</code>

SQL Tools Pro Only...

<code>SQL_DirectBindColumn,</code>	<code>SQL_DirectBindCol</code>
<code>SQL_ResultColumnBufferPtr,</code>	<code>SQL_ResColBufferPtr</code>
<code>SQL_ResultColumnIndPtr,</code>	<code>SQL_ResColIndPtr</code>

Result Count Family

Functions that provide general information about the results of a SQL statement's, such as the number of Rows affected by an [UPDATE](#) or the number of Columns in a result set.

`SQL_ResultRowCount,` `SQL_ResRowCount`

`SQL_ResultColumnCount,` `SQL_ResColCount`

Result Column Family

Functions that provide actual values (i.e. data) from the columns of a result set, provide information about a column's Indicator value, and provide information about the columns themselves (type, name, etc.).

Result Column Values:

SQL_ResultColumnSInt,	SQL_ResColSInt
SQL_ResultColumnUInt,	SQL_ResColUInt
SQL_ResultColumnBInt,	SQL_ResColBInt
SQL_ResultColumnFloat,	SQL_ResColFloat
SQL_ResultColumnDate,	SQL_ResColDate
SQL_ResultColumnTime,	SQL_ResColTime
SQL_ResultColumnDateTime,	SQL_ResColDateTime
SQL_ResultColumnStr,	SQL_ResColStr
SQL_ResultColumnText,	SQL_ResColText
SQL_LongResultColumn,	SQL_LongResCol

SQL Tools Pro Only...

SQL_ResultColumnDateTimePart	SQL_ResColDateTimePart
------------------------------	------------------------

Information about Result Columns:

SQL_ResultColumnInfoSInt,	SQL_ResColInfoSInt
SQL_ResultColumnInfoStr,	SQL_ResColInfoStr
SQL_ResultColumnType,	SQL_ResColType
SQL_ResultColumnSize,	SQL_ResColSize
SQL_ResultColumnLen,	SQL_ResColLen
SQL_ResultColumnNumber,	SQL_ResColNumber

Result Column Indicator values:

SQL_ResultColumnNull,	SQL_ResColNull
SQL_ResultColumnMore,	SQL_ResColMore
SQL_ResultColumnInd,	SQL_ResColInd

Error/Trace Family

Various functions related to error handling and tracing.

- SQL_ErrorClearAll
- SQL_ErrorClearOne
- SQL_ErrorColumnNumber
- SQL_ErrorCount
- SQL_ErrorDatabaseNumber
- SQL_ErrorFuncName
- SQL_ErrorIgnore
- SQL_ErrorNativeCode
- SQL_ErrorNumber
- SQL_ErrorPending
- SQL_ErrorQuickAll
- SQL_ErrorQuickOne
- SQL_ErrorSimulate
- SQL_ErrorStatementNumber
- SQL_ErrorText
- SQL_ErrorTime
- SQL_State

- SQL_Trace
- SQL_TraceSInt
- SQL_TraceStr
- SQL_TraceStrOLE

SQL Tools Pro Only...

- SQL_ErrorStr
- SQL_Diagnostic
- SQL_OnErrorCall
- SQL_AsyncErrors

Utility Family

Various utility functions, such as text-to-binary and binary-to-text conversions, a "string interpreter" that simplifies the use of certain characters in strings (such as quotation marks), and a simple Message Box function.

- SQL_BinaryStr
- SQL_TextStr
- SQL_TextStrOLE
- SQL_TextDateTime
- SQL_TextDate
- SQL_TextTime
- SQL_MsgBox
- SQL_MsgBoxButton
- SQL_IString
- SQL_LimitTextLength
- SQL_Okay
- SQL_SelectFile
- SQL_StringToType
- SQL_ToolsVersion

Get Info Family

SQL Tools Internal "Get" Functions. These functions are rarely used in programs because SQL Tools *automatically* uses these functions (internally) whenever an Info function is used. When an Info function is first used, SQL Tools caches all of the information that is related to the function, for faster access in the future. The Get functions can be used to force SQL Tools to "refresh" the Info data, if you have reason to believe that, while your program is running, a table has been added, a column has been deleted, etc.

SQL_GetTblCols,	SQL_GetTableColumns
SQL_GetTblInfo,	SQL_GetTableInfo

SQL Tools Pro Only...

SQL_GetDataSources
SQL_GetDrivers

SQL_GetAutoCols,	SQL_GetAutoColumns
SQL_GetColPrivs,	SQL_GetColumnPrivileges
SQL_GetDBDataTypes,	SQL_GetDatabaseDataTypes
SQL_GetFornKeys,	SQL_GetForeignKeys
SQL_GetIdxes,	SQL_GetIndexes
SQL_GetPrimKeys,	SQL_GetPrimaryKeys
SQL_GetProcCols,	SQL_GetProcedureColumns
SQL_GetProcs,	SQL_GetProcedures
SQL_GetTblPrivs,	SQL_GetTablePrivileges
SQL_GetUniqCols,	SQL_GetUniqueColumns

Handle Family

These functions can be used to obtain certain window handles, plus the actual ODBC handles of the ODBC Environment, each ODBC database connection, and each ODBC statement.

SQL_hParentWindow

SQL Tools Pro Only...

It should not be necessary to use most of these functions unless you wish to write API-level functions that SQL Tools does not provide. (Of which there are very, very few.):

SQL_hDatabase,	SQL_hDB
SQL_hStatement,	SQL_hStmt
SQL_hEnvironment	

SQL_AsyncErrors

Summary

If an asynchronous SQL statement generates one or more Error Messages, this function must be used before your program can access them.

Twin

None.

Family

Error/Trace Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_AsyncErrors(lThreadNumber&)
```

Parameters

lThreadNumber&

A thread number that was specified in a previous `SQL_AsyncStatement` or `SQL_AsyncStmt` function.

Return Values

After this function makes them visible, this function will return the *number* of errors that have been made visible to your program, from zero (0) to the maximum capacity of the SQL Tools Error Stack.

Remarks

See `SQL_AsyncStmt` for a complete discussion of this function.

Diagnostics

This function does not return Error Codes or Error Messages.

Example

See `SQL_AsyncStmt`.

Driver Issues

None.

Speed Issues

None.

See Also

Asynchronous Execution of SQL Statements

SQL_AsyncStatement

Summary

Executes a SQL statement "asynchronously", i.e. in a separate thread. (This function *can* be used by Visual Basic programs even though VB does not support true multi-threading. PowerBASIC programs have the option of using this function or the `THREAD` functions that are built into PowerBASIC. Generally speaking, using this function is more convenient than using `THREAD`.)

Twin

`SQL_AsyncStmt`

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

You should familiarize yourself with **Asynchronous Execution of SQL Statements** before attempting to use this function.

Syntax

```
lResult& = SQL_AsyncStatement(lDatabaseNumber&, _  
                               lStatementNumber&, _  
                               lAction&, _  
                               sStatement$, _  
                               lThreadNumber&)
```

Parameters

lDatabaseNumber& and lStatementNumber&

See Database Numbers and Statement Numbers for a detailed discussion.

All Other Parameters

See twin function `SQL_AsyncStmt` for complete details.

Return Values

See twin function `SQL_AsyncStmt` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_AsyncStatement` is identical to `SQL_AsyncStmt`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_AsyncStatement` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_AsyncStmt` for complete details.

Example

See twin function `SQL_AsyncStmt`.

Driver Issues

None.

Speed Issues

None.

See Also

Asynchronous Execution of SQL Statements
Multi-Threaded Programs

SQL_AsyncStatus

Summary

Indicates the current status of a SQL statement that was executed with the `SQL_AsyncStatement` or `SQL_AsyncStmt` function.

Twin

None.

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

You should familiarize yourself with **Asynchronous Execution of SQL Statements** before attempting to use this function.

Syntax

```
lResult& = SQL_AsyncStatus(lThreadNumber&)
```

Parameters

lThreadNumber&

A thread number that was specified in a previous `SQL_AsyncStatement` or `SQL_AsyncStmt` function.

Return Values

This function will return `SQL_STILL_EXECUTING` if the asynchronous SQL statement with the specified thread number has not yet finished executing.

If the specified SQL statement has finished executing, this function will return either `SUCCESS` (zero) or an ODBC Error Code to indicate the results of the statement. (These values are identical to those returned by `SQL_Stmt`.)

If you attempt to use this function to obtain the result of a statement that has not yet been started with `SQL_AsyncStatement` or `SQL_AsyncStmt` (i.e. if you use `SQL_AsyncStatus` *before* one of those two functions) it will also return `SUCCESS` (zero). This is the most logical return value for this condition, because the statement is not "still executing" and has not returned an error.

Remarks

See `SQL_AsyncStmt` for a complete discussion of this function.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

See `SQL_AsyncStmt`.

Driver Issues

None.

Speed Issues

None.

See Also

Asynchronous Execution of SQL Statements
Multi-Threaded Programs

SQL_AsyncStmt

Summary

Executes a SQL statement "asynchronously", i.e. in a separate thread. (This function *can* be used by Visual Basic programs even though VB does not support true multi-threading. PowerBASIC programs have the option of using this function or the `THREAD` functions that are built into PowerBASIC. Generally speaking, using this function is more convenient than using `THREAD`.)

Twin

`SQL_AsyncStatement`

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

You should familiarize yourself with Asynchronous Execution of SQL Statements before attempting to use this function.

Syntax

```
lResult& = SQL_AsyncStmt(lAction&, _  
                        sStatement$, _  
                        lThreadNumber&)
```

Parameters

lAction&

One of the following constants: `SQL_STMT_PREPARE`, `SQL_STMT_EXECUTE`, or `SQL_STMT_IMMEDIATE`. (A number of aliases for these values are also recognized.) See `SQL_Stmt` for a complete discussion of these values.

sStatement\$

The SQL statement to be prepared and/or executed. The exact syntax that you use will depend on the capabilities of the ODBC driver that your program uses. For a summary of the basic syntax that is recognized by all ODBC-compliant drivers, see Appendix A: SQL Statement Syntax. See `SQL_Stmt` for a complete discussion of this parameter

lThreadNumber&

A number between one (1) and the number that your program most recently used for `SQL_Thread(MAX_THREAD)`.

Return Values

This function will return...

`ERROR_BAD_PARAM_VALUE` if you use an invalid value for the *lThreadNumber&* parameter. (This is usually caused by failing to use the `SQL_Thread(THREAD_MAX)` function before using `SQL_AsyncStmt`). Or...

`ERROR_CANT_BE_DONE` if a SQL statement is currently using the specified thread number, or...

`ERROR_UNKNOWN_ERROR` if Windows fails to create the requested thread, or...

`SUCCESS` (zero) if the specified SQL statement has been executed.

IMPORTANT NOTE: A return value of `SUCCESS` does *not* indicate that the specified SQL statement executed *properly*. It simply means that SQL Tools was able to create a thread and "launch" the SQL statement. To find out whether or not the SQL statement itself generated any errors, use the `SQL_AsyncStatus` function. Also see `SQL_AsyncErrors`.

Remarks

Except for the `IThreadNumber` parameter, `SQL_AsyncStmt` is identical to `SQL_Stmt`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_AsyncStmt` is shown below.

Most programs use the `SQL_Stmt` or `SQL_Statement` function to prepare and/or execute SQL statements. When that is done, your program "pauses" until the SQL statement generates a result.

But that is not always desirable. For example, most GUI-style programs need to continuously update their screens, and because it can take seconds, minutes, or even *hours* for some SQL statements to execute, you may wish to execute a SQL statement "asynchronously". That term means "in the background, while my main program continues to run". Asynchronous execution can allow your program to do many different things while waiting, such as checking to see if the user has clicked a Cancel button, and/or displaying a "WORKING... PLEASE WAIT" animation.

In order to use the `SQL_AsyncStmt` function, you should first be familiar with using the simpler `SQL_Stmt` function. If you are not familiar with `SQL_Stmt`, we strongly suggest that you read about that function before attempting to use this one.

Here is the code for a simple program that uses `SQL_AsyncStmt`. It is based on the SQL-Dump example code that is provided and explained in *A Simple Program, Step By Step*.

```
SQL_Authorize &H.....
SQL_Init

SQL_Thread THREAD_MAX, 1

SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"

Open "\SQLTOOLS\SAMPLES\SQL-DUMP.TXT" For Output As #2

sStatement$ = "SELECT * FROM ADDRESSBOOK"

SQL_AsyncStmt SQL_STMT_IMMEDIATE, sStatement$, 1

Do
    If SQL_AsyncStatus(1) <> SQL_STILL_EXECUTING Then
        Exit Do
    End If
    'You can do other useful work here, while
    'waiting for the query to execute.
Loop
```

```

Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD Then Exit Do
    If SQL_ErrorPending Then Exit Do
    Print #2, SQL_ResColText(ALL_COLS) + "<"
Loop

```

IMPORTANT NOTE: For clarity, this program does not include Error Checking code, which is an important part of any program. See **Error Handling** below.

The `SQL_Authorize` and `SQL_Init` lines at the very beginning of the example code are required for all SQL Tools programs.

The line that says `SQL_Thread THREAD_MAX, 1` tells SQL Tools that you intend to use one thread (one asynchronous statement at a time) in this program. More complex programs may need to use larger values for the second parameter. See `SQL_Thread` for more information.

The next three lines...

```

SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
Open "\SQLTOOLS\SAMPLES\SQL-Dump.TXT" For Output As #2
sStatment$ = "SELECT * FROM ADDRESSBOOK"

```

...are covered in detail in A Simple Program, Step By Step. They are fairly straightforward so we will not explain them here.

The next line...

```

SQL_AsyncStmt SQL_STMT_IMMEDIATE, sStatement$, 1

```

...tells SQL Tools to execute the SQL statement immediately, asynchronously, using thread number 1. The `SQL_AsyncStmt` function will return almost instantly, leaving the SQL statement executing "in the background".

The first `DO/LOOP` structure waits for the `SQL_AsyncStatus(1)` function to return a value other than `SQL_STILL_EXECUTING` for thread number 1. When that happens, it means that the asynchronous SQL statement is ready to return a result. See `SQL_AsyncStatus` for more information.

The second `DO/LOOP` structure is exactly the same as the one described in A Simple Program, Step By Step. It retrieves the data from the SQL statement.

As you can see, using `SQL_AsyncStmt` is very much like using `SQL_Stmt`, but with a few extra steps and a few extra *benefits*.

Also see `SQL_StmtCancel`.

Error Handling

There is one additional complexity that must be considered when using asynchronous execution: Error Handling.

Normally, when SQL Tools is used in two or more threads of the same program, each thread is given its own Error Stack. The program's threads can use the various `SQL_Error` functions, and they will provide information about errors that have been detected *in that thread only*. (If threads could see each other's error information, it would be very difficult to figure out what was going on. For more information about this, see Multi-Threaded Programs.)

When you use a SQL Tools "async" function, however, your program operates in a single thread and all of the multi-threading operations are handled automatically. SQL Tools creates a thread, executes your SQL statement, checks the result, and terminates the thread, all automatically. Because your program operates in the main thread, it can't see errors that may have taken place in the "async" thread.

So SQL Tools provides the `SQL_AsyncErrors` function, which "transfers" errors from an async thread to your program's main thread. After the `SQL_AsyncStatus` function has told your program that an asynchronous SQL statement has finished executing (see above) you should use the `SQL_AsyncErrors` function with the same thread number. Continuing the example above, it would look like this:

```
Do
    If SQL_AsyncStatus(1) <> SQL_STILL_EXECUTING Then
        Exit Do
    End If
    'You can do other useful work here, while
    'waiting for the query to execute.
Loop

If SQL_AsyncErrors <> 0 Then
    'handle errors here
End IF
```

By the way, you might be wondering why SQL Tools doesn't *automatically* make the errors visible. It's because your program may have *two or more* asynchronous statements executing at the same time, and you may want to make the errors visible to your program one thread at a time, to make your error-handling code easier to manage.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

See **Remarks** above.

Driver Issues

None.

Speed Issues

None.

See Also

Asynchronous Execution of SQL Statements
Multi-Threaded Programs

SQL_Authorize

Summary

Tells SQL Tools that it is authorized to begin operating.

Twin

None.

Family

Configuration Family

Availability

Standard and Pro

Warning

Every program that uses SQL Tools must use this function before any other SQL Tools functions are used, including SQL_Init and SQL_Initialize.

The incorrect use of this function will cause your programs to malfunction. See **Remarks** below for more information.

Syntax

```
lResult& = SQL_Authorize(&h.....)
```

..or...

```
lResult& = SQL_Authorize(MY_SQLT_AUTHCODE)
```

(See **Example** below for recommended use.)

Parameters

&h.....

The BASIC hexadecimal prefix *&h* followed by the eight-character Authorization Code that was provided with your SQL Tools Runtime Files. Every SQL Tools licensee is provided with a unique Authorization Code in the form of an eight-digit hexadecimal number. *Note that the prefix &h must be added to your Authorization Code in order to create a hexadecimal number that your compiler will recognize.* (C and C++ users should use the C notation *0x* instead of *&h*.)

Instead of *&h* and your Authorization Code you can also use the constant/equate *MY_SQLT_AUTHCODE*, as long as the proper value has been entered into your SQL Tools declaration (INC) file.

See **Example** below.

Return Values

This function will return *ERROR_CANT_BE_DONE* if an invalid Authorization Code is used.

A return value of *SUCCESS* (zero) indicates that *either* the correct Authorization Code or a "Dummy Code" was accepted by the function. See **Remarks** below for more information.

A return value of negative one (-1) means that your program called SQL_Authorize more than once. SQL_Authorize should be called only once by each program.

Remarks

Every SQL Tools Runtime File is "serialized". That means that it contains a unique, embedded key number called an Authorization Code. In order to use SQL Tools, you must prove to the Runtime File that you know its correct Authorization Code.

If you don't use the SQL_Authorize function at all, the SQL_Init and SQL_Initialize functions will refuse to work, making it impossible for your program to use SQL Tools in any way.

If you use the SQL_Authorize function with the Authorization Code that matches your Runtime File -- the exact number that was provided *with* your Runtime Files -- then SQL Tools will work normally.

If you use the SQL_Authorize function with a "Dummy Code", SQL Tools will randomly, intentionally malfunction.

See SQL Tools Authorization Codes for a complete description of how Authorization Codes and Dummy Codes are used.

Diagnostics

This function returns Error Codes, and can generate SQL Tools Error Messages.

Example

```
'If the Authorization Code for your copy of
'SQL Tools is "ABCD1234", you would add the
' "&h" prefix and use...
```

```
IF SQL_Authorize(&hABCD1234) <> SUCCESS THEN
    SQL_MsgBox "ERROR!  WRONG AUTH CODE!", 0
    EXIT FUNCTION
END IF
```

```
'PLEASE NOTE THAT "ABCD1234" IS NOT
'A VALID AUTHORIZATION CODE!  YOU MUST
'USE THE AUTHORIZATION CODE THAT WAS
'PROVIDED WITH YOUR COPY OF SQL TOOLS.
```

```
'Alternatively, you could add your Auth Code
'to your SQL Tools declaration file (SQLT_*.INC
'or modSQLTools*.BAS) like this...
```

```
MY_SQLT_AUTHCODE = &hABCD1234
```

```
'...and then use this in your programs...
```

```
IF SQL_Authorize(MY_SQLT_AUTHCODE) <> SUCCESS THEN
    SQL_MsgBox "ERROR!  WRONG AUTH CODE!", 0
    EXIT FUNCTION
END IF
```

PowerBASIC users must add a % prefix to MY_SQLT_AUTHCODE in order to create the PB "equate" %MY_SQLT_AUTHCODE.

Driver Issues

None.

Speed Issues

None.

See Also

Four Critical Steps For Every SQL Tools Program

SQL_AutoBindCol

Summary

Automatically binds one column (or all of the columns) of a result set to a memory buffer and an Indicator buffer, so that your program can access the values. (Please note that, unless you tell it not to, SQL Tools automatically autobinds all columns in a result set, so *it is not usually necessary for your program to use this function.*)

Twin

SQL_AutoBindColumn

Family

Result Column Binding Family

Availability

Standard and Pro

Warning

If you use this function when you don't need to, your program will be performing unnecessary work. See **Remarks** below.

Syntax

```
lResult& = SQL_AutoBindCol(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of the column that you wish to autobind. The first column of a result set is column number 1, the second is column number 2, and so on. (The Bookmark Column, if it is supported by your ODBC driver, is always Column Zero (0). You should not usually autobind a Bookmark Column.)

You may specify any nonzero, positive number for *lColumnNumber&*, up to and including the highest-numbered column in a result set. Using a number that is too large will result in an `ERROR_BAD_PARAM_VALUE` error.

You may also use the constant `ALL_COLUMNS` for *lColumnNumber&*, to tell the `SQL_AutoBindCol` function to automatically bind all of the columns in a result set (except for the Bookmark Column, if it exists).

Return Values

This function will return `SUCCESS` or `SUCCESS_WITH_INFO` if the binding process is performed without any errors, or an Error Code if it is not.

Remarks

Unless you tell it not to, SQL Tools automatically autobinds all of the columns in every result set this is created by the `SQL_Stmt` function. You can deactivate this "AutoAutoBinding" feature by using the following code...

```
SQL_SetOptionSInt OPT_AUTOAUTO_BIND, 0
```

If you deactivate the AutoAutoBinding feature, your program is responsible for binding all of the columns in all result sets.

You can perform result column binding with the `SQL_ManualBindCol` and/or

SQL_DirectBindCol functions, and/or by using the SQL_AutoBindCol function on selected columns. In other words, you can use any combination of AutoBinding, Manual Binding, and Direct Binding that you choose.

If you find it necessary to Manually Bind or Direct Bind one column of a result set, it will often be desirable to AutoBind the rest of the columns. The SQL_AutoBindCol function can be used to AutoBind selected columns of a result set after the AutoAutoBind feature has been deactivated.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'autobind column 10...  
SQL_AutoBindCol 10
```

Driver Issues

None.

Speed Issues

Autobound columns are *very* slightly slower than Manually- and Direct-Bound columns. This is because your program must access the column data and Indicator via SQL Tools functions, and it takes a small amount of time to enter and exit from those functions. If you Manually or Direct-Bind a column or Indicator, your program can access the memory buffer directly, without going through a SQL Tools function. However, if you do not use AutoBinding, many different SQL Tools function (such as the SQL_ResCol functions) *cannot be used*.

If speed is an extremely *critical* design factor in your program, you may wish to use Manual and/or Direct Binding instead of AutoBinding.

See Also

Result Column Binding

SQL_AutoBindColumn

Summary

Automatically binds one column (or all of the columns) of a result set to a memory buffer and an Indicator buffer, so that your program can access the values. (Please note that, unless you tell it not to, SQL Tools automatically autobinds all columns in a result set, so *it is not usually necessary for your program to use this function.*)

Twin

SQL_AutoBindCol

Family

Result Column Binding Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_AutoBindColumn(lDatabaseNumber&, _  
                               lStatementNumber&, _  
                               lColumnNumber&)
```

Parameters

lDatabaseNumber& and lStatementNumber&

See Database Numbers and Statement Numbers for a detailed discussion.

All Other Parameters

See twin function SQL_AutoBindCol for complete details.

Return Values

See twin function SQL_AutoBindCol for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_AutoBindColumn is identical to SQL_AutoBindCol. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_AutoBindColumn is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_AutoBindCol for complete details.

Example

```
'autobind Database 2, Statement 3, column 10...  
SQL_AutoBindCol 2, 3, 10
```

Driver Issues

None.

Speed Issues

See twin function `SQL_AutoBindCol` for complete details.

See Also

Result Column Binding

SQL_AutoColCount

Summary

Indicates the number of AutoColumns that a table has.

Twin

SQL_AutoColumnCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_AutoColCount(lTableNumber&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

Return Values

This function returns zero or a positive number to indicate the number of AutoColumns that a given table has.

Remarks

An AutoColumn is a column that is automatically updated when any value in the row is updated by a transaction. For more information, see AutoColumns.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "the specified table has one AutoColumn".

Example

See AutoColumns.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Unique Columns

SQL_AutoColInfoSInt

Summary

Provides information about one AutoColumn, in numeric form.

Twin

SQL_AutoColumnInfoSInt

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_AutoColInfoSInt(lTableNumber&, _  
                                lColumnNumber&, _  
                                lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lColumnNumber&

The number of an AutoColumn, between one (1) and the number returned by the SQL_AutoColCount function.

lInfoType&

The type of information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are provided, this function returns the numeric information value. Otherwise, zero (0) is returned.

Remarks

An AutoColumn is a column that is automatically updated when any value in the row is updated by a transaction. For more information, see AutoColumns.

Please note that only *some* types of AutoColumn information are useful in numeric form. For a list of *lInfoType&* values that can be used to obtain information about an AutoColumn in string form, see SQL_AutoColInfoStr.

To obtain numeric information, the *lInfoType&* parameter must be one of the following values:

AUTOCOL_BUFFER_LENGTH

The length of the AutoColumn data in bytes. This is the amount of data that is transferred by a SQL_Fetch or SQL_FetchRel operation if a data type of SQL_DEFAULT is specified. See Buffer Size.

AUTOCOL_DATA_TYPE

The AutoColumn's SQL data type.

AUTOCOL_DECIMAL_DIGITS

The number of decimal digits that the AutoColumn has. Zero (0) will be returned for columns that have data types where decimal digits are not applicable (strings, integers, binary values, etc.).

AUTOCOL_PSEUDO_COLUMN

Indicates whether or not the column is a pseudo-column, such as an Oracle ROWID column. This function will return one of the following values:

SQL_PC_PSEUDO
SQL_PC_NOT_PSEUDO
SQL_PC_UNKNOWN

(For maximum interoperability, pseudo-column names should not be quoted with the identifier quote character that is returned by `SQL_DBInfoStr.`)

AUTOCOL_SIZE

The display size of the AutoColumn.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "the SQL data type of the specified AutoColumn is `SQL_CHAR` (value 1)". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_AutoColInfoSInt(AUTOCOL_DATA_TYPE)
```

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

AutoColumns

SQL_AutoColInfoStr

Summary

Provides information about one AutoColumn, in string form.

Twin

SQL_AutoColumnInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_AutoColInfoStr(lTableNumber&, _  
                               lColumnNumber&, _  
                               lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lColumnNumber&

The number of an AutoColumn, between one (1) and the number returned by the SQL_AutoColCount function.

lInfoType&

The type of information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are provided, this function returns the requested information value. Otherwise, an empty string is returned.

Remarks

An AutoColumn is a column that is automatically updated when any value in the row is updated by a transaction. For more information, see AutoColumns.

Please note that only *some* types of AutoColumn information are useful in string form. For a list of *lInfoType&* values that can be used to obtain information about an AutoColumn in numeric form, see SQL_AutoColInfoSInt.

To obtain string information, the *lInfoType&* parameter must be one of the following values:

AUTOCOL_NAME

The name of the AutoColumn. The driver will return an empty string if a column does not have a name.

AUTOCOL_TYPE_NAME

Datasource-dependent data type name. For example, "INTEGER", or

"COUNTER".

Diagnostics

This function does not return Error Codes because it returns string information. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_AutoColInfoStr(1,10,AUTOCOL_NAME)
```

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

AutoColumns

SQL_AutoColumnCount

Summary

Indicates how many AutoColumns a table has.

Twin

SQL_AutoColCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_AutoColumnCount(lDatabaseNumber&, _  
                                lTableNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_AutoColCount for complete details.

Return Values

See twin function SQL_AutoColCount for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_AutoColumnCount is identical to SQL_AutoColCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_AutoColumnCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_AutoColCount for complete details.

Example

See twin function SQL_AutoColCount.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

AutoColumns

SQL_AutoColumnInfoSInt

Summary

Provides information about a table's AutoColumns, in numeric form,

Twin

SQL_AutoColInfoSInt

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_AutoColumnInfoSInt(lDatabaseNumber&, _  
                                lTableNumber&, _  
                                lColumnNumber&, _  
                                lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_AutoColInfoSInt for complete details.

Return Values

See twin function SQL_AutoColInfoSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_AutoColumnInfoSInt is identical to SQL_AutoColInfoSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_AutoColumnInfoSInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_AutoColInfoSInt for complete details.

Example

See twin function SQL_AutoColInfoSInt.

Driver Issues

None

Speed Issues

See Cached Information.

See Also: AutoColumns

SQL_AutoColumnInfoStr

Summary

Provides information about a table's AutoColumns, in string form.

Twin

SQL_AutoColInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_AutoColumnInfoStr(lDatabaseNumber&, _  
                                lTableNumber&, _  
                                lColumnNumber&, _  
                                lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_AutoColInfoStr for complete details.

Return Values

See twin function SQL_AutoColInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber*& parameter, SQL_AutoColumnInfoStr is identical to SQL_AutoColInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_AutoColumnInfoStr is shown below.

For more information about using *lDatabaseNumber*& and *lStatementNumber*& in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_AutoColInfoStr for complete details.

Example

See twin function SQL_AutoColInfoStr.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also

AutoColumns

SQL_BinaryStr

Summary

Converts a string that has been coded by the `SQL_TextStr` or `SQL_TextStrOLE` function (or in a similar manner) back into a binary string.

Twin

None.

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_BinaryStr(sText$)
```

Parameters

sText\$

A string variable or literal that **1)** contains text, **2)** may or may not contain `[CHR$(x)]` markers which represent ASCII characters, and **3)** may *not* contain literal control characters (less than ASCII 32).

Return Values

The return value of this function is a copy of the *sText\$* string in which all of the `[CHR$(x)]` markers (if any) have been converted into the appropriate ASCII characters.

Remarks

A more complete discussion of the interaction between `SQL_BinaryStr` and `SQL_TextStr` is provided in the Reference Guide entry for `SQL_TextStr`.

Diagnostics

None.

Example

```
sText$ = "A[ CHR$(0) ]B"
```

```
sBinary$ = SQL_BinaryStr(sText$)
```

```
'sBinary$ now contains a three-character  
'string where the first character is A,  
'the middle character is CHR$(0), and  
'the third character is B.
```

Driver Issues:

None.

Speed Issues

None.

See Also

Utility Family

SQL_BindParam

Summary

Binds a ? placeholder in a prepared SQL statement (or in a Stored Procedure) to memory buffers that your program provides.

Twin

SQL_BindParameter

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

The incorrect use of this function will cause Application Errors. Please see Binding Statement Input Parameters for background information and complete instructions.

Syntax

```
lResult& = SQL_BindParam(lParameterNumber&, _  
                        lParamType&, _  
                        lBasType&, _  
                        lSQLType&, _  
                        lDisplaySize&, _  
                        lDigits&, _  
                        lPointerToBuffer&, _  
                        lBufferLen&, _  
                        lIndicator&)
```

Parameters

lParameterNumber&

The number of the parameter placeholder that is being bound. If a Stored Procedure is being used, a value between one (1) and the number returned by `SQL_ProcColCount`. Otherwise a value from one (1) to the number of ? markers in the SQL statement, which can be determined with the `SQL_StmtParamCount` function.

lParamType&

The type of parameter that is being bound, either `SQL_PARAM_INPUT`, `SQL_PARAM_OUTPUT`, or `SQL_PARAM_INPUT_OUTPUT`. If you are *not* using a Stored Procedure, the value of *lParamType&* must be always be `SQL_PARAM_INPUT`. See **Remarks** below for details.

lBasType&

The BASIC Data Type of the data buffer that is being used for the parameter. This should always be a constant that starts with `BAS_`.

lSQLType&

The SQL Data Type of the parameter that is being bound. This should always be a constant that starts with `SQL_`.

lDisplaySize&

The display size of the parameter's SQL data type.

lDigits&

The number of digits after the decimal point of the SQL data type. (For string, integer, and binary data types, always use zero (0)).

lPointerToBuffer&

A memory pointer to the first byte of the data buffer. This should normally be

a value that was produced by the BASIC `VarPtr` or `StrPtr` function. (Visual Basic has *undocumented* `VarPtr` and `StrPtr` functions. See MSDN article Q199824.) For `StrPtr`, see Binding Dynamic String/Binary Parameters.

lBufferLen&

The data buffer size, in bytes.

lIndicator&

The name of the `BAS_LONG` variable that will be used for the parameter's Indicator.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the parameter is successfully bound to the memory buffers. Otherwise, an ODBC Error Code or SQL Tools Error Code will be returned.

Remarks

For background information and complete instructions for using this function, see Binding Statement Input Parameters. An extensive explanation is provided, including sample source code.

If you are *not* using a Stored Procedure, the value of *lParamType*& must be always be `SQL_PARAM_INPUT`.

If you *are* using a Stored Procedure that requires bound parameters, the *lParamType*& value must be one of the following constants:

`SQL_PARAM_INPUT` (Indicates that the `?` marks a parameter in a SQL statement which does not call a procedure, such as a `SELECT` or `INSERT` statement, or that it marks an input parameter in a Stored Procedure. If a program can't determine the type of a parameter in a Stored Procedure call, it should use `SQL_PARAM_INPUT`.)

`SQL_PARAM_INPUT_OUTPUT` (The `?` marks an input/output parameter in a Stored Procedure.)

`SQL_PARAM_OUTPUT` (The `?` marks the return value *or* an output parameter of a Stored Procedure.)

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Binding Statement Input Parameters for several examples. Also see the `BindDateParam.BAS` sample program.

Driver Issues

See Binding Statement Input Parameters.

Speed Issues

See Binding Statement Input Parameters.

See Also

Stored Procedures

SQL_BindParameter

Summary

Binds a ? placeholder in a prepared SQL statement (or in a Stored Procedure) to memory buffers that your program provides.

Twin

SQL_BindParam

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

The incorrect use of this function will cause Application Errors. Please see Binding Statement Input Parameters for background information and complete instructions.

Syntax

```
lResult& = SQL_BindParameter(lDatabaseNumber&, _  
                             lStatementNumber&, _  
                             lParameterNumber&, _  
                             lParamType&, _  
                             lBasType&, _  
                             lSQLType&, _  
                             lDisplaySize, _  
                             lDigits&, _  
                             lPointerToBuffer&, _  
                             lBufferLen&, _  
                             lIndicator&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Database Numbers and Statement Numbers for a detailed discussion.

All Other Parameters

See twin function SQL_BindParam for complete details.

Return Values

See twin function SQL_BindParam for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_BindParameter is identical to SQL_BindParam. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_BindParameter is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

For background information and complete instructions for using this function, see Binding Statement Input Parameters. An extensive explanation is provided, including sample source code.

Diagnostics

See twin function `SQL_BindParam` for complete details.

Example

See Binding Statement Input Parameters for several examples.

Driver Issues

See Binding Statement Input Parameters.

Speed Issues

See Binding Statement Input Parameters.

See Also

Stored Procedures

SQL_Bkmk

Summary

Retrieves a bookmark (a string that identifies a row) for the current row of a result set.

Twin

SQL_Bookmark

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_Bkmk
```

Parameters

None.

Return Values

This function returns a string that can be used to identify a row of a result set.

Remarks

A bookmark is a string that can be used to identify a row of a result set, to make it easy (for example) to return to that row in the future.

For a complete discussion, see Bookmarks.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'save the bookmark
sResult$ = SQL_Bkmk

'(other operations which re-position the cursor)

'return to the bookmarked row
SQL_FetchRel(sResult$,0)
```

Driver Issues

The Microsoft Access 97 ODBC Driver does not support bookmarks if ODBC 2.0 behavior is used, i.e. when an *IODBCVersion*& value of 2 is used for the *SQL_Initialize* function.

Speed Issues

See Using Bookmarks for a discussion of bookmark speed issues.

See Also: Relative Fetches

SQL_Bookmark

Summary

Retrieves a bookmark (a string that identifies a row) for the current row of a result set.

Twin

SQL_Bkmk

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_Bookmark(lDatabaseNumber &, _  
                        lStatementNumber &)
```

Parameters

lDatabaseNumber & and *lStatementNumber* &

See Database Numbers and Statement Numbers for a detailed discussion.

Return Values

This function returns a string that can be used to identify a row of a result set.

Remarks

Except for the *lDatabaseNumber* & and *lStatementNumber* & parameters, `SQL_Bookmark` is identical to `SQL_Bkmk`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_Bookmark` is shown below.

For more information about using *lDatabaseNumber* & and *lStatementNumber* & in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_Bkmk` for complete details.

Example

See twin function `SQL_Bkmk`.

Driver Issues

See twin function `SQL_Bkmk` for complete details.

Speed Issues

See Using Bookmarks for a discussion of bookmark speed issues.

See Also

Relative Fetches

SQL_BulkOp

Summary

Performs a Bulk Operation (Bulk Add, Bulk Delete, Bulk Update, or Bulk Fetch) on a Multirow cursor.

Twin

SQL_BulkOperation

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None. (See **Driver Issues** below.)

Syntax

```
lResult& = SQL_BulkOp(lOperation&)
```

Parameters

lOperation&

One of the following constants, which corresponds to the desired operation:
BULK_ADD, BULK_UPDATE, BULK_DELETE, or BULK_FETCH.

Return Values

If the operation is performed without errors, this function returns SUCCESS or SUCCESS_WITH_INFO.

If errors are detected during the operation, this function will return an ODBC Error Code or a SQL Tools Error Code.

Remarks

For a complete discussion of this function, see Bulk Operations.

Diagnostics

This function returns Error Codes, and can return ODBC Error Messages and SQL Tools Error Messages.

Example

```
SQL_BulkOp BULK_DELETE
```

Driver Issues

According to the Microsoft ODBC Software Developer Kit, this function is "not widely supported".

Speed Issues

None.

See Also

Positioned Operations

SQL_BulkOperation

Summary

Performs a Bulk Operation (Bulk Add, Bulk Delete, Bulk Update, or Bulk Fetch) on a Multirow cursor.

Twin

SQL_BulkOp

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_BulkOperation(lDatabaseNumber&, _  
                             lStatementNumber&, _  
                             lOperation&)
```

Parameters

lDatabaseNumber& and lStatementNumber&

See Database Numbers and Statement Numbers for a detailed discussion.

All Other Parameters

See twin function SQL_BulkOp for complete details.

Return Values

See twin function SQL_BulkOp for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_BulkOperation is identical to SQL_BulkOp. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_BulkOperation is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_BulkOp for complete details.

Example

```
'perform bulk delete operation  
'on database 1, statement 2.  
SQL_BulkOperation 1, 2, BULK_DELETE
```

Driver Issues: None.

Speed Issues: None.

See Also: Positioned Operations

SQL_CloseDatabase

Summary

Closes a database that has been opened with `SQL_OpenDB` or `SQL_OpenDatabase`.

Twin

`SQL_CloseDB`

Family

Database Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_CloseDatabase(lDatabaseNumber&)
```

Parameters

lDatabaseNumber&

See Database Numbers and Statement Numbers for a detailed discussion.

Return Values

See twin function `SQL_CloseDB` for complete information.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_CloseDatabase` is identical to `SQL_CloseDB`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_CloseDatabase` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_CloseDB` for complete information.

Example

```
'close database #2
SQL_CloseDatabase 2
```

Driver Issues

None.

Speed Issues

None.

See Also

Opening a Database

SQL_CloseDB

Summary

Closes a database that has been opened with `SQL_OpenDB` or `SQL_OpenDatabase`.

Twin

`SQL_CloseDatabase`

Family

Database Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_CloseDB
```

Parameters

None.

Return Values

If the database is closed without errors, this function returns `SUCCESS` or `SUCCESS_WITH_INFO`.

If errors are detected, this function will return an ODBC Error Code or a SQL Tools Error Code.

Remarks

The `SQL_CloseDB` operation automatically performs all of the steps that are necessary to close a database, including the unbinding of all result set columns, the closing of all open statements (see `SQL_CloseStmt`), and the releasing of all internal SQL Tools buffers and handles that are related to the database.

After a database has been closed, your program can no longer use SQL Tools functions to access it, or any statements or result columns that are related to it. (Unless, of course, your program first uses `SQL_OpenDB` to re-open the database.) If you attempt to use a SQL Tools function to access a database that has been closed, an `ERROR_DB_NOT_OPEN` error message will be generated.

Generally speaking, most programs do not really need to use the `SQL_CloseDB` function. The `SQL_Shutdown` function -- which all programs are required to use -- automatically uses the `SQL_CloseDB` function to close all open databases, so it is not necessary to explicitly use the `SQL_CloseDB` function in your programs.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
SQL_CloseDB
```

Driver Issues

None.

Speed Issues

None.

See Also

Opening a Database

SQL_CloseStatement

Summary

Closes a statement that was previously opened with the `SQL_Stmt`, `SQL_Statement`, `SQL_OpenStmt`, or `SQL_OpenStatement` function.

Twin

`SQL_CloseStmt`

Family

Statement Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_CloseStatement(lDatabaseNumber&, _  
                               lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Database Numbers and Statement Numbers for a detailed discussion.

Return Values

If the statement is closed without errors, this function returns `SUCCESS` or `SUCCESS_WITH_INFO`.

If errors are detected, this function will return an ODBC Error Code or a SQL Tools Error Code.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_CloseStatement` is identical to `SQL_CloseStmt`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_Statement` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_CloseStmt` for complete information.

Example

```
'close database 2's statement 4...  
SQL_CloseStatement 2, 4
```

Driver Issues: None.

Speed Issues: None.

See Also: Opening and Closing Statements

SQL_CloseStmt

Summary

Closes a statement that was previously opened with the `SQL_Stmt`, `SQL_Statement`, `SQL_OpenStmt`, or `SQL_OpenStatement` function.

Twin

`SQL_CloseStatement`

Family

Statement Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_CloseStmt
```

Parameters

None.

Return Values

If the statement is closed without errors, this function returns `SUCCESS` or `SUCCESS_WITH_INFO`.

If errors are detected, this function returns an ODBC Error Code or a SQL Tools Error Code.

Remarks

The `SQL_CloseStmt` operation automatically performs all of the steps that are necessary to close a statement, including the unbinding of all result set columns, and the releasing of all internal SQL Tools buffers and handles that are related to the statement.

After a statement has been closed, your program can no longer use SQL Tools functions to access it, or result columns that are related to it. (Unless, of course, your program first re-opens the statement.) If you attempt to use a SQL Tools function to access a statement that has been closed, an `ERROR_STMT_NOT_OPEN` error message will be generated.

Generally speaking, most programs do not really need to use the `SQL_CloseStmt` function. The `SQL_Shutdown` function -- which all programs are required to use -- automatically uses the `SQL_CloseStmt` function to close all open statement, so it is not necessary to explicitly use the `SQL_CloseStmt` function in your programs.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

SQL_CloseStmt

Driver Issues

None.

Speed Issues

None.

See Also

SQL_OpenStmt

SQL_ColPrivCount

Summary

Indicates the number of Column Privileges that a column has.

Twin

SQL_TablePrivilegeCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ColPrivCount(lTableNumber&, _  
                             lColumnNumber&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lColumnNumber&

The number of a column, between one (1) and the number returned by the SQL_TblColCount function.

Return Values

If valid parameters are used, this function will return the number of Column Privileges that are associated with a particular column. This number may be zero or a positive number. If invalid parameters are used, zero (0) will be returned.

Remarks

A Column Privilege is an "access right" that is granted to a user, called the Grantee, by another user, called the Grantor. See Column Privileges for more information.

Diagnostics

This function does not return Error Codes because an Error Code like SUCCESS_WITH_INFO (value 1) could be confused with a legitimate return value like "the specified column has one privilege". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the number of Column Privileges  
'for table 2, column 8:  
Print SQL_ColPrivCount(2,8)
```

Driver Issues: None.

Speed Issues: See Cached Information.

See Also: Table Privileges and Column Privileges

SQL_ColPrivInfoStr

Summary

Provides information about a Column Privilege, in string form.

Twin

SQL_ColumnPrivilegeInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_ColPrivInfoStr(lTableNumber&, _  
                               lColumnNumber&, _  
                               lPrivilegeNumber&, _  
                               lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lColumnNumber&

The number of a column, between one (1) and the number returned by the SQL_TblColCount function.

lPrivilegeNumber&

The number of a privilege, between one (1) and the number returned by the SQL_ColPrivCount function.

lInfoType&

The type of numeric information that is being requested. See **Remarks** below for a list of valid values.

Return Values

If valid parameters are used, this function will return the requested numeric information. Otherwise, an empty string will be returned.

Remarks

A Column Privilege is an "access right" that is granted to a user, called the Grantee, by another user, called the Grantor. For example, if Column Privileges have been specified for a certain column like `ANNUALSALARY`, a certain user may have a `SELECT` privilege (the right to use the `SELECT` statement to retrieve data from the column) but not an `UPDATE` privilege (the right to change the values in the column). Other users might not have any rights to access the `ANNUALSALARY` column in any way.

See Column Privileges for more information.

Please note that all of the information about a column's Privileges is useful in string form, so there is no `SQL_ColPrivSInt` function.

To obtain information about a column's Privileges, use one of the following *InfoType*& values:

COLPRIV_GRANTEE

The name of the user to whom the privilege has been granted.

COLPRIV_GRANTOR

The name of the user that granted the privilege. If the value of COLPRIV_GRANTEE (just above) is the owner of the object, the COLPRIV_GRANTOR value will be "_SYSTEM".

COLPRIV_IS_GRANTABLE

Indicates whether the grantee is permitted to grant the privilege to other users.

This *InfoType*& will return "YES" or "NO", or an empty string if the grantability is unknown or not applicable to the Datasource.

COLPRIV_PRIVILEGE

Identifies the column privilege. May be one of the following values, or other values that may be supported by the Datasource: Please note that the quotation marks that are shown below are *not* part of the value that will be returned by this *InfoType*&.

"SELECT" (The grantee is permitted to retrieve data from the column.)

"INSERT" (The grantee is permitted to provide data for the column in new rows that are inserted into the associated table.)

"UPDATE" (The grantee is permitted to update data in the column.)

"REFERENCES" (The grantee is permitted to refer to the column within a constraint (for example, a unique, referential, or table check constraint).

COLPRIV_TABLE_CATALOG,
COLPRIV_TABLE_SCHEMA,
COLPRIV_TABLE_NAME, and
COLPRIV_COLUMN_NAME

The catalog, schema, table, and column name to which the privilege applies.

Diagnostics

This function does not return Error Codes because it returns string values, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the name of the person who  
'granted column privilege number 1  
'for table 2, column 8:  
Print SQL_TblColumnInfoStr(2,8,1,COLPRIV_GRANTOR)
```

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Table Privileges and Column Privileges

SQL_ColumnPrivilegeCount

Summary

Indicates how many Column Privileges are associated with one column of a table.

Twin

SQL_ColPrivCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ColumnPrivilegeCount(lDatabaseNumber&, _  
                                     lTableNumber&, _  
                                     lColumnNumber&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ColPrivCount for complete details.

Return Values

See twin function SQL_ColPrivCount for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_ColumnPrivilegeCount is identical to SQL_ColPrivCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ColumnPrivilegeCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ColPrivCount for complete details.

Example

See twin function SQL_ColPrivCount.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

SQL_ColumnPrivilegeCount

SQL_ColumnPrivilegeInfoStr

Summary

Provides information about a Column Privilege, in string form.

Twin

SQL_ColPrivInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_ColumnPrivilegeInfoStr(lDatabaseNumber&, _  
                                     lTableNumber&, _  
                                     lColumnNumber&, _  
                                     lPrivilegeNumber&, _  
                                     lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ColPrivInfoStr for complete details.

Return Values

See twin function SQL_ColPrivInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_ColumnPrivilegeInfoStr is identical to SQL_ColPrivInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ColumnPrivilegeInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ColPrivInfoStr for complete details.

Example

See twin function SQL_ColPrivInfoStr.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also: SQL_ColumnPrivilegeCount

SQL_CurName

Summary

Returns the name that has been assigned to a cursor with the `SQL_NameCur` or `SQL_NameCursor` function.

Twin

`SQL_CursorName`

Family

Statement Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_CurName
```

Parameters

None.

Return Values

This function returns the name of a cursor, in string form, that was assigned by using the `SQL_NameCur` or `SQL_NameCursor` function.

If no name has been assigned, this function will return either the default cursor name (as assigned by the ODBC driver) or an empty string (" ") if the driver does not support Named Cursors.

Remarks

See Using Named Cursors for information about using this function.

Diagnostics

This function does not return Error Codes, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_CurName
```

Driver Issues

None.

Speed Issues

None.

See Also

Named Cursors

SQL_CurrentDB

Summary

Returns the Database Number of the current database, i.e. the database number that is currently being used by SQL Tools "abbreviated" functions.

Twin

None.

Family

Use Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_CurrentDB
```

Parameters

None.

Return Values

This function always returns an integer value between one (1) and the maximum database number that was specified with the *IMaxDatabaseNumber&* parameter of the *SQL_Initialize* function.

If the *SQL_Init* function was used instead of *SQL_Initialize*, this function will always return the number one (1) or two (2).

Remarks

If your program uses the *SQL_UseDB* and/or *SQL_UseDBStmt* functions to change the default database number, it can obtain the current setting by using this function.

Diagnostics

This function does not generate errors of any type.

Example

```
Print SQL_CurrentDB
```

Driver Issues

None.

Speed Issues

None.

See Also

Using Database Numbers

SQL_CurrentStmt

Summary

Returns the Statement Number of the current statement, i.e. the statement number that is currently being used by SQL Tools "abbreviated" functions.

Twin

None.

Family

Use Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_CurrentStmt
```

Parameters

None.

Return Values

This function always returns an integer value between one (1) and the maximum statement number that was specified with the *lMaxStatementNumber*& parameter of the `SQL_Initialize` function.

If the `SQL_Init` function was used instead of `SQL_Initialize`, this function will always return the number one (1) or two (2).

Remarks

For more information about Statement Numbers, please see Using Database Numbers and Statement Numbers.

Diagnostics

This function does not generate errors of any type.

Example

```
Print SQL_CurrentStmt
```

Driver Issues

None.

Speed Issues

None.

See Also

Using Statement Numbers

SQL_CursorName

Summary

Returns the name that has been assigned to a cursor with the `SQL_NameCur` or `SQL_NameCursor` function.

Twin

`SQL_CurName`

Family

Statement Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_CursorName(lDatabaseNumber&, _  
                           lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber*&

See Database Numbers and Statement Numbers for a detailed discussion.

Return Values

See twin function `SQL_CurName` for complete information.

Remarks

Except for the *lDatabaseNumber*& and *lStatementNumber*& parameters, `SQL_CursorName` is identical to `SQL_CurName`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_CursorName` is shown below.

For more information about using *lDatabaseNumber*& and *lStatementNumber*& in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_CurName` for complete information.

Example

```
'print name of cursor assigned to  
'database 1, statement 2.  
Print SQL_CursorName(1,2)
```

Driver Issues

None.

Speed Issues

None.

See Also

Named Cursors

SQL_DatabaseAttribStr

Summary

Provides the value of a database Attribute, in string form. (Generally speaking, "information" values cannot be changed by your program, and "attribute" values can be changed.)

Twin

SQL_DBAtribStr

Family

Database Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_DatabaseAttribStr(lDatabaseNumber&, _  
                                lAttribute&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_DBAtribStr for complete details.

Return Values

See twin function SQL_DBAtribStr for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_DatabaseAttribStr is identical to SQL_DBAtribStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_DatabaseAttribStr is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_DBAtribStr for complete details.

Example

See twin function SQL_DBAtribStr for complete details.

Driver Issues

None.

Speed Issues

None.

See Also: Database Information and Attributes

SQL_DatabaseAttribUInt

Summary

Provides the value of a database Attribute, in numeric form. (Generally speaking, "information" values cannot be changed by your program, and "attribute" values can be changed.)

Twin

SQL_DBAtribUInt

Family

Database Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_DatabaseAttribUInt(lDatabaseNumber&, _  
                                   lAttribute&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_DBAtribUInt for complete details.

Return Values

See twin function SQL_DBAtribUInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_DatabaseAttribUInt is identical to SQL_DBAtribUInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_DatabaseAttribUInt is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_DBAtribUInt for complete details.

Example

See twin function SQL_DBAtribUInt for complete details.

Driver Issues

None.

Speed Issues

None.

See Also: Database Information and Attributes

SQL_DatabaseAutoCommit

Summary

Changes the AutoCommit status of a database.

Twin

SQL_DBAutoCommit

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DatabaseAutoCommit(lDatabaseNumber&, _  
                                   lOnOff&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_DBAutoCommit for complete details.

Return Values

See twin function SQL_DBAutoCommit for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_DatabaseAutoCommit is identical to SQL_DBAutoCommit. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_DatabaseAutoCommit is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_DBAutoCommit for complete details.

Example

See twin function SQL_DBAutoCommit for complete details.

Driver Issues

None.

Speed Issues

None.

See Also

Committing Transactions Manually

SQL_DatabaseDataTypeCount

Summary

Returns the number of Datasource-Dependent Data Types that a database supports.

Twin

SQL_DBDataTypeCount

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DatabaseDataTypeCount(lDatabaseNumber&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

Return Values

See twin function SQL_DBDataTypeCount for complete information.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_DatabaseDataTypeCount is identical to SQL_DBDataTypeCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_DatabaseDataTypeCount is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_DBDataTypeCount for complete information.

Example

See twin function SQL_DBDataTypeCount for complete information.

Driver Issues

None.

Speed Issues

None.

See Also

Datasource-dependent Data Types, SQL Data Types, BASIC Data Types.

SQL_DatabaseDataTypeInfoSInt

Summary

Provides information about a Datasource-dependent Data Type that is supported by a database, in numeric form. (Generally speaking, "information" values cannot be changed by your program and "attribute" values can be changed.)

Twin

SQL_DBDataTypeInfoSInt

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DatabaseDataTypeInfoSInt(lDatabaseNumber&, _  
                                         lDataTypeNumber&, _  
                                         lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_DBDataTypeInfoSInt for complete details.

Return Values

See twin function SQL_DBDataTypeInfoSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_DatabaseDataTypeInfoSInt is identical to SQL_DBDataTypeInfoSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_DatabaseDataTypeInfoSInt is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_DBDataTypeInfoSInt for complete details.

Example

See twin function SQL_DBDataTypeInfoSInt for complete details.

Driver Issues

None.

Speed Issues

None.

See Also: Datasource-dependent Data Types, SQL Data Types, BASIC Data Types.

SQL_DatabaseDataTypeInfoStr

Summary

Provides information about a Datasource-dependent Data Type that is supported by a database, in string form. (Generally speaking, "information" values cannot be changed by your program and "attribute" values can be changed.)

Twin

SQL_DBDataTypeInfoStr

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_DatabaseDataTypeInfoStr(lDatabaseNumber&, _  
                                         lDataTypeNumber&, _  
                                         lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_DBDataTypeInfoStr for complete details.

Return Values

See twin function SQL_DBDataTypeInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_DatabaseDataTypeInfoStr is identical to SQL_DBDataTypeInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_DatabaseDataTypeInfoStr is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_DBDataTypeInfoStr for complete details.

Example

See twin function SQL_DBDataTypeInfoStr for complete details.

Driver Issues: None.

Speed Issues: None.

See Also: Datasource-dependent Data Types

SQL_DatabaseDataTypeNumber

Summary

Returns the Data Type Number (if any) that corresponds to a Datasource-dependent Data Type Name.

Twin

SQL_DBDataTypeNumber

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

It is very important to avoid confusion between Data Type Numbers and the numeric values that are associated with `SQL_INTEGER`, `SQL_CHAR`, etc. The first Datasource-dependent Data Type that a database reports that it supports is Data Type Number 1, the second reported type is number 2, and so on. Data Type Number 1 *may or may not* correspond to `SQL_CHAR`, which has a numeric identifier value of 1.

See Datasource-dependent Data Types for more information.

Syntax

```
lResult& = SQL_DatabaseDataTypeNumber(lDatabaseNumber&, _  
                                     sTypeName$)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function `SQL_DBDataTypeNumber` for complete details.

Return Values

See twin function `SQL_DBDataTypeNumber` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_DatabaseDataTypeNumber` is identical to `SQL_DBDataTypeNumber`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_DatabaseDataTypeNumber` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_DBDataTypeNumber` for complete details.

Example

See twin function `SQL_DBDataTypeNumber` for complete details.

Driver Issues

None.

Speed Issues

None.

See Also

Datasource-dependent Data Types

SQL_DatabaseInfoStr

Summary

Provides information about a database, in string form. (Generally speaking, "information" values cannot be changed by your program and "attribute" values can be changed.)

Twin

SQL_DBInfoStr

Family

Database Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_DatabaseInfoStr(lDatabaseNumber&, _  
                               lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_DBInfoStr for complete details.

Return Values

See twin function SQL_DBInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_DatabaseInfoStr is identical to SQL_DBInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_DatabaseInfoStr is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_DBInfoStr for complete details.

Example

See twin function SQL_DBInfoStr for complete details.

Driver Issues

None.

Speed Issues

None.

See Also

Database Information and Attributes

SQL_DatabaseInfoUInt

Summary

Provides information about a database, in numeric form. (Generally speaking, "information" values cannot be changed by your program and "attribute" values can be changed.)

Twin

SQL_DBInfoUInt

Family

Database Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_DatabaseInfoStr(lDatabaseNumber&, _  
                                lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_DBInfoUInt for complete details.

Return Values

See twin function SQL_DBInfoUInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_DatabaseInfoStr is identical to SQL_DBInfoUInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_DatabaseInfoStr is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_DBInfoUInt for complete details.

Example

See twin function SQL_DBInfoUInt for complete details.

Driver Issues

None.

Speed Issues

None.

See Also: Database Information and Attributes

SQL_DatabasesOpen

Summary

Indicates whether or not a database is currently open.

Twin

SQL_DBIsOpen

Family

Database Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_DatabaseIsOpen(lDatabaseNumber)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

Return Values

Logical True (-1) if the specified database is open, or False if it is not open.

Remarks

Except for the *lDatabaseNumber&* parameters, `SQL_DatabaseIsOpen` is identical to `SQL_DBIsOpen`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_DatabaseIsOpen` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_DBIsOpen` for complete information.

Example

```
If SQL_DatabaseIsOpen(1) Then Beep
```

Driver Issues

None.

Speed Issues

None.

See Also

Manually Opening and Closing Databases

SQL_DataSourceAdd

Summary

Allows your program's user to add a Data Source to their system by navigating through a series of standard ODBC dialogs.

Twin

None.

Family

Environment Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DataSourceAdd(sDataSourceName$)
```

Parameters

sDataSourceName\$

An empty string (" "), or the name of the Data Source you wish to add. This can be an arbitrary name like "My New Data Source" but it must be a legal Data Source name, i.e. it must not contain backslashes (\), control characters, or other invalid characters.

Return Values

This function normally returns `SUCCESS` (zero) if a Data Source is created, or `ERROR_USER_CANCEL` if it is not. This function can also return `ERROR_CANT_BE_DONE` if the `ODBCCP32.DLL` file is not properly installed on the system (or if the file is corrupt) but this should be extremely rare because that file is a standard ODBC component.

Remarks

This function gives your programs the ability to access certain parts of the ODBC Data Source Administrator *programmatically*, i.e. without instructing the user to manually open the Control Panel and the ODBC Administrator program.

It is important to note that if you specify a Data Source Name by using the *sDataSourceName\$* parameter, *the user will be given the opportunity to change that name* so this function may not always create the Data Source that you intend. We recommend that even if this function returns `SUCCESS` you should enumerate the available Data Sources using the technique shown in `SQL_DataSourceCount`, to make sure that the user created the Data Source that you intended.

Because this function displays dialogs, it requires a "parent window" to be specified. SQL Tools will automatically use the Windows desktop as the parent window for the dialogs, unless you specify one of your program's windows or forms by using the `SQL_SetOptionSInt` `OPT_H_PARENT_WINDOW` option. See `SQL_hParentWindow` for more information.

Diagnostics

This function returns Error Codes and can generate SQL Tools Error Messages.

Example

```
lResult& = SQL_DataSourceAdd("My Data Source")
```

Driver Issues

None.

Speed Issues

None.

See Also

SQL_DataSourceAdmin, SQL_DataSourceModify, SQL_DataSourceCount,
SQL_DataSourceInfoStr.

SQL_DataSourceAdmin

Summary

Displays the main dialog of the ODBC Data Source Administrator program, and allows the user to access all of its functions.

Twin

None.

Family

Environment Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DataSourceAdmin
```

Parameters

None.

Return Values

This function always returns `SUCCESS` unless the `ODBCCP32.DLL` file is not installed or is corrupt, in which case it returns `ERROR_CANT_BE_DONE`. This error should be extremely rare because that file is a standard ODBC component.

Remarks

This function gives your programs the ability to access the ODBC Data Source Administrator *programmatically*, i.e. without instructing the user to manually open the Control Panel and the ODBC Administrator program.

Because this function displays dialogs, it requires a "parent window" to be specified. SQL Tools will automatically use the Windows desktop as the parent window for the dialogs, unless you specify one of your program's windows or forms by using the `SQL_SetOptionSInt OPT_H_PARENT_WINDOW` option. See `SQL_hParentWindow` for more information.

Diagnostics

None.

Example

```
lResult& = SQL_DataSourceAdmin
If lResult& <> SUCCESS Then
    'The Database Administrator
    'program failed to display.
End If
```

Driver Issues

None.

Speed Issues

None.

See Also

SQL_DataSourceModify, SQL_DataSourceAdd, SQL_DataSourceCount,
SQL_DataSourceInfoStr.

SQL_DataSourceCount

Summary

Returns the number of Datasources that are available at runtime.

Twin

None.

Family

Environment Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DataSourceCount
```

Parameters

None.

Return Values

This function returns zero or a positive integer, indicating the number of Datasources that are available on a computer at runtime.

Remarks

The `SQL_DataSourceCount`, `SQL_DataSourceInfoStr` and `SQL_DataSourceNumber` functions can be used to obtain information about the Datasources that are available to your program. This is basically the same information that is displayed by the Microsoft ODBC Datasource Administrator program, but it is available to your program.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with an answer like "there is one Datasource available". However this function can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
For lDS& = 1 To SQL_DataSourceCount
    Print SQL_DataSourceInfoStr(lDS&,DATASOURCE_NAME)
Next
```

Driver Issues

None.

Speed Issues

None.

See Also

`SQL_DataSourceInfoStr` and `SQL_DataSourceNumber`

SQL_DataSourceInfoStr

Summary

Returns the name and/or description of a Datasource.

Twin

None.

Family

Environment Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DataSourceInfoStr(lDataSourceNumber&, _  
                                lInfoType&)
```

Parameters

lDataSourceNumber&

A number between one (1) and the maximum datasource number (as reported by the `SQL_DataSourceCount` function).

lInfoType&

Either `DATASOURCE_NAME` or `DATASOURCE_DESCRIPTION`.

Return Values

This function will return either a Datasource Name, or the Datasource Description, depending on the value of *lInfoType&*.

Remarks

The `SQL_DataSourceCount`, `SQL_DataSourceInfoStr` and `SQL_DataSourceNumber` functions can be used to obtain information about the Datasources that are available to your program. This is basically the same information that is displayed by the Microsoft ODBC Datasource Administrator program, but it is available to your program.

Diagnostics

This function does not return Error Codes because it returns string values, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

See `SQL_DataSourceCount` example.

Driver Issues

None.

Speed Issues

None.

See Also

`SQL_DataSourceCount` and `SQL_DataSourceNumber`

SQL_DataSourceModify

Summary

Allows your program to modify Data Sources programmatically, with or without displaying dialogs which allow the user to change certain values. (When used with Microsoft Access databases, this function can also be used to create new databases and to compact/repair databases.)

Twin

None.

Family

Environment Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DataSourceModify(lRequestType&, _  
                                sDriverName$, _  
                                sAttributes$, _  
                                lPrompt&)
```

Parameters

lRequestType&

One of the following constants: ODBC_ADD_DSN, ODBC_CONFIG_DSN, ODBC_REMOVE_DSN, ODBC_ADD_SYS_DSN, ODBC_CONFIG_SYS_DSN, ODBC_REMOVE_SYS_DSN, or ODBC_REMOVE_DEFAULT_DSN. See **Remarks** below for details.

sDriverName\$

If the *lRequestType&* parameter is ODBC_REMOVE_DEFAULT_DSN then the *sDriverName\$* parameter can be an empty string. Otherwise it must contain the name of an ODBC Driver that is currently installed on the system. See **Remarks** below for details.

sAttributes\$

The attributes of the Data Source. If the *lRequestType&* parameter is ODBC_REMOVE_DEFAULT_DSN then the *sAttributes\$* parameter can be an empty string. Otherwise it must contain *at least* a DSN= value. See **Remarks** below for details.

lPrompt&

If this parameter is False (zero) then no "prompt" dialogs will be displayed. If this parameter is Logical True (-1) or any other nonzero value, dialogs will be displayed to allow the user to view and/or modify the Data Source attributes. Note that if True or another nonzero value is used, the dialogs will *a/ways* be displayed, even if the attributes are complete and correct. (This is different from the `SQL_OpenDB` function, for example, which displays dialogs only if the supplied information is not sufficient to allow a connection.)

Return Values

This function normally returns `SUCCESS` (zero). If an invalid parameter is used, it will return `ERROR_BAD_PARAM_VALUE`. If the `ODBCCP32.DLL` file is not installed

or is corrupt, this function will return `ERROR_CANT_BE_DONE`. (This error should be extremely rare because that file is a standard ODBC component.)

Remarks

(For information about creating and/or compacting Access databases, see **Access Extensions** below.)

In order to use this function you will need to be familiar with the process of creating and modifying Data Sources. See Appendix G: Connection String Syntax for more information. It may also be helpful to familiarize yourself with the ODBC Data Source Administrator program, which can be invoked from the Windows Control Panel or with the `SQL_DataSourceAdmin` function. It has its own Help File.

IMPORTANT NOTE: This function does not make very many "judgements" about the correctness or completeness of the Data Source attributes that you supply. This is *intentional*. It will, for example, return an Error Code if you attempt to use a `DSN=` values that ends in a backslash, because that is clearly not valid. But is entirely *possible* to create or modify a Data Source that refers to a non-existent database, or that has required attributes that are missing. Keep in mind that if the `SQL_OpenDB` function encounters an incomplete or incorrect Data Source, it will automatically display the dialogs that are necessary to allow a connection. It is *completely normal* under some circumstances for a Data Source to be purposely "incomplete", so `SQL_DataSourceModify` will not complain if you create an incomplete or incorrect Data Source.

The *lRequestType*& parameter must be one of the following values:

`ODBC_ADD_DSN` or `ODBC_ADD_SYS_DSN`

Adds a new Data Source or System Data Source.

The *sDriverName*\$ parameter must be the name of an ODBC Driver that is currently installed on the system. You can obtain a list of the currently installed Drivers by using the technique shown under `SQL_DriverCount`.

The *sAttributes*\$ parameter must contain a list of Data Source attributes, delimited with the "pipe" symbol (|) or with Carriage Returns and/or Line Feeds (ASCII 13 or 10).

Example:

```
sDriverName$ = "Microsoft Access Driver (*.mdb)"

sAttributes$ = "DSN=MyDataSource|DBQ=C:\SQLTools\SQL-
DUMP\SQL-DUMP.MDB|DEFAULTDIR=C:\"

lResult& = SQL_DataSourceModify(ODBC_ADD_DSN,
sDriverName$, sAttributes$)
```

For more complete information about the requirements for the *sAttributes*\$ parameter, see Appendix G: Connection String Syntax.

`ODBC_CONFIG_DSN` or `ODBC_CONFIG_SYS_DSN`

Modifies an existing Data Source or System Data Source.

This example reconfigures the ODBC_ADD_DSN example (above) to use drive D instead of drive C:

```
sDriverName$ = "Microsoft Access Driver (*.mdb)"

sAttributes$ = "DSN=MyDataSource|DBQ=D:\SQLTools\SQL-
DUMP\SQL-DUMP.MDB|DEFAULTDIR=D:\ "

lResult& = SQL_DataSourceModify(ODBC_CONFIG_DSN,
sDriverName$, sAttributes$)
```

ODBC_REMOVE_DSN or ODBC_REMOVE_SYS_DSN

Removes an existing Data Source or System Data Source.

The *sDriverName\$* parameter must be the name of an ODBC Driver that is currently installed on the system, such as "Microsoft Access Driver (*.mdb)".

The *sAttributes\$* parameter must contain a valid DSN= value, in order to identify the Data Source to be removed.

Example:

```
sDriverName$ = "Microsoft Access Driver (*.mdb)"

sAttributes$ = "DSN=MyDataSource"

lResult& = SQL_DataSourceModify(ODBC_REMOVE_DSN,
sDriverName$, sAttributes$, 0)
```

ODBC_REMOVE_DEFAULT_DSN.

Removes the current default Data Source. The *sDriverName\$* and *sAttributes\$* parameters are ignored.

Access Extensions

The Microsoft Access ODBC Driver provides a number of additional functions that can be used with SQL_DataSourceModify %ODBC_ADD_DSN.

```
CREATE_DBV2=
CREATE_DBV3=
CREATE_DBV4=
CREATE_DB=
CREATE_SYSDB=
```

These *sAttribute\$* values can be used to create a new Access database. CREATE_DBV2 creates a database that is compatible with Access 2 (16-bit). CREATE_DBV3 creates a database that is compatible with Access 95, Access 97, and later versions of Access. CREATE_DBV4 creates a database that is compatible with Access 2000 and later versions of Access. CREATE_DB (without a V-number) creates a database using the most recent

version of Access that the current ODBC driver supports. `CREATE_SYSDb` creates a system database file.

Example:

```
'Create an MDB file that is compatible with Access 97.
```

```
sDriver$ = "Microsoft Access Driver (*.mdb)"
```

```
sAttribs$ = "CREATE_DBV3=C:\MyNew.mdb"
```

```
SQL_DataSourceModify ODBC_ADD_DSN, sDriver$, sAttribs$, 0
```

IMPORTANT NOTE: The `CREATE_` functions do not allow quotation marks to be used around the MDB file name and path, so you can't use file names or directory names which contain spaces. This is a limitation of the Access ODBC Driver, not SQL Tools. It is possible, however, to create an MDB file with an 8.3-compatible name and then rename and/or move the file using long file and directory names. See `KILL` and `NAME` in your BASIC documentation.

The `CREATE_` functions also support an optional parameter called "sort order", which must be one of the following keywords: `General`, `Traditional Spanish`, `Dutch`, `Swedish/Finnish`, `Norwegian/Danish`, `Icelandic`, `Czech`, `Hungarian`, `Polish`, `Russian`, `Turkish`, `Arabic`, `Hebrew`, or `Greek`. To create a database that uses Polish sorting (for example), change the sample code above like this:

```
sAttribs$ = "CREATE_DBV3=C:\MyNew.mdb Polish"
```

If no sort order is specified, `General` will be used.

Note that the `ODBC_ADD_DSN` constant is used, even though a new DSN is not actually created.

`COMPACT_Db=`

This `sAttributes$` value can be used to repair a damaged Access database. This process is usually called "compacting" the database, because it can also be used to remove wasted space from a database. (Wasted space can be created by deleting tables, columns, data, or just about anything else from an Access database. Wasted space can also be created by [UPDATE](#) operations.)

Note that you must use *two* file names with this function, a "source" file and a "target" file, separated by a single space character. The source file is the MDB file that should be compacted. The target file is the name of the resulting (compacted) database. The two file names *may* be the same, but for maximum safety we recommend using two *different* names. Then, if (and only if) the `COMPACT_Db` operation is successful, delete the source file and rename the target file.

Example:

'Compact the SQL-Dump.MDB file:

```
sDriver$ = "Microsoft Access Driver (*.mdb)"

sAttribs$ = "COMPACT_DB=C:\SQLTools\Samples\SQL-Dump.MDB
C:\Temp.MDB"

lResult& = SQL_DataSourceModify ODBC_ADD_DSN, sDriver$,
sAttribs$, 0

If lResult& = %SUCCESS THEN
    'delete C:\SQLTools\Samples\SQL-Dump.MDB
    'then rename C:\Temp.MDB to
    'C:\SQLTools\Samples\SQL-Dump.MDB.
END IF
```

IMPORTANT NOTE: The COMPACT_DB function does not allow quotation marks to be used around the MDB file name and path, so you can't use file names or directory names which contain spaces. This is a limitation of the Access ODBC Driver, not SQL Tools. You should use BASIC to obtain an 8.3-compatible path/file string for the database, and use that instead of the "long" path/file name.

Note that the ODBC_ADD_DSN constant is used, even though a new DSN is not actually created.

Note also that the COMPACT_DB function accepts the same optional "sort order" parameter as the CREATE_ functions. See above for details.

Diagnostics

This function returns Error Codes and can generate SQL Tools Error Messages.

Example

See **Remarks** above for examples.

Driver Issues

None.

Speed Issues

None.

See Also

SQL_DataSourceAdmin, SQL_DataSourceAdd, SQL_DataSourceCount,
SQL_DataSourceInfoStr.

SQL_DataSourceNumber

Summary

Returns the Datasource Number (if any) that corresponds to a Datasource Name.

Twin

None.

Family

Environment Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DataSourceNumber(sDataSourceName$)
```

Parameters

sDataSourceName\$

A string that contains the name of a datasource.

Return Values

If a Datasource with the name *sDataSourceName\$* is found, its number will be returned. Otherwise, negative one (-1) will be returned.

Remarks

This function is *not* case-sensitive. If a datasource named "dBase Files" exists, then searching for "DBASE FILES", "dBase files", etc. will result in a match.

Diagnostics

This function does not return Error Codes because an Error Code like SUCCESS_WITH_INFO (value 1) could be confused with the answer "that string corresponds to Datasource number one". It can, however, generate ODBC Error Messages.

Example

```
PRINT SQL_DataSourceNumber("dBase Files")
```

Driver Issues

None.

Speed Issues

None.

See Also

SQL_DataSourceCount, SQL_DataSourceInfoStr and
SQL_DataSourceNumber

SQL_DBAttribStr

Summary

Returns a Database Attribute in string form. (Generally speaking, an "Attribute" is a value that can be changed by your program.)

Twin

SQL_DatabaseAttribStr

Family

Database Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_DBAttribStr(lAttribute&)
```

Parameters

lAttribute&

A DB_ATTR_ constant. See **Remarks** below for more information.

Return Values

If *lAttribute&* has a valid value, and if the requested attribute type is supported by the ODBC driver that you are using, this function will return the attribute in string form. Otherwise, an empty string will be returned.

Remarks

Only *certain* Database Attributes are useful in string form. For a list of *numeric* Database Attributes, see SQL_DBAttribUInt.

The following constants can be used to obtain database attributes in string form:

DB_ATTR_CURRENT_CATALOG

The name of the catalog that is used by the Datasource.

DB_ATTR_TRACEFILE

The name of the trace file that will be used if ODBC API Tracing is activated.

DB_ATTR_TRANSLATE_LIB

The name of a library that contains the ODBC API functions called SQLDriverToDataSource and SQLDataSourceToDriver, which the ODBC driver uses to perform tasks such as character set translation.

Diagnostics

If you attempt to access an attribute that is not supported by your ODBC driver, an ODBC Error Message will be generated.

Example

```
Print SQL_DBAtribStr(DB_ATTR_TRACEFILE)
```

Driver Issues

None.

Speed Issues

These values are *not* cached by SQL Tools, they are requested from the ODBC driver each time that the `SQL_DBAtribStr` function is used. So if your program needs to use these values repeatedly, you may be able to increase your program's performance by using `SQL_DBAtribStr` to obtain a value and then storing it in a variable.

See Also

`SQL_SetDBAtribStr` and Database Information and Attributes

SQL_DBAAttrUInt

Summary

Returns a Database Attribute in numeric form. (Generally speaking, an "Attribute" is a value that can be changed by your program.)

Twin

SQL_DatabaseAttrUInt

Family

Database Info/Attr Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_DBAAttrUInt(lAttribute&)
```

Parameters

lAttribute&

A `DB_ATTR_` constant. See **Remarks** below for more information.

Return Values

If *lAttribute&* has a valid value, and if the requested attribute type is supported by the ODBC driver that you are using, this function will return the attribute in numeric form. Otherwise, a value of zero (0) will be returned.

Remarks

Only *certain* Database Attributes are useful in numeric form. For a list of *string* Database Attributes, see `SQL_DBAAttrStr`.

The following constants can be used to obtain database attributes in numeric form:

`DB_ATTR_ACCESS_MODE`

This value will always be `SQL_MODE_READ_WRITE` (value zero) or `SQL_MODE_READ_ONLY` (value one).

`DB_ATTR_AUTOCOMMIT`

This value will always be `SQL_AUTOCOMMIT_OFF` (value zero) or `SQL_AUTOCOMMIT_ON` (value one).

`DB_ATTR_CONNECTION_DEAD` **Read Only**

ODBC 3.x+ ONLY: This value will be zero (0) if the connection to the database is active, or one (1) if it is dead.

This is a "read only" attribute that can be obtained with the `SQL_DBAAttrUInt` function but can't be set with `SQL_SetDBAttrUInt`.

DB_ATTR_CONNECTION_TIMEOUT

The number of seconds that the ODBC driver will wait for any request to be completed before returning to your program. The default value is zero (0), which indicates that the driver should wait indefinitely.

DB_ATTR_DISCONNECT_BEHAVIOR

This attribute is not fully documented by the Microsoft ODBC Software Developer Kit. It appears to be related to connection pooling. This attribute will always be `SQL_DB_RETURN_TO_POOL` (value zero) or `SQL_DB_DISCONNECT` (value one).

DB_ATTR_LOGIN_TIMEOUT

The number of seconds that the database will wait for a login request to be completed before returning to your program. The default value is driver-dependent but it is usually zero (0), which indicates that the driver should wait indefinitely.

DB_ATTR_METADATA_ID

This value determines how certain characters are interpreted in "Info" requests. Since SQL Tools handles all Info ("catalog") functions internally, this value should always be zero (0).

DB_ATTR_ODBC_CURSORS

This value indicates how the ODBC Driver Manager uses the ODBC Cursor Library, which is used to simulate certain cursor behavior if an ODBC driver does not support the behavior. This value will always be one of the following values:

`SQL_CUR_USE_IF_NEEDED` (value zero) to indicate that the ODBC Driver Manager uses the ODBC Cursor Library as it needs to, in order to simulate cursor behaviors that are requested by your program. This is the default SQL Tools value, but it is not the native ODBC default. (In other words, SQL Tools explicitly sets this value instead of relying on the ODBC default value.)

`SQL_CUR_USE_ODBC` (value one) to indicate that the Driver Manager uses the ODBC Cursor Library for all cursor functions, even if a driver supports the function.

`SQL_CUR_USE_DRIVER` (value two) to indicate that the Driver Manager does not use the ODBC Cursor Library. This is the ODBC native default, but it is not the SQL Tools default. (In other words, SQL Tools explicitly sets this value instead of relying on the ODBC default value.)

DB_ATTR_PACKET_SIZE

An unsigned integer value that indicates the network packet size, in bytes. Many Datasources do not support this option.

DB_ATTR_QUIET_MODE

If this value is zero (0), the ODBC driver operates in the "quiet mode" and does not display any dialog boxes. (This setting does *not* affect the dialog boxes that are provided by the SQL Tools `SQL_OpenDB` and `SQL_OpenDatabase` functions.) If this value is nonzero, it represents the handle of the window that the dialog boxes should use as a parent window. The default value is zero.

DB_ATTR_TRACE

The current state of the ODBC API Trace Mode, either `SQL_TRACE_OFF` (value zero) or `SQL_TRACE_ON` (value one).

DB_ATTR_TRANSLATE_OPTION

A 32-bit bitmasked value that is passed to the translation DLL. (See `SQL_DBAtribStr(DB_ATTR_TRANSLATE_LIB)`.)

DB_ATTR_TXN_ISOLATION

A 32-bit bitmasked value that describes the database's Transaction Isolation Level. For more information, please refer to the Microsoft ODBC Software Developer Kit.

Diagnostics

This function does not return Error Codes because they could be confused with attribute values, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_DBAtribUInt(DB_ATTR_LOGIN_TIMEOUT)
```

Driver Issues

None.

Speed Issues

These values are *not* cached by SQL Tools, they are requested from the ODBC driver each time that the `SQL_DBAtribUInt` function is used. So if your program needs to use these values repeatedly, you may be able to increase your program's performance by using `SQL_DBAtribUInt` to obtain a value and then storing it in a variable.

See Also

Database Information and Attributes

SQL_DBAutoCommit

Summary

Sets a database's AutoCommit status.

Twin

SQL_DatabaseAutoCommit

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DBAutoCommit(lOnOff&)
```

Parameters

lOnOff&

Use False (zero) to disable a database's AutoCommit function, or True or any nonzero value to enable it. The default setting is True (AutoCommit enabled).

Return Values

If the AutoCommit mode is successfully changed, this function will return `SUCCESS` or `SUCCESS_WITH_INFO`. If the operation is not successful, an Error Code will be returned.

Remarks

The default AutoCommit behavior for all databases is `AutoCommit True` which tells the database that it should automatically commit all transactions as soon as they are executed. You can use this function to turn off the AutoCommit feature, and then use the `SQL_EndTrans` or `SQL_EndTransaction` function to manually Commit or Roll-Back each transaction.

See [Committing Transactions Manually](#) for more information.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
SQL_DBAutoCommit False
```

Driver Issues

None.

Speed Issues

None.

See Also: [Committing Transactions Manually](#)

SQL_DBDataTypeCount

Summary

Returns the number of Datasource-dependent Data Types that are supported by a database.

Twin

SQL_DatabaseDataTypeCount

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DBDataTypeCount
```

Parameters

None.

Return Values

This function returns an integer value that indicates the number of Datasource-dependent Data Types that are supported by a database. If the function encounters an error it will return zero (0).

Remarks

SQL Tools can provide a great deal of information about the various Datasource-dependent Data Types that are supported by a database. These data types are referenced by numbers between one (1) and the number of data types that are supported.

Be very careful to avoid confusing these numbers with SQL Data Type identifier values.

For example, the first Datasource-dependent Data Type that a database reports as being available is always referred to as Data Type 1, the second type that is reported is Data Type 2, and so on. Data Type 1 *may or may not* be the `SQL_CHAR` data type, which has a value of one (1).

Diagnostics

This function does not return Error Codes because they might be confused with numeric return values. For example, this function does not return `SUCCESS_WITH_INFO`, which has a numeric value of one (1), because it might be confused with the result "this database supports one data type". This function can, however, generate ODBC Error Messages.

Example

```
For lType& = 1 To SQL_DBDataTypeCount
    Print SQL_DBDataTypeInfoStr(lType&, DTYPE_NAME)
Next
```

Driver Issues

None.

Speed Issues

None.

See Also

Datasource-dependent Data Types

SQL_DBDataTypeInfoSInt

Summary

Returns information about a Datasource-dependent Data Type that is supported by a database, in numeric form.

Twin

SQL_DatabaseDataTypeInfoSInt

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DBDataTypeInfoSInt(lDataTypeNumber&, _  
                                lInfoType&)
```

Parameters

lDataTypeNumber&

A number between one (1) and the value returned by the `SQL_DBDataTypeCount` function, i.e. the number of Datasource-dependent data types that are supported by a database.

lInfoType&

A constant that indicates the type of information that is being requested. See **Remarks** below for details.

Return Values

This function returns signed integers in the `BAS_LONG` range. The value that is returned will depend on the type of information that is being requested.

Remarks

This function and the `SQL_DBDataTypeInfoStr` function are used to obtain information about the data types that are supported by a database.

When using this function, it is very important for you to avoid confusing a Data Type Number with the value that is associated with a SQL Data Type. For example, the first Datasource-dependent Data Type that is reported by a database is always referenced as Data Type number one (1), the second is always number two (2), and so on. Data Type Number one *may or may not be* the `SQL_CHAR` data type, which has a numeric value of one (1).

It is also important to avoid confusing the Data Types that are supported by a database with the "native" SQL Data Types. For example, a Data Type called `COUNTER` is supported by many different databases. It is usually implemented as a `SQL_INTEGER` column that is not nullable, but some databases use a different SQL Data Type to create `COUNTER` columns. The `SQL_DBDataTypeInfoSInt` function reports information about `COUNTER` columns as they are implemented by the database, *not* about the native `SQL_INTEGER` data type.

Only certain types of Data Type Info are useful in numeric form. See

SQL_DBDataTypeInfoStr for Info types that are useful in string form.

The *InfoType*& parameter that is passed to this function should always be one of the following values.

DTYPE_AUTO_UNIQUE_VALUE

This *InfoType*& returns a SQL_TRUE (value 1) or SQL_FALSE (value 0) value to indicate whether or not the data type is auto-incrementing. Example: a COUNTER data type is usually auto-incrementing to ensure that duplicate values are never used.

DTYPE_CASE_SENSITIVE

This value indicates whether or not a character (string) data type is case-sensitive in collations and comparisons. This *InfoType*& will always return one of the following values:

SQL_TRUE (value 1) if the data type is a character data type which is case-sensitive

SQL_FALSE (value 0) if the data type is not a characters data type, or is a character data type that is not case-sensitive.

DTYPE_COLUMN_SIZE

The display size of the column

DTYPE_FIXED_PREC_SCALE

This *InfoType*& returns a SQL_TRUE (value 1) or SQL_FALSE (value 0) value to indicate whether or not the data type has predefined fixed precision and scale.

DTYPE_INTERVAL_PRECISION

ODBC 3.0 ONLY: If the data type is a SQL_INTERVAL, this *InfoType*& can be used to obtain the value of the interval's leading precision.

DTYPE_MINIMUM_SCALE and DTYPE_MAXIMUM_SCALE

These *InfoType*& values are used to obtain the minimum and maximum scales of the data type. If a data type has a fixed scale, these values are the same. For example, a SQL_TIMESTAMP column might have a fixed scale for fractional seconds.

DTYPE_NULLABLE

This value indicates whether or not a Data Type is nullable. This *InfoType*& will always return one of the following values:

SQL_NULLABLE if the data type *does* accept Null values.

SQL_NO_NULLS if the data type *does not* accept Null values.

SQL_NULLABLE_UNKNOWN if it is not known whether or not the column accepts Null values.

Please note that DTYPE_NULLABLE information is available from all ODBC drivers. Compare DTYPE_ISNULLABLE below, which is available only from drivers that support ODBC 3.x and above.

DTYPE_NUM_PREC_RADIX

ODBC 3.x+ ONLY: See Num Prec Radix.

DTYPE_SEARCHABLE

This value indicates how the data type is used in a SQL statement's *WHERE* clause. This *InfoType* will always return one of the following values:

SQL_PRED_NONE (value 0) means that the column cannot be used in a *WHERE* clause.

SQL_PRED_CHAR (value 1) means that the column can be used in a *WHERE* clause, but only with the *LIKE* predicate.

SQL_PRED_BASIC (value 2) means that the column can be used in a *WHERE* clause with all the comparison operators except *LIKE* (comparison, quantified comparison, *BETWEEN*, *DISTINCT*, *IN*, *MATCH*, and *UNIQUE*).

DTYPE_SQL_DATA_TYPE

ODBC 3.x+ ONLY: The SQL Data Type of the column. If an ODBC driver supports this *InfoType*, the return value will be the same value that is returned for DTYPE_DATA_TYPE, except for interval and datetime data types. For intervals and datetimes, the DTYPE_SQL_DATA_TYPE value will be SQL_INTERVAL or SQL_TIMESTAMP, and the DTYPE_SQL_DATETIME_SUB value (see below) will contain the subcode for the specific interval or datetime data type. Also see DTYPE_TYPE below, which is supported by all ODBC drivers.

DTYPE_SQL_DATETIME_SUB

ODBC 3.x+ ONLY: If the value of DTYPE_SQL_DATA_TYPE (above) is SQL_DATETIME or SQL_INTERVAL, this *InfoType* can be used to obtain the datetime/interval subcode. For example, the DTYPE_SQL_DATA_TYPE might be SQL_INTERVAL, and the DTYPE_SQL_DATETIME_SUB might be ODBC2_INTERVAL_SECOND to indicate the *type* of SQL_INTERVAL.

DTYPE_TYPE

The SQL Data Type of the Data Type, i.e. a numeric value that corresponds to SQL_CHAR, SQL_INTEGER, and so on. In some cases, this value is driver-specific. This column is sometimes called the Concise Data Type and is almost always a more reliable Indicator of a Data Type's type than the DTYPE_SQL_DATA_TYPE (see above).

DTYPE_UNSIGNED_ATTRIBUTE

This *InfoType* will return SQL_TRUE (value 1) if the data type is a Signed numeric data type, and SQL_FALSE (value 0) if it is an Unsigned numeric data type or a non-numeric data type like SQL_CHAR.

Diagnostics

This function does not return Error Codes because they could be confused with legitimate return values, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'display data type of data type 1  
Print SQL_DBDataTypeInfoSInt(1, DTYPE_TYPE)
```

Driver Issues

None.

Speed Issues

None.

See Also

Datasource-dependent Data Types

SQL_DBDataTypeInfoStr

Summary

Returns information about a Datasource-dependent Data Type that is supported by a database, in string form.

Twin

SQL_DatabaseDataTypeInfoStr

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_DBDataTypeInfoStr(lDataTypeNumber&, _  
                                lInfoType&)
```

Parameters

lDataTypeNumber&

A number between one (1) and the value that is returned by the `SQL_DBDataTypeCount` function, i.e. the number of datasource-dependent data types that are supported by a database.

lInfoType&

A constant that indicates the type of information that is being requested. See **Remarks** below for details.

Return Values

This function returns strings. The string value that is returned will depend on the type of information being requested.

Remarks

This function and the `SQL_DBDataTypeInfoSInt` function are used to obtain information about the data types that are supported by a database.

When using this function, it is very important for you to avoid confusing a Data Type Number with the value that is associated with a SQL Data Type. For example, the first datasource-dependent data type that is reported by a database is always referenced as data type number one (1), the second is always data type number two (2), and so on. Data type number one *may or may not be* the `SQL_CHAR` data type, which has a numeric value of one (1).

It is also important to avoid confusing the Data Types that are supported by a database with the "native" SQL Data Types. For example, a Data Type called `COUNTER` is supported by many different databases. It is usually implemented as a `SQL_INTEGER` column that is not nullable, but some databases use a different SQL Data Type to create `COUNTER` columns. The `SQL_DBDataTypeInfoStr` function reports information about `COUNTER` columns as they are implemented by the database, *not* about the native `SQL_INTEGER` data type.

Only certain types of Data Type Info are useful in string form. See

SQL_DBDataTypeInfoSInt for Info types that are useful in numeric form.

The *InfoType*& that is passed to this function should always be one of the following values.

DTYPE_CREATE_PARAMS

This *InfoType*& value can be used to obtain a list of keywords, separated by commas, in the language of the country where it is used, corresponding to the parameters that a program may specify (in parentheses) when using the name that is returned in the DTYPE_TYPE_NAME field. The keywords will vary, depending on the data type and the database that supports it. The keywords will always appear in the order that the syntax requires them to be used. Example: a Microsoft Access database TEXT column might specify the DTYPE_CREATE_PARAMS string "MAX LENGTH", meaning that you must use a string like TEXT(MAX LENGTH 10) when referring to the column in a SQL statement which is intended to create a new TEXT column.

DTYPE_NAME

The name of the data type. Keep in mind that this name is *not* the name of the SQL Data Type on which the data type is based. For example, this *InfoType*& might return the name "COUNTER" for a column, while the name of the SQL Data Type is SQL_INTEGER, not COUNTER.

DTYPE_LITERAL_PREFIX and DTYPE_LITERAL_SUFFIX

These *InfoType*& values can be used to return the strings that are used as the "literal value" identifiers for a data type. For example, the string "0x" (zero-ex) might be returned for a binary column to indicate that the prefix 0x can be used to denote a literal binary value. Or a string containing a single quote (') might be returned for a string column, to indicate that a single quote should be used (instead of a double quote) to delimit strings in SQL statements.

DTYPE_LOCAL_TYPE_NAME

This *InfoType*& can be used to obtain a localized version of the datasource-dependent name of the data type. An empty string is returned if a localized name is not supported by the datasource. The DTYPE_LOCAL_TYPE_NAME string is intended for display purposes only.

Diagnostics

This function does not return Error Codes (because it returns only string values), but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'display data type name of data type 1
Print SQL_DBDataTypeInfoStr(1,DTYPE_NAME)
```

Driver Issues: None.

Speed Issues: None.

See Also: Datasource-dependent Data Types

SQL_DBDataTypeNumber

Summary

Returns the Data Type Number that corresponds to a Datasource-dependent Data Type Name.

Twin

SQL_DatabaseDataTypeNumber

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_DBDataTypeNumber( sTypeName$ )
```

Parameters

sTypeName\$

The name of a data type that is supported by a database, such as "COUNTER".

Return Values

If a data type with the specified name is supported by the database, this function will return the Data Type Number that corresponds to the name.

If no matching data type is found, this function will return negative one (-1).

Remarks

See Datasource-dependent Data Types for information about using this function.

Diagnostics

This function does not return Error Codes because they could be confused with legitimate return values. For example, the Error Code `SUCCESS_WITH_INFO` (value 1) could be confused with the answer "that string corresponds to Data Type number 1". This function can, however, return ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the data type number  
'for the type called COUNTER  
Print SQL_DBDataTypeNumber( "COUNTER" )
```

Driver Issues: None.

Speed Issues: None.

See Also: Datasource-dependent Data Types

SQL_DBInfoStr

Summary

Provides information about a database, in string form. (Generally speaking, "Information" values cannot be changed. "Attributes" are settings that can be changed by your program.)

Twin

SQL_DatabaseInfoStr

Family

Database Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_DBInfoStr(lInfoType&)
```

Parameters

lInfoType&

A constant that indicates the type of information that is being requested. See **Remarks** below for valid values.

Return Values

If a valid *lInfoType&* is used, the return value of this function will be a string that represents the information that is being requested.

If an invalid value is used for *lInfoType&*, an empty string will be returned.

Remarks

Only certain types of database information are useful in string form. For a list of *lInfoType&* values that are useful in numeric form, see `SQL_DBInfoUInt`.

Please note that nearly 200 different types of information can be obtained with the `SQL_DBInfoStr` and `SQL_DBInfoUInt` functions, and many of the numeric values are bitmasked values that are capable of returning as many as many as 32 different sub-values.

The following *lInfoType&* values can be used to obtain information about a database, in string form.

`DB_ACCESSIBLE_PROCEDURES`

Returns "Y" if your program can execute all of the Stored Procedures in the database, or "N" if it cannot.

`DB_ACCESSIBLE_TABLES`

Returns "Y" if your program is guaranteed access to all tables in the database, or "N" if there are some tables that may be inaccessible.

DB_CATALOG_NAME

Returns "Y" if the server supports catalog names, or "N" if it does not.

DB_CATALOG_NAME_SEPARATOR

The character that the Datasource defines as the separator between a catalog name and the qualified name element that follows or precedes it. An empty string is returned if catalogs are not supported by the Datasource. (The ODBC 2.0 terminology for this value was DB_QUALIFIER_NAME_SEPARATOR)

DB_CATALOG_TERM

A string containing the Datasource vendor's name for a catalog, like "DATABASE" or "DIRECTORY". This string can be in upper, lower, or mixed case. An empty string is returned if catalogs are not supported by the Datasource. (The ODBC 2.0 name for this value was DB_QUALIFIER_TERM.)

DB_COLLATION_SEQ

ODBC 3.x+ ONLY: A string that contains the name of the default collation method for the default character set, like ISO 8859-1 or EBCDIC. If this value is unknown, an empty string will be returned

DB_COLUMN_ALIAS

Returns "Y" if the Datasource supports column aliases. Otherwise, this function returns "N".

DB_CONNECTION_STRING

The connection string that was used to open a database.

DB_DATA_SOURCE_NAME

A string containing the Datasource name used during connection. If the connection string did not contain the DSN keyword (such as when it contains the DRIVER keyword), this will be an empty string.

DB_DATA_SOURCE_READ_ONLY

Returns "Y" if the Datasource is set to the Read Only mode, or "N" if it is not.

DB_DATABASE_NAME

A string that contains the name of the current database in use (if the Datasource defines an object called a "database".)

DB_DBMS_NAME

A string that contains the name of the DBMS product that is being accessed by the ODBC driver.

DB_DBMS_VER

A string that contains the version of the DBMS product that is being accessed by the ODBC driver.

The version is presented in the form `##.##.####`, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version. The driver can optionally append DBMS product-specific version information. Example, "02.01.0003 Abc 4.1".

DB_DESCRIBE_PARAMETER

Returns "Y" if parameters can be described, or "N", if they cannot.

DB_DM_VER

ODBC 3.x+ ONLY: A string containing the version of the ODBC Driver Manager.

The version is presented in the form `##.##.####.####`, where the first set of (two) digits represent the major ODBC version, the second set of (two) digits is the minor ODBC version, the third set of (four) digits is the Driver Manager major build number, and the last set of (four) digits is the Driver Manager minor build number.

DB_DRIVER_NAME

A string that contains the file name of the ODBC driver that is being used to access the Datasource.

DB_DRIVER_ODBC_VER

A string that contains the version of ODBC that the ODBC driver supports. The version is presented in the form `##.##`, where the first two digits are the major version number and the next two digits are the minor version number.

DB_DRIVER_VER

A string that contains the version of the ODBC driver and, optionally, a description of the driver. At a minimum, the version is presented in the form `##.##.####`, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version.

DB_DSN_FILENAME

The name of the DSN File (if any) that was used to open the database.

DB_EXPRESSIONS_IN_ORDERBY

Returns "Y" if the Datasource supports expressions in an `ORDER BY` list. Otherwise, this function returns "N".

DB_IDENTIFIER_QUOTE_CHAR

The character that is used as the starting and ending delimiter of a quoted

(delimited) identifier in SQL statements. If the Datasource does not support quoted identifiers, an empty string is returned.

DB_INTEGRITY

Returns "Y" if the Datasource supports the Integrity Enhancement Facility. Otherwise, this function returns "N". (The ODBC 2.0 name for this function was DB_ODBC_SQL_OPT_IEF.)

DB_KEYWORDS

A string that contains a comma-delimited list of all datasource-specific keywords. This list does *not* contain keywords that are specific to ODBC, or keywords that are used by both the Datasource and ODBC. Applications should not use these words in object names (table names, column names, etc.).

DB_LIKE_ESCAPE_CLAUSE

Returns "Y" if the Datasource supports an escape character for the percent character (%) and underscore character (_) in a *LIKE* predicate, and if the driver supports the ODBC syntax for defining a *LIKE* predicate escape character. Otherwise, this function returns "N".

DB_MAX_ROW_SIZE_INCLUDES_LONG

Returns "Y" if the maximum row size returned for the DB_MAX_ROW_SIZE information type includes the length of all SQL_LONGVARCHAR, SQL_WLONGVARCHAR, and SQL_LONGVARBINARY columns in the row. Otherwise, this function returns "N".

DB_MULT_RESULT_SETS

Returns "Y" if the Datasource supports multiple result sets, or "N" if it does not.

DB_MULTIPLE_ACTIVE_TXN

Returns "Y" if the driver supports more than one active transaction at the same time, or "N" if only one transaction can be active at any given time.

DB_NEED_LONG_DATA_LEN

Returns "Y" if the Datasource requires the length of a long data value (such as SQL_LONGVARCHAR or SQL_LONGVARBINARY) before that value can be sent to the Datasource, or "N" if it does not require the length.

DB_ODBC_VER

A string that contains the version of ODBC to which the ODBC Driver Manager conforms. The version is presented in the form ##.##.0000, where the first two digits are the major version and the next two digits are the minor version.

DB_ORDER_BY_COLUMNS_IN_SELECT

Returns "Y" if the columns in an `ORDER BY` clause must be in the select list. Otherwise, this function returns "N".

DB_OUTER_JOINS

This function is listed by, but not documented in, the Microsoft ODBC Software Developer Kit. It appears to return "Y" if the database supports outer joins, and "N" if it does not.

DB_PROCEDURE_TERM

A string that contains the Datasource vendor's name for a procedure, like example, "DATABASE PROCEDURE", "STORED PROCEDURE", "PROCEDURE", "PACKAGE", or "STORED QUERY".

DB_PROCEDURES

Returns "Y" if the Datasource supports Stored Procedures *and* the driver supports the ODBC procedure invocation syntax (*call*). Otherwise, this function returns "N".

DB_ROW_UPDATES

Returns "Y" if a keyset-driven or mixed cursor maintains row versions or values for all fetched rows and can therefore detect any updates made to a row by any user since the row was last fetched. (This applies only to updates, not to deletions or insertions.) The driver can return the `SQL_ROW_UPDATED` flag to the row status array when the `SQL_Fetch` or `SQL_FetchRel` function is used. Otherwise, this function returns "N".

DB_SCHEMA_TERM

A string that contains the Datasource vendor's name for a schema. For example, "OWNER", "Authorization ID", or "Schema". The string can be returned in upper, lower, or mixed case. (The ODBC 2.0 name for this function was `DB_OWNER_TERM`.)

DB_SEARCH_PATTERN_ESCAPE

A string that contains the character that the driver supports as an escape character, to permit the use of the underscore (`_`) and percent sign (`%`) as valid characters in search patterns. The escape character applies only for those Info ("catalog") function arguments that support search strings. If this string is empty, the driver does not support a search-pattern escape character.

DB_SERVER_NAME

A string that contains the actual datasource-specific server name.

DB_SPECIAL_CHARACTERS

A string that contains all of the special characters (i.e. all characters except

"a" through "z", "A" through "Z", "0" through "9", and the underscore character) that can be used in an identifier name, such as a table name or column name. If an identifier contains one or more of these characters, the identifier must be delimited.

DB_TABLE_TERM

A string that contains the Datasource vendor's name for a table; for example, "TABLE" or "FILE". This string can be in upper, lower, or mixed case.

DB_USER_NAME

A string that contains the name that is used in a particular database. This can be different from the login name.

DB_XOPEN_CLI_YEAR

A string that contains the year of publication of the X/Open specification with which the version of the ODBC Driver Manager fully complies.

Diagnostics

This function does not return Error Codes, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_DBInfoStr(DB_USER_NAME)
```

Driver Issues

None.

Speed Issues

SQL Tools does *not* cache the values that are returned by this function. If your program needs one or more of these values repeatedly, you may be able to improve the speed of your program by obtaining a `SQL_DBInfoStr` or `SQL_DBInfoUInt` value and storing it in a variable, instead of repeatedly using this function.

See Also

Database Information and Attributes

SQL_DBInfoUInt

Summary

Provides information about a database, in numeric form. (Generally speaking, "information" values cannot be changed. "Attributes" are settings that can be changed by your program.)

Twin

SQL_DatabaseInfoUInt

Family

Database Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
dwResult??? = SQL_DBInfoUInt(lInfoType&)
```

...Or...

```
lResult& = SQL_DBInfoUInt(lInfoType&)
```

Parameters

lInfoType&

A constant that indicates the type of information that is being requested. See **Remarks** below for valid values.

Return Values

If a valid *lInfoType&* is used, the return value of this function will be a numeric value that represents the information that is being requested. In *most* cases the return value will be within the positive range of `BAS_LONG` variables, but some *lInfoType&* values return values that are larger than a `BAS_LONG` variable can hold, so this function returns `BAS_DWORD` values.

If an invalid value is used for *lInfoType&*, zero (0) will be returned.

Remarks

Only certain types of database information are useful in numeric form. For a list of *lInfoType&* values that are useful in string form, see `SQL_DBInfoStr`.

Please note that nearly 200 different types of information can be obtained with the `SQL_DBInfoStr` and `SQL_DBInfoUInt` functions, and many of the numeric values are bitmasked values that are capable of returning as many as many as 32 different sub-values.

The following *lInfoType&* values can be used to obtain information about a database, in numeric form.

DB_ACTIVE_ENVIRONMENTS

ODBC 3.x+ ONLY: The maximum number of active environments that the ODBC driver can support. If there is no specified limit or the limit is unknown, zero is returned. (SQL Tools supports only one active environment per program.)

DB_AGGREGATE_FUNCTIONS

ODBC 3.x+ ONLY: A bitmasked numeric value that describes support for the ODBC aggregate functions. The bitmask identifiers are:

SQL_AF_ALL
SQL_AF_AVG
SQL_AF_COUNT
SQL_AF_DISTINCT
SQL_AF_MAX
SQL_AF_MIN
SQL_AF_SUM

DB_ALTER_DOMAIN

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses in an [ALTER DOMAIN](#) statement that are supported by the Datasource. A return value of zero (0) means that the [ALTER DOMAIN](#) statement is not supported. The following bitmask identifiers are used to determine which clauses are supported:

SQL_AD_ADD_DOMAIN_CONSTRAINT (Adding a domain constraint is supported)

SQL_AD_ADD_DOMAIN_DEFAULT (<alter domain><set domain default clause> is supported)

SQL_AD_CONSTRAINT_NAME_DEFINITION (<constraint name definition clause> is supported for naming a domain constraint.)

SQL_AD_DROP_DOMAIN_CONSTRAINT (<drop domain constraint clause> is supported)

SQL_AD_DROP_DOMAIN_DEFAULT (<alter domain> <drop domain default clause> is supported)

The following bitmask identifiers describe the supported <constraint attributes> if <add domain constraint> is supported

SQL_AD_ADD_CONSTRAINT_DEFERRABLE
SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE
SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED
SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE

DB_ALTER_TABLE

A bitmasked value that describes the clauses in the [ALTER TABLE](#) statement that are supported by the Datasource. The following bitmask identifiers are

used:

SQL_AT_ADD_COLUMN_COLLATION (<add column> clause is supported, with the ability to specify column collation.)

SQL_AT_ADD_COLUMN_DEFAULT (<add column> clause is supported, with the ability to specify column defaults.)

SQL_AT_ADD_COLUMN_SINGLE (<add column> is supported.)

SQL_AT_ADD_CONSTRAINT (<add column> clause is supported, with the ability to specify column constraints.)

SQL_AT_ADD_TABLE_CONSTRAINT (<add table constraint> clause is supported.)

SQL_AT_CONSTRAINT_NAME_DEFINITION (<constraint name definition> is supported for naming column and table constraints.)

SQL_AT_DROP_COLUMN_CASCADE (<drop column> CASCADE is supported.)

SQL_AT_DROP_COLUMN_DEFAULT (<alter column> <drop column default clause> is supported.)

SQL_AT_DROP_COLUMN_RESTRICT (<drop column> RESTRICT is supported.)

SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE (<drop column> CASCADE is supported.)

SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT (<drop column> RESTRICT is supported.)

SQL_AT_SET_COLUMN_DEFAULT (<alter column> <set column default clause> is supported.)

The following bitmask identifiers describe the support for <constraint attributes> if the specifying of column or table constraints is supported:

SQL_AT_CONSTRAINT_INITIALLY_DEFERRED
SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE
SQL_AT_CONSTRAINT_DEFERRABLE
SQL_AT_CONSTRAINT_NON_DEFERRABLE

DB_ASYNC_MODE

This value indicates the level of Asynchronous Execution support that is provided by the ODBC driver. *ODBC-based Asynchronous Execution is not supported by SQL Tools.* See Asynchronous Execution. (SQL Tools does, however, support thread-based asynchronous execution of SQL statements.)

DB_BATCH_ROW_COUNT

ODBC 3.x+ ONLY: A bitmasked value that describes the availability of row counts. The following bitmask identifiers are used:

SQL_BRC_ROLLED_UP (Row counts for consecutive *INSERT*, *DELETE*, or *UPDATE* statements are "rolled up" into one value. If this bit is not set, then row counts are available for each individual statement.)

SQL_BRC_PROCEDURES (Row counts, if any, are available when a batch is executed in a stored procedure. If row counts are available, they may be rolled up or individually available, depending on the value of the SQL_BRC_ROLLED_UP bit.)

SQL_BRC_EXPLICIT (Row counts, if any, are available when a batch is executed with SQL_Stmt(SQL_STMT_EXECUTE) or SQL_Stmt(SQL_STMT_IMMEDIATE). If row counts are available, they may be rolled up or individually available, depending on the value of the SQL_BRC_ROLLED_UP bit.)

DB_BATCH_SUPPORT

ODBC 3.x+ ONLY: A bitmasked value that describes the driver's support for batched SQL statements. The following bitmask identifiers are used to determine which level is supported:

SQL_BS_SELECT_EXPLICIT (The ODBC driver supports explicit batches that can have statements which generate result sets.)

SQL_BS_ROW_COUNT_EXPLICIT (The driver supports explicit batches that can have statements which generate row counts.)

SQL_BS_SELECT_PROC (The driver supports explicit procedures that can have statements which generate result sets.)

SQL_BS_ROW_COUNT_PROC (The driver supports explicit procedures that can have statements which generate row counts.)

DB_BOOKMARK_PERSISTENCE

A bitmasked value that describes the database operations through which bookmarks persist. The following bitmask identifiers are used:

SQL_BP_CLOSE and SQL_BP_DROP (Bookmarks are valid after an application closes a statement. When SQL Tools closes a statement, in ODBC terminology it both "closes" and "drops" the statement.)

SQL_BP_DELETE (The bookmark for a row is still valid after that row has been deleted.)

SQL_BP_TRANSACTION (Bookmarks are still valid after an application commits or rolls back a transaction.)

SQL_BP_UPDATE (The bookmark for a row is still valid after any column in the row, including key columns, has been updated.)

SQL_BP_OTHER_HSTMT (A bookmark that is associated with one statement can be used with a different statement.)

DB_CATALOG_LOCATION

A numeric value that describes the position of the catalog in a qualified table name, either SQL_CL_START or SQL_CL_END. (The ODBC 2.0 name for this value was DB_QUALIFIER_LOCATION.)

DB_CATALOG_USAGE

A bitmasked value that describes the statements in which catalogs can be used. The following bitmask identifiers are used:

SQL_CU_DML_STATEMENTS (Catalogs are supported in [SELECT](#), [INSERT](#), [UPDATE](#), [DELETE](#), and, if supported, [SELECT FOR UPDATE](#) and positioned update and delete statements.)

SQL_CU_PROCEDURE_INVOCATION (Catalogs are supported in the ODBC stored procedure invocation statement [call](#).)

SQL_CU_TABLE_DEFINITION (Catalogs are supported in [CREATE TABLE](#), [CREATE VIEW](#), [ALTER TABLE](#), [DROP TABLE](#), and [DROP VIEW](#) statements.)

SQL_CU_INDEX_DEFINITION (Catalogs are supported in [CREATE INDEX](#) and [DROP INDEX](#) statements)

SQL_CU_PRIVILEGE_DEFINITION (Catalogs are supported in [GRANT](#) and [REVOKE](#) statements)

A value of zero (0) is returned if catalogs are not supported by the Datasource.

(The ODBC 2.0 name for this value was DB_QUALIFIER_USAGE.)

DB_CONCAT_NULL_BEHAVIOR

A numeric value that indicates how the Datasource handles the concatenation of null-valued character columns with non-null-valued character columns:

SQL_CB_NULL (Result is a null value.)

SQL_CB_NON_NULL (Result is the concatenation of *non*-null-valued column or columns.)

DB_CONVERT...

All of the DB_CONVERT functions are covered in this section except for DB_CONVERT_FUNCTIONS, which has its own section below.

Each of the DB_CONVERT_ values that are listed below returns a bitmasked value that describes the data-type conversions that are supported by the Datasource with the CONVERT scalar function for data of the specified type. If a bit of the bitmask equals zero (0) the Datasource does not support any conversions from data of the named type.

The following DB_CONVERT values all work the same way...

DB_CONVERT_BIGINT
DB_CONVERT_BINARY
DB_CONVERT_BIT
DB_CONVERT_CHAR
DB_CONVERT_DATE
DB_CONVERT_DECIMAL
DB_CONVERT_DOUBLE
DB_CONVERT_FLOAT
DB_CONVERT_INTEGER
DB_CONVERT_INTERVAL_DAY_TIME (ODBC 3.x ONLY)
DB_CONVERT_INTERVAL_YEAR_MONTH (ODBC 3.x ONLY)
DB_CONVERT_LONGVARBINARY
DB_CONVERT_LONGVARCHAR
DB_CONVERT_NUMERIC
DB_CONVERT_REAL
DB_CONVERT_SMALLINT
DB_CONVERT_TIME
DB_CONVERT_TIMESTAMP
DB_CONVERT_TINYINT
DB_CONVERT_VARBINARY
DB_CONVERT_VARCHAR
DB_CONVERT_WCHAR (ODBC 3.x ONLY)
DB_CONVERT_WLONGVARCHAR (ODBC 3.5 ONLY)
DB_CONVERT_WVARCHAR (ODBC 3.5 ONLY)

After you have obtained a bitmasked value for one of the functions above, you can use the following bitmask identifiers to find out whether or not the conversion is supported.

SQL_CVT_CHAR
SQL_CVT_NUMERIC
SQL_CVT_DECIMAL
SQL_CVT_INTEGER
SQL_CVT_SMALLINT
SQL_CVT_FLOAT
SQL_CVT_REAL
SQL_CVT_DOUBLE
SQL_CVT_VARCHAR
SQL_CVT_LONGVARCHAR
SQL_CVT_BINARY
SQL_CVT_VARBINARY
SQL_CVT_BIT
SQL_CVT_TINYINT

SQL_CVT_BIGINT
SQL_CVT_DATE
SQL_CVT_TIME
SQL_CVT_TIMESTAMP
SQL_CVT_LONGVARBINARY
SQL_CVT_INTERVAL_YEAR_MONTH
SQL_CVT_INTERVAL_DAY_TIME
SQL_CVT_WCHAR
SQL_CVT_WLONGVARCHAR
SQL_CVT_WVARCHAR

For example, to find out whether or not a conversion from a TINYINT to a NUMERIC value is supported, you would obtain the SQL_DBInfoUInt value for DB_CONVERT_TINYINT and check the SQL_CVT_NUMERIC bit. See Using Bitmasked Values for more information.

DB_CONVERT_FUNCTIONS

A bitmasked value that describes the scalar conversion functions that are supported by the driver and associated Datasource. The following bitmask identifiers are used:

SQL_FN_CVT_CAST
SQL_FN_CVT_CONVERT

DB_CORRELATION_NAME

A numeric value that describes whether or not table correlation names are supported:

SQL_CN_NONE (Correlation names are not supported.)

SQL_CN_DIFFERENT (Correlation names are supported, but they must differ from the names of the tables they represent.)

SQL_CN_ANY (Correlation names are supported and can be any valid user-defined name.)

DB_CREATE_ASSERTION

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses in a *CREATE ASSERTION* statement which are supported by the Datasource. The following bitmask identifier is used to determine which clauses are supported:

SQL_CA_CREATE_ASSERTION

The following bits specify the supported constraint attribute if the ability to explicitly specify constraint attributes is supported:

SQL_CA_CONSTRAINT_INITIALLY_DEFERRED
SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE
SQL_CA_CONSTRAINT_DEFERRABLE
SQL_CA_CONSTRAINT_NON_DEFERRABLE

DB_CREATE_CHARACTER_SET

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses in a *CREATE CHARACTER SET* statement which are supported by the Datasource. The following bitmask identifiers are used:

SQL_CCS_CREATE_CHARACTER_SET
SQL_CCS_COLLATE_CLAUSE
SQL_CCS_LIMITED_COLLATION

DB_CREATE_COLLATION

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses in a *CREATE COLLATION* statement which are supported by the Datasource. The following bitmask identifier is used:

SQL_CCOL_CREATE_COLLATION

DB_CREATE_DOMAIN

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses in a *CREATE DOMAIN* statement which are supported by the Datasource. The following bitmask identifiers are used:

SQL_CDO_CREATE_DOMAIN (The *CREATE DOMAIN* statement is supported.)

SQL_CDO_CONSTRAINT_NAME_DEFINITION (<constraint name definition> is supported for naming domain constraints).

The following bits specify the ability to create column constraints:

SQL_CDO_DEFAULT (Specifying domain constraints is supported)

SQL_CDO_CONSTRAINT (Specifying domain defaults is supported)

SQL_CDO_COLLATION (Specifying domain collation is supported)

The following bits specify the supported constraint attributes if the specifying of domain constraints is supported:

SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED
SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE
SQL_CDO_CONSTRAINT_DEFERRABLE
SQL_CDO_CONSTRAINT_NON_DEFERRABLE

A return value of zero (0) means that the *CREATE DOMAIN* statement is not supported.

DB_CREATE_SCHEMA

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses in a *CREATE SCHEMA* statement which are supported by the Datasource. The following bitmask identifiers are used:

SQL_CS_CREATE_SCHEMA
SQL_CS_AUTHORIZATION
SQL_CS_DEFAULT_CHARACTER_SET

DB_CREATE_TABLE

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses in a [CREATE TABLE](#) statement which are supported by the Datasource. The following bitmask identifiers are used:

SQL_CT_CREATE_TABLE (The [CREATE TABLE](#) statement is supported)

SQL_CT_TABLE_CONSTRAINT (Specifying table constraints is supported)

SQL_CT_CONSTRAINT_NAME_DEFINITION (The <constraint name definition> clause is supported for naming column and table constraints)

The following bits specify the ability to create temporary tables:

SQL_CT_COMMIT_PRESERVE (Deleted rows are preserved on commit.)

SQL_CT_COMMIT_DELETE (Deleted rows are deleted on commit.)

SQL_CT_GLOBAL_TEMPORARY (Global temporary tables can be created.)

SQL_CT_LOCAL_TEMPORARY (Local temporary tables can be created.)

The following bits specify the ability to create column constraints:

SQL_CT_COLUMN_CONSTRAINT (Specifying column constraints is supported.)

SQL_CT_COLUMN_DEFAULT (Specifying column defaults is supported.)

SQL_CT_COLUMN_COLLATION (Specifying column collation is supported.)

The following bits specify the supported constraint attributes if specifying column or table constraints is supported:

SQL_CT_CONSTRAINT_INITIALLY_DEFERRED
SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE
SQL_CT_CONSTRAINT_DEFERRABLE
SQL_CT_CONSTRAINT_NON_DEFERRABLE

DB_CREATE_TRANSLATION

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses in a [CREATE TRANSLATION](#) statement which are supported by the Datasource. The following bitmask identifier is used:

SQL_CTR_CREATE_TRANSLATION

DB_CREATE_VIEW

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses in a

`CREATE VIEW` statement which are supported by the Datasource. The following bitmask identifiers are used:

SQL_CV_CREATE_VIEW
SQL_CV_CHECK_OPTION
SQL_CV_CASCADED
SQL_CV_LOCAL

A return value of zero (0) means that the `CREATE VIEW` statement is not supported.

DB_CURSOR_COMMIT_BEHAVIOR

A numeric value that indicates how a `TRANS_COMMIT` operation affects cursors and prepared statements in the Datasource:

SQL_CB_DELETE (Close cursors and delete prepared statements. To use the cursor again, your program must re-prepare and re-execute the statement.)

SQL_CB_CLOSE (Close cursors. Your program can use `SQL_Stmt (SQL_STMT_EXECUTE)` on a prepared statement without using `SQL_Stmt (SQL_STMT_PREPARE)` again.

SQL_CB_PRESERVE (Preserve cursors in the same position as before the `TRANS_COMMIT` operation. Your program can continue to fetch data or it can close the statement and re-execute it without re-preparing it.)

DB_CURSOR_ROLLBACK_BEHAVIOR

A numeric value that indicates how a `TRANS_ROLLBACK` operation affects cursors and prepared statements in the Datasource:

SQL_CB_DELETE (Close cursors and delete prepared statements. To use the cursor again, your program must re-prepare and re-execute the statement.)

SQL_CB_CLOSE (Close cursors. Your program can use `SQL_Stmt (SQL_STMT_EXECUTE)` on a prepared statement without using `SQL_Stmt (SQL_STMT_PREPARE)` again.

SQL_CB_PRESERVE (Preserve cursors in the same position as before the `TRANS_ROLLBACK` operation. Your program can continue to fetch data or it can close the cursor and re-execute the statement without re-preparing it.)

DB_CURSOR_SENSITIVITY

A numeric value that indicates the database's support for cursor sensitivity:

SQL_INSENSITIVE (All cursors on a statement show the result set without reflecting any changes made to it by any other cursor within the same transaction.)

SQL_UNSPECIFIED (It is not specified whether or not cursors make visible the changes that are made to a result set by another cursor within the same

transaction. Cursors on the statement may make visible none, some, or all such changes.)

`SQL_SENSITIVE` (Cursors are sensitive to changes made by other cursors within the same transaction.)

`DB_DATETIME_LITERALS`

ODBC 3.x+ ONLY: A bitmasked value that describes the SQL92 datetime literals that are supported by the Datasource.

Note that these are the datetime literals listed in the SQL92 specification and are separate from the datetime literal escape clauses defined by ODBC. A bit value of zero (0) means that SQL92 datetime literals are not supported. The following bitmask identifiers are used

`SQL_DL_SQL92_DATE`
`SQL_DL_SQL92_TIME`
`SQL_DL_SQL92_TIMESTAMP`
`SQL_DL_SQL92_INTERVAL_YEAR`
`SQL_DL_SQL92_INTERVAL_MONTH`
`SQL_DL_SQL92_INTERVAL_DAY`
`SQL_DL_SQL92_INTERVAL_HOUR`
`SQL_DL_SQL92_INTERVAL_MINUTE`
`SQL_DL_SQL92_INTERVAL_SECOND`
`SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH`
`SQL_DL_SQL92_INTERVAL_DAY_TO_HOUR`
`SQL_DL_SQL92_INTERVAL_DAY_TO_MINUTE`
`SQL_DL_SQL92_INTERVAL_DAY_TO_SECOND`
`SQL_DL_SQL92_INTERVAL_HOUR_TO_MINUTE`
`SQL_DL_SQL92_INTERVAL_HOUR_TO_SECOND`
`SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND`

`DB_DDL_INDEX`

ODBC 3.x+ ONLY: A numeric value that indicates support for creation and dropping of indexes. This function will return either `SQL_DI_CREATE_INDEX` or `SQL_DI_DROP_INDEX`.

`DB_DEFAULT_TXN_ISOLATION`

A numeric value that indicates the default transaction isolation level that is supported by the driver or Datasource, or zero if the Datasource does not support transactions.

The following terms are used to define transaction isolation levels:

Dirty Read: Transaction 1 changes a row. Transaction 2 reads the changed row before transaction 1 commits the change. If transaction 1 rolls back the change, transaction 2 will have read a row that is considered to have never existed.

Non-repeatable Read: Transaction 1 reads a row. Transaction 2 updates or deletes that row and commits this change. If transaction 1 attempts to reread the row, it will receive different row values or discover that the row has been deleted.

Phantom: Transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 generates one or more rows (either through inserts or updates) that match the search criteria. If transaction 1 re-executes the statement that reads the rows, it receives a different set of rows.

If a Datasource supports transactions, the ODBC driver will return one of the following values:

SQL_TXN_READ_UNCOMMITTED (Dirty reads, non-repeatable reads, and phantoms are possible.)

SQL_TXN_READ_COMMITTED (Dirty reads are not possible. Non-repeatable reads and phantoms are possible.)

SQL_TXN_REPEATABLE_READ (Dirty reads and non-repeatable reads are not possible. Phantoms are possible.)

SQL_TXN_SERIALIZABLE (Transactions are serializable. Serializable transactions do not allow dirty reads, non-repeatable reads, or phantoms.)

DB_DRIVER_HLIB

The *hInstance* value that was returned to the Driver Manager when it loaded the driver DLL. The handle is only valid for the current database.

DB_DROP...

ODBC 3.x+ ONLY: The following DB_DROP_ constants return bitmasked values that can be used (with the corresponding SQL_Dx_DROP constants) to determine which clauses are supported by the various *DROP* statements:

DB_DROP_ASSERTION (with SQL_DA_DROP_ASSERTION)

DB_DROP_CHARACTER_SET (with SQL_DCS_DROP_CHARACTER_SET)

DB_DROP_COLLATION (with SQL_DC_DROP_COLLATION)

DB_DROP_DOMAIN (with SQL_DD_DROP_DOMAIN, SQL_DD_CASCADE, and SQL_DD_RESTRICT)

DB_DROP_SCHEMA (with SQL_DS_DROP_SCHEMA, SQL_DS_CASCADE, and SQL_DS_RESTRICT)

DB_DROP_TABLE (with SQL_DT_DROP_TABLE, SQL_DT_CASCADE, and SQL_DT_RESTRICT)

DB_DROP_TRANSLATION (SQL_DTR_DROP_TRANSLATION)

DB_DROP_VIEW (with SQL_DV_DROP_VIEW, SQL_DV_CASCADE, and SQL_DV_RESTRICT)

DB_DSN_FILENAME

The name of the DSN File (if any) that was used to open the database.

DB_DYNAMIC_CURSOR_ATTRIBUTES1

ODBC 3.x+ ONLY: A bitmasked value that describes the attributes of a dynamic cursor that are supported by the driver. This bitmasked value contains only the *first* subset of attributes. For the second subset, see DB_DYNAMIC_CURSOR_ATTRIBUTES2 below.

NOTE: This list of constants is used for DYNAMIC, STATIC, FORWARD-ONLY and KEYSET-DRIVEN cursors. Where the word dynamic is used below, you may need to substitute the word static, forward-only, or keyset-driven.

The following bitmask identifiers are used:

SQL_CA1_NEXT (NEXT_ROW is supported in a call to SQL_Fetch when the cursor is a dynamic cursor.)

SQL_CA1_ABSOLUTE (FIRST_ROW, LAST_ROW, and absolute row numbers are supported in a call to SQL_Fetch when the cursor is a dynamic cursor. (Note that in all cases, the row that will be fetched is independent of the current cursor position.)

SQL_CA1_RELATIVE (The SQL_FetchRel function is supported when used for simple relative fetches.)

SQL_CA1_BOOKMARK (The SQL_FetchRel function is supported when used with bookmarks.)

SQL_CA1_LOCK_EXCLUSIVE (A *ILockType*& value of LOCK_EXCLUSIVE is supported in a call to SQL_SetPos when the cursor is a dynamic cursor.)

SQL_CA1_LOCK_NO_CHANGE (A *ILockType*& value of LOCK_NO_CHANGE is supported in a call to SQL_SetPos when the cursor is a dynamic cursor.)

SQL_CA1_LOCK_UNLOCK (A *ILockType*& value of LOCK_UNLOCK is supported in a call to SQL_SetPos when the cursor is a dynamic cursor.)

SQL_CA1_POS_POSITION (An *IOperation*& value of SET_POSITION is supported in a call to SQL_SetPos when the cursor is a dynamic cursor.)

SQL_CA1_POS_UPDATE (An *IOperation*& value of SET_UPDATE is supported in a call to SQL_SetPos when the cursor is a dynamic cursor.)

SQL_CA1_POS_DELETE (An *IOperation*& value of SET_DELETE is supported in a call to SQL_SetPos when the cursor is a dynamic cursor.)

SQL_CA1_POS_REFRESH (An *IOperation*& value of SET_REFRESH is supported in a call to SQL_SetPos when the cursor is a dynamic cursor.)

SQL_CA1_POSITIONED_UPDATE (An "*UPDATE WHERE CURRENT OF*" SQL

statement is supported when the cursor is a dynamic cursor.)

SQL_CA1_POSITIONED_DELETE (A "*DELETE WHERE CURRENT OF*" SQL statement is supported when the cursor is a dynamic cursor.)

SQL_CA1_SELECT_FOR_UPDATE (A "*SELECT FOR UPDATE*" SQL statement is supported when the cursor is a dynamic cursor.)

SQL_CA1_BULK_ADD (An *IOperation*& value of BULK_ADD is supported in a call to SQL_BulkOp when the cursor is a dynamic cursor.)

SQL_CA1_BULK_UPDATE_BY_BOOKMARK (An *IOperation*& value of BULK_UPDATE is supported in a call to SQL_BulkOp when the cursor is a dynamic cursor.)

SQL_CA1_BULK_DELETE_BY_BOOKMARK (An *IOperation*& value of BULK_DELETE is supported in a call to SQL_BulkOp when the cursor is a dynamic cursor.)

SQL_CA1_BULK_FETCH_BY_BOOKMARK (An *IOperation*& value of BULK_FETCH is supported in a call to SQL_BulkOp when the cursor is a dynamic cursor.)

DB_DYNAMIC_CURSOR_ATTRIBUTES2

ODBC 3.x+ ONLY: A bitmasked value that describes the attributes of a dynamic cursor that are supported by the driver. This bitmasked value contains only the *second* subset of attributes. For the first subset, see DB_DYNAMIC_CURSOR_ATTRIBUTES1 above.

NOTE: This list of constants is used for DYNAMIC, STATIC, FORWARD-ONLY and KEYSSET-DRIVEN cursors. Where the word dynamic is used below, you may need to substitute the word static, forward-only, or keyset-driven.

The following bitmask identifiers are used:

SQL_CA2_READ_ONLY_CONCURRENCY (A read-only dynamic cursor, in which no updates are allowed, is supported.

SQL_CA2_LOCK_CONCURRENCY (A dynamic cursor that uses the lowest level of locking sufficient to ensure that the row can be updated is supported. These locks must be consistent with the transaction isolation level set by the value that is returned by the SQL_DBAtribUInt(DB_ATTR_TXN_ISOLATION) function.)

SQL_CA2_OPT_ROWVER_CONCURRENCY (A dynamic cursor that uses the optimistic concurrency control comparing row versions is supported.

SQL_CA2_OPT_VALUES_CONCURRENCY (A dynamic cursor that uses the optimistic concurrency control comparing values is supported.

SQL_CA2_SENSITIVITY_ADDITIONS (Added rows are visible to a dynamic cursor; the cursor can scroll to those rows. Where these rows are added to the cursor is driver-dependent.

SQL_CA2_SENSITIVITY_DELETIONS (Deleted rows are no longer available to a dynamic cursor, and do not leave a "hole" in the result set. After the dynamic cursor scrolls from a deleted row, it cannot return to that row.)

SQL_CA2_SENSITIVITY_UPDATES (Updates to rows are visible to a dynamic cursor. If a dynamic cursor scrolls from and returns to an updated row, the data returned by the cursor is the updated data, not the original data.)

SQL_CA2_MAX_ROWS_SELECT (The SQL Tools function SQL_StmtMode(STMT_ATTR_MAX_RESULT_ROWS) affects *SELECT* statements when the cursor is a dynamic cursor.)

SQL_CA2_MAX_ROWS_INSERT (The SQL Tools function SQL_StmtMode(STMT_ATTR_MAX_RESULT_ROWS) affects *INSERT* statements when the cursor is a dynamic cursor.

SQL_CA2_MAX_ROWS_DELETE (The SQL Tools function SQL_StmtMode(STMT_ATTR_MAX_RESULT_ROWS) affects *DELETE* statements when the cursor is a dynamic cursor.

SQL_CA2_MAX_ROWS_UPDATE (The SQL Tools function SQL_StmtMode(STMT_ATTR_MAX_RESULT_ROWS) affects *UPDATE* statements when the cursor is a dynamic cursor.

SQL_CA2_MAX_ROWS_CATALOG (The SQL Tools function SQL_StmtMode(STMT_ATTR_MAX_RESULT_ROWS) affects Info ("catalog") functions when the cursor is a dynamic cursor.

SQL_CA2_MAX_ROWS_AFFECTS_ALL (The SQL Tools function SQL_StmtMode(STMT_ATTR_MAX_RESULT_ROWS) affects *SELECT*, *INSERT*, *DELETE*, and *UPDATE* statements, and Info functions, when the cursor is a dynamic cursor.

SQL_CA2_CRC_EXACT (The exact row count is available in the DIAG_CURSOR_ROW_COUNT diagnostic field (from the SQL_Diagnostic function) when the cursor is a dynamic cursor.)

SQL_CA2_CRC_APPROXIMATE (An approximate row count is available in the DIAG_CURSOR_ROW_COUNT diagnostic field (from the SQL_Diagnostic function) when the cursor is a dynamic cursor.)

SQL_CA2_SIMULATE_NON_UNIQUE (The ODBC driver does not guarantee that simulated positioned update or delete statements will affect only one row when the cursor is a dynamic cursor. It is your program's responsibility to guarantee this. If a statement affects more than one row, SQL_Stmt(SQL_STMT_EXECUTE) or SQL_Stmt(SQL_STMT_IMMEDIATE) will return SQL State 01001 (Cursor operation conflict).

SQL_CA2_SIMULATE_TRY_UNIQUE (The ODBC driver attempts to guarantee that simulated positioned update or delete statements will affect only one row when the cursor is a dynamic cursor. The driver always

executes such statements, even if they might affect more than one row, such as when there is no unique key. If a statement affects more than one row, `SQL_Stmt (SQL_STMT_EXECUTE)` or `SQL_Stmt (SQL_STMT_IMMEDIATE)` will return SQL State 01001 (Cursor operation conflict).

`SQL_CA2_SIMULATE_UNIQUE` (The ODBC driver guarantees that simulated positioned update or delete statements will affect only one row when the cursor is a dynamic cursor. If the driver cannot guarantee this for a given statement, `SQL_Stmt (SQL_STMT_EXECUTE)` or `SQL_Stmt (SQL_STMT_PREPARE)` returns SQL State 01001 (Cursor operation conflict).

`DB_FETCH_DIRECTION`

This is a "deprecated" function in ODBC 3.x and should not be used.

`DB_FILE_USAGE`

A numeric value that indicates how a single-tier driver directly treats files in a Datasource:

`SQL_FILE_NOT_SUPPORTED` (The driver is not a single-tier driver. For example, the Oracle ODBC driver is a two-tier driver.)

`SQL_FILE_TABLE` (A single-tier driver treats files in a Datasource as tables. For example, an Xbase driver treats each Xbase file as a table.)

`SQL_FILE_CATALOG` (A single-tier driver treats files in a Datasource as a catalog. For example, a Microsoft Access driver treats each Microsoft Access file as a complete database.

Your program can use the `DB_FILE_USAGE` value to determine how users will select data. For example, Xbase users usually think of data as being stored in files, while Oracle and Access users generally think of data as being stored in tables. When a user selects an Xbase datasource, your program could display the Windows File-Open common dialog box. When the user selects an Oracle or Access datasource, your program could display a custom "Select Table" dialog box.

`DB_FORWARD_ONLY_CURSOR_ATTRIBUTES1` and `DB_FORWARD_ONLY_CURSOR_ATTRIBUTES2`

ODBC 3.x+ ONLY: These functions are virtually identical to the `DB_DYNAMIC_CURSOR_ATTRIBUTES1` and `2` functions that are described above. For complete information, read the descriptions of `DB_DYNAMIC_CURSOR_ATTRIBUTES1` and `2` above and substitute "forward-only" wherever it says dynamic.

`DB_GETDATA_EXTENSIONS`

A bitmasked value that describes restrictions on the `SQL_LongResCol` function. The following bitmask identifiers are used

`SQL_GD_ANY_COLUMN` (`SQL_LongResCol` can be used with any unbound column, including those before the last bound column. Note that the columns

must be accessed in order of ascending column number unless SQL_GD_ANY_ORDER is also returned.)

SQL_GD_ANY_ORDER (SQL_LongResCol can be used with unbound columns in any order. Note that SQL_LongResCol can only be used for columns after the last bound column unless SQL_GD_ANY_COLUMN is also returned.)

SQL_GD_BLOCK (SQL_LongResCol can be used for an unbound column in any row in a block (where the rowset size is greater than 1) of data after positioning to that row with SQL_SetPos.)

SQL_GD_BOUND (SQL_LongResCol can be used for bound columns as well as unbound columns. A driver cannot return this value unless it also returns SQL_GD_ANY_COLUMN. SQL_LongResCol is only required to return data from unbound columns that **1)** occur after the last bound column, **2)** are called in order of increasing column number, and **3)** are not in a row in a MultiRow cursor.

If a driver supports bookmarks (either fixed- or variable-length), it must support using SQL_LongResCol for Column Zero. This support is required regardless of what the driver returns for SQL_DBInfoUInt (DB_GETDATA_EXTENSIONS).

DB_GROUP_BY

A numeric value that describes the relationship between the columns in a *GROUP BY* clause and the non-aggregated columns in the select list:

SQL_GB_COLLATE (A *COLLATE* clause can be specified at the end of each grouping column.)

SQL_GB_NOT_SUPPORTED (*GROUP BY* clauses are not supported.)

SQL_GB_GROUP_BY_EQUALS_SELECT (The *GROUP BY* clause must contain all of the non-aggregated columns in the select list. It *cannot* contain any other columns. For example, *SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT.*)

SQL_GB_GROUP_BY_CONTAINS_SELECT (The *GROUP BY* clause must contain all of the non-aggregated columns in the select list. It *can* contain columns that are not in the select list. For example, *SELECT DEPT, MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT, AGE.*)

SQL_GB_NO_RELATION (The columns in the *GROUP BY* clause and the columns in the select list are not related. The meaning of non-grouped, non-aggregated columns in the select list is Datasource-dependent. For example, *SELECT DEPT, SALARY FROM EMPLOYEE GROUP BY DEPT, AGE.*)

DB_IDENTIFIER_CASE

A numeric value that describes how identifiers (table names, column names, etc.) are used:

SQL_IC_UPPER (Identifiers are not case-sensitive and are stored in upper case in the system catalog.)

SQL_IC_LOWER (Identifiers are not case-sensitive and are stored in lower case in the system catalog.)

SQL_IC_SENSITIVE (Identifiers are case-sensitive and are stored in mixed case in the system catalog.)

SQL_IC_MIXED (Identifiers are not case-sensitive and are stored in mixed case in the system catalog.)

DB_INDEX_KEYWORDS

ODBC 3.x+ ONLY: A numeric value that describes keywords in a *CREATE INDEX* statement that are supported by the driver:

SQL_IK_NONE (None of the keywords are supported.)

SQL_IK_ASC (*ASC* keyword (Ascending) is supported.)

SQL_IK_DESC (*DESC* keyword (Descending) is supported.)

SQL_IK_ALL (All keywords are supported.)

DB_INFO_DRIVER_START

This is a "deprecated" function in ODBC 3.x and should not be used.

DB_INFO_SCHEMA_VIEWS

ODBC 3.x+ ONLY: A bitmasked value that describes the views in the INFORMATION_SCHEMA (as defined by SQL92) that are supported by the ODBC driver. The following bitmask identifiers are used:

SQL_ISV_ASSERTIONS (Identifies the catalog's assertions that are owned by a given user.)

SQL_ISV_CHARACTER_SETS (Identifies the catalog's character sets that are accessible to a given user.)

SQL_ISV_CHECK_CONSTRAINTS (Identifies the CHECK constraints that are owned by a given user.)

SQL_ISV_COLLATIONS (Identifies the character collations for the catalog that are accessible to a given user.)

SQL_ISV_COLUMN_DOMAIN_USAGE (Identifies columns for the catalog that are dependent on domains defined in the catalog and are owned by a given user.)

SQL_ISV_COLUMN_PRIVILEGES (Identifies the privileges on columns of persistent tables that are available to or granted by a given user.)

SQL_ISV_COLUMNS (Identifies the columns of persistent tables that are

accessible to a given user.)

`SQL_ISV_CONSTRAINT_COLUMN_USAGE` (Similar to `CONSTRAINT_TABLE_USAGE` view, columns are identified for the various constraints that are owned by a given user.)

`SQL_ISV_CONSTRAINT_TABLE_USAGE` (Identifies the tables that are used by constraints (referential, unique, and assertions), and are owned by a given user.)

`SQL_ISV_DOMAIN_CONSTRAINTS` (Identifies the domain constraints (of the domains in the catalog) that are accessible to a given user.)

`SQL_ISV_DOMAINS` (Identifies the domains defined in a catalog that are accessible to the user.)

`SQL_ISV_KEY_COLUMN_USAGE` (Identifies columns defined in the catalog that are constrained as keys by a given user.)

`SQL_ISV_REFERENTIAL_CONSTRAINTS` (Identifies the referential constraints that are owned by a given user.)

`SQL_ISV_SCHEMATA` (Identifies the schemas that are owned by a given user.)

`SQL_ISV_SQL_LANGUAGES` (Identifies the SQL conformance levels, options, and dialects supported by the SQL implementation.)

`SQL_ISV_TABLE_CONSTRAINTS` (Identifies the table constraints that are owned by a given user.)

`SQL_ISV_TABLE_PRIVILEGES` (Identifies the privileges on persistent tables that are available to or granted by a given user.)

`SQL_ISV_TABLES` (Identifies the persistent tables defined in a catalog that are accessible to a given user.)

`SQL_ISV_TRANSLATIONS` (Identifies character translations for the catalog that are accessible to a given user.)

`SQL_ISV_USAGE_PRIVILEGES` (Identifies the `USAGE` privileges on catalog objects that are available to or owned by a given user.)

`SQL_ISV_VIEW_COLUMN_USAGE` (Identifies the columns on which the catalog's views that are owned by a given user are dependent.)

`SQL_ISV_VIEW_TABLE_USAGE` (Identifies the tables on which the catalog's views that are owned by a given user are dependent.)

`SQL_ISV_VIEWS` (Identifies the viewed tables defined in this catalog that are accessible to a given user.)

`DB_INFO_VOLUME`

This function tells you how many bytes of data SQL Tools is currently storing

in its Info cache (i.e. in your computer's memory). This value can be useful if your program needs to determine whether or not Info was added to the SQL Tools cache while the program was running, so that it can decide whether or not it needs to re-save a DBI file with `SQL_InfoExport`.

`DB_INSERT_STATEMENT`

ODBC 3.x+ ONLY: A bitmasked value that describes support for *INSERT* statements:

`SQL_IS_INSERT_LITERALS`
`SQL_IS_INSERT_SEARCHED`
`SQL_IS_SELECT_INTO`

`DB_KEYSET_CURSOR_ATTRIBUTES1` and
`DB_KEYSET_CURSOR_ATTRIBUTES2`

ODBC 3.x+ ONLY: These functions are virtually identical to the `DB_DYNAMIC_CURSOR_ATTRIBUTES1` and `2` functions that are described above. For complete information, read the descriptions of `DB_DYNAMIC_CURSOR_ATTRIBUTES1` and `2` above and substitute "keyset-driven" wherever it says dynamic.

`DB_LOCK_TYPES`

This is a "deprecated" function in ODBC 3.x and should not be used.

`DB_MAX_ASYNC_CONCURRENT_STATEMENTS`

ODBC-based Asynchronous Execution is not supported by SQL Tools, so this value is not useful. (SQL Tools *does* support thread-based asynchronous execution of SQL statements, but this value does not apply to that technique.)

`DB_MAX_BINARY_LITERAL_LEN`

A numeric value that specifies the maximum length (in bytes, excluding the literal prefix and suffix) of a binary literal value in a SQL statement. For example, assuming that the standard binary prefix "0x" (zero-ex) is used, the binary literal value `0xABCD` has a length of 4. If there is no maximum length or the length is unknown, this function returns zero.

`DB_MAX_CATALOG_NAME_LEN`

ODBC 3.x+ ONLY: A numeric value that specifies the maximum length of a catalog name. If there is no maximum length or the length is unknown, this function returns zero. (The ODBC 2.0 name for this function was `DB_MAX_QUALIFIER_NAME_LEN`.)

`DB_MAX_CHAR_LITERAL_LEN`

A numeric value that specifies the maximum length (in bytes, excluding the literal prefix and suffix) of a character (string) literal in a SQL statement. If there is no maximum length or the length is unknown, this function returns zero.

DB_MAX_COLUMN_NAME_LEN

A numeric value that specifies the maximum length of a column name. If there is no maximum length or the length is unknown, this function returns zero.

DB_MAX_COLUMNS_IN_GROUP_BY

A numeric value that specifies the maximum number of columns that are allowed in a *GROUP BY* clause. If there is no specified limit or the limit is unknown, this function returns zero.

DB_MAX_COLUMNS_IN_INDEX

A numeric value that specifies the maximum number of columns that are allowed in an index. If there is no specified limit or the limit is unknown, this function returns zero.

DB_MAX_COLUMNS_IN_ORDER_BY

A numeric value that specifies the maximum number of columns that are allowed in an *ORDER BY* clause. If there is no specified limit or the limit is unknown, this function returns zero.

DB_MAX_COLUMNS_IN_SELECT

A numeric value that specifies the maximum number of columns that are allowed in a *SELECT* list. If there is no specified limit or the limit is unknown, this function returns zero.

DB_MAX_COLUMNS_IN_TABLE

A numeric value that specifies the maximum number of columns that a table can contain. If there is no specified limit or the limit is unknown, this function will return zero.

DB_MAX_CONCURRENT_ACTIVITIES

A numeric value that specifies the maximum number of active ("concurrent") statements that the driver can support for a database connection. A statement is defined as active if it has results pending, with "results" defined as **1**) rows from a *SELECT* operation, **2**) rows affected by an *INSERT*, *UPDATE*, or *DELETE* operation (such as a row count), or **3**) if the statement is in a *SQL_NEED_DATA* state. This value can reflect a limitation imposed by either the driver or the Datasource. If there is no specified limit or the limit is unknown, this function will return zero. (The ODBC 2.0 name for this function was *DB_ACTIVE_STATEMENTS*.)

DB_MAX_CURSOR_NAME_LEN

A numeric value that specifies the maximum length of a cursor name. If there is no maximum length or the length is unknown, this function returns zero.
IMPORTANT NOTE: Many ODBC drivers limit this value to 18, and interoperable applications should always use names that are less than 19

characters long. For this reason, SQL Tools limits all cursor names to 18 characters.

DB_MAX_DRIVER_CONNECTIONS

A numeric value that specifies the maximum number of open databases that the driver can support in one program. This value can reflect a limitation imposed by either the driver or the Datasource. If there is no specified limit or the limit is unknown, this function returns zero. (The ODBC 2.0 name for this function was DB_ACTIVE_CONNECTIONS.)

DB_MAX_IDENTIFIER_LEN

A numeric value that specifies the maximum number of characters that can be used for user-defined names, like table names and column names.

DB_MAX_INDEX_SIZE

A numeric value that specifies the maximum number of bytes that are allowed in the combined fields of an index. If there is no specified limit or the limit is unknown, this function returns zero.

DB_MAX_PROCEDURE_NAME_LEN

A numeric value that specifies the maximum length of a stored procedure name. If there is no maximum length or the length is unknown, this function returns zero.

DB_MAX_ROW_SIZE

A numeric value that specifies the maximum length of a single row in a table. If there is no specified limit or the limit is unknown, this function returns zero.

DB_MAX_SCHEMA_NAME_LEN

A numeric value that specifies the maximum length of a schema name. If there is no maximum length or the length is unknown, this function returns zero. (The ODBC 2.0 name for this function was DB_MAX_OWNER_NAME_LEN)

DB_MAX_STATEMENT_LEN

A numeric value that specifies the maximum length (number of characters, including all spaces) of a SQL statement. If there is no maximum length or the length is unknown, this function returns zero.

DB_MAX_TABLE_NAME_LEN

A numeric value that specifies the maximum length of a table name. If there is no maximum length or the length is unknown, this function returns zero.

DB_MAX_TABLES_IN_SELECT

A numeric value that specifies the maximum number of tables that are allowed in a *FROM* clause of a *SELECT* statement. If there is no specified limit or the limit is unknown, this function returns zero.

DB_MAX_USER_NAME_LEN

A numeric value that specifies the maximum length of a user name. If there is no maximum length or the length is unknown, this function returns zero.

DB_NON_NULLABLE_COLUMNS

A numeric value that specified whether or not the Datasource supports *NOT NULL* in column definitions:

SQL_NNC_NON_NULL (Columns cannot be nullable. The Datasource supports the *NOT NULL* column constraint in *CREATE TABLE* statements).

SQL_NNC_NULL (All columns must be nullable.)

DB_NULL_COLLATION

A numeric value that specifies where Null values are sorted in a result set:

SQL_NC_END (Null values are sorted at the end of the result set, *regardless* of the *ASC* or *DESC* keywords.)

SQL_NC_HIGH (Null values are sorted at the high end of the result set, depending on the *ASC* or *DESC* keywords.)

SQL_NC_LOW (Null values are sorted at the low end of the result set, depending on the *ASC* or *DESC* keywords.)

SQL_NC_START (Null values are sorted at the start of the result set, *regardless* of the *ASC* or *DESC* keywords.)

DB_NUMERIC_FUNCTIONS

A bitmasked value that describes the scalar numeric functions that are supported by the driver and associated Datasource. The following bitmask identifiers are used:

SQL_FN_NUM_ABS
SQL_FN_NUM_ACOS
SQL_FN_NUM_ASIN
SQL_FN_NUM_ATAN
SQL_FN_NUM_ATAN2
SQL_FN_NUM_CEILING
SQL_FN_NUM_COS
SQL_FN_NUM_COT
SQL_FN_NUM_DEGREES
SQL_FN_NUM_EXP
SQL_FN_NUM_FLOOR
SQL_FN_NUM_LOG
SQL_FN_NUM_LOG10
SQL_FN_NUM_MOD
SQL_FN_NUM_PI
SQL_FN_NUM_POWER
SQL_FN_NUM_RADIANS

SQL_FN_NUM_RAND
SQL_FN_NUM_ROUND
SQL_FN_NUM_SIGN
SQL_FN_NUM_SIN
SQL_FN_NUM_SQRT
SQL_FN_NUM_TAN
SQL_FN_NUM_TRUNCATE

DB_ODBC_API_CONFORMANCE

This is a "deprecated" function in ODBC 3.x and should not be used.

DB_ODBC_INTERFACE_CONFORMANCE

ODBC 3.x+ ONLY: A numeric value that specifies the level of the ODBC 3.x interface to which the driver conforms.

SQL_OIC_CORE (The minimum conformance level to which all ODBC drivers are expected to conform. This level includes basic interface elements such as connection functions; functions for preparing and executing a SQL statement; basic result set metadata functions; basic catalog functions; and so on.)

SQL_OIC_LEVEL1 (A conformance level that includes the core functionality, plus scrollable cursors, bookmarks, positioned updates and deletes, and so on.)

SQL_OIC_LEVEL2 (A conformance level that includes all level 1 functionality, plus advanced features such as sensitive cursors; update, delete, and refresh by bookmarks; stored procedure support; catalog functions for primary and foreign keys; multi-catalog support; and so on.)

DB_ODBC_SQL_CONFORMANCE

This is a "deprecated" function in ODBC 3.x and should not be used.

DB_STANDARD_CLI_CONFORMANCE

This function is listed by, but not documented in, the Microsoft ODBC Software Developer Kit.

DB_OJ_CAPABILITIES

A bitmasked value that describes the types of outer joins that are supported by the driver and the Datasource. The following bitmask identifiers are used:

SQL_OJ_LEFT (Left outer joins are supported.)

SQL_OJ_RIGHT (Right outer joins are supported.)

SQL_OJ_FULL (Full outer joins are supported.)

SQL_OJ_NESTED (Nested outer joins are supported.)

SQL_OJ_NOT_ORDERED (The column names in an [ON](#) clause of an outer join

do not have to be in the same order as their respective table names in the *OUTER JOIN* clause.)

SQL_OJ_INNER (The inner table -- i.e. the right table in a left outer join or the left table in a right outer join -- can also be used in an inner join. This value does not apply to full outer joins, which do not have an inner table.)

SQL_OJ_ALL_COMPARISON_OPS (The comparison operator in an *ON* clause can be any of the ODBC comparison operators. If this bit is *not* set, only the equal sign (=) operator can be used in outer joins.

If none of these options are supported, no outer join clause is supported.

DB_PARAM_ARRAY_ROW_COUNTS

ODBC 3.x+ ONLY: A numeric value that specifies the driver's properties regarding the availability of row counts in a parameterized execution. This function always returns one of the following values:

SQL_PARC_BATCH (Individual row counts are available for each set of parameters. This is conceptually equivalent to the ODBC driver generating a batch of SQL statements, one for each parameter set in the array.)

SQL_PARC_NO_BATCH (There is only one row count available, which is the cumulative row count resulting from the execution of the statement for the entire array of parameters. This is conceptually equivalent to treating the statement along with the entire parameter array as one unit. Errors are handled as if one statement were executed.)

DB_PARAM_ARRAY_SELECTS

ODBC 3.x+ ONLY: A numeric value that specifies the driver's properties regarding the availability of result sets in a parameterized execution. This function always returns one of the following three values:

SQL_PAS_BATCH (There is one result set available per set of parameters. This is conceptually equivalent to the ODBC driver generating a batch of SQL statements, one for each parameter set in the array.)

SQL_PAS_NO_BATCH (There is only one result set available, which represents the cumulative result set resulting from the execution of the statement for the entire array of parameters. This is conceptually equivalent to treating the statement along with the entire parameter array as one unit.)

SQL_PAS_NO_SELECT (The driver does not allow a statement which generates a result set to be executed with an array of parameters.)

DB_POSITIONED_STATEMENTS

This is a "deprecated" function in ODBC 3.x and should not be used.

DB_POS_OPERATIONS

This is a "deprecated" function in ODBC 3.x and should not be used.

DB_QUOTED_IDENTIFIER_CASE

A numeric value that specifies how quoted identifiers are handled:

SQL_IC_UPPER (Quoted identifiers are not case-sensitive and are stored in uppercase in the system catalog.)

SQL_IC_LOWER (Quoted identifiers are not case-sensitive and are stored in lowercase in the system catalog.)

SQL_IC_SENSITIVE (Quoted identifiers are case-sensitive and are stored in mixed case in the system catalog. Note that in a SQL92-compliant database, quoted identifiers are *always* case-sensitive.)

SQL_IC_MIXED (Quoted identifiers are not case-sensitive and are stored in mixed case in the system catalog.)

DB_SCHEMA_USAGE

A bitmasked value that describes the statements in which schemas can be used:

SQL_SU_DML_STATEMENTS (Schemas are supported in [SELECT](#), [INSERT](#), [UPDATE](#), [DELETE](#), and, if they are supported, [SELECT FOR UPDATE](#) and positioned update and delete statements.)

SQL_SU_PROCEDURE_INVOCATION (Schemas are supported in the ODBC procedure invocation statement [call](#).)

SQL_SU_TABLE_DEFINITION (Schemas are supported in [CREATE TABLE](#), [CREATE VIEW](#), [ALTER TABLE](#), [DROP TABLE](#), and [DROP VIEW](#) statements)

SQL_SU_INDEX_DEFINITION (Schemas are supported in [CREATE INDEX](#) and [DROP INDEX](#) statements)

SQL_SU_PRIVILEGE_DEFINITION (Schemas are supported in [GRANT](#) and [REVOKE](#) statements)

(The ODBC 2.0 name for this function was DB_OWNER_USAGE.)

DB_SCROLL_CONCURRENCY

This is a "deprecated" function in ODBC 3.x and should not be used.

DB_SCROLL_OPTIONS

A bitmasked value that describes the scroll options that are supported for scrollable cursors. The following bitmask identifiers are used:

SQL_SO_FORWARD_ONLY (The cursor can only scroll forward.)

SQL_SO_STATIC (The data in the result set is static.)

SQL_SO_KEYSET_DRIVEN (The driver saves and uses the keys for every row in the result set.)

SQL_SO_DYNAMIC (The driver keeps the keys for every row in the rowset. The keyset size is the same as the rowset size).

SQL_SO_MIXED (The driver keeps the keys for every row in the keyset, and the keyset size is greater than the rowset size. The cursor is keyset-driven inside the keyset and dynamic outside the keyset).

DB_SQL_CONFORMANCE

A numeric value that indicates the level of SQL92 that is supported by the driver:

SQL_SC_SQL92_ENTRY (Entry level SQL92 compliant)

SQL_SC_FIPS127_2_TRANSITIONAL (FIPS 127-2 transitional level compliant)

SQL_SC_SQL92_FULL (Full level SQL92 compliant)

SQL_SC_SQL92_INTERMEDIATE (Intermediate level SQL92 compliant)

DB_SQL92_DATETIME_FUNCTIONS

ODBC 3.x+ ONLY: A bitmasked value that describes the datetime scalar functions that are supported by the driver and the Datasource. The following bitmask identifiers are used:

SQL_SDF_CURRENT_DATE

SQL_SDF_CURRENT_TIME

SQL_SDF_CURRENT_TIMESTAMP

DB_SQL92_FOREIGN_KEY_DELETE_RULE

ODBC 3.x+ ONLY: A bitmasked value that describes the rules that are supported for a foreign key in a [DELETE](#) statement. The following bitmask identifiers are used:

SQL_SFKD_CASCADE

SQL_SFKD_NO_ACTION

SQL_SFKD_SET_DEFAULT

SQL_SFKD_SET_NULL

DB_SQL92_FOREIGN_KEY_UPDATE_RULE

ODBC 3.x+ ONLY: A bitmasked value that described the rules that are supported for a foreign key in an [UPDATE](#) statement. The following bitmask identifiers are used:

SQL_SFKU_CASCADE

SQL_SFKU_NO_ACTION

SQL_SFKU_SET_DEFAULT

SQL_SFKU_SET_NULL

DB_SQL92_GRANT

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses that are supported in the [GRANT](#) statement. The following bitmask identifiers are used:

SQL_SG_DELETE_TABLE
SQL_SG_INSERT_COLUMN
SQL_SG_INSERT_TABLE
SQL_SG_REFERENCES_TABLE
SQL_SG_REFERENCES_COLUMN
SQL_SG_SELECT_TABLE
SQL_SG_UPDATE_COLUMN
SQL_SG_UPDATE_TABLE
SQL_SG_USAGE_ON_DOMAIN
SQL_SG_USAGE_ON_CHARACTER_SET
SQL_SG_USAGE_ON_COLLATION
SQL_SG_USAGE_ON_TRANSLATION
SQL_SG_WITH_GRANT_OPTION

DB_SQL92_NUMERIC_VALUE_FUNCTIONS

ODBC 3.x+ ONLY: A bitmasked value that describes the numeric scalar functions that are supported by the driver and the Datasource. The following bitmask identifiers are used

SQL_SNVF_BIT_LENGTH
SQL_SNVF_CHAR_LENGTH
SQL_SNVF_CHARACTER_LENGTH
SQL_SNVF_EXTRACT
SQL_SNVF_OCTET_LENGTH
SQL_SNVF_POSITION

DB_SQL92_PREDICATES

ODBC 3.x+ ONLY: A bitmasked value that describes the predicates that are supported in a [SELECT](#) statement. The following bitmask identifiers are used:

SQL_SP_BETWEEN
SQL_SP_COMPARISON
SQL_SP_EXISTS
SQL_SP_IN
SQL_SP_ISNOTNULL
SQL_SP_ISNULL
SQL_SP_LIKE
SQL_SP_MATCH_FULL
SQL_SP_MATCH_PARTIAL
SQL_SP_MATCH_UNIQUE_FULL
SQL_SP_MATCH_UNIQUE_PARTIAL
SQL_SP_OVERLAPS
SQL_SP_QUANTIFIED_COMPARISON
SQL_SP_UNIQUE

DB_SQL92_RELATIONAL_JOIN_OPERATORS

ODBC 3.x+ ONLY: A bitmasked value that describes the relational join operators that are supported in a *SELECT* statement. The following bitmask identifiers are used:

SQL_SRJO_CORRESPONDING_CLAUSE
SQL_SRJO_CROSS_JOIN
SQL_SRJO_EXCEPT_JOIN
SQL_SRJO_FULL_OUTER_JOIN
SQL_SRJO_INTERSECT_JOIN
SQL_SRJO_LEFT_OUTER_JOIN
SQL_SRJO_NATURAL_JOIN
SQL_SRJO_RIGHT_OUTER_JOIN
SQL_SRJO_UNION_JOIN
SQL_SRJO_INNER_JOIN (indicates support for the *INNER JOIN* syntax, *not* for the inner join capability)

DB_SQL92_REVOKE

ODBC 3.x+ ONLY: A bitmasked value that describes the clauses that are supported in a *REVOKE* statement. The following bitmask identifiers are used:

SQL_SR_CASCADE
SQL_SR_DELETE_TABLE
SQL_SR_GRANT_OPTION_FOR
SQL_SR_INSERT_COLUMN
SQL_SR_INSERT_TABLE
SQL_SR_REFERENCES_COLUMN
SQL_SR_REFERENCES_TABLE
SQL_SR_RESTRICT
SQL_SR_SELECT_TABLE
SQL_SR_UPDATE_COLUMN
SQL_SR_UPDATE_TABLE
SQL_SR_USAGE_ON_DOMAIN
SQL_SR_USAGE_ON_CHARACTER_SET
SQL_SR_USAGE_ON_COLLATION
SQL_SR_USAGE_ON_TRANSLATION

DB_SQL92_ROW_VALUE_CONSTRUCTOR

ODBC 3.x+ ONLY: A bitmasked value that describes the row value constructor expressions that are supported in a *SELECT* statement. The following bitmask identifiers are used:

SQL_SRVC_VALUE_EXPRESSION
SQL_SRVC_NULL
SQL_SRVC_DEFAULT
SQL_SRVC_ROW_SUBQUERY

DB_SQL92_STRING_FUNCTIONS

ODBC 3.x+ ONLY: A bitmasked value that describes the string scalar functions that are supported by the driver and the Datasource. The following bitmask identifiers are used

SQL_SSF_CONVERT
SQL_SSF_LOWER
SQL_SSF_UPPER
SQL_SSF_SUBSTRING
SQL_SSF_TRANSLATE
SQL_SSF_TRIM_BOTH
SQL_SSF_TRIM_LEADING
SQL_SSF_TRIM_TRAILING

DB_SQL92_VALUE_EXPRESSIONS

ODBC 3.x+ ONLY: A bitmasked value that describes the value expressions that are supported. The following bitmask identifiers are used

SQL_SVE_CASE
SQL_SVE_CAST
SQL_SVE_COALESCE
SQL_SVE_NULLIF

DB_STANDARD_CLI_CONFORMANCE

ODBC 3.x+ ONLY: A bitmasked value that describes the CLI standard(s) to which the driver conforms. The following bitmask identifiers are used:

SQL_SCC_XOPEN_CLI_VERSION1
SQL_SCC_ISO92_CLI

DB_STATIC_CURSOR_ATTRIBUTES1 and DB_STATIC_CURSOR_ATTRIBUTES2

ODBC 3.x+ ONLY: These functions are virtually identical to the DB_DYNAMIC_CURSOR_ATTRIBUTES1 and 2 functions that are described above. For complete information, read the descriptions of DB_DYNAMIC_CURSOR_ATTRIBUTES1 and 2 above and substitute "static" wherever it says dynamic.

DB_STATIC_SENSITIVITY

This is a "deprecated" function in ODBC 3.x and should not be used.

DB_STRING_FUNCTIONS

A bitmasked value that describes the scalar string functions that are supported by the driver and associated Datasource. The following bitmask identifiers are used:

SQL_FN_STR_ASCII
SQL_FN_STR_BIT_LENGTH
SQL_FN_STR_CHAR
SQL_FN_STR_CHAR_LENGTH
SQL_FN_STR_CHARACTER_LENGTH
SQL_FN_STR_CONCAT
SQL_FN_STR_DIFFERENCE
SQL_FN_STR_INSERT
SQL_FN_STR_LCASE
SQL_FN_STR_LEFT

SQL_FN_STR_LENGTH
 SQL_FN_STR_LOCATE (see below)
 SQL_FN_STR_LOCATE_2 (see below)
 SQL_FN_STR_LTRIM
 SQL_FN_STR_OCTET_LENGTH
 SQL_FN_STR_POSITION
 SQL_FN_STR_REPEAT
 SQL_FN_STR_REPLACE
 SQL_FN_STR_RIGHT
 SQL_FN_STR_RTRIM
 SQL_FN_STR_SOUNDEX
 SQL_FN_STR_SPACE
 SQL_FN_STR_SUBSTRING
 SQL_FN_STR_UCASE

Note: If an application can call the [LOCATE](#) function with the *string_exp1*, *string_exp2*, and *start* arguments, the driver returns the `SQL_FN_STR_LOCATE` bit. If an application can call the [LOCATE](#) function with only the *string_exp1* and *string_exp2* arguments, the driver returns the `SQL_FN_STR_LOCATE_2` bit. Drivers that fully support the [LOCATE](#) function return *both* bits.

DB_SUBQUERIES

A bitmasked value that describes the predicates that support subqueries:

SQL_SQ_CORRELATED_SUBQUERIES
 SQL_SQ_COMPARISON
 SQL_SQ_EXISTS
 SQL_SQ_IN
 SQL_SQ_QUANTIFIED

The `SQL_SQ_CORRELATED_SUBQUERIES` bit indicates that all of the predicates that support subqueries support correlated subqueries.

DB_SYSTEM_FUNCTIONS

A bitmasked value that describes the scalar system functions that are supported by the driver and Datasource. The following bitmask identifiers are used

SQL_FN_SYS_DBNAME
 SQL_FN_SYS_IFNULL
 SQL_FN_SYS_USERNAME

DB_TABLE_COUNT

The number of tables that a database contains. Also see `SQL_TblCount`.

DB_TIMESTAMP_ADD_INTERVALS

A bitmasked value that describes the timestamp intervals that are supported by the driver and Datasource for the [TIMESTAMPADD](#) scalar function. The following bitmask identifiers are used:

```

SQL_FN_TSI_FRAC_SECOND
SQL_FN_TSI_SECOND
SQL_FN_TSI_MINUTE
SQL_FN_TSI_HOUR
SQL_FN_TSI_DAY
SQL_FN_TSI_WEEK
SQL_FN_TSI_MONTH
SQL_FN_TSI_QUARTER
SQL_FN_TSI_YEAR

```

DB_TIMEDATE_DIFF_INTERVALS

A bitmasked value that describes the timestamp intervals that are supported by the driver and Datasource for the [*TIMESTAMPDIFF*](#) scalar function. The following bitmask identifiers are used:

```

SQL_FN_TSI_FRAC_SECOND
SQL_FN_TSI_SECOND
SQL_FN_TSI_MINUTE
SQL_FN_TSI_HOUR
SQL_FN_TSI_DAY
SQL_FN_TSI_WEEK
SQL_FN_TSI_MONTH
SQL_FN_TSI_QUARTER
SQL_FN_TSI_YEAR

```

DB_TIMEDATE_FUNCTIONS

A bitmasked value that describes the scalar date and time functions that are supported by the driver and Datasource. The following bitmask identifiers are used:

```

SQL_FN_TD_CURRENT_DATE
SQL_FN_TD_CURRENT_TIME
SQL_FN_TD_CURRENT_TIMESTAMP
SQL_FN_TD_CURDATE
SQL_FN_TD_CURTIME
SQL_FN_TD_DAYNAME
SQL_FN_TD_DAYOFMONTH
SQL_FN_TD_DAYOFWEEK
SQL_FN_TD_DAYOFYEAR
SQL_FN_TD_EXTRACT
SQL_FN_TD_HOUR
SQL_FN_TD_MINUTE
SQL_FN_TD_MONTH
SQL_FN_TD_MONTHNAME
SQL_FN_TD_NOW
SQL_FN_TD_QUARTER
SQL_FN_TD_SECOND
SQL_FN_TD_TIMESTAMPADD
SQL_FN_TD_TIMESTAMPDIFF
SQL_FN_TD_WEEK
SQL_FN_TD_YEAR

```

DB_TXN_CAPABLE

A numeric value that describes the transaction support that is provided by the driver or the Datasource:

SQL_TC_NONE (Transactions are not supported.)

SQL_TC_DML (Transactions can only contain [SELECT](#), [INSERT](#), [UPDATE](#), and [DELETE](#). If other syntax is encountered in a transaction, an error message will be generated.)

SQL_TC_DDL_COMMIT (Transactions can only contain [SELECT](#), [INSERT](#), [UPDATE](#), and [DELETE](#). If other syntax is encountered in a transaction, the transaction will be committed.)

SQL_TC_DDL_IGNORE (Transactions can only contain [SELECT](#), [INSERT](#), [UPDATE](#), and [DELETE](#). If other syntax is encountered in a transaction, it will be ignored.)

SQL_TC_ALL (Transactions can contain any statements in any order.)

DB_TXN_ISOLATION_OPTION

A bitmasked value that describes the transaction isolation levels that are available from the driver or the Datasource. The following bitmask identifiers are used:

SQL_TXN_READ_UNCOMMITTED
SQL_TXN_READ_COMMITTED
SQL_TXN_REPEATABLE_READ
SQL_TXN_SERIALIZABLE

For descriptions of these isolation levels, see the descriptions under DB_DEFAULT_TXN_ISOLATION (above).

DB_UNIDENTIFIED_115

This function is not described by the Microsoft ODBC Software Developer Kit. It appears to return a numeric value.

DB_UNION

A bitmasked value that describes the support for the [UNION](#) clause:

SQL_U_UNION (The Datasource supports the [UNION](#) clause.)

SQL_U_UNION_ALL (The Datasource supports the [ALL](#) keyword in the [UNION](#) clause.)

Diagnostics

This function does not return Error Codes because values like SUCCESS_WITH_INFO (value 1) could be confused with legitimate return values, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_DBInfoUInt(DB_TXN_CAPABLE)
```

Driver Issues

None.

Speed Issues

SQL Tools does *not* cache the values that are returned by this function. If your program needs one or more of these values repeatedly, you may be able to improve the speed of your program by obtaining a `SQL_DBInfoStr` or `SQL_DBInfoUInt` value and storing it in a variable, instead of repeatedly using this function.

See Also

Database Information and Attributes

SQL_DBIsOpen

Summary

Indicates whether or not a database is open.

Twin

SQL_DatabaseIsOpen

Family

Database Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_DBIsOpen
```

Parameters

None.

Return Values

Logical True (-1) if the current database (as specified with the `SQL_UseDB` function) is open, or False (zero) if it is not open.

Remarks

This function can be used to determine whether or not a database is open. For information about opening databases, see [Opening A Database](#).

Diagnostics

This function does not return Error Codes, but it can generate SQL Tools Error Messages.

Example

```
If SQL_DBIsOpen Then Beep
```

Driver Issues

None.

Speed Issues

None.

See Also

SQL_StmtIsOpen

SQL_Diagnostic

Summary

Provides additional diagnostic information about the most-recently-used SQL Tools function, if it returned an Error Message.

Twin

None. You must use this function, with an appropriate *IDatabaseNumber*& and *IStatementNumber*& value, in order to obtain diagnostic information.

Family

Error/Trace Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_Diagnostic(lDatabaseNumber&, _  
                          lStatementNumber&, _  
                          lInfoType&)
```

Parameters

IDatabaseNumber

The database number of the database that experienced the error. If the error is related to all databases (as would be the case when your program attempts to set an environment attribute with the `SQL_SetEnvironAttribSInt` function, for example), use `ALL_DBs`.

IStatementNumber&

The statement number of the statement that experienced the error. If the error is related to all statements (as would be the case when your program attempts to set a database attribute with `SQL_SetDBAttribUInt`, for example), use `ALL_STMTs`.

IInfoType&

One of the `DIAG_` constants described in **Remarks**, below.

Return Values

An empty string (if no diagnostic information is available), or a string that contains one or more diagnostic values. If more than one diagnostic value is returned, the values will usually be comma-delimited.

Remarks

When SQL Tools executes a function that returns an Error Message (i.e. a result other than `SUCCESS`) it *may* be possible to retrieve additional information about the error by using the `SQL_Diagnostic` function. The term "error", in this context, also includes `SUCCESS_WITH_INFO`.

IMPORTANT NOTE: This function can only return information about the *most-recently-used* SQL Tools function. If you use a SQL Tools function that returns an error and then use another SQL Tools function, it is very likely that you will then be *unable* to use `SQL_Diagnostic` to obtain information about the original error, regardless of whether or not the second function returned an error.

IMPORTANT NOTE: Multiple values can be returned by this function, so *all* values are returned as comma-delimited strings, even if they are basically numeric values. It is possible to change the string that is used to delimit multiple return values by using the `SQL_SetOptionStr(OPT_COL_DELIMITER)` function. If you change this option, please insert the name of the delimiter that you specified wherever this document says "comma-delimited".

The *InfoType* parameter must be one of the following values:

The first group of *InfoType* values represent "global" diagnostic information. Only one diagnostic value will be returned for each of these *InfoType* values:

`DIAG_CURSOR_ROW_COUNT`

This field contains the cursor's row count. The exact meaning of this value will depend on the attributes of the current cursor, which can be obtained with the `SQL_StmtAttrib(DB_type_CURSOR_ATTRIBUTES2)` function.

You *must* specify a database number and a statement number when using this function. You may not specify `ALL_DBS` or `ALL_STMTS`.

`DIAG_DYNAMIC_FUNCTION` and
`DIAG_DYNAMIC_FUNCTION_CODE`

These functions can only be used to obtain diagnostic information about errors that are reported by `SQL_Stmt(SQL_STMT_EXECUTE)`, `SQL_Stmt(SQL_STMT_IMMEDIATE)`, or `SQL_MoreRes`. They return strings that describe the SQL statement that the underlying function executed.

Each unique string value that is returned by `DIAG_DYNAMIC_FUNCTION` (such as *DELETE WHERE* or *DROP TABLE*) corresponds to a unique numeric value that is returned by `DIAG_DYNAMIC_FUNCTION_CODE`, so most programs use one or the other.

You *must* specify a database number and a statement number when using these functions. You may not specify `ALL_DBS` or `ALL_STMTS`.

`DIAG_NUMBER`

The number of diagnostic records that are available for the specified database or statement, i.e. the number of diagnostic values that non-global *InfoType* values (see below) will return.

`DIAG_RETURNCODE`

The Error Code that was returned by the function. This information can also be obtained by using the various `SQL_Error` functions.

`DIAG_ROW_COUNT`

The number of rows that were affected by an *INSERT*, *DELETE* or *UPDATE* performed with `SQL_Stmt(SQL_STMT_EXECUTE)`, `SQL_Stmt(SQL_STMT_IMMEDIATE)`, `SQL_BulkOp`, or `SQL_SetPos`. This value can also be obtained with the `SQL_ResRowCount` function.

You *must* specify a database number and a statement number when using this function. You may not specify ALL_DBS or ALL_STMTs.

The second group of *InfoType*& values represent non-global diagnostic information. Multiple diagnostic values (see DIAG_NUMBER above) can be returned for each of these *InfoType*& values:

DIAG_CLASS_ORIGIN

A string that contains the ODBC specification document which defines the "class" portion of the SQL State value for this error.

The return value of this function will be "ISO 9075" for all SQL States defined by the X/Open and ISO call-level interface.

The return value of this function will be "ODBC 3.0" for ODBC-specific SQL States (all those that have a SQL State class of "IM").

DIAG_COLUMN_NUMBER

If the DIAG_ROW_NUMBER value (see below) is **1**) a valid row number in a rowset or **2**) a valid row number in a set of parameters, this field will contain (respectively) **1**) the column number in the result set or **2**) the parameter number in the set of parameters

Result set column numbers always start at 1. If an error pertains to a bookmark column, then this value can be zero.

Parameter numbers start at 1. This value will be negative one (-1) if the error is not associated with a particular column number or parameter number.

If the driver cannot determine the column number or parameter number, this value will be negative two (-2).

You *must* specify a database number and a statement number when using this function. You may not specify ALL_DBS or ALL_STMTs.

DIAG_CONNECTION_NAME

A string that contains the name of the connection that the error relates to. This value is driver-defined.

You *must* specify a database number when using this function, and you *must* use a statement number of ALL_STMTs.

DIAG_MESSAGE_TEXT

This string will contain an informational message on the error or warning. This information can also be obtained from the SQL_ErrorText function.

DIAG_NATIVE

A driver-specific or datasource-specific native error code. If there is no native error code, this value will be zero (0).

This information can also be obtained from the `SQL_ErrorNativeCode` function.

DIAG_ROW_NUMBER

The row number in the rowset, or the parameter number in the set of parameters, with which the error is associated.

Row numbers and parameter numbers start with 1. This value will be negative one (-1) if the error is not associated with a particular row number or parameter number.

If the driver cannot determine the row number or parameter number with which the error is associated, this value will be negative two (-2)

You *must* specify a database number and a statement number when using this function. You may not specify `ALL_DBS` or `ALL_STMTS`.

DIAG_SERVER_NAME

A string that contains the server name to which the error relates.

DIAG_SQLSTATE

A five-character SQL State diagnostic code. This value can also be obtained with the `SQL_State` function.

DIAG_SUBCLASS_ORIGIN

A string with the same format and valid values as `DIAG_CLASS_ORIGIN` (see above) which identifies the defining portion of the subclass portion of the SQL State code.

The ODBC-specific SQL States for which "ODBC 3.0" is returned include:

01S00, 01S01, 01S02, 01S06, 01S07, 07S01, 08S01, 21S01, 21S02, 25S01, 25S02, 25S03, 42S01, 42S02, 42S11, 42S12, 42S21, 42S22, HY095, HY097, HY098, HY099, HY100, HY101, HY105, HY107, HY109, HY110, HY111, HYT00, HYT01, IM001, IM002, IM003, IM004, IM005, IM006, IM007, IM008, IM010, IM011, IM012.

Diagnostics

This function does not generate Error Messages because it is used to obtain information *about* Error Messages.

Example

```
Print SQL_Diagnostic(1,1,DIAG_CURSOR_ROW_COUNT)
```

Driver Issues: None.

Speed Issues: None.

See Also

The Error/Trace Family

SQL_DirectBindCol

Summary

Binds one column of a result set to a memory buffer that your program provides, while allowing SQL Tools to bind the Indicator. (Most programs do not use this function, because SQL Tools can AutoBind all of the columns in a result set. Also compare `SQL_ManualBindCol`.)

Twin

`SQL_DirectBindColumn`

Family

Result Column Binding Family

Availability

SQL Tools Pro Only (see)

Warning

If your program uses this function to bind a result column to a memory buffer but then fails to maintain that buffer, an Application Error will result. See **Remarks** below for more information.

Syntax

```
lResult& = SQL_DirectBindCol(lColumnNumber&, _  
                             lDataType&, _  
                             lPointerToBuffer&, _  
                             lBufferLength&)
```

Parameters

lColumnNumber&

The number of the column that is to be bound, from one (1) to the number of columns in the result set. If a bookmark column is being bound (not recommended) this value can be zero (0).

lDataType&

The SQL Data Type of the column's data. See the SQL Tools Declaration Files for a list of legal values. **Technical note for experienced ODBC users: Do not attempt to use `SQL_C_` data types for this parameter.**

lPointerToBuffer&

A 32-bit pointer to the memory location where the memory buffer begins.

lBufferLength&

The length of the memory buffer, in bytes.

Return Values

This function will return `SUCCESS` or `SUCCESS_WITH_INFO` if the binding operation is successful, or an Error Code if it is not.

Remarks

SQL Tools can perform three types of Result Column Binding. **1)** AutoBinding, where the data buffer and Indicator buffer are managed by SQL Tools, **2)** Manual Binding, where the data buffer and Indicator buffer are managed by your program, and **3)** Direct Binding (which uses this function) where the data buffer is managed by your program and the Indicator buffer is managed by SQL Tools. See Result Column Binding for more information.

In order for a program to access a value in a column of a result set, the column must be bound to a memory buffer that is large enough to hold the value. The `SQL_DirectBindCol` function can be used to perform this operation.

NOTE: Most SQL Tools programs use AutoBinding, so the `SQL_DirectBindCol` function is rarely used. You should only attempt to use this function if **1)** you need to squeeze every *drop* of speed from your application, or **2)** if the SQL Tools AutoBind function does not bind a column in the way that you need it to be bound.

Once you have bound a column of a result set to a memory buffer with `SQL_DirectBindCol`, your program is responsible for maintaining that buffer. Most importantly, *you must make sure that the buffer does not move* or, if it does move, you *must* re-bind the buffer before the `SQL_Fetch` or `SQL_FetchRel` function is used again. Failure to do this will almost certainly result in an Application Error. If you use a BASIC `ASCII` string, fixed-length string, or numeric variable (or an array) for a buffer it will be *fixed* in memory and it will not move, so this is not a concern. If you use a BASIC *dynamic* string, however, the string will move whenever you assign a value to it, so you must take great care to avoid assigning a value to a string variable that is used for a buffer.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

None. The use of this function is too complex for a brief example to be meaningful. The User's Guide section of this document contains extensive explanations and examples.

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Binding (Basic), Result Column Binding (Advanced)

SQL_DirectBindColumn

Summary

Binds one column of a result set to a memory buffer that your program provides, while allowing SQL Tools to bind the Indicator. (Most programs do not use this function, because SQL Tools can AutoBind all of the columns in a result set. Also compare `SQL_ManualBindCol`.)

Twin

`SQL_DirectBindCol`

Family

Result Column Binding Family

Availability

SQL Tools Pro Only (see)

Warning

If your program uses this function to bind a result column to a memory buffer but then fails to maintain that buffer, an Application Error will result. See `SQL_DirectBindCol` for more information.

Syntax

```
lResult& = SQL_DirectBindColumn(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&, _  
                                lDataType&, _  
                                lPointerToBuffer&, _  
                                lBufferLength&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function `SQL_DirectBindCol` for complete details.

Return Values

See twin function `SQL_DirectBindCol` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_DirectBindColumn` is identical to `SQL_DirectBindCol`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_DirectBindColumn` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_DirectBindCol` for complete details.

Example

See twin function `SQL_DirectBindCol` for complete details.

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Binding (Basic), Result Column Binding (Advanced)

SQL_DriverCount

Summary

Returns the number of ODBC Drivers that are available to your program.

Twin

None.

Family

Environment Family

Availability

SQL Tools Pro Only (see)

Warning

See **Remarks** regarding "cached" information, below.

Syntax

```
lResult& = SQL_DriverCount
```

Parameters

None.

Return Values

This function will return zero or a positive number, to indicate the number of ODBC drivers that are available to your program.

Remarks

To improve program performance, the very first time that the `SQL_DriverCount`, `SQL_DriverInfoStr` or `SQL_DriverNumber` function is used, SQL Tools reads *all* of the available ODBC driver information and caches it (i.e. stores it internally), so that future uses of these functions will be significantly faster.

Under normal circumstances this technique works well, but if your program uses one of these functions and then an ODBC driver is added to your system *while your program is still running*, it will not be detected. If you have reason to believe that the ODBC driver information may have changed since the last time your program used one of these driver functions, you can use the `SQL_GetDrivers` function to re-read all of the available ODBC driver information. Keep in mind that this process can take several seconds.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with the answer "there is one ODBC driver available." This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display all currently installed Drivers:
For lDriver& = 1 To SQL_DriverCount
    Print SQL_DriverInfo(lDriver&, DRIVER_NAME)
Next
```

Driver Issues

None.

Speed Issues

See notes regarding "cached" information, in **Remarks** above.

See Also

ODBC Drivers

SQL_DriverInfoStr

Summary

Returns information about an ODBC Driver, in string form. (All driver information is string-based, so there is no numeric function for obtaining driver information.)

Twin

None.

Family

Environment Family

Availability

SQL Tools Pro Only (see)

Warning

See **Remarks** below, regarding "cached" information.

Syntax

```
strResult$ = SQL_DriverInfoStr(lDriverNumber&, _  
                                lInfoType&)
```

Parameters

lDriverNumber&

A number between one (1) and the number of ODBC Drivers that are available, as returned by the `SQL_DriverCount` function.

lInfoType&

`DRIVER_NAME`, `DRIVER_DESCRIPTION` or a numeric value greater than 200. See **Remarks** below for more information.

Return Values

This function will return a string that contains the requested information, or an empty string if an invalid parameter is used.

Remarks

If you use an *lInfoType&* value of `DRIVER_NAME`, a string like "Microsoft Access Driver (*.mdb)" or "SQL Server" will be returned.

If you use an *lInfoType&* value of `DRIVER_DESCRIPTION`, a string that contains a complete ODBC Driver description will be returned. A driver description string contains one or more pieces of information about the ODBC driver, delimited with Carriage Return characters (`CHR$(13)`). Here is a typical driver description for Microsoft Access 97, with the `CHR$(13)` delimiters represented by **<CR>**.

```
UsageCount=19<CR>APILevel=1<CR>ConnectFunctions=YYN<CR>DriverOD  
BCVer=02.50<CR>FileUsage=2<CR>FileExtns=*.mdb<CR>SQLLevel=0<CR>  
s=YYN<CR>
```

The `DRIVER_NAME` constant has a numeric value of one (1), and the `DRIVER_DESCRIPTION` constant has a numeric value of two (2). The individual elements of the `DRIVER_DESCRIPTION` string can be accessed individually by using an *lInfoType&* value of 200 (meaning "element of *lInfoType&* 2") plus an element number. For example, if the example string above was returned for `DRIVER_DESCRIPTION`, using an *lInfoType&* value of 202 would return

"APILevel=1" because that is the second element of the string.

To improve program performance, the very first time that the `SQL_DriverInfoStr`, `SQL_DriverCount`, or `SQL_DriverNumber` function is used, SQL Tools reads *all* of the available ODBC driver information and caches it (i.e. stores it internally), so that future uses of these functions will be significantly faster.

Under normal circumstances this technique works well, but if your program uses one of these functions and then an ODBC driver is added to your system *while your program is still running*, it will not be detected. If you have reason to believe that the ODBC driver information may have changed since the last time your program used one of these driver functions, you can use the `SQL_GetDrivers` function to re-read all of the available ODBC driver information. Keep in mind that this process can take several seconds.

Diagnostics

This function does not return Error Codes because it returns only string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_DriverInfoStr(1,DIVER_NAME)
```

Driver Issues

None.

Speed Issues

See notes regarding "cached" information, in **Remarks** above.

See Also

ODBC Drivers

SQL_DriverNumber

Summary

Returns the ODBC Driver number (if any) that is associated with an ODBC Driver name.

Twin

None.

Family

Environment Family

Availability

SQL Tools Pro Only (see)

Warning

See **Remarks** regarding "cached" information, below.

Syntax

```
lResult& = SQL_DriverNumber(sDriverName$)
```

Parameters

sDriverName\$

A string that contains the exact name of an ODBC Driver, such as "SQL Server" or "Microsoft Access Driver (*.mdb)".

Return Values

If an ODBC Driver with the specified name is found, the corresponding driver number will be returned. If no match is found, negative one (-1) will be returned.

Remarks

This function is *not* case-sensitive. If an ODBC Driver named "SQL Server" exists, using an *sDriverName\$* value of "SQL Server", "SQL SERVER", "sql Server", (etc.) would produce the same results.

To improve program performance, the very first time that the `SQL_DriverNumber`, `SQL_DriverInfoStr`, or `SQL_DriverCount` function is used, SQL Tools reads *all* of the available ODBC driver information and caches it (i.e. stores it internally), so that future uses of these functions will be significantly faster.

Under normal circumstances this technique works well, but if your program uses one of these functions and then an ODBC driver is added to your system *while your program is still running*, it will not be detected. If you have reason to believe that the ODBC driver information may have changed since the last time your program used one of these driver functions, you can use the `SQL_GetDrivers` function to re-read all of the available ODBC driver information. Keep in mind that this process can take several seconds.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with the result "the specified string matches ODBC Driver number 1". This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_DriverNumber("SQL SERVER")
```

Driver Issues

None.

Speed Issues

See notes regarding "cached" information, in **Remarks** above.

See Also

ODBC Drivers

SQL_EndOfData

Summary

This function returns a Logical True value (-1) if the most recent `SQL_Fetch`, `SQL_FetchRel`, `SQL_FetchResult`, or `SQL_FetchRelative` operation failed because **1)** there were no rows in the result set, **2)** you attempted to fetch a row beyond the last row of the result set, or **3)** you attempted to fetch a row before the first row of the result set. Otherwise it returns a False (zero) value. (Note that it does *not* return True if the fetch operation failed because of an *error*.)

Twin

`SQL_EOD`

Family

Statement Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_EndOfData(lDatabaseNumber, _  
                        lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers for a detailed discussion.

Return Values

See twin function `SQL_EOD`.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_EndOfData` is identical to `SQL_EOD`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_EndOfData` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Example

```
Do  
    SQL_FetchResult 2, 1, NEXT_ROW  
    If SQL_EndOfData(2, 1) Then Exit Do  
    'process a row of data  
Loop
```

Driver Issues

See twin function `SQL_EOD` for complete details.

Speed Issues

See twin function `SQL_EOD` for complete details.

See Also

Detecting the End Of Data

SQL_EndTrans

Summary

Instructs a database to end a transaction by either "committing it" or "rolling it back". (Most programs use the AutoCommit mode, so this function is not commonly used.)

Twin

SQL_EndTransaction

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_EndTrans(lOperation&)
```

Parameters

lOperation&

Either TRANS_COMMIT or TRANS_ROLLBACK.

Return Values

This function will return SUCCESS or SUCCESS_WITH_INFO if the transaction is ended according to *lOperation&*, or an Error Code if it is not.

Remarks

If the AutoCommit mode (which is the default mode for SQL Tools) is used, this function has no effect on a transaction.

If the AutoCommit mode is turned off (by using the SQL_DBAutoCommit function), then your program is responsible for telling the database to either **1**) "commit" a transaction (i.e. make it final by changing the database) or **2**) "roll back" a transaction (i.e. cancel it, and undo all of the changes that may have been made in the database).

See Transactions for more information about using this function.

Diagnostics

This function can return Error Codes, and can also return ODBC Error Messages and SQL Tools Error Messages.

Example

```
lResult& = SQL_EndTrans(TRANS_COMMIT)
```

Driver Issues: None.

Speed Issues: None.

See Also

Committing Transactions Manually

SQL_EndTransaction

Summary

Instructs a database to end a transaction by either "committing it" or "rolling it back". (Most programs use the AutoCommit mode, so this function is not commonly used.)

Twin

SQL_EndTrans

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_EndTransaction(lDatabaseNumber&, _  
                               lOperation&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_EndTrans for complete details.

Return Values

See twin function SQL_EndTrans for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_EndTransaction is identical to SQL_EndTrans. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_EndTransaction is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_EndTrans for complete details.

Example

See twin function SQL_EndTrans.

Driver Issues

None.

Speed Issues

None.

See Also

Committing Transactions Manually

SQL_EnvironAttribSInt

Summary

Returns information about the ODBC environment (which affects all databases and statements) in numeric form.

Twin

None.

Family

Environment Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_EnvironAttribSInt(lInfoType&)
```

Parameters

lInfoType&

One of the following constants: ENV_ODBC_VERSION, ENV_CONNECTION_POOLING, ENV_CP_MATCH, or ENV_OUTPUT_NTS. See **Remarks** below for details.

Return Values

If a valid value is specified for *lInfoType&*, this function will return the corresponding ODBC environment attribute. If an invalid value is used, zero (0) will be returned.

Remarks

If *lInfoType&* is...

ENV_CONNECTION_POOLING

This function will return one of the following numeric values:

SQL_CP_OFF (Connection pooling is turned off. This is the default.)

SQL_CP_ONE_PER_DRIVER (A single connection pool is supported for each driver. Every database connection in a pool is associated with one driver.)

SQL_CP_ONE_PER_HENV (A single connection pool is supported for each environment. Every database connection in a pool is associated with one environment, i.e. one program.)

See SQL_Initialize for more information.

ENV_CP_MATCH

This function will return one of the following numeric values:

`SQL_CP_STRICT_MATCH` (Only connections that exactly match the connection options in the call and the connection attributes set by the program are reused. If connection pooling is turned on, this is the default.)

`SQL_CP_RELAXED_MATCH` (Connections with matching connection string keywords can be used. Keywords must match, but not all connection attributes must match.)

See `SQL_Initialize` for more information.

`ENV_ODBC_VERSION`

This function will return a value of either two (2) or three (3), to indicate the ODBC Version that is being provided by the environment. If an ODBC function (and therefore a SQL Tools function) behaves differently if ODBC 2 or 3 is used, this function tells you which behavior is being emulated.

By default, SQL Tools sets this attribute to 3 because all databases can support at least some ODBC 3.x behavior. See `SQL_Initialize` for more information.

`ENV_OUTPUT_NTS`

In a Windows environment, this function ("Output Null Terminated Strings") will always return a value of one (1), indicating that Null Terminated Strings (also known as `ASCIIZ` strings) are used internally by the ODBC driver. SQL Tools removes the null terminators from *dynamic* strings before returning them to your program.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value. This function can (but does not usually) return ODBC Error Messages or SQL Tools Error Messages.

Example

```
Print SQL_EnvironAttribSInt(ENV_ODBC_VERSION)
```

Driver Issues

None.

Speed Issues

None.

See Also

Environment Attributes

SQL_EOD

Summary

This function returns a Logical True value (-1) if the most recent `SQL_Fetch`, `SQL_FetchRel`, `SQL_FetchResult`, or `SQL_FetchRelative` operation failed because **1)** there were no rows in the result set, **2)** you attempted to fetch a row beyond the last row of the result set, or **3)** you attempted to fetch a row before the first row of the result set. Otherwise it returns a False (zero) value. (Note that it does *not* return True if the fetch operation failed because of an *error*.)

Twin

`SQL_EndOfData`

Family

Statement Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_EOD
```

Parameters

None.

Return Values

Logical True (-1) or False (0).

Remarks

This function is conceptually similar to the BASIC `Eof` (End Of File) function that is used to detect whether or not a file-input operation has reached the end of the available data. An important distinction, however, is that `SQL_EOD` only returns a True value *after* a `SQL_Fetch` or `SQL_FetchRel` operation has *failed* as the result of reaching then end (or beginning) of the available data.

For a complete discussion of this function, see [Detecting The End Of Data](#).

Example

```
Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD THEN Exit Do
    'process a row of data
LOOP
```

Driver Issues

None.

Speed Issues

None.

See Also: [Detecting "No Data At All"](#)

SQL_ErrorClearAll

Summary

Removes all error messages from the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

Once error messages have been cleared, they cannot be recovered. Make sure that your program has examined and handled all errors before using this function.

Syntax

```
lResult& = SQL_ErrorClearAll
```

Parameters

None.

Return Values

This function returns the number of errors that were in the SQL Tools Error Stack before this function was used, i.e. the number of errors that were cleared.

Remarks

See Error Handling for a complete discussion of this function.

Diagnostics

None.

Example

```
SQL_ErrorClearAll
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorClearOne

Summary

Removes one error message from the SQL Tools Error Stack.

Twin

None

Family

Error/Trace Family

Availability

Standard and Pro

Warning

Once an error message has been cleared, it cannot be recovered. Make sure that your program has examined and handled an error before using this function.

Syntax

```
lResult& = SQL_ErrorClearOne
```

Parameters

None.

Return Values

This function returns a value of one (1) if an error is cleared, or zero (0) if there were no errors in the SQL Tools Error Stack when this function was called. In other words, it is like the `SQL_ErrorClearAll` function. It returns the number of errors that were cleared.

Remarks

See Error Handling for a complete discussion of this function.

Diagnostics

None.

Example

```
SQL_ErrorClearOne
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorColumnName

Summary

Returns the Column Number that is associated with the oldest Error Message in the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ErrorColumnName
```

Parameters

None.

Return Values

This function returns negative one (-1) if no column number was associated with the oldest Error Message in the SQL Tools Error Stack, or a number between one (1) and the number of columns in the result set to which the error relates. If there are no Error Messages in the SQL Tools Error Stack, this function will return zero (0). If the error is associated with a bookmark column the return value of this function can also be zero.

Remarks

See Error Handling In SQL Tools Programs for a complete discussion of this function.

Diagnostics

None.

Example

```
PRINT SQL_ErrorColumnName
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Messages

SQL_ErrorCount

Summary

Returns the number of Error Messages that are currently in the SQL Tools Error Stack.

Twin

None

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ErrorCount
```

Parameters

None.

Return Values

This function will return zero (0) if there are no Error Messages in the SQL Tools Error Stack, or a positive number if the stack contains one or more error messages.

Remarks

See Error Handling in SQL Tools Programs for a complete discussion of this function..

Diagnostics

None.

Example

```
If SQL_ErrorCount > 0 Then  
    'handle error messages  
End If
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Messages

SQL_ErrorDatabaseNumber

Summary

Returns the database number that is associated with the oldest Error Message in the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ErrorDatabaseNumber
```

Parameters

None.

Return Values

This function will return negative one (-1) if the oldest Error Message in the SQL Tools Error Stack is not associated with a database number (as would be the case with a failed attempt to change the ODBC environment), or a number between one (1) and the maximum database number specified in the `SQL_Initialize` function. If there are no Error Messages in the SQL Tools Error Stack, this function will return zero (0).

Remarks

See Error Handling in SQL Tools Programs for a complete discussion of this function..

Diagnostics

None.

Example

```
Print SQL_ErrorDatabaseNumber
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Messages

SQL_ErrorFuncName

Summary

Returns the name of the function that is associated with the oldest Error Message in the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

See IMPORTANT NOTES in Remarks below.

Syntax

```
sResult$ = SQL_ErrorFuncName
```

Parameters

None.

Return Values

This function will return the name of the function that is associated with the oldest Error Message in the SQL Tools Error Stack. This will *usually* be the name of a SQL Tools function, but it can also be the name of a sub or function in *your* program if you have used the `SQL_ErrorSimulate` function. If there are no Error Messages in the SQL Tools Error Stack, this function will return an empty string.

Remarks

IMPORTANT NOTE: This function always returns the name of a "verbose" SQL Tools function, even if your program used an "abbreviated" function. For example, if your program made an error when using the `SQL_OpenDB` function, the `SQL_ErrorFuncName` return value would be `SQL_OpenDatabase` not `SQL_OpenDB`

IMPORTANT NOTE: It is entirely possible that the `SQL_ErrorFuncName` return value will be the name of a function that your program did not use directly. For example, if your program uses the `SQL_Stmt` function and SQL Tools automatically (internally) uses the `SQL_AutoBindCol` function to bind the columns in the statement's result set, an error may be reported for `SQL_AutoBindColumn` even though your program did not use that function directly. In order to make troubleshooting easier, you can use the `SQL_Trace` function to determine the *exact* source of any error.

See Error Handling in SQL Tools Programs for a complete discussion of this function.

Diagnostics

None.

Example

```
Print SQL_ErrorFuncName
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Messages

SQL_ErrorFunction

Summary

This function has been replaced by the `SQL_ErrorFuncName` function.

`SQL_ErrorFunction` should no longer be used.

SQL_ErrorIgnore

Summary

Tells SQL Tools to ignore Error Messages, under certain conditions.

Twin

None

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_ErrorIgnore(lDatabaseNumber&, _  
                           lStatementNumber&, _  
                           sSQLStates$)
```

Parameters

lDatabaseNumber

The Database Number of the database where the error(s) should be ignored, between one (1) and the *lMaxDatabaseNumber*& value that you specified with the SQL_Initialize function. (The default value of *lMaxDatabaseNumber*& is two (2) if you use the SQL_Init function instead of SQL_Initialize.) Note that you may *not* use ALL_DBs for this parameter.

lStatementNumber&

The Statement Number of the statement where the error(s) should be ignored, between zero (0) and the *lMaxStatementNumber*& value that you specified with the SQL_Initialize function. (The default value of *lMaxStatementNumber*& is two (2) if you use the SQL_Init function instead of SQL_Initialize.) You may also use the value ALL_STMTs for this parameter, to specify that errors should be ignored for Database Number *lDatabaseNumber*&, regardless of the Statement Number.

sSQLStates\$

One or more five-character SQL State strings. If two or more SQL States are specified, they must be delimited with commas.

Return Values

This function returns the *sSQLStates*\$ value that was specified the *last* time that the SQL_ErrorIgnore function was used. See **Remarks** below for more information.

Remarks

This function is used to tell SQL Tools not to report Error Messages with certain SQL State values. See Ignoring Predictable Errors for background information.

This function's return value is the *sSQLStates*\$ that was used the *last* time that the SQL_ErrorIgnore function was used. For example, if the first time the function is used for a particular Database/Statement Number combination it is used to ignore the SQL State "12345", the return value will be an empty string. If the next time the function is used for the same Database/Statement Number combination, it is used to

ignore the SQL State "54321", the return value of the function would be the previous `sSQLStates$` value, "12345".

This seemingly-unusual behavior makes it possible for you to use code like this...

```
SQL_ErrorIgnore 1, 1, "12345"
```

```
'(your code goes here)
```

```
SQL_ErrorIgnore 1 ,1, SQL_ErrorIgnore(1, 1, "")+ ",98765"
```

...to *add* a SQL State code to an existing ignore list. In the example just above, the `SQL_ErrorIgnore 1, 1, "12345"` line would set the ignore-list for Database 1, Statement 1 to the string '12345'.

Then, in the second line, the `SQL_ErrorIgnore(1,1, " ")` function would be executed first. (If you have any questions about the execution order, consult your BASIC documentation.) It would clear the ignore-list for Database 1, Statement 1 to an empty string (" "), but it would also return previous value, the string "12345". The program would then add ",98765" to the end of that string to produce "12345,98765", and that string would be submitted to the `SQL_ErrorIgnore` function as the new ignore-list.

You *must* use commas to separate two or more SQL State strings, or this function will appear to malfunction. For example, if you used the string "1234598765" instead of "12345,98765", SQL Tools would ignore all Error Messages with SQL States that were found *anywhere* in the string "1234598765". It would ignore SQL States 12345, 23459, 34598, and so on.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate SQL Tools Error Messages.

Example

See Ignoring Predictable Errors for several examples.

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorNativeCode

Summary

Returns the Native Error Code that is associated with the oldest Error Message in the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ErrorNativeCode
```

Parameters

None.

Return Values

This function will return the Native Error Code (see **Remarks** below) that is associated with the oldest Error Message in the SQL Tools Error Stack. If there are no errors in the stack, this function will return zero (0).

Remarks

A "Native Error Code" is a numeric value that corresponds to an error condition, as *assigned by the program, subprogram, or driver that detected the error*. Native Codes are not standardized in any way, and no lists of Native Codes are provided in this document.

Native Error Codes can vary greatly. The same SQL State value may be associated with different Native Codes from different ODBC Drivers.

If a certain SQL State value can indicate more than one specific error condition, it may be possible to use the Native Code to determine the cause of the error more precisely.

If you need to know the exact meaning of a Native Code, it will be necessary for you to contact the company that originated the database format that you are using.

Diagnostics

None.

Example

```
Print SQL_ErrorNativeCode
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorNumber

Summary

The Error Codes that is associated with the oldest Error Message in the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ErrorNumber
```

Parameters

None.

Return Values

This function returns the numeric Error Codes value that is associated with the oldest Error Message in the SQL Tools Error Stack. See **Remarks** below for more information.

Remarks

For ODBC Error Messages, the Error Codes will represent a condition like SUCCESS_WITH_INFO (value 1) or SQL_ERROR (value -1).

For SQL Tools Error Messages, the Error Code will represent a condition like ERROR_BAD_PARAM_VALUE (value 999000030) or ERROR_STMT_NOT_OPEN (value 999000034).

See ODBC Error Codes and SQL Tools Error Codes for more information, including a complete list of the possible return values.

Diagnostics

None.

Example

```
Print SQL_ErrorNumber
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorPending

Summary

Indicates, by returning a Logical True (-1) or False (zero) value, whether or not there are any errors in the SQL Tools Error Stack.

Twin

None

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ErrorPending
```

Parameters

None.

Return Values

This function returns Logical True (-1) if there are one or more errors in the SQL Tools Error Stack, or False (zero) if there are no errors in the stack.

Remarks

This function can be used to quickly test whether or not your program needs to examine the SQL Tools Error Stack for error messages.

Diagnostics

None.

Example

```
If SQL_ErrorPending Then  
    'process and clear error message(s)  
End If
```

Driver Issues

None.

Speed Issues

This is the fastest way to check whether or not any errors have been detected since your program began running, or since the last time the Error Stack was cleared.

See Also

Error Handling

SQL_ErrorQuickAll

Summary

Returns a string that contains all of the Error Messages in the SQL Tools Error Stack, and clears the stack.

Twin

None

Family

Error/Trace Family

Availability

Standard and Pro

Warning

Once an error message has been cleared, it cannot be recovered. Make sure that your program examines the return value of this function and handles all of the errors.

Syntax

```
sResult$ = SQL_ErrorQuickAll
```

Parameters

None.

Return Values

This function returns a string that contains all of the Error Messages in the SQL Tools Error Stack. The individual errors are usually delimited by the "pipe" symbol ("|").

Remarks

If the string that is returned by this function contains more than one Error Message, the individual Error Messages will be delimited with the pipe symbol ("|") unless the `SQL_SetOptionStr(OPT_ROW_DELIMITER)` function has been used to specify a different "row" delimiter.

Each individual Error Message will be formatted in the manner described for the `SQL_ErrorQuickOne` function.

Diagnostics

None.

Example

```
SQL_MsgBox SQL_ErrorQuickAll, MSGBOX_OK
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorQuickOne

Summary

Returns a string that contains the oldest Error Message in the SQL Tools Error Stack, and automatically removes that Error Message from the stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

Once an error message has been cleared, it cannot be recovered. Make sure that your program examines the return value of this function and handles the error.

Syntax

```
sResult$ = SQL_ErrorQuickOne
```

Parameters

None.

Return Values

If there are no Error Messages in the SQL Tools Error Stack, this function returns an empty string.

If there are one or more Error Messages in the stack, the oldest error will be returned by this function in string form. See **Remarks** below for the string's format.

Remarks

Error Messages that are returned by this function will always have the following format:

```
Chars 01-12: SQL_ErrorTime in [square brackets]
Chars 14-37: SQL_ErrorFuncName (see note just below)
Chars 39-41: SQL_ErrorDatabaseNumber
Chars 43-45: SQL_ErrorStatementNumber
Chars 47-49: SQL_ErrorColumnNumber
Chars 51-59: SQL_ErrorNumber
Chars 62-66: SQL_State
Chars 68-77: SQL_ErrorNativeCode
Chars 78-80: reserved; currently always "--"
Chars 82+ : SQL_ErrorText (length varies)
```

It is important to note that this function returns only the first 24 characters of the function name, so the following function names (which are more than 24 characters long) will be truncated where you see the pipe (|) symbol.

```
SQL_ColumnPrivilegeInfoS|tr
SQL_DatabaseDataTypeCoun|t
SQL_DatabaseDataTypeInfo|SInt
SQL_DatabaseDataTypeInfo|Str
```

SQL_DatabaseDataTypeNumb	er
SQL_ProcedureColumnInfoS	Int
SQL_ProcedureColumnInfoS	tr
SQL_ResultColumnBufferPt	r
SQL_ResultColumnDateTime	Part
SQL_SetDatabaseAttribUIn	t
SQL_StatementNativeSynta	x
SQL_StatementParameterCo	unt
SQL_TablePrivilegeInfoSt	r

In most cases this will not interfere with your ability to determine which function produced an error. If you need the function's full name, use the `SQL_ErrorFuncName` function, which always returns the entire name.

Diagnostics

None.

Example

```
SQL_MsgBox SQL_ErrorQuickOne, MSGBOX_OK
```

Typical results...

```
[123456.789] SQL_OpenDatabase      1  -1  -1  999000030
#0030 999000030  -- [Perfect Sync][SQL Tools]Bad Parameter
Value
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorSimulate

Summary

Allows your program to add Error Messages to the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ErrorSimulate(lDatabaseNumber&, _  
                             lStatementNumber&, _  
                             lColumnNumber&, _  
                             sFunctionName$, _  
                             lErrorNumber&, _  
                             sSQLState$, _  
                             lNativeError&, _  
                             sErrorMessage)
```

Parameters

All Parameters

You should refer to the corresponding SQL Tools Error function description for the values that are legal for each parameter. For example, to find out the legal values for the *sFunctionName\$* parameter, see *SQL_ErrorFuncName*.

Return Values

This function returns the new value of *SQL_ErrorCount*, after your error has been added to the stack.

Remarks

You can use this function to simulate errors, and add Error Messages to the SQL Tools Error Stack as if they had been detected by SQL Tools.

Diagnostics

None.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorStatementNumber

Summary

Returns the statement number that is associated with the oldest Error Message in the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ErrorStatementNumber
```

Parameters

None.

Return Values

This function will return negative one (-1) if the oldest Error Message in the SQL Tools Error Stack is not associated with a statement number (as would be the case with a failed attempt to open a database), or a number between one (1) and the maximum statement number specified in the `SQL_Initialize` function. If there are no Error Messages in the SQL Tools Error Stack, this function will return zero (0)

Remarks

See Error Handling in SQL Tools Programs a complete discussion of this function.

Diagnostics

None.

Example

```
Print SQL_ErrorStatementNumber
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorStr

Summary

Returns Error Message values from the SQL Tools Error Stack in a "random access" manner.

Twin

None

Family

Error/Trace Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_ErrorStr(lRecordNumber&, _  
                        lInfoType&)
```

Parameters

lRecordNumber&

This parameter must be a number between one (1) and the number of Error Messages that are currently in the SQL Tools Error Stack. (A better name for this parameter might have been *lErrorMessageNumber&* but that could be confused with other error-related values.)

lInfoType&

See **Remarks** below for valid values.

Return Values

This function returns a string that corresponds to the requested property of the requested Error Message. If an invalid value is specified for either parameter, an empty string is returned.

Remarks

Most SQL Tools error-related functions work with either **1)** the oldest Error Message in the SQL Tools Error Stack, or **2)** the entire Error Stack. Unlike other functions, the `SQL_ErrorStr` function can be used to access any property of any Error Message that is currently in the stack. You can think of the Error Stack as a database table that is normally accessed row-by-row. The `SQL_ErrorStr` function gives your program "random access" to error information.

The *lInfoType&* parameter must have one of the following values:

ERROR_COL

The `SQL_ErrorColumnNumber` value.

ERROR_DB

The `SQL_ErrorDatabaseNumber` value.

ERROR_FUNCTION

The SQL_ErrorFuncName value.

ERROR_NATIVE_CODE

The SQL_ErrorNativeCode value.

ERROR_NUMBER

The SQL_ErrorNumber value.

ERROR_STMT

The SQL_ErrorStatementNumber value.

ERROR_SQL_STATE

The SQL_State value.

ERROR_TEXT

The SQL_ErrorText value.

ERROR_TIME

The SQL_ErrorTime value.

Diagnostics

None.

Example

```
Print SQL_ErrorStr(3,ERROR_FUNCTION)
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorText

Summary

Returns the text message that is associated with the oldest Error Message in the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_ErrorText
```

Parameters

None.

Return Values

This function will return a string that contains the text message that is associated with the oldest Error Message in the SQL Tools Error Stack.

If the text came from an error that your program simulated (see `SQL_ErrorSimulate`), the text was formatted by your program.

If the text came from an ODBC Driver, the ODBC Driver Manager, or SQL Tools, it will usually have the following format:

```
[Company][Program]Message
```

For example, here is a typical message:

```
[Microsoft][ODBC Driver Manager] Information type out of range
```

And here is a typical message from SQL Tools:

```
[Perfect Sync][SQL Tools] Bad Parameter Value
```

Remarks

See Error Handling in SQL Tools Programs for a complete discussion of this function.

Diagnostics

None.

Example

```
SQL_MsgBox SQL_ErrorText, MSGBOX_OK
```

Typical results...

[Perfect Sync][SQL Tools] DB Not Open

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_ErrorTime

Summary

Returns the time that the oldest Error Message in the SQL Tools Error Stack was added to the stack, in seconds and fractional seconds past midnight, in string form.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_ErrorTime
```

Parameters

None.

Return Values

This function returns a string in the format "#####.###" which represents the number of seconds and fractional seconds past midnight that the oldest Error Message in the SQL Tools Error Stack was originally added to the stack.

Remarks

See Error Handling in SQL Tools Programs for a complete discussion of this function.

Diagnostics

None.

Example

```
Print SQL_ErrorTime
```

Typical results...

```
123456.789
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_Fetch

Summary

Retrieves one row of data (or one rowset if a MultiRow cursor is being used) from the result set that was generated by a SQL statement.

Twin

SQL_FetchResult

Family

Statement Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_Fetch(lRowNumber&)
```

Parameters

lRowNumber&

A number that specifies a specific row number (such as row 13), or a constant that specifies a "named" row: `FIRST_ROW`, `NEXT_ROW`, `PREV_ROW`, or `LAST_ROW`.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH__INFO` if a row of data (or a rowset) is successfully retrieved from the result set.

If there is no data to retrieve (for example, if the result set is empty or if the final row of data has already been retrieved), this function returns `SQL_NO_DATA` (value 100).

If an error is detected, this function can return other Error Codes.

Remarks

The most common (and most widely-supported) use of this function is...

```
SQL_Fetch NEXT_ROW
```

...which operates in a manner that is similar to the `BASIC Line Input` function. If no rows have yet been read from the result set, the first row is automatically retrieved. If one or more rows have already been retrieved from the result set, the next row is retrieved from the current cursor position.

If the ODBC driver that you are using supports them, you can also use the following options:

<code>SQL_Fetch PREV_ROW</code>	(retrieves the <i>previous</i> row)
<code>SQL_Fetch FIRST_ROW</code>	(retrieves row number one)
<code>SQL_Fetch LAST_ROW</code>	(retrieves the final row)
<code>SQL_Fetch lRowNumber&</code>	(retrieves row <i>lRowNumber&</i>)

You can, of course, experimentally determine whether or not the various `SQL_Fetch` options are supported by your ODBC driver.

Or you can use the `SQL_StmtAttrib(DB_STATIC_CURSOR_ATTRIBUTES2)` function to programmatically determine whether or not an option is available. (If you have used the `SQL_StmtMode(STMT_ATTR_CURSOR_TYPE)` function to select a non-static cursor, you should use the appropriate `DB_type_CURSOR_ATTRIBUTES2` option.)

For more information about the `SQL_Fetch` function, see [Fetching Rows From Result Sets](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD Then Exit Do
Loop
```

Driver Issues

Some drivers do not support options other than `NEXT_ROW`.

Speed Issues

`SQL_Fetch NEXT_ROW` can be *significantly* faster than any of the other options. If you can limit your program to using `NEXT_ROW` you will probably obtain the maximum speed that is available from your ODBC driver. In fact, if you limit your program to `NEXT_ROW` you can use the `SQL_StmtMode` function to actually *disable* other types of fetching, and your program will (usually) run faster.

See Also

[Fetching Rows From Result Sets \(Basic\)](#), [Fetching Rows From Result Sets \(Advanced\)](#), [Relative Fetches](#)

SQL_FetchPos

Summary

Returns the current row number (the "position") of a Result Set that was created with a *SELECT* statement, i.e. the row number of the most recent fetch operation.

Twin

SQL_FetchPosition

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warnings

Certain types of fetch operations can cause the *SQL_FetchPos* function to lose track of the current row number, and your program may need to use the *SQL_SyncFetchPos* function to re-synchronize the row-counting system. See **Remarks** below for complete information.

This function should be used only with Static Cursors. See **Remarks** below for complete information.

Syntax

```
lResult& = SQL_FetchPos
```

Parameters

None.

Return Values

This will function return one of the following values:

- 1) A number greater than zero, indicating the row number of the most recent fetch operation.
- 2) Zero (0) if no fetch operation has yet been performed on the statement, or if the most recent fetch operation placed the statement at the Beginning Of Data point (the point before row 1). Zero will also be returned if the current statement is not a *SELECT* statement.
- 3) Negative one (-1) if the last fetch operation failed because the statement reached the End Of Data point. This indicates that "there is no *current* row because the last fetch failed".
- 4) Negative two (-2) if the current row is not known. See **Remarks** below.

Remarks

The *SQL_FetchPos* and *SQL_FetchPosition* function automatically track your program's use of the *SQL_Fetch*, *SQL_FetchResult*, *SQL_FetchRel*, and *SQL_FetchRelative* functions, in order to keep track of each *SELECT* statement's current row number. Whenever your program uses one of those functions, SQL Tools will attempt to determine the fetch operation's effect on the result set, and which row number was fetched.

IMPORTANT NOTE: We say "attempt" because certain types of fetch operations will cause SQL Tools to lose track of the current row number. For example, if you use `SQL_Fetch LAST_ROW` the very last row of the result set will be fetched, but there is no way for SQL Tools to find out the *row number* of that row. (ODBC drivers do not provide a function that reliably returns the number of rows in a result set. For more information see Why You Can't Use `SQL_ResRowCount` for `SELECT` Statements.)

These are the four operations that can cause SQL Tools to lose track of the current row number:

`SQL_Fetch LAST_ROW` (see above)

`SQL_Fetch row number` using a row number that does not exist, i.e. that is larger than the highest-numbered row in the result set. This effectively moves the statement to the End Of Data position.

`SQL_FetchRel` using a positive offset value that causes the fetch to fail. For example, using an offset of `+10` when the result set only has two items. This too moves the statement to the End Of Data position.

`SQL_FetchRel` using a bookmark. This moves the cursor to some point in the middle of the result set, but it does not allow SQL Tools to determine the row number.

If any of those functions (or their verbose equivalents) are used, SQL Tools will lose track of the current row number and the `SQL_FetchPos` function will begin returning negative two (`-2`). So before performing those operations, you may want to determine the row number yourself. For example, if your program has *counted* the rows in the result set, it already knows the row number that be fetched by a `LAST_ROW` operation. If that is the case, you can use `SQL_Fetch LAST_ROW` and then use the `SQL_SyncFetchPos` function to *tell* SQL Tools what the row number is *after* the fetch. Doing that will re-synchronize SQL Tools with the `SELECT` statement, and allow you to continue using the `SQL_FetchPos` function normally.

Another method of re-synchronizing the row count is to perform a fetch to a known row number. For example, if SQL Tools loses track of the row number but your program then performs a `SQL_Fetch FIRST_ROW` operation, SQL Tools will automatically re-synchronize to row 1. The same is true for "absolute" fetches that return a specific row number, such as `SQL_Fetch 2`, as long as the fetch is successful.

Tip: If your program uses bookmarks, each time you use the `SQL_Bkmk` function you should also use the `SQL_FetchPos` function to get the row number that goes *with* the bookmark string. Then your program should *store both the bookmark and the row number*. That way, when you use `SQL_FetchRel` to return to that bookmark you can immediately use `SQL_SyncFetchPos` to re-synchronize the row number.

IMPORTANT NOTE: *SQL Tools uses Static Cursors by default, so unless you have purposely created a Dynamic Cursor you don't need to be concerned about this next potential problem.* (For information about Static and Dynamic Cursors, see the section of this document that is titled Problems with Scrollable Cursors.) If you use the `SQL_FetchPos` function with a *Dynamic* Cursor, it is very likely to provide

incorrect row numbers. For example, let's say that you have a Dynamic result set that returned two rows of data. You fetch the first row, and the `SQL_FetchPos` function returns 1 to indicate that the first row was fetched. But imagine that in the meantime, another program has added a row to the database that matches your `SELECT` statement. Because the cursor is Dynamic, the new row might be added to your result set before the first row, before the second row, or after the second row. So "row 1" isn't necessarily the first row any more, and the return value of the `SQL_FetchPos` function is no longer valid. *Row Numbers are not meaningful with Dynamic Cursors, because unlike Static Cursors, the Row Numbers can change!*

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "row 1". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD Then Exit Do
    sRowContents$ = SQL_ResColText(ALL_COLS)
    lRowNumber = SQL_FetchPos
Loop
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Sets

SQL_FetchPosition

Summary

Returns the current row number (the "position") of a Result Set that was created with a [SELECT](#) statement, i.e. the row number of the most recent fetch operation.

Twin

[SQL_FetchPos](#)

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

Certain types of fetch operations can cause the [SQL_FetchPosition](#) function to lose track of the current row number, and your program may need to use the [SQL_SyncFetchPosition](#) function to re-synchronize the row-counting system. See the **Remarks** section of [SQL_FetchPos](#) for complete information.

This function should be used only with Static Cursors. See the **Remarks** section of [SQL_FetchPos](#) for complete information.

Syntax

```
lResult& = SQL_FetchPosition(lDatabaseNumber&, _  
                             lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

This function returns values that are identical to those returned by the [SQL_FetchPos](#) function. To avoid errors when this document is updated, information that is common to both functions is not duplicated here.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, [SQL_FetchPosition](#) is identical to [SQL_FetchPos](#). To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to [SQL_FetchPosition](#) is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

This function returns diagnostic information that is identical to that returned by the [SQL_FetchPos](#) function. To avoid errors when this document is updated, information that is common to both functions is not duplicated here.

Example

See [SQL_FetchPos](#).

Driver Issues

None.

Speed Issues

None.

See Also

Result Sets

SQL_FetchRel

Summary

Performs a relative fetch operation on a result set, according to the number of rows specified by the *lOffset&* parameter. This function is also used for bookmark fetches, which can have an optional *lOffset&* value.

Twin

SQL_FetchRelative

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_FetchRel(sBookmark$, _  
                        lOffset&)
```

Parameters

sBookmark\$

An empty string, or a bookmark string from the SQL_Bkmk function.

lOffset&

The row where the fetch operation should take place, in terms of the number of rows **1)** before or after the current row, or **2)** before or after the row specified by the *sBookmark\$* parameter.

Return Values

This function returns Error Codes that are identical to those returned by the SQL_Fetch function. To avoid errors when this document is updated, information that is common to both functions is not duplicated here.

Remarks

For background information, see SQL_Fetch and Relative Fetch Operations.

A "relative" fetch operation is a fetch which is based on an "offset" value, which can be zero (0) or a positive or negative number of rows.

There are two basic ways to use this function:

If the *sBookmark\$* parameter is an empty string, this function fetches the row that is *lOffset&* rows from the current cursor position. For example, if the cursor was located on row 100 of a result set and a "+10" relative fetch was performed, row 110 would be fetched. If a "-3" fetch was then performed, row 107 would be retrieved. Relative fetches that do not use bookmarks are always performed relative to the cursor location *at the time of the operation*, not relative to the original cursor location.

If you attempt to perform a relative fetch that refers to a row that is before the beginning or after the end of the result set, the SQL_FetchRel function will return SQL_ERROR_NO_DATA and the SQL_EOD function will return Logical True until a valid row is fetched. Also, the SQL_FetchPos function will return negative two (-2).

Not all ODBC drivers support relative fetches. You can determine the types of fetches that your driver supports **1)** experimentally, or **2)** by examining the result of the `SQL_DBInfoUInt (DB_type_CURSOR_ATTRIBUTES1)` function, where *type* is the type of cursor being used (`STATIC`, `DYNAMIC`, etc).

If the *sBookmark\$* string is not an empty string, it must contain a string that was produced by the `SQL_Bkmk` or `SQL_Bookmark` function. If an invalid string is used, Application Errors are possible. If a valid bookmark string is used, this function will fetch the row that is *IOffset&* rows from the bookmarked row. For example, if an *IOffset&* value of zero (0) was used, the originally-bookmarked row would be retrieved. If a value of +1 was used for *IOffset&*, the row immediately after the bookmarked row would be retrieved. If a value of --6 was used, the row that was six rows before the bookmark would be retrieved. Relative fetches that use bookmarks are always performed relative to the bookmark's location, not the current cursor location.

Please note that the use of bookmark-based fetches affects the `SQL_FetchPos` function's ability to determine the current row number.

If you attempt to perform a relative-bookmark fetch that refers to a row that is before the beginning or after the end of the result set, the `SQL_FetchRel` function will return `SQL_ERROR_NO_DATA` and the `SQL_EOD` function will return Logical True until a valid row is fetched.

Not all ODBC drivers support bookmark-based fetches, and others may support bookmark fetches but require that the *IOffset&* value be zero (0). You can determine the types of fetches that your driver supports **1)** experimentally, or **2)** by examining the result of the `SQL_DBInfoUInt (DB_type_CURSOR_ATTRIBUTES1)` function, where *type* is the type of cursor being used (`STATIC`, `DYNAMIC`, etc).

For more information, see Bookmarks.

C, C++, AND DELPHI PROGRAMMERS PLEASE NOTE: Because they can contain ASCII character zero (`CHR$(0)`), bookmarks must be passed to this function as OLE strings, not ASCIIZ strings. BASIC programmers do not need to worry about this distinction.

Diagnostics

This function returns diagnostic information that is identical to that returned by the `SQL_Fetch` function. To avoid errors when this document is updated, information that is common to both functions is not duplicated here.

Example

```
'Get the row that is 100 rows
'after the current cursor location.
SQL_FetchRel "", 100

'(For an example of using SQL_FetchRel
'with bookmarks, see SQL_Bkmk.)
```

Driver Issues

The Microsoft Access 97 ODBC Driver does not support bookmarks if ODBC 2.0 behavior is used, i.e. when an *IODBCVersion&* value of 2 is used for the `SQL_Initialize` function.

Speed Issues

See Bookmarks for a discussion of speed issues related to bookmarks.

See Also

Relative Fetches, Bookmarks

SQL_FetchRelative

Summary

Performs a relative fetch operation on a result set, according to the number of rows specified by the *IOffset*& parameter. This function is also used for bookmark fetches, which can have an optional *IOffset*& value.

Twin

SQL_FetchRel

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_FetchRelative(lDatabaseNumber&, _  
                             lStatementNumber&, _  
                             sBookmark$, _  
                             lOffset&)
```

Parameters

lDatabaseNumber& and *lStatementNumber*&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_FetchRel for complete details.

Return Values

This function returns Error Codes that are identical to those returned by the SQL_Fetch function. To avoid errors when this document is updated, information that is common to both functions is not duplicated here.

Remarks

Except for the *lDatabaseNumber*& and *lStatementNumber*& parameters, SQL_FetchRelative is identical to SQL_FetchRel. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_FetchRelative is shown below.

For more information about using *lDatabaseNumber*& and *lStatementNumber*& in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

This function returns diagnostic information that is identical to that returned by the SQL_FetchRel function. To avoid errors when this document is updated, information that is common to both functions is not duplicated here.

Example

See SQL_FetchRel.

Driver Issues

Our test programs indicate that the Microsoft Access 97 Driver does not support bookmarks if ODBC 2.0 behavior is used, i.e. when an *IODBCVersion*& value of 2 is used for the *SQL_Initialize* function.

Speed Issues

See Using Bookmarks for a discussion of speed issues related to bookmarks.

See Also

Relative Fetches

SQL_FetchResult

Summary

Retrieves one row of data (or a rowset from a MultiRow cursor) from the result set that was generated by a SQL statement.

Twin

SQL_Fetch

Family

Statement Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_FetchResult(lDatabaseNumber&, _  
                           lStatementNumber&, _  
                           lRowNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function `SQL_Fetch` for complete details.

Return Values

See twin function `SQL_Fetch` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_FetchResult` is identical to `SQL_Fetch`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_FetchResult` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_Fetch` for complete details.

Example

```
SQL_FetchResult 1, 2, NEXT_ROW
```

Driver Issues

See twin function `SQL_Fetch` for complete details.

Speed Issues

See twin function `SQL_Fetch` for complete details.

See Also: `SQL_FetchRelative`

SQL_ForeignKeyCount

Summary

Returns the total number of columns that are used to define Foreign Keys for a table.

Twin

SQL_FornKeyCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ForeignKeyCount(lDatabaseNumber&, _  
                                lTableNumber&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_FornKeyCount for complete details.

Return Values

See twin function SQL_FornKeyCount for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_ForeignKeyCount is identical to SQL_FornKeyCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ForeignKeyCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_FornKeyCount for complete details.

Example

See twin function SQL_FornKeyCount.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Foreign Keys

SQL_ForeignKeyInfoSInt

Summary

Provides information about a column that is used as a Foreign Key, in numeric form.

Twin

SQL_FornKeyInfoSInt

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ForeignKeyInfoSInt(lDatabaseNumber&, _  
                                   lTableNumber&, _  
                                   lKeyNumber&, _  
                                   lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_FornKeyInfoSInt for complete details.

Return Values

See twin function SQL_FornKeyInfoSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_ForeignKeyInfoSInt is identical to SQL_FornKeyInfoSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_FornKeyInfoSInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_FornKeyInfoSInt for complete details.

Example

See twin function SQL_FornKeyInfoSInt.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also

Foreign Keys

SQL_ForeignKeyInfoStr

Summary

Provides information about a Foreign Key, in string form.

Twin

SQL_FornKeyInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_ForeignKeyInfoStr(lDatabaseNumber&, _  
                                lTableNumber&, _  
                                lKeyNumber&, _  
                                lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_FornKeyInfoStr for complete details.

Return Values

See twin function SQL_FornKeyInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_ForeignKeyInfoStr is identical to SQL_FornKeyInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ForeignKeyInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_FornKeyInfoStr for complete details.

Example

See twin function SQL_FornKeyInfoStr.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also

Foreign Keys

SQL_FornKeyCount

Summary

Returns the total number of columns that are used to define Foreign Keys for a table.

Twin

SQL_ForeignKeyCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_FornKeyCount(lTableNumber&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

Return Values

If a valid *lTableNumber&* is used, this function will return the number of columns that are used by the Foreign Keys that are associated with the specified table.

Remarks

A *Foreign Key* is a column (or a set of columns) in one table which matches the *Primary Key* in another table.

This function returns a value which indicates the total number of columns that are used for Foreign Keys. This is not *necessarily* the same as the total number of Foreign Keys. For example, if a table has two foreign keys, each of which uses one column, this function would return two (2). On the other hand, if a table has two foreign keys, each of which requires two columns to create a unique key value, this function would return four (4).

See Foreign Keys and Primary Keys for more information.

Also see `SQL_FornKeyInfoSInt` and `SQL_FornKeyInfoStr`.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "the table has 1 foreign key". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_FornKeyCount
```

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Foreign Keys

SQL_FornKeyInfoSInt

Summary

Provides information about a column that is used as a Foreign Key, in numeric form.

Twin

SQL_ForeignKeyInfoSInt

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_FornKeyInfoSInt(lTableNumber&, _  
                                lKeyNumber&, _  
                                lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

lKeyNumber&

The number of a Foreign Key, between one (1) and the number returned by the `SQL_FornKeyCount` function.

lInfoType&

The type of information being requested. See **Remarks** below for a list of valid values.

Return Values

If valid parameters are used, this function will return the requested numeric information. Otherwise, zero (0) will be returned.

Remarks

A *Foreign Key* is a column (or a set of columns) in one table that matches a *Primary Key* in another table. The `SQL_FornKeyInfoSInt` function can be used to obtain information about a column that is used in a Foreign Key.

See Foreign Keys and Primary Keys for more information.

Please note that not *all* of the information about a table's Foreign Keys is useful in numeric form. For a list of *lInfoType&* values that can be used to obtain *string* information about a table's Foreign Keys, see `SQL_FornKeyInfoStr`.

To obtain numeric information about a table's columns, use one of the following *lInfoType&* values:

`FKEY_DEFERRABILITY`

This *lInfoType&* will always have one of the following values:

SQL_INITIALLY_DEFERRED
SQL_INITIALLY_IMMEDIATE
SQL_NOT_DEFERRABLE

FKEY_DELETE_RULE

The action that is to be applied to the foreign key when the SQL operation is [DELETE](#).

In the following definitions, the *referenced table* is the table that has the primary key, and the *referencing table* is the table that has the foreign key).

This *InfoType* can have one of the following values.

SQL_CASCADE (When a row in the referenced table is deleted, all of the matching rows in the referencing tables are also deleted.)

SQL_NO_ACTION (The update is rejected if the deletion of a row in the referenced table would cause a "dangling reference" in the referencing table, i.e. if rows in the referencing table would have no counterparts in the referenced table. This was called SQL_RESTRICT in ODBC 2.0.)

SQL_SET_NULL (When one or more rows in the referenced table are deleted, each component of the foreign key of the referencing table is set to Null in all matching rows of the referencing table.)

SQL_SET_DEFAULT (When one or more rows in the referenced table are deleted, each component of the foreign key of the referencing table is set to the applicable default in all matching rows of the referencing table.)

Zero (0) if this *InfoType* is not applicable to the Datasource.

FKEY_SEQ

The column sequence number. If a Foreign Key use more than one column from another table to produce a unique key value, the SQL_ForeignKeyInfoSInt function will return more than one Foreign Key, each with a different column name. This *InfoType* value can be used to determine the order in which those column names are assembled to create the unique key.

FKEY_UPDATE_RULE

The action that is to be applied to the foreign key when the SQL operation is [UPDATE](#).

In the following definitions, the *referenced table* is the table that has the primary key, and the *referencing table* is the table that has the foreign key).

This *InfoType* can have one of the following values.

SQL_CASCADE (When the primary key of the referenced table is updated, the foreign key of the referencing table is also updated.)

SQL_NO_ACTION (The update is rejected if **1**) an update of the primary key

of the referenced table would cause a "dangling reference" in the referencing table (i.e. rows in the referencing table would have no counterparts in the referenced table), or **2**) an update of the foreign key of the referencing table would introduce a value that does not exist as a value of the primary key of the referenced table. This was called `SQL_RESTRICT` action in ODBC 2.0.)

`SQL_SET_NULL` (When one or more rows in the referenced table are updated such that one or more components of the primary key are changed, the components of the foreign key in the referencing table that correspond to the changed components of the primary key are set to Null in all matching rows of the referencing table.)

`SQL_SET_DEFAULT` (When one or more rows in the referenced table are updated such that one or more components of the primary key are changed, the components of the foreign key in the referencing table that correspond to the changed components of the primary key are set to the applicable default values in all matching rows of the referencing table. Null will be returned if this is not applicable to the datasource.)

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Foreign Keys.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Table Column Info Family

SQL_FornKeyInfoStr

Summary

Provides information about a column that is used as a Foreign Key, in string form.

Twin

SQL_ForeignKeyInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_FornKeyInfoStr(lTableNumber&, _  
                               lKeyNumber&, _  
                               lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

lKeyNumber&

The number of a Foreign Key, between one (1) and the number returned by the `SQL_FornKeyCount` function.

lInfoType&

The type of information being requested. See **Remarks** below for a list of valid values.

Return Values

If valid parameters are used, this function will return the requested string information. Otherwise, an empty string will be returned.

Remarks

A *Foreign Key* is a column (or a set of columns) in a table that match a *Primary Key* in another table. The `SQL_FornKeyInfoStr` function can be used to obtain information about a column that is used in a Foreign Key.

See Foreign Keys and Primary Keys for more information.

Please note that not *all* of the information about a table's Foreign Keys is useful in string form. For a list of *lInfoType&* values that can be used to obtain *numeric* information about a table's Foreign Keys, see `SQL_FornKeyInfoSInt`.

To obtain string information about a table's columns, use one of the following *lInfoType&* values:

FKEY_NAME

The name of the Foreign Key.

FKEY_TABLE_CATALOG,
FKEY_TABLE_SCHEMA,
FKEY_TABLE_NAME, and
FKEY_COLUMN_NAME

The catalog, schema, table, and column names of the Foreign Key.

PKEY_NAME

The name of the Primary Key.

PKEY_TABLE_CATALOG,
PKEY_TABLE_SCHEMA,
PKEY_TABLE_NAME, and
PKEY_COLUMN_NAME

The catalog, schema, table, and column names of the Primary Key (in another table) to which the Foreign Key applies.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Foreign Keys.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Table Column Info Family

SQL_FuncAvail

Summary

Reports whether or not your ODBC driver supports a given function.

Twin

SQL_FunctionAvailable

Family

Database Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_FuncAvail(lFunctionID&)
```

Parameters

lFunctionID&

One of the 78 different `API_` constants. See **Remarks** below for details.

Return Values

This function returns Logical True (-1) if the specified function is supported by your ODBC driver, or False (0) if it is not.

Remarks

With very few exceptions, if an ODBC driver supports a function, SQL Tools allows you to use that function. It is sometimes necessary, therefore, to determine whether or not and ODBC driver supports a certain function.

For example, you can use `SQL_FuncAvail(API_SQLTABLEPRIVILEGES)` to determine whether or not your ODBC driver supports "table privileges". If it does not (i.e. if the function returns False) then the SQL Tools functions that are related to table privileges (`SQL_TblPrivCount`, etc.) are effectively disabled by the driver.

Another example: whenever SQL Tools opens a database, it automatically uses the `SQL_FuncAvail(API_SQLFETCHSCROLL)` function to determine whether or not to allow your programs to attempt "fetch scrolling", i.e. the use of `SQL_Fetch` without `NEXT_ROW`.

Generally speaking, it is safe to assume that virtually all ODBC drivers support the list of `API_` constants that are listed under "ODBC CORE COMPLIANCE" below. If your program uses more advanced features (Level 1 or 2) and is likely to be used with more than one ODBC driver, you can use the `SQL_FuncAvail` function to programmatically determine whether or not certain features are supported.

We suggest that you consult the Microsoft ODBC Software Developer Kit for precise information about the API-level functions that are affected by each of these values:

ODBC CORE COMPLIANCE...

API_SQLALLOCCONNECT
API_SQLALLOCENV
API_SQLALLOCSTMT
API_SQLBINDCOL
API_SQLBULKOPERATIONS
API_SQLCANCEL
API_SQLCOLATTRIBUTE
API_SQLCONNECT
API_SQLDESCRIBECOL
API_SQLDISCONNECT
API_SQLError
API_SQLEXCEDIRECT
API_SQLEXECUTE
API_SQLFETCH
API_SQLFREECONNECT
API_SQLFREEENV
API_SQLFREESTMT
API_SQLGETCURSORNAME
API_SQLNUMRESULTCOLS
API_SQLPREPARE
API_SQLROWCOUNT
API_SQLSETCURSORNAME
API_SQLSETPARAM
API_SQLTRANSACT

COMPLIANCE LEVEL 1 AND ABOVE...

API_SQLCOLUMNS
API_SQLDRIVERCONNECT
API_SQLGETCONNECTOPTION
API_SQLGETDATA
API_SQLGETFUNCTIONS
API_SQLGETINFO
API_SQLGETSTMTOPTION
API_SQLGETTYPEINFO
API_SQLPARAMDATA
API_SQLPUTDATA
API_SQLSETCONNECTOPTION
API_SQLSETSTMTOPTION
API_SQLSPECIALCOLUMNS
API_SQLSTATISTICS
API_SQLTABLES

COMPLIANCE LEVEL 2 AND ABOVE...

API_LOADBYORDINAL (**NOT SUPPORTED BY ODBC 3.x+**)
API_SQLALLOCHANDLE
API_SQLALLOCHANDLESTD
API_SQLBINDPARAM
API_SQLBINDPARAMETER
API_SQLBROWSECONNECT
API_SQLCLOSECURSOR
API_SQLCOLUMNPRIVILEGES
API_SQLCOPYDESC
API_SQLDATASOURCES

API_SQLDESCRIBEPARAM
 API_SQLDRIVERS
 API_SQLENDTRAN
 API_SQLEXTENDEDFETCH
 API_SQLFETCHSCROLL
 API_SQLFOREIGNKEYS
 API_SQLFREEHANDLE
 API_SQLGETCONNECTATTR
 API_SQLGETDESCFIELD
 API_SQLGETDESCREC
 API_SQLGETDIAGFIELD
 API_SQLGETDIAGREC
 API_SQLGETENVATTR
 API_SQLGETSTMTATTR
 API_SQLMORERESULTS
 API_SQLNATIVESQL
 API_SQLNUMPARAMS
 API_SQLPARAMOPTIONS
 API_SQLPRIMARYKEYS
 API_SQLPROCEDURECOLUMNS
 API_SQLPROCEDURES
 API_SQLSETCONNECTATTR
 API_SQLSETDESCFIELD
 API_SQLSETDESCREC
 API_SQLSETENVATTR
 API_SQLSETPOS
 API_SQLSETSCROLLOPTIONS
 API_SQLSETSTMTATTR
 API_SQLTABLEPRIVILEGES

Diagnostics

This function does not return Error Codes, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```

If SQL_FuncAvail(API_SQLPROCEDURES) = 0 Then
    'ODBC driver does not support
    'stored procedures.
End If
  
```

Driver Issues

None. All ODBC drivers are required to support this function.

Speed Issues

None.

See Also

Database Info/Attrib Family

SQL_FunctionAvailable

Summary

Reports whether or not your ODBC driver supports a given function.

Twin

SQL_FuncAvail

Family

Database Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_FunctionAvailable(lDatabaseNumber&, _  
                                lFunctionID&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_FuncAvail for complete details.

Return Values

See twin function SQL_FuncAvail for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_FunctionAvailable is identical to SQL_FuncAvail. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_FunctionAvailable is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_FuncAvail for complete details.

Example

See twin function SQL_FuncAvail.

Driver Issues

See twin function SQL_FuncAvail for complete details.

Speed Issues

None.

See Also

Database Info/Attrib Family

SQL_GetAutoCols

Summary

Refreshes cached information about a table's AutoColumns.

Twin

SQL_GetAutoColumns

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetAutoCols(lTableNumber&)
```

Parameters

lTableNumber&

The table number of the table that should have its AutoColumn information refreshed, between one (1) and the number that is returned by the SQL_TblCount function.

Return Values

This function returns SUCCESS or SUCCESS_WITH_INFO if the AutoColumn information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'refresh the AutoColumn  
'info for table #7.  
SQL_GetAutoCols 7
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see Cached Information.

See Also

AutoColumns

SQL_GetAutoColumns

Summary

Refreshes cached AutoColumn information. (See Cached Information.)

Twin

SQL_GetAutoCols

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetAutoColumns(lDatabaseNumber&, _  
                               lTableNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_GetAutoCols for complete details.

Return Values

See twin function SQL_GetAutoCols for complete details.

Remarks

Except for the *lDatabaseNumber&* parameters, SQL_GetAutoColumns is identical to SQL_GetAutoCols. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_GetAutoColumns is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'refresh the autocolumn info for database 1, table 8.  
SQL_GetAutoColumns 1, 8
```

Driver Issues: None.

Speed Issues

For a general discussion of speed issues related to Info functions, see Cached Information.

See Also: AutoColumns

SQL_GetColPrivs

Summary

Refreshes cached Column Privilege information. (See Cached Information.)

Twin

SQL_GetColumnPrivileges

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetColPrivs(lTableNumber&, _  
                           lColumnNumber&)
```

Parameters

lTableNumber&

The table number that contains the column that should have its column privilege information refreshed, between one (1) and the number that is returned by the SQL_TblCount function.

lColumnNumber&

The column number of the column that should have its column privilege information refreshed, between one (1) and the number that is returned by the SQL_TblColCount function.

Return Values

This function returns SUCCESS or SUCCESS_WITH_INFO if the column-privilege information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'refresh the privilege information  
'for table 17, column 88.  
SQL_GetColPrivs 17, 88
```

Driver Issues: None.

Speed Issues: For a general discussion of speed issues related to Info functions, see Cached Information.

See Also

Column Privileges

SQL_GetColumnPrivileges

Summary

Refreshes cached Column Privilege information. (See Cached Information.)

Twin

SQL_GetColPrivs

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetColumnPrivileges(lDatabaseNumber&, _  
                                   lTableNumber&, _  
                                   lColumnNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_GetColPrivs for complete details.

Return Values

See twin function SQL_GetColPrivs for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_GetColumnPrivileges is identical to SQL_GetColPrivs. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_GetColumnPrivileges is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'refresh the table-column-privilege  
'info for database 1, table 10.  
SQL_GetColumnPrivileges 1, 10
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Column Privileges](#)

SQL_GetDatabaseDataTypes

Summary

Refreshes cached Datasource-dependent Data Type information. (See Cached Information.)

Twin

SQL_GetDBDataTypes

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetDatabaseDataTypes(lDatabaseNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

Return Values

See twin function SQL_GetDBDataTypes for complete information.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_GetDatabaseDataTypes is identical to SQL_GetDBDataTypes. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_GetDatabaseDataTypes is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

For a general discussion, see Cached Information.

Diagnostics

See twin function SQL_GetDBDataTypes for complete information.

Example

```
'refresh the data type info for database 2:  
SQL_GetDatabaseDataTypes 2
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see Cached Information.

See Also: Datasource-dependent Data Types

SQL_GetDataSources

Summary

Refreshes cached Datasource information. (See Cached Information.)

Twin

None.

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetDataSources
```

Parameters

None.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the Datasource information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
SQL_GetDataSources
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see Cached Information.

See Also

`SQL_DatasourceInfoStr`, `SQL_DatasourceCount`

SQL_GetDBDataTypes

Summary

Refreshes cached Datasource-dependent Data Type information. (See Cached Information.)

Twin

SQL_GetDatabaseDataTypes

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetDBDataTypes
```

Parameters

None.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the Datasource-dependent Data Type information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
SQL_GetDBDataTypes
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see Cached Information.

See Also

Datasource-dependent Data Types

SQL_GetDrivers

Summary

Refreshes cached ODBC Driver information. (See Cached Information.)

Twin

None

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetDrivers
```

Parameters

None.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the ODBC Driver information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
SQL_GetDrivers
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see Cached Information.

See Also

ODBC Drivers

SQL_GetForeignKeys

Summary

Refreshes cached information about a table's Foreign Keys. (See Cached Information.)

Twin

SQL_GetFornKeys

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetForeignKeys(lDatabaseNumber&, _  
                               lTableNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_GetFornKeys for complete details.

Return Values

See twin function SQL_GetFornKeys for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_GetForeignKeys is identical to SQL_GetFornKeys. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_GetForeignKeys is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'refresh the foreign key info for  
'database 1, table 6.  
SQL_GetForeignKeys 1,6
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Foreign Keys](#)

SQL_GetFornKeys

Summary

Refreshes cached information about a table's Foreign Keys. (See [Cached Information](#).)

Twin

SQL_GetForeignKeys

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetFornKeys(lTableNumber&)
```

Parameters

lTableNumber&

The number of the table that should have its Foreign Key information refreshed, between one (1) and the number returned by the `SQL_TblCount` function.

Return Values

This function returns `SUCCESS` if the table's Foreign Key information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the foreign key  
'info for table #928.  
SQL_GetFornKeys 928
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Foreign Keys](#)

SQL_GetIndexes

Summary

Refreshes cached information about a table's Indexes. (See Cached Information.)

Twin

SQL_GetIndexes

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetIndexes(lDatabaseNumber&, _  
                           lTableNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function `SQL_GetIndexes` for complete details.

Return Values

See twin function `SQL_GetIndexes` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameters, `SQL_GetIndexes` is identical to `SQL_GetIndexes`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_GetIndexes` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the table-index info for database 1, table 4  
SQL_GetIndexes 1, 4
```

Driver Issues: None.

Speed Issues: For a general discussion of speed issues related to Info functions, see Cached Information.

See Also: Indexes

SQL_GetIndexes

Summary

Refreshes cached information about a table's Indexes. (See Cached Information.)

Twin

SQL_GetIndexes

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetIndexes(lTableNumber&)
```

Parameters

lTableNumber&

The number of the table that should have its Index information refreshed, between one (1) and the number that is returned by the `SQL_TblCount` function.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the table's Index information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see Cached Information.

Please note that the phrase "refreshing a table's indexes" is not meant to imply that this function changes the database in any way. Indexes *themselves* never need to be "refreshed" unless a database is damaged, and this function cannot be used to repair a damaged database. This function simply refreshes the *information* about indexes that SQL Tools has cached *internally*.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the index info for table #2.  
SQL_GetIndexes 2
```

Driver Issues: None.

Speed Issues: For a general discussion of speed issues related to Info functions, see Cached Information.

See Also: Indexes

SQL_GetPrimaryKeys

Summary

Refreshes cached information about a table's Primary Keys. (See [Cached Information](#).)

Twin

SQL_GetPrimKeys

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetPrimaryKeys(lDatabaseNumber&, _  
                               lTableName&)
```

Parameters

lDatabaseNumber&

See [Using Database Numbers and Statement Numbers](#).

All Other Parameters

See twin function `SQL_GetPrimKeys` for complete details.

Return Values

See twin function `SQL_GetPrimKeys` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_GetPrimaryKeys` is identical to `SQL_GetPrimKeys`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_GetPrimaryKeys` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see [Using Database Numbers and Statement Numbers](#).

For a general discussion, see [Refreshing Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the primary key info  
'for database 1, table 7.  
SQL_GetPrimaryKeys 1, 7
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Primary Keys](#)

SQL_GetPrimKeys

Summary

Refreshes cached information about a table's Primary Keys. (See [Cached Information](#).)

Twin

SQL_GetPrimaryKeys

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetPrimKeys(lTableNumber&)
```

Parameters

lTableNumber&

The number of the table that should have its Primary Key information refreshed, between one (1) and the number that is returned by the `SQL_TblCount` function.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the table's Primary Key information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the primary key  
'info for table #17.  
SQL_GetPrimKeys 17
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Primary Keys](#)

SQL_GetProcCols

Summary

Refreshes cached information about the columns that a Stored Procedure uses.
(See [Cached Information](#).)

Twin

SQL_GetProcedureColumns

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetProcCols(lProcedureNumber&)
```

Parameters

lProcedureNumber&

A number between one (1) and the number of Stored Procedures that a database contains, as returned by the `SQL_ProcCount` function. (Keep in mind that if you are refreshing this value you also may need to use `SQL_GetProcs` to refresh the `SQL_ProcCount` value.)

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the Stored Procedure Column information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the column information  
'for stored procedure #3.  
SQL_GetProcCols 3
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Stored Procedures](#)

SQL_GetProcedureColumns

Summary

Refreshes cached information about the columns that a Stored Procedure uses.
(See [Cached Information](#).)

Twin

SQL_GetProcCols

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetProcedureColumns(lDatabaseNumber&, _  
                                   lProcedureNumber&)
```

Parameters

lDatabaseNumber&

See [Using Database Numbers and Statement Numbers](#).

All Other Parameters

See twin function `SQL_GetProcCols` for complete details.

Return Values

See twin function `SQL_GetProcCols` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_GetProcedureColumns` is identical to `SQL_GetProcCols`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_GetProcedureColumns` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see [Using Database Numbers and Statement Numbers](#).

For a general discussion, see [Cached Information](#).

Diagnostics

See twin function `SQL_GetProcCols` for complete details.

Example

```
'Refresh column info for database 1, procedure #2  
SQL_GetProcedureColumns 1, 2
```

Driver Issues: None.

Speed Issues: For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also: [Stored Procedure](#)

SQL_GetProcedures

Summary

Refreshes cached information about the Stored Procedures that a database contains.
(See [Cached Information](#).)

Twin

`SQL_GetProcs`

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetProcedures(lDatabaseNumber&)
```

Parameters

lDatabaseNumber&

See [Using Database Numbers and Statement Numbers](#).

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the Stored Procedure information is successfully refreshed, or an Error Code if it is not.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_GetProcedures` is identical to `SQL_GetProcs`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_GetProcedures` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see [Using Database Numbers and Statement Numbers](#).

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Get the procedures that are stored in database #1.  
SQL_GetProcedures 1
```

Driver Issues: None.

Speed Issues: For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also: [Stored Procedure](#)

SQL_GetProcs

Summary

Refreshes cached information about the Stored Procedures that a database contains.
(See Cached Information.)

Twin

SQL_GetProcedures

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetProcs
```

Parameters

None.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the Stored Procedure information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
SQL_GetProcs
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see Cached Information.

See Also

Stored Procedure

SQL_GetTableColumns

Summary

Refreshes cached information about the columns that a table contains. (See [Cached Information](#).)

Twin

SQL_GetTblCols

Family

Get Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_GetTableColumns(lDatabaseNumber&, _  
                               lTableNumber&)
```

Parameters

lDatabaseNumber&

See [Using Database Numbers and Statement Numbers](#).

All Other Parameters

See twin function `SQL_GetTblCols` for complete details.

Return Values

See twin function `SQL_GetTblCols` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_GetTableColumns` is identical to `SQL_GetTblCols`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_GetTableColumns` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see [Using Database Numbers and Statement Numbers](#).

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the table-column info  
'for database 1, table #8.  
SQL_GetTableColumns 1, 8
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Tables, Rows, and Columns](#)

SQL_GetTableInfo

Summary

Refreshes cached information about the tables in a database. (See [Cached Information](#).)

Twin

SQL_GetTblInfo

Family

Get Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_GetTableInfo(lDatabaseNumber&)
```

Parameters

lDatabaseNumber&

See [Using Database Numbers and Statement Numbers](#).

Return Values

See twin function `SQL_GetTblInfo` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameters, `SQL_GetTableInfo` is identical to `SQL_GetTblInfo`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_GetTableInfo` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see [Using Database Numbers and Statement Numbers](#).

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the information about  
'the tables in database #3.  
SQL_GetTableInfo 3
```

Driver Issues: None.

Speed Issues: For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also: [Tables, Rows, and Columns](#)

SQL_GetTablePrivileges

Summary

Refreshes cached information about a table's Table Privileges. (See [Cached Information](#).)

Twin

SQL_GetTblPrivs

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetTablePrivileges(lDatabaseNumber&, _  
                                   lTableNumber&)
```

Parameters

lDatabaseNumber&

See [Using Database Numbers and Statement Numbers](#).

All Other Parameters

See twin function `SQL_GetTblPrivs` for complete details.

Return Values

See twin function `SQL_GetTblPrivs` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_GetTablePrivileges` is identical to `SQL_GetTblPrivs`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_GetTablePrivileges` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see [Using Database Numbers and Statement Numbers](#).

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the table-privilege info  
'for database 1, table #9.  
SQL_GetTablePrivileges 1, 9
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Table Privileges](#)

SQL_GetTblCols

Summary

Refreshes cached information about a table's columns. (See [Cached Information](#).)

Twin

SQL_GetTableColumns

Family

Get Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_GetTblCols(lTableNumber&)
```

Parameters

lTableNumber&

The number of the table that should have its column information refreshed, between one (1) and the number that is returned by the `SQL_TblCount` function.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the table's column information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the column information  
'for table #12.  
SQL_GetTblCols 12
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Tables, Rows, and Columns](#)

SQL_GetTblInfo

Summary

Loads information (or refreshes cached information) about a database's tables.

Twin

SQL_GetTableInfo

Family

Get Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_GetTblInfo
```

Parameters

None.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the database's table information is successfully loaded or refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see [Cached Information](#).

If your database contains a very large number of tables, or if you are accessing the database through a slow network connection, or if you are using a relatively slow computer or hard drive, or if your computer does not have enough RAM to allow SQL Tools to store Table Info in memory, this function can take a very long time to execute. And that can cause other SQL Tools Info functions (many of which use `SQL_GetTblInfo` internally) to take a very long time to execute.

For example, one SQL Tools user reported that the `SQL_TblInfoStr` function was "hanging" when in fact it was simply taking a very long time to finish its work. Their database contained well over 20,000 tables, and it took SQL Tools nearly an hour to retrieve, analyze, and cache the requested data.

Fortunately, there are several different ways to speed up the `SQL_GetTblInfo` function.

By default, the `SQL_GetTblInfo` function automatically retrieves information about all types of tables. It is possible to tell SQL Tools to only retrieve Info about certain types of tables by using this code:

```
SQL_SetOptionStr OPT_TABLE_TYPES, types
```

...where *types* is a string that contains one or more table types. For example, using this code:

```
SQL_SetOptionSInt OPT_TABLE_TYPES, "TABLE"
```

...would tell SQL Tools to ignore System Tables, Views, Aliases, and so on. Only information for tables with the type "TABLE" would be retrieved.

You must specify table types in UPPER CASE, and if more than one type is specified you must separate them with commas. Do not add leading or trailing spaces.

IMPORTANT NOTE: You must change the value of the `OPT_TABLE_TYPES` option very early in your program, before your program uses any Info function of any type. Failure to do so will result in the new option setting being ignored. We suggest that you set this option's value before you open a database, to ensure that the requested value will be used whenever information about the database is requested.

It is also possible to use these options...

```
SQL_SetOptionStr OPT_TABLE_NAME, name
SQL_SetOptionStr OPT_TABLE_SCHEMA, schema
SQL_SetOptionStr OPT_TABLE_CATALOG, catalog
```

...to tell the `SQL_GetTblInfo` function to retrieve Info for only one table, one schema, one catalog, or any combination of those values. (Another name for a schema is an "owner", and another name for a catalog is a "qualifier". Consult your database documentation for more information.) Unlike the `OPT_TABLE_TYPES` option, these three options cannot accept comma-delimited lists of values. It is not usually necessary to use these options, but they are provided for special circumstances.

For example, if you have used the `SQL_OpenDB` function to open a Sybase database, you may find that the various SQL Tools Info functions will return information about several other "related" Sybase databases. In that case, you may need to use the `OPT_TABLE_SCHEMA` option to tell SQL Tools to only retrieve the desired information, and ignore all of the other databases.

IMPORTANT NOTE: You must change the values of these `OPT_TABLE` options very early in your program, before your program uses any Info function of any type. Failure to do so will result in the new option settings being ignored. We suggest that you set these values before you open a database, to ensure that the requested values will be used whenever information about the database is requested.

The `SQL_TblInfoStr` function can be used to obtain a table's type, name, schema name and catalog name. During development and testing you may need to use empty strings for all of the `OPT_TABLE` options so that you can obtain the necessary values. Then, when the appropriate type, schema and/or catalog names have been obtained, you can add them to your program as necessary. (In other words, it may be necessary for you to tolerate a two-hour test run during development, in order to obtain the information necessary to make the Info function execute faster.)

For another technique that can be used to speed up the Table Info functions, see `SQL_InfoImport`.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

SQL_GetTblInfo

Driver Issues

None.

Speed Issues

See **Remarks** above.

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Table Metadata](#)

SQL_GetTblPrivs

Summary

Refreshes cached information about a table's Table Privileges. (See Cached Information.)

Twin

SQL_GetTablePrivileges

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetTblPrivs(lTableNumber&)
```

Parameters

lTableNumber&

The number of the table that should have its Table Privilege information refreshed, between one (1) and the number that is returned by the SQL_TblCount function.

Return Values

This function returns SUCCESS or SUCCESS_WITH_INFO if the table's Table Privilege information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see Cached Information.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the privilege info  
'for table #23.  
SQL_GetTblPrivs 23
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see Cached Information.

See Also

Table Privileges

SQL_GetUniqCols

Summary

Refreshes cached information about a table's Unique Columns. (See [Cached Information](#).)

Twin

SQL_GetUniqueColumns

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetUniqCols(lTableNumber&)
```

Parameters

lTableNumber&

The number of the table that should have its Unique Column information refreshed, between one (1) and the number that is returned by the `SQL_TblCount` function.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the table's Unique Column information is successfully refreshed, or an Error Code if it is not.

Remarks

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the unique-column  
'info for table #17.  
SQL_GetUniqCols 17
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Unique Columns](#)

SQL_GetUniqueColumns

Summary

Refreshes cached information about a table's Unique Columns. (See [Cached Information](#).)

Twin

SQL_GetUniqCols

Family

Get Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_GetUniqueColumns(lDatabaseNumber&, _  
                                lTableNumber&)
```

Parameters

lDatabaseNumber&

See [Using Database Numbers and Statement Numbers](#).

All Other Parameters

See twin function `SQL_GetUniqCols` for complete details.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the Unique Column information is successfully refreshed, or an Error Code if it is not.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_GetUniqueColumns` is identical to `SQL_GetUniqCols`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_GetUniqueColumns` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see [Using Database Numbers and Statement Numbers](#).

For a general discussion, see [Cached Information](#).

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Refresh the unique-column info for  
'database 1, table #77.  
SQL_GetUniqueColumns 1,77
```

Driver Issues

None.

Speed Issues

For a general discussion of speed issues related to Info functions, see [Cached Information](#).

See Also

[Unique Columns](#)

SQL_hDatabase

Summary

Provides the ODBC handle of a database.

Twin

SQL_hDB

Family

Handle Family

Availability

SQL Tools Pro Only (see)

Warning

Please see SQL_hDB for several important warnings.

Syntax

```
lResult& = SQL_hDatabase(lDatabaseNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

Return Values

See twin function SQL_hDB for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_hDatabase is identical to SQL_hDB. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_hDatabase is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

None.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

SQL Handles

SQL_hDB

Summary

Provides the ODBC handle of the current database.

Twin

SQL_hDatabase

Family

Handle Family

Availability

SQL Tools Pro Only (see)

Warning

The incorrect use of ODBC handles can cause Application Errors.

Syntax

```
lResult& = SQL_hDB
```

Parameters

None.

Return Values

This function returns a handle value that can be used for ODBC functions that SQL Tools does not support. (Of which there are very few.)

Remarks

In order to use ODBC functions directly, without going through SQL Tools, you will need the handles of **1)** the ODBC Environment, **2)** the various databases that SQL Tools has opened, and **3)** the various statements that SQL Tools has opened. The various `SQL_h` functions can be used to obtain those handles if you wish to implement ODBC features that SQL Tools does not support.

WARNING: SQL Tools supports virtually 100% of the functions that ODBC provides. If an ODBC feature is not supported by SQL Tools, there is probably a very good reason for it, and you should consider whether or not you *really* need to use the feature.

For example, while SQL Tools does support thread-based asynchronous execution of SQL statements, it does not support ODBC-based asynchronous execution. According to the Microsoft ODBC Software Developer Kit, "*In general, applications should execute functions asynchronously only on single-threaded operating systems. On multithread operating systems,*" [such as Windows] "*applications should execute functions on separate threads, rather than executing them asynchronously on the same thread. No functionality is lost if drivers that operate only on multithread operating systems do not support asynchronous execution.*" If you attempt to add support for this feature to SQL Tools, you will probably find that several of the Info function will fail to work properly, and you will have to manually add support for those functions as well.

After all of that, you're probably asking yourself "so why are the `SQL_h` functions even *provided* by SQL Tools?" The primary reason is something called "descriptors". Here is what the ODBC SDK has to say about them: "*An application calling ODBC*

functions need not concern itself with descriptors. No database operation requires that the application gain direct access to descriptors. However, for some applications, gaining direct access to descriptors streamlines many operations. For example, direct access to descriptors provides a way to rebind column data that may be more efficient than calling SQLBindCol again."

Diagnostics

None.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

SQL Handles

SQL_hEnvironment

Summary

Provides the ODBC handle of the ODBC environment.

Twin

None

Family

Handle Family

Availability

SQL Tools Pro Only (see)

Warning

Please see `SQL_hDB` for several important warnings.

Syntax

```
lResult& = SQL_hEnvironment
```

Parameters

None.

Return Values

This function returns the handle of the ODBC Environment.

Remarks

Please see `SQL_hDB` for several important warnings regarding the use of ODBC Handles.

Diagnostics

None.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

SQL Handles

SQL_hParentWindow

Summary

Returns the handle of the window that SQL Tools is currently using for the parent window of various dialog boxes.

Twin

None.

Family

Handle Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_hParentWindow
```

Parameters

None.

Return Values

This function returns the handle of the window that SQL Tools is currently using for the parent window of various dialog boxes, such as those displayed by `SQL_MsgBox`, `SQL_SelectFile`, and `SQL_OpenDB`.

Remarks

Your program can specify the window that should be used as the parent window of various SQL Tools dialog boxes by using the `SQL_SetOptionSInt(OPT_h_PARENT_WINDOW)` function. If you do not set this value, SQL Tools automatically uses the handle of the Windows Desktop.

If you specify a value that is not a window, SQL Tools will not use it.

If you specify the handle of a valid window and SQL Tools accepts the value, but then the window is destroyed, SQL Tools will revert to using the handle of the Windows Desktop.

Effectively, this function returns the handle that SQL Tools will use for the parent window of various dialog boxes *if* the handle is still valid when the dialog box is displayed.

Your program can use the `SQL_hParentWindow` function as a convenience, to allow your program's dialog boxes to have an automatic parent-window-selection feature.

Diagnostics

None.

Example

None. Your program's use of the `SQL_hParentWindow` function will depend entirely on your program's design.

Driver Issues

None.

Speed Issues

None.

See Also

Handle Family

SQL_hStatement

Summary

Provides the ODBC handle of a statement.

Twin

SQL_hStmt

Family

Handle Family

Availability

SQL Tools Pro Only (see)

Warning

Please see SQL_hDB for several important warnings.

Syntax

```
lResult& = SQL_hStatement(lDatabaseNumber&, _  
                           lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

This function returns the ODBC handle of a SQL statement that was opened by SQL Tools.

Remarks

Please see SQL_hDB for several important warnings regarding the use of ODBC Handles.

Diagnostics

None.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

SQL Handles

SQL_hStmt

Summary

Provides the ODBC handle of the current statement.

Twin

SQL_hStatement

Family

Handle Family

Availability

SQL Tools Pro Only (see)

Warning

Please see [SQL_hDB](#) for several important warnings.

Syntax

```
lResult& = SQL_hStmt
```

Parameters

None.

Return Values

This function returns the ODBC handle of a SQL statement that was opened by SQL Tools.

Remarks

Please see [SQL_hDB](#) for several important warnings regarding the use of ODBC Handles.

Diagnostics

None.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

SQL Handles

SQL_IndexCount

Summary

Indicates how many Indexesa table has.

Twin

SQL_IndxCOUNT

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_IndexCount(lDatabaseNumber&, _  
                           lTableNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_IndxCOUNT for complete details.

Return Values

See twin function SQL_IndxCOUNT for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_IndexCount is identical to SQL_IndxCOUNT. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_IndexCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_IndxCOUNT for complete details.

Example

See twin function SQL_IndxCOUNT.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Indexes

SQL_IndexInfoInt

Summary

Provides information about an Index, in numeric form.

Twin

SQL_IndxInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_IndexInfoInt(lDatabaseNumber&, _  
                             lTableNumber&, _  
                             lIndexNumber&, _  
                             lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_IndxInfoStr for complete details.

Return Values

See twin function SQL_IndxInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_IndexInfoInt is identical to SQL_IndxInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_IndexInfoInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_IndxInfoStr for complete details.

Example

See twin function SQL_IndxInfoStr.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also: Indexes

SQL_IndexInfoStr

Summary

Provides information about an Index, in string form.

Twin

SQL_IndxInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_IndexInfoStr(lDatabaseNumber&, _  
                             lTableNumber&, _  
                             lIndexNumber&, _  
                             lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_IndxInfoStr for complete details.

Return Values

See twin function SQL_IndxInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber*& parameter, SQL_IndexInfoStr is identical to SQL_IndxInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_IndexInfoStr is shown below.

For more information about using *lDatabaseNumber*& and *lStatementNumber*& in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_IndxInfoStr for complete details.

Example

See twin function SQL_IndxInfoStr.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also: Indexes

SQL_IdxCount

Summary

Returns the number of Indexes that a table has.

Twin

SQL_IndexCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_IdxCount(lTableNumber&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number that is returned by the SQL_TblCount function.

Return Values

If a valid *lTableNumber&* is used, and if the database is open, this function will return the number of Indexes that are associated with the table. Otherwise, this function will return zero (0).

Remarks

An Index is a structure that is maintained by a database, in order to speed up access to columns that have been indexed.

This function returns the number of Indexes that are associated with a table.

Diagnostics

This function does not return Error Codes because an Error Code like SUCCESS_WITH_INFO (value 1) could be confused with a legitimate return value like "this table has one index". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Indexes.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Indexes

SQL_IdxInfoSInt

Summary

Provides information about an Index, in numeric form.

Twin

SQL_IndexInfoSInt

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_IdxInfoSInt(lTableNumber&, _  
                           lIndexNumber&, _  
                           lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

lIndexNumber&

The number of an Index, between one (1) and the number returned by the `SQL_IdxCount` function.

lInfoType&

The type of information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, this function will return the requested information. Otherwise, zero (0) will be returned.

Remarks

An Index is a structure that is maintained by a database, in order to speed up access to columns that have been indexed.

Please note that not *all* of the information about a table's Indexes is useful in numeric form. For a list of *lInfoType&* values that can be used to obtain *string* information about an Index, see `SQL_IdxInfoStr`.

In order to obtain numeric information about an Index, the *lInfoType&* parameter must be one of the following values:

INDEX_NON_UNIQUE

Indicates whether or not the index allows or prohibits duplicate values. This *lInfoType&* will return `SQL_TRUE` (1) if the index values are allowed to be non-unique, or `SQL_FALSE` (0) if the index values are required to be unique.

INDEX_ORDINAL_POSITION

The column sequence number in the index, starting with 1.

INDEX_PAGECOUNT

The number of pages that are used to store the index. Zero (0) is returned if this information is not available from the datasource, or if it is not applicable to the datasource.

INDEX_ROWCOUNT

The number of unique values in the index. Zero (0) is returned if this information is not available from the datasource. (This value is also known as the "cardinality" of the index.)

INDEX_TYPE

The Index type. This *InfoType* will return one of the following values:

SQL_INDEX_CLUSTERED
SQL_INDEX_HASHED
SQL_INDEX_OTHER

The Microsoft ODBC Software Developer Kit also lists the following two values, but does not provide the *numeric* values for the symbols.

SQL_INDEX_BTREE
SQL_INDEX_CONTENT

If you are able to determine the correct numeric values for these symbols, please contact Perfect Sync Tech Support (support@perfectsync.com) and we will add the values to future versions of the SQL Tools documentation.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Indexes.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Indexes

SQL_IdxInfoStr

Summary

Provides information about an Index, in string form.

Twin

SQL_IndexInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_IdxInfoStr(lTableNumber&, _  
                           lIndexNumber&, _  
                           lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

lIndexNumber&

The number of an Index, between one (1) and the number returned by the `SQL_IdxCount` function.

lInfoType&

The type of information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, this function will return the requested information. Otherwise, an empty string will be returned.

Remarks

An Index is a structure that is maintained by a database, in order to speed up access to columns that have been indexed.

Please note that not *all* of the information about a table's Indexes is useful in string form. For a list of *lInfoType&* values that can be used to obtain *numeric* information about an Index, see `SQL_IdxInfoStr`.

In order to obtain string information about an Index, the *lInfoType&* parameter must be one of the following values:

INDEX_ASC_OR_DESC

The sort sequence for the column. This *lInfoType&* will return one of the following values:

"A" (for Ascending)

"D" (for Descending)

An empty string will be returned if column sort sequence is not supported by the Datasource.

INDEX_CATALOG, INDEX_SCHEMA, and INDEX_TABLE

The catalog, schema, and table name of the table with which the index is associated.

INDEX_COLUMN_NAME

The Index column name.

If the column is based on an expression, such as *SALARY + FRINGES*, the expression is returned. If the expression cannot be determined by the ODBC driver, an empty string is returned.

INDEX_FILTER_CONDITION

If the index is a filtered index, this *InfoType* returns a string that contains the filter condition, such as *AGE > 100*. If the filter condition cannot be determined, or if the index is not a filtered index, an empty string will be returned

INDEX_NAME

The name of the Index.

INDEX_QUALIFIER

A string value that contains the identifier which is used to qualify the index name when you are performing a *DROP INDEX* operation. If this *InfoType* returns a value, it must be used to qualify the index name in a *DROP INDEX* statement. Otherwise the INDEX_SCHEMA value should be used.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Indexes.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Indexes

SQL_InfoExport

Summary

Creates a disk file that contains all of the Info values (Table Info, Column Info, etc.) that SQL Tools has collected about a database.

Twin

None. (See **Remarks** below.)

Family

Configuration Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_InfoExport(lDatabaseNumber&, _  
                           sFileName$)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

sFileName\$

A string that contains the name (with optional drive/path) of the disk file that should be created. We recommend the extension `.DBI` (for **D**atabase **I**nf) but it is not required.

Return Values

This function returns `SUCCESS` if the requested file was created, or an Error Code if it was not.

If this function returns `ERROR_CANT_BE_DONE`, it means that no Info is available to be saved.

If this function returns a value between `ERROR_FIRST_BAS_ERROR` and `ERROR_LAST_BAS_ERROR`, it means that a runtime error (such as disk full, etc.) was encountered. You can obtain a BASIC-compatible `ERR` value by subtracting `ERROR_FIRST_BAS_ERROR` from the numeric return value, to help you determine the cause of the error.

Remarks

Obtaining information (Info) about a large database can be a slow process. (For a list of some of the things that can cause a slowdown, see `SQL_GetTblInfo`.)

Once Info has been retrieved from a database, the `SQL_InfoExport` function can be used to create a disk file that contains all of the Info values. Then, the next time a program is run, it can use the `SQL_InfoImport` function to re-load the Info instead of re-retrieving it from the database. This can greatly speed up the initialization of a program.

IMPORTANT NOTE: If your database's structure is dynamic -- if tables, columns, privileges, etc. are frequently added or deleted -- it may not be a good idea for you to

use the Info Export and Import functions. If the database structure is modified and the Info values are not refreshed properly, your program will get "out of sync" with the database and the results will be unpredictable. **Tip:** You may wish to have your program check the date stamp on the Info Export file when your program starts, and automatically refresh the Info (by using the `SQL_GetTblInfo` function) when it reaches a certain age. Or you might want to create a utility program that runs overnight (every night) and re-builds the Info Export files, for use by other programs the following day.

IMPORTANT NOTE: It is *extremely* important that you make sure that you do not Import the wrong Info file for a database. For example, if a file called `MYDB.DBI` is created for a database called `MYDB`, and you accidentally load the `MYDB.DBI` file when a program is using a different database, the results are unpredictable. That's why there are no "twin" functions for `SQL_InfoExport` and `SQL_InfoImport`: you must always specify a database number, so that the probability of errors is reduced.

Diagnostics

This function returns Error Codes, and can generate SQL Tools Error Messages.

Example

```
SQL_InfoExport 1, "MYTABLES.DBI"
```

Driver Issues

None.

Speed Issues

See **Remarks** above.

See Also

`SQL_GetTblInfo`, `SQL_InfoImport`

SQL_InfoImport

Summary

Loads database Info from a file that was created with the `SQL_InfoExport` function, or from a string that was obtained from the `SQL_TblInfoStr(0,0)` function.

Twin

None.

Family

Configuration Family

Availability

SQL Tools Pro Only (see)

Warning

None.

```
lResult& = SQL_InfoImport(lDatabaseNumber&, _  
                          sInfoSource$)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

sInfoSource\$

A string that contains the name (with optional drive/path) of the disk file from which the Info should be loaded. We recommend that you use the filename extension `.DBI` (for **D**atabase **I**nterface) but it is not required. Alternatively, this parameter can be a string variable that contains actual Info, or an empty string. See **Remarks** below for details.

Return Values

This function returns `SUCCESS` if the requested file was loaded, or an Error Code if it was not.

If this function returns `ERROR_CANT_BE_DONE`, it means that the specified file does not exist.

If this function returns a value between `ERROR_FIRST_BAS_ERROR` and `ERROR_LAST_BAS_ERROR`, it means that a runtime error (such as a disk-media error, etc.) was encountered. You can obtain a BASIC-compatible `ERR` value by subtracting `ERROR_FIRST_BAS_ERROR` from the numeric return value, to help you determine the cause of the error.

Remarks

See `SQL_InfoExport` for a complete discussion of exporting and importing Info by using disk files, which is the most common technique.

It is also possible to save and restore Info by using memory instead of disk files. The following function...

```
sInfo$ = SQL_TblInfoStr(0,0)
```

...can be used to obtain a string that contains all of the Info that SQL Tools has accumulated about a database. If you are using the verbose SQL Tools functions,

you would use...

```
sInfo$ = SQL_TableInfoStr(lDatabaseNumber,0,0)
```

Using 0,0 means "all tables, all info." A string that has been obtained in this way can then be re-imported like this...

```
SQL_InfoImport lDatabaseNumber, sInfo$
```

It is also possible to clear all of a database's cached info by doing this...

```
SQL_InfoImport lDatabaseNumber, ""
```

Then, the next time that an Info function is used, SQL Tools will detect that the cache is empty and will automatically use the `SQL_GetTblInfo` function to re-read the requested Info. (You can also use `SQL_GetTblInfo` directly, to accomplish the same thing.)

Diagnostics

This function returns Error Codes, and can generate SQL Tools Error Messages.

Example

```
SQL_InfoImport 1, "MYTABLES.DBI"
```

Driver Issues

None.

Speed Issues

See `SQL_InfoExport` for a complete discussion.

See Also

`SQL_GetTblInfo`, `SQL_InfoExport`

SQL_Init

Summary

Initializes SQL Tools, using initialization values that work well for most programs.

Twin

SQL_Initialize

Family

Configuration Family

Availability

Standard and Pro

Warning

Every program that uses SQL Tools *must* use `SQL_Authorize` and then either `SQL_Init` or `SQL_Initialize` before it uses any other SQL Tools functions. See [Four Critical Steps For Every SQL Tools Program](#) for more information.

Syntax

```
lResult& = SQL_Init
```

Parameters

None.

Return Values

See `SQL_Initialize` for complete information.

Remarks

Using `SQL_Init` is exactly the same as using...

```
SQL_Initialize 2, 2, 32, 3, 3, 0, 0, 0
```

For the meaning of each of the parameters, please see `SQL_Initialize`.

Diagnostics

None.

Example

```
Function MyProgram As Long
    SQL_Authorize AuthCode 'see
    SQL_Init
    MyProgram = MainProgram
    SQL_Shutdown
End Function
```

Driver Issues

See `SQL_Initialize` for complete information.

Speed Issues

None.

See Also

[Four Critical Steps For Every SQL Tools Program](#)

SQL_Initialize

Summary

Initializes SQL Tools, using values that you specify.

Twin

SQL_Init

Family

Configuration Family

Availability

Standard and Pro

Warning

Every program that uses SQL Tools *must* use `SQL_Authorize` and then either `SQL_Init` or `SQL_Initialize` before it uses any other SQL Tools functions. See [Four Critical Steps For Every SQL Tools Program](#) for more information.

Syntax

```
lResult& = SQL_Initialize(lMaxDatabaseNumber&, _  
                          lMaxStatementNumber&, _  
                          lMaxColumnNumber&, _  
                          lMaxParameterNumber&, _  
                          lODBCVersion&, _  
                          lConnPooling&, _  
                          lPoolMatching&, _  
                          hEXEInstance)
```

Parameters

lMaxDatabaseNumber&

The maximum Database Number that your program will use, between 1 and 256. The `SQL_Init` default value (and the maximum value that is allowed by SQL Tools Standard) is 2.

lMaxStatementNumber&

The maximum Statement Number that your program will use, between 0 and 256. The `SQL_Init` default value (and the maximum value this is allowed by SQL Tools Standard) is 2.

lMaxColumnNumber&

The maximum Column Number that your program will use, between 32 and 999. The `SQL_Init` default value is 32.

lMaxParameterNumber&

The maximum Bound Statement Parameter Number that your program will use, between 1 and 256. The `SQL_Init` default value is 3.

lODBCVersion&

Either 2 or 3, depending on the ODBC Version that you want SQL Tools to emulate. The `SQL_Init` default value is 3. See **Remarks** below for more information.

lConnPooling&

See **Remarks** below.

lPoolMatching&

See **Remarks** below.

hExeInstance&

See **Remarks** below.

Return Values

If the initialization is successful, `SUCCESS` or `SUCCESS_WITH_INFO` is returned.

If `SQL_Init` or `SQL_Initialize` is used before `SQL_Authorize`, `ERROR_DLL_NOT_AUTHORIZED` will be returned.

If an attempt is made to re-initialize SQL Tools after it has been successfully initialized, `ERROR_CANT_BE_DONE` will be returned.

If an error is detected during the initialization process, other Error Codes may be returned. Depending on the error, it may or may not be possible to use `SQL_Initialize` a second time (using different values) to initialize SQL Tools.

Remarks

IMPORTANT NOTE regarding *IMaxDatabaseNumber*&, *IMaxStatementNumber*&, *IMaxColumnNumber*&, and *IMaxParameterNumber*&. Using *IMax* values that are unnecessarily large will cause SQL Tools to use memory that it really doesn't need to. While values as high as 256, 256, 999, and 256 can be used (respectively), the use of those values would result in SQL Tools reserving an *extremely* large block of memory for its own use. In most cases, you will need to increase one, two, or three of these values, but not all four of them.

IMaxDatabaseNumber&: The `SQL_Init` value of two (2) allows the use of two different databases by the same program at the same time. If your program uses only one database at a time, you can save a small amount of memory by using `SQL_Initialize` and a value of 1 (the minimum value) for this parameter. If your program needs to open more than two databases at a time, you can use values up to 256 for this parameter.

IMaxStatementNumber&: The `SQL_Init` default value of two (2) allows the use of two different SQL statements by the same program at the same time. If your program uses only one statement at a time, you can save a small amount of memory by using `SQL_Initialize` and a value of 1 for this parameter. Under normal circumstances, the minimum value for this parameter should be one (1). For information about using zero (0) for this value, see Statement Zero Operation. If your program needs to use more than two concurrent statements per databases, you can use values up to 256 for this parameter.

IMaxColumnNumber&: This parameter cannot be set to a value below 32 because SQL Tools uses up to 32 columns internally, for various Info functions. You must use a minimum value of 32 even if your program does not require 32 columns per statement. The maximum value for this parameter is 999.

IMaxParameterNumber&: This parameter is used to specify the largest number of Bound Statement Parameters that your program will use. The default value is three (3), to allow up to 3 Bound Parameters to be used without changing from `SQL_Init` to `SQL_Initialize`. If your program does not use any Bound Parameters, you can save a small amount of memory by using `SQL_Initialize` and a value of 1 (the minimum value) for this parameter. The maximum value for this parameter is 256.

IODBCVersion&: The `SQL_Init` default value for this parameter is 3, because most ODBC drivers can emulate at least *some* ODBC 3.x behavior. Using 3 often results in `SUCCESS_WITH_INFO` Error Messages such as...

[Microsoft][ODBC Driver Manager] The driver doesn't support the

version of ODBC behavior that the application requested.

...when a database is opened with `SQL_OpenDB`. The message above was generated when a test program used 3 for `IODBCVersion` and then used `SQL_OpenDB` to open a Microsoft Access 97 database. *This is not a problem.* See `SQL_OpenDB` and Ignoring Predictable Errors for more information.

`lConnPooling` must always be one of the following values: `SQL_CP_OFF` (0), `SQL_CP_ONE_PER_DRIVER` (1), or `SQL_CP_ONE_PER_HENV` (2). SQL Tools Standard only accepts `SQL_CP_OFF`. See the Microsoft ODBC Software Developer Kit for more information about Connection Pooling. The default `SQL_Init` value is zero (`SQL_CP_OFF`).

`lPoolMatching` must always be `SQL_CP_STRICT_MATCH` (0) or `SQL_CP_RELAXED_MATCH` (1). See the Microsoft ODBC Software Developer Kit for more information. The default `SQL_Init` value is zero (`SQL_CP_STRICT_MATCH`).

The `hExeInstance` Parameter

It is only necessary to set this value if you want SQL Tools to use an icon that is embedded in your program (or in a DLL) when it displays dialog boxes. If that is not the case, simply use zero (0) for this parameter.

`hExeInstance` must be either zero (0) or the Instance handle of an EXE or DLL module that contains one or more icons.

If a Visual Basic EXE program contains the icons, the Visual Basic `App.hInstance` property can be used to obtain this value.

PowerBASIC programmers can use the `GetModuleHandle` API function to obtain this value.

```
hCurInst& = GetModuleHandle(ByVal 0)
SQL_Initialize 2,2,32,3,3,0,0,hCurInst&
```

...or it can be obtained this way...

```
FUNCTION WinMain(hCurInst&, _
                hPrevInstance&, _
                lpzCmdLine, _
                nCmdShow&)

    SQL_Initialize 2,2,32,3,3,0,0,hCurInst&
```

Note that the variable names that are used for the *first* parameter of `WinMain` and the *last* parameter of `SQL_Initialize` are the same. This accomplishes the transfer of the value from the EXE to the DLL.

Diagnostics

None.

Example

```
SQL_Authorize  AuthCode  'see  
SQL_Initialize 2,2,32,3,3,0,0,hCurInst&
```

Driver Issues

None.

Speed Issues

None.

See Also

Four Critical Steps for Every SQL Tools Program

SQL_IString

Summary

"Interprets" a string, converting certain text codes (called "shorthands") into certain hard-to-type characters or strings.

Twin

None.

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_IString(sString$)
```

Parameters

sString\$

A string that may or may not contain shorthand strings. **IMPORTANT NOTE:** In its default mode, the `SQL_IString` function only recognizes *lower-case* shorthand strings.

Return Values

This function returns a string that is a copy of *sString\$*, except that any shorthand strings will have been replaced with the specified characters or strings.

Remarks

When SQL Tools is first initialized, the following shorthand strings and their interpretations are recognized:

<code>\q</code>	Double Quotation Mark (ASCII 34: ")
<code>\t</code>	Tab Character (ASCII 9)
<code>\r</code>	Carriage Return (ASCII 13)
<code>\n</code>	"NewLine", also known as Line Feed (ASCII 10)
<code>\e</code>	"Enter" (ASCII 13,10,32)
<code>\ascii</code>	Any ASCII character

For example, if you use a string like this in your source code...

```
sString$ = "The last word is \qQUOTED\q."  
Print SQL_IString(sString$)
```

...the result will be...

The last word is "QUOTED".

The `\ascii` function is used by entering a three-character decimal number after the shorthand, like this:

The last character of this string is `CHR$(0):\ascii000`

...Or...

The last character of this string is `CHR$(255):\ascii255`

The backslash character (`\`) is called the Shorthand Prefix character. (In C programming it is called the Escape Character, but this often causes confusion about ASCII character 27, which is also called "escape".)

The Prefix Character can be used to specify that a shorthand string should *not* be interpreted. For example, if a string contains the following characters...

The File Name is `\newdir\newfile.txt`

...and you wanted to use `SQL_IString` to add quotation marks around the file name, like this...

The File Name is `\q\newdir\newfile.txt\q`

...you would *not* want the `SQL_IString` function to interpret the `\n` strings as NewLine characters (ASCII 10) because they are actually part of a directory name. You can tell the `SQL_IString` function to *not* interpret the string in two different ways. **1)** Convert the file name to upper case. `SQL_IString` only recognizes lower-case shorthand strings. **2)** Add a second prefix character to all of the `\n` prefix characters in the string, like this

The File Name is `\q\\newdir\\newfile.txt\q`

The double backslash (`\\`) tells the `SQL_IString` function "this is a literal backslash, not a shorthand prefix".

You can specify new values for the Shorthand Prefix and all of the Shorthands (`q`, `t`, `r`, etc.) by using the `SQL_SetOptionStr` function and the following values...

```
OPT_ISTRING_PREFIX
OPT_ISTRING_CR
OPT_ISTRING_LF
OPT_ISTRING_TAB
OPT_ISTRING_QUOTE
OPT_ISTRING_ENTER
OPT_ISTRING_ASCII
```

For example, you could change the Shorthand Prefix to the tilde character like this:

```
SQL_SetOptionStr(OPT_ISTRING_PREFIX) = "~"
```

From that point forward, the Shorthands would be `~q`, `~t`, `~r`, and so on.

You can also specify an `OPT_ISTRING_SUFFIX` string, so that (for example) all Shorthands would start with `[` and end with `]`. The default value of the suffix is an empty string.

Finally, you can specify one pair of user-defined search-and-replace strings. For example...

```
SQL_SetOptionStr(OPT_ISTRING_SEARCH) = "@"  
SQL_SetOptionStr(OPT_ISTRING_REPLACE)= "atsign"
```

...could be used to define a \@ shortcut. Whenever it was found, it would be replaced with the string "atsign".

Diagnostics

None.

Example

See **Remarks** above for several examples.

Driver Issues

None.

Speed Issues

None,

See Also

Utility Family

SQL_LimitTextLength

Summary

Limits a string to a certain maximum length.

Twin

None.

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_LimitTextLength(sString$)
```

Parameters

sString\$
Any string.

Return Values

The return value of this function will be a copy of *sString\$* which, if *sString\$* is longer than a certain length, will be truncated. An ellipsis (. . .) will be added to the end of the string to indicate that it was truncated.

Remarks

In addition to being available to your programs, this function is used internally by SQL Tools. Specifically, the `SQL_ResColText` function uses it to produce a printable, reasonable-length string for any type of result column data.

The default maximum string length for this function is 32 characters. This value can be changed (to any value greater than 2) with the `SQL_SetOptionSInt(OPT_TEXT_MAXLENGTH)` function.

Diagnostics

None.

Example

```
'Assuming that the OPT_TEXT_MAXLENGTH  
'value has been changed to 6 for this test:
```

```
For lLen& = 1 To 9  
    sString$ = String$(lLen&, "X")  
    Print lLen&;  
    Print SQL_LimitTextLength(sString$)  
Next
```

...would display...

1 X
2 XX
3 XXX
4 XXXX
5 XXXXX
5 XXXXXXX
7 XXX...
8 XXX...
9 XXX...

Driver Issues

None.

Speed Issues

None.

See Also

SQL_ResColText, Utility Family

SQL_LongParam

Summary

Sends Long data to a bound statement input parameter, or to a `SQL_BulkOp` or `SQL_SetPos` operation.

Twin

`SQL_LongParameter`

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_LongParam(sValue$, _  
                        lIndicator&)
```

Parameters

sValue\$

The Long data, or a *portion* of the Long data, that is to be sent.

lIndicator&

For string or binary data, the length of the *sValue\$* parameter. For parameters with the Null value, `SQL_NULL_DATA`. Under unusual circumstances, for numeric data (in string form), the value `SQL_NUMERIC_DATA`.

Return Values

Returns `SUCCESS` or `SUCCESS_WITH_INFO` if the data is successfully sent to the parameter, or an Error Code if it isn't.

Remarks

See Binding Statement Input Parameters, Binding Long Parameter Values, and Using Long Values with Bulk and Positioned Operations for detailed discussions of this function.

Diagnostics

This function can return Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Binding Statement Input Parameters, Binding Long Parameter Values, and Using Long Values with Bulk and Positioned Operations for code examples.

Driver Issues: See Binding Statement Input Parameters and Using Long Values with Bulk and Positioned Operations.

Speed Issues: See Binding Statement Input Parameters and Using Long Values with Bulk and Positioned Operations.

See Also: `SQL_BulkOp`, `SQL_SetPos`

SQL_LongParameter

Summary

Sends Long data to a bound statement input parameter, or to a SQL_BulkOp or SQL_SetPos operation.

Twin

SQL_LongParam

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_LongParam(lDatabaseNumber&, _  
                          lStatementNumber&, _  
                          sValue$, _  
                          lIndicator&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_LongParam for complete details.

Return Values

See twin function SQL_LongParam for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_LongParameter is identical to SQL_LongParam. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_LongParameter is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_LongParam for complete details.

Example

See twin function SQL_LongParam.

Driver Issues: See twin function SQL_LongParam for complete details.

Speed Issues: See twin function SQL_LongParam for complete details.

See Also: SQL_BulkOp, SQL_SetPos

SQL_LongResCol

Summary

Returns a "chunk" of a Long Result Column (usually either `SQL_LONGVARCHAR`, `SQL_WLONGVARCHAR`, `SQL_LONGVARBINARY`).

Twin

`SQL_LongResultColumn`

Family

Result Column Family

Availability

Standard and Pro

Warning

Some ODBC Drivers place restrictions on the columns that can be accessed with `SQL_LongResCol`. See **Remarks** below for more information.

Syntax

```
sResult$ = SQL_LongResCol(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of the result column for which you wish to obtain data, between one (1) and the number that is returned by the `SQL_ResColCount` function. If you are accessing a bookmark column, this parameter can be zero (0).

Return Values

This function returns a string that contains a "chunk" of the value of a `SQL_LONGVARCHAR`, `SQL_WLONGVARCHAR`, or `SQL_LONGVARBINARY` column. See **Remarks** below for more information.

Remarks

For background information, see Long Columns.

The default "chunk" size for this function is 32,768 (32k) bytes. This value works well for most programs, but it can be changed with the `SQL_SetOptionSInt(OPT_DATALEN_CHUNK)` function.

If a Long Column value contains fewer than 32k bytes, `SQL_LongResCol` will return the entire Long Column value.

If a Long Column value contains more than 32k bytes, the first use of `SQL_LongResCol` after each fetch will return the first 32k bytes of data. To obtain additional chunks (which can be added together to reconstruct the entire Long value) do *not* repeat the fetch operation, use `SQL_LongResCol` repeatedly, until the `SQL_ResColMore` function returns False (0) to indicate that the entire Long Column value has been read.

IMPORTANT NOTE: Many ODBC drivers support Long columns up to 1 *gigabyte* in length (that's 1,000 megabytes), so it may not be practical for your program to handle

a Long value as a single string. You must write your program accordingly.

IMPORTANT NOTE: It is *normal* for an ODBC Error Message to be generated when the `SQL_LongResCol` function is used to retrieve a *portion* of a Long Column. The message is typically "Data Truncated" (or something similar) and it simply indicates that the data-fetch operation retrieved only part of the column's data. Your program can safely ignore these error messages because they are a normal part of the process of retrieving data from Long Columns.

IMPORTANT NOTE: Some ODBC drivers place restrictions on the columns that can be accessed with `SQL_LongResCol`. For example, you may be required to use result sets that have all of their Long columns *after* the non-Long columns. For complete information and a method for determining the restrictions that your ODBC driver imposes, see Possible Driver Restrictions on Long Columns.

Problems with Missing Bytes

Long Columns are usually used for three different types of data, `SQL_LONGVARCHAR`, `SQL_WLONGVARCHAR`, and `SQL_LONGVARBINARY`.

When a Long Column contains a `SQL_LONGVARCHAR` value, that *usually* means that it contains text. For example, it might be a "memo" field that allows the user to type lengthy notes. A `SQL_LONGVARCHAR` field is allowed to contain "control characters" such as Carriage Returns, but not `CHR$(0)`. The data is not truly "binary", it is considered to be text or "character" data, hence the name `SQL_LONGVARCHAR`.

When a Long Column contains text, the ODBC driver will automatically add a "null terminator" (`CHR$(0)`) to the end of *each chunk* of data. To make using Long Columns easier, SQL Tools automatically trims the last character from every block of data that is returned by the `SQL_LongResCol` function, so your programs don't have to worry about removing the `CHR$(0)`. Your program can simply add the "chunks" together to create one long string, if that's what you need to do. Keep in mind that this means that each string that is returned by `SQL_LongResCol` will be *one byte shorter* than the "chunk" size that is described above. This is perfectly normal.

Removing the last character may cause a problem, however, if the Long Column contains `SQL_WLONGVARCHAR` or `SQL_LONGVARBINARY` data. Binary data and Unicode data are not always null-terminated, so when SQL Tools removes the last character from a chunk, it may actually be removing *data*, not a null terminator.

`SQL_LONGVARCHAR` columns will always work properly with the `SQL_LongResCol` function. `SQL_WLONGVARCHAR` and `SQL_LONGVARBINARY` columns will *usually* work properly, but unfortunately, not all ODBC drivers are 100% consistent with the way they return Long data, so SQL Tools can't always *automatically* determine what it needs to do.

If you are using the `SQL_LongResCol` function to retrieve Unicode data or *binary* data from a database (bitmaps, sounds, BLOBs, etc.) you may find that the chunks of data that it returns are always one byte shorter than they should be. If that happens, you must *tell* SQL Tools not to remove the last byte from each chunk. To do that, use this line of code in your program:

```
SQL_SetOptionSInt OPT_LONGRES_COLTYPE, SQL_LONGVARIABLE
```

Please note that this setting affects *all* Long Columns at the same time. If your database contains SQL_LONGVARCHAR *and* SQL_LONGVARIABLE/SQL_wLONGVARCHAR columns, you will need to add code to "manually" remove the null terminators from *text* values that are returned by the SQL_LongResCol function. See SQL_SetOptionSInt for more information.

Diagnostics

This function does not return Error Codes because it returns string values, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Get long data from column 10:
sLongCol$ = SQL_LongResCol(10)
Do
    If SQL_ResColMore(10) Then
        sLongCol$ = sLongCol$ + SQL_LongResCol(10)
    Else
        Exit Do
    End If
Loop
```

Driver Issues

See **Remarks** above.

Speed Issues

None.

See Also

Long Columns

SQL_LongResultColumn

Summary

Returns a "chunk" of a Long Result Column (usually either SQL_LONGVARCHAR, SQL_wLONGVARCHAR, SQL_LONGVARBINARY).

Twin

SQL_LongResCol

Family

Result Column Family

Availability

Standard and Pro

Warning

Some ODBC Drivers place restrictions on the columns that can be accessed with SQL_LongResCol. See SQL_LongResCol for more information.

Syntax

```
sResult$ = SQL_LongResultColumn(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_LongResCol for complete details.

Return Values

See twin function SQL_LongResCol for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_LongResultColumn is identical to SQL_LongResCol. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_LongResultColumn is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_LongResCol for complete details.

Example

See SQL_LongResCol for an example.

Driver Issues: See SQL_LongResCol.

Speed Issues: None.

See Also: Long Columns

SQL_ManualBindCol

Summary

Binds one column of a result set, and its Indicator, to memory buffers that your program provides. (Most programs do not need to perform this step because SQL Tools can AutoBind all of the columns in a result set. Compare `SQL_DirectBindCol`.)

Twin

`SQL_ManualBindColumn`

Family

Result Column Binding Family

Availability

Standard and Pro

Warning

If your program uses this function to bind a result column and Indicator to memory buffers but then fails to properly maintain those buffers, an Application Error will result. See `SQL_DirectBindCol` for more information.

Syntax

```
lResult& = SQL_ManualBindCol(lColumnNumber&, _  
                             lDataType&, _  
                             lPointerToBuffer&, _  
                             lBufferLength&, _  
                             lIndicator&)
```

Parameters

lColumnNumber&, *lDataType&*, *lPointerToBuffer&*, and *lBufferLength&*

See `SQL_DirectBindCol` for information about these parameters. This function uses exactly the same parameters in exactly the same ways.

lIndicator&

The *variable* that should be used for the column's Indicator. You must not use a literal numeric value for this parameter.

Return Values

See `SQL_DirectBindCol` for complete details.

Remarks

Except for the *lIndicator&* parameter, `SQL_ManualBindCol` is identical to `SQL_DirectBindCol`. Manual Binding is just like Direct Binding except that it also binds an Indicator to a variable that your program provides. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_ManualBindCol` is shown below.

Diagnostics: See `SQL_DirectBindCol` for complete details.

Example: See `SQL_DirectBindCol`.

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Binding (Basic), Result Column Binding (Advanced)

SQL_ManualBindColumn

Summary

Binds one column of a result set, and its Indicator, to memory buffers that your program provides. (Most programs do not need to perform this step because SQL Tools can AutoBind all of the columns in a result set. Compare SQL_DirectBindCol.)

Twin

SQL_ManualBindCol

Family

Result Column Binding Family

Availability

Standard and Pro

Warning

If your program uses this function to bind a result column and Indicator to memory buffers but then fails to properly maintain those buffers, an Application Error will result. See SQL_DirectBindCol for more information.

Syntax

```
lResult& = SQL_ManualBindColumn(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&, _  
                                lDataType&, _  
                                lPointerToBuffer&, _  
                                lBufferLength&, _  
                                lIndicator&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ManualBindCol for complete details.

Return Values

See twin function SQL_ManualBindCol for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ManualBindColumn is identical to SQL_ManualBindCol. SQL_ManualBindCol, in turn, is identical to SQL_DirectBindCol (except for one parameter). To avoid errors when this document is updated, information that is common to all of these functions is not duplicated here. Only information that is unique to SQL_ManualBindColumn is shown below.

Diagnostics

See twin function SQL_ManualBindCol for complete details.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Binding (Basic), Result Column Binding (Advanced)

SQL_MoreRes

Summary

Indicates whether or not there are More Results available from a batched SQL statement, i.e. whether or not an *additional* result set or row count is available to be retrieved.

Twin

SQL_MoreResults

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_MoreRes
```

Parameters

None.

Return Values

This function will return one of the following values:

%SUCCESS if another result set or row count is available.

%SUCCESS_WITH_INFO if another result set or row count is available and the statement attributes (cursor type, concurrency, etc.) have changed. You can use the various SQL_Error... functions to determine what changed.

%SQL_NO_DATA if no additional result sets or row counts are available

%SQL_ERROR if an error is detected.

Remarks

SQL statements that use *SELECT* return result sets, and most other SQL statements return row counts that indicate how many rows were affected by the statement.

If SQL Statements are batched, they can return *multiple* result sets and/or multiple row counts.

When a batch is executed, the first result set or row count is immediately made available to your program, just as if the first SQL statement was not part of a batch. Your program should handle the first result set or row count, and then use the SQL_MoreRes function to determine whether or not an additional result set or row count is available. If more results are available, the SQL_MoreRes function will return %SUCCESS or %SUCCESS_WITH_INFO and the next result set or row count will be made available to your program.

IMPORTANT NOTE: If SQL_MoreRes is used to make a new result set available,

you must remember to bind the new result set's columns. This is usually done by using the `SQL_AutoBindCol(ALL_COLS)` function *immediately* after `SQL_MoreRes`, but other binding techniques (direct binding, etc.) can also be used. (It is not necessary to perform this step if `SQL_MoreRes` is being used to make a new *row count* available to your program.)

You should not use `SQL_MoreRes` until you are *finished* with the first result set or row count, because once the function has been used the first results are discarded.

Diagnostics

This function does not return Error Codes, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

Many ODBC drives do not support batched SQL statements.

Speed Issues

None.

See Also

Appendix A: SQL Statement Syntax

SQL_MoreResults

Summary

Indicates whether or not there are More Results available from a batched SQL statement, i.e. whether or not an *additional* result set or row count is available to be retrieved.

Twin

SQL_MoreRes

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_MoreResults(lDatabaseNumber&, _  
                           lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

See twin function `SQL_MoreRes` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_MoreResults` is identical to `SQL_MoreRes`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_MoreResults` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_MoreRes` for complete details.

Example

None.

Driver Issues

See twin function `SQL_MoreRes` for complete details.

Speed Issues

None.

See Also

Appendix A: SQL Statement Syntax

SQL_MsgBox

Summary

Displays a standard Windows Message Box.

Twin

None.

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_MsgBox(sMessage$, _  
                      lStyle&)
```

Parameters

sMessage\$

The text message that should be displayed in the message box.

lStyle&

The type of message box that should be displayed, i.e. the number and types of buttons that the message box should have. See **Remarks** below for a list of valid values.

Return Values

This function returns a numeric value that indicates which button was selected by the user: OK_BUTTON, CANCEL_BUTTON, ABORT_BUTTON, RETRY_BUTTON, IGNORE_BUTTON, YES_BUTTON, or NO_BUTTON.

Remarks

In addition to using the *sMessage\$* and *lStyle&* values, your program can change the appearance of the message box in other ways. See the notes regarding the use of the SQL_SetOption functions below.

The *sMessage\$* parameter may contain certain characters that are used to control text formatting, such as the NewLine (Line Feed) character. The *sMessage\$* string that is submitted to SQL_MsgBox is always processed by the SQL_IString function, to make it easy to include NewLine, Quote, and other characters.

The *lStyle&* parameter must be one of the following values. The names of the constants indicate the message box buttons that are created by the values: MSGBOX_OK, MSGBOX_OKCANCEL, MSGBOX_ABORTRETRYIGNORE, MSGBOX_YESNOCANCEL, MSGBOX_YESNO, or MSGBOX_RETRYCANCEL.

The default message box title is "SQL Tools". If you use the SQL_SetOptionStr(OPT_MY_PROGRAM) function to tell SQL Tools the name of your program, that string will be used for message box titles.

The default message box icon is the Perfect Sync "S" logo. You can use the SQL_SetOptionSInt(OPT_ICON_ID) function to specify a different icon. You may

use any one of the following values, which correspond to the standard names of the standard Windows icons: `ICON_APPLICATION`, `ICON_HAND`, `ICON_ERROR`, `ICON_QUESTION`, `ICON_EXCLAMATION`, `ICON_WARNING`, `ICON_ASTERISK`, `ICON_INFORMATION`, or `ICON_WINLOGO`. You may also use the Resource ID Number of an icon that is embedded in your EXE program *if* you also tell SQL Tools the instance handle of your program. This is usually done by passing the appropriate *hInstance* value to the `SQL_Initialize` function.

The default parent window for all SQL Tools message boxes is the Windows Desktop. You can specify a different window by using the `SQL_SetOptionSInt(OPT_h_PARENT_WINDOW)` function. See the `SQL_hParentWindow` function for more details.

Note that the `SQL_MsgBox` function returns a numeric value that corresponds to the button that is selected by the user, and that the `SQL_MsgBoxButton` function can be used to obtain the same information. Both methods will return one of the `...BUTTON` return values shown above. If the `SQL_MsgBoxButton` function is used before the `SQL_MsgBox` function is used for the first time, it will return `BUTTON_NOT_SELECTED`.

Diagnostics

None.

Example

```
SQL_MsgBox "CLICK OK: ", MSGBOX_OK
```

Driver Issues

None.

Speed Issues

None.

See Also

Utility Family

SQL_MsgBoxButton

Summary

Returns the ID number of the button that was selected the last time that the `SQL_MsgBox` or `SQL_SelectFile` function was used.

Twin

None.

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_MsgBoxButton
```

Parameters

None.

Return Values

If the `SQL_MsgBox` and `SQL_SelectFile` functions have not yet been used, this function will return `BUTTON_NOT_SELECTED`. If `SQL_MsgBox` or `SQL_SelectFile` has been used at least once, this function will return one of the following values, depending on which button was most recently selected by the user: `OK_BUTTON`, `CANCEL_BUTTON`, `ABORT_BUTTON`, `RETRY_BUTTON`, `IGNORE_BUTTON`, `YES_BUTTON`, or `NO_BUTTON`.

Please note the important difference between `BUTTON_NOT_SELECTED` and `NO_BUTTON`. `NO_BUTTON` means that the button with the label "No" was selected. `NO_BUTTON` does *not* mean "No button has yet been selected".

Note also that if the ODBC Driver displays any dialogs (such as the ODBC Connection Dialogs that can be displayed by the `SQL_OpenDB` function), those dialogs will *not* affect the return value of this function. This function is affected by the `SQL_SelectFile` and `SQL_MsgBox` functions *only*.

Remarks

In most cases, your program will detect which `SQL_MsgBox` or `SQL_SelectFile` button was selected by examining the return values of those functions.

This function is provided primarily as a programming convenience.

Diagnostics

None.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Utility Family

SQL_NameCur

Summary

Assigns a name to a cursor.

Twin

SQL_NameCursor

Family

Statement Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_NameCur(sName$)
```

Parameters

sName\$

The name that is to be assigned to the cursor. The name must be less than 19 characters long, and no other cursor may have the same name. For efficient processing, the cursor name should not include any leading or trailing spaces, and if the name includes a delimited identifier, the delimiter should be the first character of the name.

Return Values

If the name is assigned successfully, this function returns SUCCESS or SUCCESS_WITH_INFO.

If an error is detected and the name is not assigned, this function returns an Error Code.

Remarks

Cursor names are used only in "positioned" update and delete statements, such as

```
UPDATE tablename... WHERE CURRENT OF cursorname
```

You must execute a SQL statement, and thereby create a cursor before it can be named.

See Named Cursors for more information.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
SQL_NameCur "MyCursor"
```

Driver Issues

None.

Speed Issues

None.

See Also

Named Cursors

SQL_NameCursor

Summary

Assigns a name to a cursor.

Twin

SQL_NameCur

Family

Statement Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_NameCursor(lDatabaseNumber&, _  
                           lStatementNumber&, _  
                           sName$)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_NameCur for complete details.

Return Values

See twin function SQL_NameCur for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_NameCursor is identical to SQL_NameCur. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_NameCursor is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_NameCur for complete details.

Example

```
'assign a name to database 1,  
'statement #2:  
SQL_NameCursor 1, 2, "MyCursor"
```

Driver Issues: None.

Speed Issues: None.

See Also

Named Cursors

SQL_NewDBNumber and SQL_NewDatabaseNumber

Summary

These functions return an available database number, i.e. a database number that is not currently open.

Twin

These twin functions are identical. Two different spellings are provided as a programmer convenience.

Family

Database Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_NewDBNumber
```

...Or...

```
lResult& = SQL_NewDataBaseNumber
```

Parameters

None.

Return Values

These functions return the lowest unused database number, between one (1) and the *IMaxDatabaseNumber*& value that was specified with *SQL_Initialize*. If all of the database numbers between 1 and *IMaxDatabaseNumber*& are currently open, these functions return negative one (-1).

Remarks

These functions are conceptually similar to the BASIC *FREEFILE* function. Programs that use multiple databases can use these functions to dynamically assign database numbers instead of hard-coding them.

Keep in mind that *SQL_NewDBNumber* and *SQL_NewDatabaseNumber* will continue to return the same value until the database number that is returned is actually *opened*. For example, it would be a mistake to do this:

```
lDB1& = SQL_NewDBNumber
lDB2& = SQL_NewDBNumber
SQL_OpenDatabase lDB1&, "DSN1,DSN", PROMPT_TYPE_NOPROMPT
SQL_OpenDatabase lDB2&, "DSN2,DSN", PROMPT_TYPE_NOPROMPT
```

Assuming that database number 1 was not open when that code was run, the first *SQL_NewDBNumber* return value would be 1. And when the function was used the second time, database number 1 would still not be open, so the function would return 1 again. The correct way to structure that code would be:

```
lDB1& = SQL_NewDBNumber  
SQL_OpenDatabase lDB1&, "DSN1,DSN", PROMPT_TYPE_NOPROMPT  
lDB2& = SQL_NewDBNumber  
SQL_OpenDatabase ldb2&, "DSN2,DSN", PROMPT_TYPE_NOPROMPT
```

Diagnostics

None.

Example

```
lDBNo& = SQL_NewDBNumber  
SQL_OpenDatabase lDBNo&, "MY.DSN", PROMPT_TYPE_NOPROMPT
```

Driver Issues

None.

Speed Issues

None.

See Also

Opening a Database

SQL_NewStatementNumber

Summary

Returns a statement number that is available to be used (i.e. a statement number that is not currently open) for a given database number.

Twin

SQL_NewStmtNumber

Family

Statement Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_NewStatementNumber(lDatabaseNumber&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

Return Values

See twin function `SQL_NewStmtNumber` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_NewStatementNumber` is identical to `SQL_NewStmtNumber`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_NewStatementNumber` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_NewStmtNumber` for complete details.

Example

See twin function `SQL_NewStmtNumber`.

Driver Issues

None.

Speed Issues

None.

See Also

SQL Statements

SQL_NewStmtNumber

Summary

Returns a statement number that is available to be used (i.e. a statement number that is not currently open) for the current database.

Twin

SQL_NewStatementNumber

Family

Statement Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_NewStmtNumber
```

Parameters

None.

Return Values

This function returns the lowest unused statement number, between one (1) and the *lMaxStatementNumber*& value that was specified with `SQL_Initialize`, for the current database. If all of the database's statement numbers between 1 and *lMaxStatementNumber*& are currently open, this function returns negative one (-1).

Remarks

This function is conceptually similar to the BASIC `FREEFILE` function. Programs which use multiple concurrent statements can use this function to dynamically assign statement numbers instead of hard-coding them.

Keep in mind that `SQL_NewStmtNumber` will continue to return the same value until the statement number that is returned is actually opened. For an example of this, see `SQL_NewDBNumber`.

IMPORTANT NOTE: Not all ODBC drivers support more than one concurrent statement. This function simply returns a SQL Tools statement number than can be used in an *attempt* to open a new statement. It does not perform a test to find out whether or not the ODBC driver is actually capable of opening another statement.

Diagnostics

None.

Example

```
lStmt& = SQL_NewStmtNumber
SQL_OpenStatement 1, lStmt&
```

Driver Issues

None.

Speed Issues

None.

See Also

SQL Statements

SQL_NextParam

Summary

Tells SQL Tools that you are ready to begin (or are finished) sending Long data to a bound statement input parameter, or to a `SQL_BulkOp` or `SQL_SetPos` operation. Also returns the next parameter number for which the ODBC driver needs data, if any.

Twin

`SQL_NextParameter`

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_NextParam
```

Parameters

None.

Return Values

Returns either `SUCCESS` (zero) if all of the required Long data values have been sent, or the parameter number of the next parameter number that needs Long data. Under certain circumstances, this function can also return Error Codes.

Remarks

For a complete discussion of this function, see *Binding Statement Input Parameters and/or Using Long Values with Bulk and Positioned Operations*.

Diagnostics

This function does not normally return Error Codes, but it is possible for it to do so. See *Binding Statement Input Parameters and/or Using Long Values with Bulk and Positioned Operations* for complete information. This function can also generate ODBC Error Messages and SQL Tools Error Messages.

Example

See *Binding Statement Input Parameters and/or Using Long Values with Bulk and Positioned Operations* for example code.

Driver Issues

See *Binding Statement Input Parameters and/or Using Long Values with Bulk and Positioned Operations*.

Speed Issues

See *Binding Statement Input Parameters and/or Using Long Values with Bulk and Positioned Operations*.

See Also: *Binding Long Parameter Values*

SQL_NextParameter

Summary

Tells SQL Tools that you are ready to begin (or are finished) sending Long data to a bound statement input parameter, or to a `SQL_BulkOp` or `SQL_SetPos` operation. Also returns the next parameter number for which the ODBC driver needs data, if any.

Twin

`SQL_NextParam`

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_NextParameter(lDatabaseNumber&, _  
                             lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

Except for the *lDatabaseNumber&* parameter, `SQL_NextParameter` is identical to `SQL_NextParam`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_NextParameter` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Remarks

For a complete discussion of this function, see Binding Statement Input Parameters.

Diagnostics

This function does not normally return Error Codes, but it is possible for it to do so. See Binding Statement Input Parameters for complete information.

Example

See twin function `SQL_NextParam`.

Driver Issues

See twin function `SQL_NextParam`.

Speed Issues

See twin function `SQL_NextParam`.

See Also: Binding Long Parameter Values

SQL_Okay

Summary

Recognizes `SUCCESS` and `SUCCESS_WITH_INFO` as being "okay" conditions, and all other Error Codes as being "not okay".

Twin

None.

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_Okay(lErrorCode&)
```

Parameters

lErrorCode&

A numeric value that represents an Error Code.

Return Values

This function returns Logical True (-1) if the value of *lErrorCode&* is `SUCCESS` or `SUCCESS_WITH_INFO`, or False (zero) if *lErrorCode&* is any other value.

Remarks

This is a programming-convenience function. Instead of using this code throughout your program...

```
IF lResult& = SUCCESS OR _  
    lResult& = SUCCESS_WITH_INFO THEN  
    'it worked  
ELSE  
    'handle an error message  
END IF
```

...you can use this:

```
IF SQL_Okay(lResult&) THEN  
    'it worked  
ELSE  
    'handle an error message  
END IF
```

Since `SQL_Okay` returns a Logical True value, you can also use...

```
IF NOT SQL_Okay(lErrorCode&) THEN...
```

Diagnostics

None.

Example

See **Remarks** above.

Driver Issues

None.

Speed Issues

None.

See Also

Utility Family

SQL_OnErrorCall

Summary

Provides SQL Tools with the memory location of an error-handling routine in *your* program.

Twin

None.

Family

Error/Trace Family

Availability

SQL Tools Pro Only (see)

Warning

Passing an invalid value to this function, or improperly designing your error-handling routine, will result in Application Errors.

This function cannot be used by programming languages that do not support Code Pointers.

Syntax

```
SQL_OnErrorCall dwCodePtr???
```

...Or...

```
SQL_OnErrorCall lCodePtr&
```

Parameters

dwCodePtr??? or *lCodePtr&*

A memory pointer from the PowerBASIC CODEPTR function, the Visual Basic AddressOf function, a C expression that uses the Address-Of operator (&) to point to a function, or a similar value. (Either a DWORD or a LONG integer can be used, as long as it represents a valid pointer to a function.)

Return Values

This function always returns SUCCESS, so it is safe to ignore the return value of this function.

Remarks

If you pass a CODEPTR value of a properly formatted error-handling routine to this function (see **Example** below), SQL Tools will call your routine whenever an error is detected.

Your error handling function *must* have the following structure:

```
FUNCTION MyErrorHandler(BYVAL lOneLongParam&) AS LONG
```

You may use any function name, and of course you may use the syntax that is required by your programming language, but the return value of the function must be a BAS_LONG integer (or equivalent) and the function must have exactly one BAS_LONG integer parameter (or equivalent), passed BYVAL.

When an error is detected by SQL Tools, it will perform all of the normal error processing that SQL Tools provides, and then it will call your function. The numeric parameter that is passed to your function will be the current `SQL_ErrorCount` value. In other words, SQL Tools will call your error handling routine and pass to it the number of errors that are currently in the Error Stack.

Once your error handling routine has been called, all normal SQL Tools error handling remains in effect *except for your error handler*, until your error handler exits. So if an error is detected and your error handler is called, you are free to use SQL Tools functions *in* your error handler without worrying that another error will be detected and your error handler will be called again, resulting in a possibly-endless loop.

The most common use of `SQL_OnErrorCall` is to display a custom Error Message.

After it has been enabled, you can disable your error handler by using a value of zero (0) for `dwCodePtr` or `lCodePtr`.

Your error handling function is known as a "callback" routine, because after your program calls (uses) a SQL Tools function, the SQL Tools error handling routines can "call back" to a function in your program.

Diagnostics

None.

Example

```
SQL_OnErrorCall CODEPTR("MyHandlerFunc")
```

Driver Issues

None.

Speed Issues

None.

See Also

Error Handling

SQL_OpenDatabase

Summary

Opens a database and prepares it for use with other SQL Tools functions.

Twin

SQL_OpenDB

Family

Database Open/Close Family

Availability

Standard and Pro

Warning:

None.

Syntax

```
lResult& = SQL_OpenDatabase(lDatabaseNumber&, _  
                             sDatabaseString$, _  
                             lPrompt&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

sDatabaseString\$

See twin function SQL_OpenDB for complete details.

lPrompt&

One of the following constants: PROMPT_TYPE_NOPROMPT, PROMPT_TYPE_COMPLETE, PROMPT_TYPE_PROMPT, or PROMPT_TYPE_REQUIRED. See **Remarks** below for details.

Return Values

See twin function SQL_OpenDB for complete details.

Remarks

Except for the *lDatabaseNumber&*, *lStatementNumber&*, and *lPrompt&* parameters, SQL_OpenDatabase is identical to SQL_OpenDB. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_OpenDatabase is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

For information about the *sConnectionString\$* parameter, see SQL_OpenDB and Connection Strings.

The following values may be used for *lPrompt&*:

PROMPT_TYPE_NOPROMPT tells the SQL_OpenDatabase function that it should not display any dialog boxes to prompt the user for a database connection. If the function is not able to establish a connection by using the information supplied in *sConnectionString\$*, it will fail and return an ODBC Error Code.

PROMPT_TYPE_PROMPT tells the SQL_OpenDatabase function that it should display

dialog boxes to display the connection information, and allow the user to change it, even if the *sConnectionString* information is valid and sufficient to establish a connection.

PROMPT_TYPE_COMPLETE and PROMPT_TYPE_REQUIRED tell the SQL_OpenDatabase function that if the connection string contains enough valid information, it should make the connection without displaying any dialogs. If any information is invalid or incomplete, the same dialog boxes as PROMPT_TYPE_PROMPT are displayed. (If *IPrompt* is PROMPT_TYPE_REQUIRED, the dialog boxes do not allow the user to change any already-valid information.)

Diagnostics

This function returns Error Codes and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
lResult& = SQL_OpenDatabase(2,"MY.DSN",PROMPT_TYPE_NOPROMPT)
```

Driver Issues

None.

Speed Issues

None.

See Also

Opening a Database

SQL_OpenDatabase1

Summary

Begins the process of opening a database by allocating a database handle that can be used by the `SQL_OpenDatabase2` function. (The `SQL_OpenDatabase1` and `SQL_OpenDatabase2` functions are rarely used by programs. Most programs use `SQL_OpenDB` or `SQL_OpenDatabase` with no number at the end. See `SQL_OpenDB` for information about using the 1 and 2 functions.)

Twin

None

Family

Database Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_OpenDatabase1(lDatabaseNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the ODBC driver provides a database handle, or an Error Code if it does not.

Remarks

This function is not commonly used. See `SQL_OpenDB` for more information.

Diagnostics

This function returns Error Codes and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Opening a Database

SQL_OpenDatabase2

Summary

Completes the process of opening a database which was started by the SQL_OpenDatabase1 function. (The SQL_OpenDatabase1 and SQL_OpenDatabase2 functions are rarely used by programs. Most program use SQL_OpenDB or SQL_OpenDatabase with no number at the end. See SQL_OpenDB for information about using the 1 and 2 functions.)

Twin

None

Family

Database Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_OpenDatabase2(lDatabaseNumber&, _  
                             sConnectionString$, _  
                             lPrompt&)
```

Parameters

All Parameters

See SQL_OpenDatabase for complete details.

Return Values

This function returns SUCCESS or SUCCESS_WITH_INFO if the open-database process is completed without errors, or an Error Code if it is not.

Remarks

This function is not commonly used. See SQL_OpenDB for more information.

Diagnostics

This function returns Error Codes and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Opening a Database

SQL_OpenDB

Summary

Opens a database and prepares it for use with other SQL Tools functions.

Twin

SQL_OpenDatabase

Family

Database Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_OpenDB(sDatabaseString$)
```

Parameters

sDatabaseString\$

A string that describes a database. This can be any one of the following:

- 1) The name of a DSN file in the default directory,
- 2) The name of a DSN file with a drive/path specification,
- 3) A partial DSN file name, such as *.DSN, with or without a drive/path spec,
- 4) A complete Connection String such as the text that is found *inside* a DSN file,
- 5) A partial Connection String, or
- 6) An empty string.

See **Remarks** below for a discussion of each option.

Return Values

If the `SQL_OpenDB` function is able to connect with a database, `SUCCESS` or `SUCCESS_WITH_INFO` will be returned. (See **Diagnostics** below for more information about `SUCCESS_WITH_INFO`.)

If the `SQL_OpenDB` function displays a Select File Dialog and/or a Connection Dialog and the user selects Cancel or Quit, this function will return `ERROR_USER_CANCEL`.

If the connection to the specified database is not successful for some other reason, the return value of `SQL_OpenDB` will be either an ODBC Error Code or a SQL Tools Error Code.

Remarks

The default prompt mode for the `SQL_OpenDB` function is called

PROMPT_TYPE_COMPLETE. This means that, unless you use `SQL_SetOptionSInt` to change the default mode (see below), the `SQL_OpenDB` function will behave in the following way:

If you provide a complete DSN file name, and if the DSN file exists in the specified location, and if the DSN file is valid, the `SQL_OpenDB` function will connect to the database without displaying any dialog boxes. (The dialog boxes are also called "prompts".)

If you provide a partial DSN file name (such as `*.DSN` or `MYDB?.DSN`), or if you specify a complete DSN file name but the file does not exist in the specified location, the function will display a standard Open File dialog box to allow you to browse for the file. You may use the Open File dialog box to select either a DSN file or a Windows shortcut to a DSN file. If the selected DSN file is valid, this function will connect to the database without displaying any further dialog boxes.

If, instead of a file name, you provide valid Connection String that contains enough information for the ODBC driver to open the database, this function will open the database without displaying any dialog boxes. (DSN files *contain* Connection Strings. Passing the name of a DSN file accomplishes exactly the same thing as passing the contents of the DSN file to the `SQL_OpenDB` function. The primary advantages of passing the Connection String itself are **1)** it allows for the hard-coding of connection strings, and **2)** it allows for the runtime construction of connection strings.)

If you provide a partial Connection String or an empty string, or if a DSN file that was selected (above) is not complete and valid, this function will display a series of dialog boxes that will allow the user to create, save, and select a DSN file.

For additional information, see Appendix G: Connection String Syntax.

In addition to allowing you to specify a Database Number, the twin function `SQL_OpenDatabase` also allows you to specify the level of prompting that you want the dialog boxes to provide. If you simply want the `SQL_OpenDatabase` operation to fail if the connection is not immediate, without displaying any of the built-in dialog boxes, you should use `SQL_OpenDatabase` function and specify the `PROMPT_TYPE_NOPROMPT` option.

It is also possible to change the default prompting method that is used by the `SQL_OpenDB` function, by using this code...

```
SQL_SetOptionSInt OPT_OPENDB_PROMPT, lPrompt&
```

...where `lPrompt&` is one of the values described under `SQL_OpenDatabase`.

The default maximum length for `sConnectionString$` is 1,024 bytes. You can increase this value up to 65,535 bytes by using the following code...

```
SQL_SetOptionSInt OPT_MAXCONNSTRINGLEN, lLen&
```

...where `lLen&` is a number between 1024 and 65535. Because of the way ODBC drivers use this string, this value must be large enough for **1)** the `sConnectionString$` that you pass to this function, and **2)** the actual Connection String after it is finalized by the ODBC driver. In other words, if you specify the 10-character string "MYDATA.DSN" the `OPT_MAXCONNSTRINGLEN` value must be large enough to hold the *contents* of the MYDATA.DSN file, plus any modifications that are made by the

dialog boxes. Most DSN files are around 300 bytes long, so the default value of 1024 (which is recommended by the ODBC 3.x specification) will work well for almost all programs. Only programs that use extremely complicated connection strings will need to change this value.

The parent window or form for the Open File dialog and other dialog boxes can be specified with the following code...

```
SQL_SetOptionSInt OPT_h_PARENT_WINDOW, hWindow&
```

... where *hWindow&* is the window's Handle. If a parent window is not specified in this way, or if the specified handle is invalid when it comes time to display a dialog box, SQL Tools will automatically revert to using the Windows Desktop as the parent window.

The title bar of the Open File dialog defaults to "SELECT A DSN FILE". You can change the title with the following code...

```
SQL_SetOptionStr OPT_SELECTDSN, sTitle$
```

... where *sTitle\$* is the desired text. (This option is provided primarily for non-English programs, but it can also be used if you want to customize the dialog boxes.) The titles of the other dialog boxes are hard-coded by Microsoft and can't be changed with SQL Tools. The Microsoft dialogs may or may not automatically use the native language of the runtime computer.

After a database has been opened, the `SQL_OpenDB` function automatically checks to make sure that it is capable of performing something called "Fetch Scroll" operations. If it is *not* capable, the database cannot perform `SQL_Fetch` operations except in a forward-only mode, so SQL Tools automatically sets an internal switch to allow only forward-only fetching. This switch can be manually set with the following code...

```
SQL_SetOptionSInt OPT_USE_FETCHSCROLL, lTrueFalse&
```

... where *lTrueFalse&* is Logical True or any nonzero value if you want SQL Tools to attempt to perform "normal" fetch operations, and a zero value (0) if you want it to perform only forward-only fetches. It should only be necessary to set this option under very unusual circumstances, but the following code.

```
lResult& = SQL_SetOptionSInt(OPT_USE_FETCHSCROLL)
```

...may be useful for troubleshooting if you suspect that a database is not *capable* of Fetch Scroll operation. The value of *lResult&* will be a Logical True or False value, depending on the current setting of the switch.

Using SQL_OpenDatabase1 and SQL_OpenDatabase2

The `SQL_OpenDB` function is actually a "wrapper" function actually performs three separate operations:

- 1) It uses the `SQL_OpenDatabase1` function to begin the process,
- 2) It uses the `SQL_SetDatabaseAttribUInt` function to specify the "as-needed" use of the ODBC Cursor Library (see **Speed Issues** below), and

3) It uses the `SQL_OpenDatabase2` function to complete the process.

You can perform these steps individually, instead of using `SQL_OpenDB`, if you need to open a database in an unusual way. For more information, please refer to the Reference Guide entries for `SQL_OpenDatabase1`, `SQL_SetDatabaseAttribUInt(DB_ODBC_CURSORS)`, and `SQL_OpenDatabase2`.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

It is very common and *completely normal* for this function to return `SUCCESS_WITH_INFO` and an Error Message that says...

"The driver doesn't support the version of ODBC behavior that the application requested".

That message means that your program specified ODBC 3.x behavior (via the `SQL_Init` or `SQL_Initialize` function) and that you have opened a database such as Access 97 that does not fully support ODBC 3.x behavior. Most ODBC drivers can emulate at least *some* 3.x behavior, so it is not usually a good idea to use a different *IODBCVersion* value with `SQL_Initialize`. If you do that, for instance, the `SUCCESS_WITH_INFO` message will no longer be generated but you will not be able to use certain ODBC functions such as Bookmarks.

Examples

```
lResult& = SQL_OpenDB("MYDATA.DSN")
```

...Or...

```
lResult& = SQL_OpenDB("DSN=SYS1;UID=JOHN;PWD=HELLO")
```

...Or...

```
lResult& = SQL_OpenDB("")
```

Driver Issues

None.

Speed Issues

By default, whenever it opens a database, SQL Tools tells your ODBC driver to use something called the "ODBC Cursor Library" on an as-needed basis. The ODBC Cursor Library simulates certain types of cursor operations if an ODBC driver does not support them directly. For example, if an ODBC driver supports only forward-only fetches, the ODBC Cursor Library can simulate other types of fetches.

While this is usually a good thing, it can impact the speed of database access. If speed is an extremely critical factor in your program design, and if your program does not need cursor behavior that is not directly supported by the ODBC driver, you might want to consider bypassing the use of the ODBC Cursor Library.

This can be accomplished by using the `SQL_OpenDatabase1` and `SQL_OpenDatabase2` functions *instead of* the `SQL_OpenDB` function. The

SQL_OpenDB and SQL_OpenDatabase functions are simply "wrappers" for the SQL_OpenDatabase1 and 2 functions, and they automatically tell the ODBC driver to use the ODBC Cursor Library in between those two steps. If you use the 1 and 2 functions directly, the "ODBC Cursor Library" step will be skipped, and the Library will not be used.

See Also

Opening a Database

SQL_OpenStatement

Summary

Opens a SQL statement and prepares it for use by the `SQL_Stmt` function. (This function is not used very often because the `SQL_Stmt` function automatically performs this step for you.)

Twin

`SQL_OpenStmt`

Family

Statement Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_OpenStatement(lDatabaseNumber&, _  
                             lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

See twin function `SQL_OpenStmt` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_OpenStatement` is identical to `SQL_OpenStmt`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_OpenStatement` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_OpenStmt` for complete details.

Example

See twin function `SQL_OpenStmt`.

Driver Issues

None.

Speed Issues

None.

See Also

Manually Opening and Closing Statements

SQL_OpenStmt

Summary

Opens a SQL statement and prepares it for use by the `SQL_Stmt` function. (This function is not used very often because the `SQL_Stmt` function automatically performs this step for you.)

Twin

`SQL_OpenStatement`

Family

Statement Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_OpenStmt
```

Parameters

None.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the statement is opened successfully, or an Error Code if it is not.

Remarks

Normally, SQL Tools automatically opens a statement whenever you use the `SQL_Stmt` function, so it is not usually necessary for your programs to use this function.

This function performs two different operations: **1)** It allocates a statement handle for a new statement, and **2)** It uses the values that your program specified with the `SQL_StmtMode` function (or the default values) to configure the statement handle. A wide variety of modes can be specified; see `SQL_StmtMode` for complete information.

If you have disabled the Statement-Auto-Open feature by using...

```
SQL_SetOptionSInt OPT_AUTOOPEN_STMT, 0
```

...then your program is responsible for manually opening statements by using the `SQL_OpenStmt` function.

If you have disabled the Statement Auto-Close feature by using...

```
SQL_SetOptionSInt OPT_AUTOCLOSE_STMT, 0
```

...then your program is responsible for using the `SQL_CloseStmt` function to close an already-open statement *before* using the `SQL_OpenStmt` function.

Diagnostics

This statement returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Manually Opening and Closing Statements

SQL_OptionResetAll

Summary

Resets all of the SQL Tools Options to their default values.

Twin

None.

Family

Configuration Family

Availability

Standard and Pro

Warning

None.

Syntax

```
SQL_OptionResetAll
```

Parameters

None.

Return Values

This function always returns `SUCCESS`, so it is safe to ignore the return value.

Remarks

This function re-initializes all of the various `SQL_SetOptionStr` and `SQL_SetOptionSInt` values that your program may have changed.

Diagnostics

None.

Example

```
Print SQL_OptionStr(OPT_MY_PROGRAM)

SQL_SetOptionStr OPT_MY_PROGRAM, "Hello World"
Print SQL_OptionStr(OPT_MY_PROGRAM)

SQL_OptionResetAll
Print SQL_OptionStr(OPT_MY_PROGRAM)
```

Results:

```
My Program
Hello World
My Program
```

Driver Issues: None.

Speed Issues: None.

See Also

Configuration Family

SQL_OptionSInt

Summary

This function can be used to obtain the current values of a wide variety of SQL Tools options, in numeric form.

Twin

None

Family

Configuration Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_OptionSInt(lOption&)
```

Parameters

lOption&

See **Remarks** below.

Return Values

If a valid *lOption&* value is used, this function will return the current value of the specified option, in numeric form. If an invalid *lOption&* value is used, zero (0) will be returned.

Remarks

Not *all* SQL Tools Option values are useful in numeric form. For example, the `OPT_MY_PROGRAM` option is usually used to store the name of your program, and using `SQL_OptionSInt` to return a numeric value for this string would usually return zero. It is possible, however, to assign a value like the string "2000" to the `OPT_MY_PROGRAM` option, in which case the `SQL_OptionSInt` function would return 2000.

For that reason, SQL Tools allows all options to be changed *and* read with both string and numeric functions.

In order to avoid errors when this document is updated in the future, a single list of all of the various SQL Tools Options is provided in the Reference Guide's `SQL_SetOptionStr` entry.

Diagnostics

None.

Example

```
'print the current setting of OPT_MAX_ERRORS:
Print SQL_OptionSInt(OPT_MAX_ERRORS)
```

Driver Issues

None

Speed Issues

None

See Also

Configuration Family

SQL_OptionStr

Summary

This function can be used to obtain the current values of a wide variety of SQL Tools options, in string form.

Twin

None

Family

Configuration Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_OptionStr(lOption&)
```

Parameters

lOption&

See **Remarks** below.

Return Values

If a valid *lOption&* value is used, this function will return the current value of the specified option, in string form. If an invalid *lOption&* value is used, an empty string will be returned.

Remarks

Not *all* SQL Tools Option values are useful in string form. For example, the `OPT_MAX_ERRORS` option is used to store the maximum number of errors that SQL Tools will store in the Error Stack, and using `SQL_OptionStr` to return a string value like "64" for this option would not normally be useful. It is possible, however, that you might want to obtain the string representation of an option's value for display purposes.

For that reason, SQL Tools allows all options to be changed *and* read with both string and numeric functions.

In order to avoid errors when this document is updated in the future, a single list of all of the various SQL Tools Options is provided in the Reference Guide's `SQL_SetOptionStr` entry.

Diagnostics

None.

Example

```
'Print the name of the current program.  
'(Unless you set this value, it defaults  
'to "My Program".)  
Print SQL_OptionStr(OPT_MY_PROGRAM)
```

Driver Issues

None.

Speed Issues

None.

See Also

Configuration Family

SQL_ParameterInfoUInt

Summary

Provides information about a bound statement parameter (a ? value-placeholder in a SQL statement).

Twin

SQL_ParamInfoUInt

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
dwResult??? = SQL_ParameterInfoUInt(lDatabaseNumber&, _  
                                     lStatementNumber&, _  
                                     lParameterNumber&, _  
                                     lInfoType&)
```

...Or...

```
lResult& = SQL_ParameterInfoUInt(lDatabaseNumber&, _  
                                 lStatementNumber&, _  
                                 lParameterNumber&, _  
                                 lInfoType&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ParamInfoUInt for complete details.

Return Values

See twin function SQL_ParamInfoUInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_ParameterInfoUInt is identical to SQL_ParamInfoUInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ParameterInfoUInt is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ParamInfoUInt for complete details.

Example

See Binding Statement Input Parameters for example code.

Driver Issues

See Binding Statement Input Parameters.

Speed Issues

See Binding Statement Input Parameters

See Also

Execution of SQL Statements

SQL_ParamInfoUInt

Summary

Provides information about a bound statement parameter (a ? value-placeholder in a SQL statement).

Twin

SQL_ParameterInfoUInt

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
dwResult??? = SQL_ParamInfoUInt(lParameterNumber&, _  
                                lInfoType&)  
  
...or...  
  
lResult& = SQL_ParamInfoUInt(lParameterNumber&, _  
                             lInfoType&)
```

Parameters

lParameterNumber&

The number of the parameter for which information is being retrieved, between one (1) and the number that is returned by the `SQL_StmtParamCount` function (i.e. the number of ? markers in a SQL statement).

lInfoType&

The type of information being requested. See **Remarks** below for a complete list of legal values.

Return Values

If valid parameters are used, this function will return the requested information. Otherwise, zero (0) will be returned.

Remarks

See Binding Statement Input Parameters for background information.

Please note that, unlike most Info values, these Info values are *not* cached by SQL Tools. They are requested from the ODBC driver whenever you use this function.

The *lInfoType&* parameter must be one of the following values:

`PARAM_DATA_TYPE`

The SQL Data Type of the bound parameter. This numeric value will correspond to a `SQL_` data-type constant. See SQL Data Types..

PARAM_DIGITS

The decimal digits value, for certain data types.

PARAM_NULLABLE

Returns one (1) if the parameter can accept a Null value, or zero (0) if it cannot.

PARAM_SIZE

The display size of the parameter's Data Type.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate value like "the data type of this parameter is 1 (`SQL_CHAR`)". It can, however, generate ODBC Error Messages and SQL Tools Error messages.

Example

See Binding Statement Input Parameters for example code.

Driver Issues

See Binding Statement Input Parameters.

Speed Issues

See Binding Statement Input Parameters

See Also

Execution of SQL Statements

SQL_PrimaryKeyCount

Summary

Indicates how many Primary Keys a table has.

Twin

SQL_PrimKeyCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_PrimaryKeyCount(lDatabaseNumber&, _  
                                lTableNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_PrimKeyCount for complete details.

Return Values

See twin function SQL_PrimKeyCount for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_PrimaryKeyCount is identical to SQL_PrimKeyCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_PrimaryKeyCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_PrimKeyCount for complete details.

Example

See twin function SQL_PrimKeyCount.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Unique Columns

SQL_PrimaryKeyInfoSInt

Summary

Provides information about a Primary Key, in numeric form.

Twin

SQL_PrimKeyInfoSInt

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_PrimaryKeyInfoSInt(lDatabaseNumber&, _  
                                   lTableNumber&, _  
                                   lKeyNumber&, _  
                                   lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_PrimKeyInfoSInt for complete details.

Return Values

See twin function SQL_PrimKeyInfoSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_PrimaryKeyInfoSInt is identical to SQL_PrimKeyInfoSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_PrimaryKeyInfoSInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_PrimKeyInfoSInt for complete details.

Example

See twin function SQL_PrimKeyInfoSInt.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also: Unique Columns

SQL_PrimaryKeyInfoStr

Summary

Provides information about a Primary Key, in string form.

Twin

SQL_PrimKeyInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_PrimaryKeyInfoStr(lDatabaseNumber&, _  
                                lTableNumber&, _  
                                lKeyNumber&, _  
                                lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_PrimKeyInfoStr for complete details.

Return Values

See twin function SQL_PrimKeyInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber*& parameter, SQL_PrimaryKeyInfoStr is identical to SQL_PrimKeyInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_PrimaryKeyInfoStr is shown below.

For more information about using *lDatabaseNumber*& and *lStatementNumber*& in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_PrimKeyInfoStr for complete details.

Example

See twin function SQL_PrimKeyInfoStr.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also

Unique Columns

SQL_PrimKeyCount

Summary

Returns the number of Primary Keys that are associated with a table.

Twin

SQL_PrimaryKeyCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_PrimKeyCount(lTableNumber&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

Return Values

If a valid *lTableNumber&* is used, and if the database is open, this function will return the number of Primary Keys that the table has. Otherwise, it will return zero (0).

Remarks

A Primary Key is a column (or a set of columns) that uniquely identifies a row in a table.

Diagnostics

This function does not return Error Codes because an Error Code like SUCCESS_WITH_INFO (value 1) could be confused with a legitimate return value such as "the table has 1 primary key". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the number of Primary Keys  
'that are associated with table #1.  
Print SQL_PrimKeyCount(1)
```

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Unique Columns

SQL_PrimKeyInfoSInt

Summary

Provides information about a Primary Key, in numeric form.

Twin

SQL_PrimaryKeyInfoSInt

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_PrimKeyInfoSInt(lTableNumber&, _  
                                lKeyNumber&, _  
                                lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

lKeyNumber&

The number of a Primary Key, between one (1) and the number returned by the `SQL_PrimKeyCount` function.

lInfoType&

The type of numeric information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, this function will return the requested information. Otherwise, zero (0) will be returned.

Remarks

A Primary Key is a column (or a set of columns) that uniquely identifies a row in a table. See Primary Keys for more information.

Please note that not *all* of the information that is available about a table's Primary Keys is useful in numeric form. For a list of *lInfoType&* values that can be used to obtain *string* information about a table's Primary Keys, see `SQL_PrimKeyInfoStr`.

In order to obtain numeric information about a table's Primary Keys, the *lInfoType&* parameter must have the following value:

PKEY_SEQ

The Primary Key's column sequence number in key, starting with 1. If a Primary Key is made up of two or more columns (in order to provide a unique value), this *lInfoType&* can be used to determine the order in which the columns are assembled.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

`SQL_PrimKeyCount`

SQL_PrimKeyInfoStr

Summary

Provides information about a Primary Key, in string form.

Twin

SQL_PrimaryKeyInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_PrimKeyInfoStr(lTableNumber&, _  
                               lKeyNumber&, _  
                               lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lKeyNumber&

The number if a Primary Key, between one (1) and the number returned by the SQL_PrimKeyCount function.

lInfoType&

The type of string information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, this function will return the requested information. Otherwise, an empty string will be returned.

Remarks

A *Primary Key* is a column (or a set of columns) that uniquely identifies a row in a table. See Primary Keys for more information.

Please note that not *all* of the information that is available about a table's Primary Keys is useful in string form. For a list of *lInfoType&* values that can be used to obtain *numeric* information about a table's Primary Keys, see SQL_PrimKeyInfoSInt.

In order to obtain string information about a table's Primary Keys, the *lInfoType&* parameter must be one of the following values:

PKEY_TABLE_CATALOG, PKEY_TABLE_SCHEMA, and PKEY_TABLE_NAME

The catalog, schema, and table names that are associated with the Primary Key.

PKEY_COLUMN_NAME

The column name of the Primary key.

PKEY_NAME

The Primary Key's name.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Unique Columns

SQL_ProcColCount

Summary

Returns the number of columns (result columns, input parameters, etc.) that a Stored Procedure has.

Twin

SQL_ProcedureColumnCount

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcColCount(lProcedureNumber&)
```

Parameters

lProcedureNumber&

The number of a procedure, between one (1) and the number of procedures that is returned by the `SQL_ProcCount` function.

Return Values

This function will return zero (0) if the procedure does not have any columns, or a positive number that indicates the total number of columns.

Remarks

Procedures can have three types of columns:

- 1) Input columns (i.e. parameters that must be defined before a procedure can be executed),
- 2) Output columns (i.e. the columns of the result set that will be produced when the procedure is executed), and
- 3) A "column" that contains the return value of the procedure.

The `SQL_ProcColCount` function returns the *total* number of columns that a procedure has.

See Stored Procedures for more information.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value, such as "this procedure has one column". This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

None.

See Also

Binding Statement Input Parameters
Execution of SQL Statements

SQL_ProcColInfoSInt

Summary

Provides information about a column (result column, input parameter, etc.) of a Stored Procedure, in numeric form.

Twin

SQL_ProcedureColumnInfoSInt

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcColInfoSInt(lProcedureNumber&, _  
                                lProcColumnNumber&, _  
                                lInfoType&)
```

Parameters

lProcedureNumber&

The number of a procedure, between one (1) and the value returned by SQL_ProcCount.

lProcColumnNumber&

The number of a column of a procedure, between one (1) and the value returned by SQL_ProcColCount.

lInfoType&

The type of numeric information being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, this function will return a numeric value that contains the information that was requested. If an invalid parameter is used, zero (0) will be returned.

Remarks

This function is used to obtain numeric information about a procedure's columns.

Keep in mind that procedures have three different kinds of columns:

- 1)** Input columns (i.e. parameters that must be defined before a procedure can be executed),
- 2)** Output columns (i.e. the columns of the result set that will be produced when the procedure is executed), and
- 3)** A "column" that contains the return value of the procedure.

Your program should use the `PROC_COLUMN_TYPE` value (see below) to determine each column's type, to help put the rest of the column's information in context. See Stored Procedures for more information.

Please note that not *all* procedure column information is useful in numeric form. For a list of *InfoType&* values that can be used to obtain *string* information about a column, see `SQL_ProcColInfoStr`.

The *InfoType&* parameter must be one of the following values when you are getting numeric information about a column:

`PROC_COLUMN_BUFFER_LENGTH`

The buffer size (in bytes) that is required for the column.

`PROC_COLUMN_CHAR_OCTET_LENGTH`

ODBC 3.x+ ONLY: The maximum length (in bytes) of a character or binary column. For all other data types, this column returns zero (0).

`PROC_COLUMN_DATA_TYPE`

The column's SQL Data Type. This will always be one of the standard SQL Data Types, such as `SQL_CHAR` or `SQL_INTEGER`.

`PROC_COLUMN_DECIMAL_DIGITS`

The number of decimal digits that the column has.

`PROC_COLUMN_DEFAULT_VALUE`

ODBC 3.x+ ONLY: The column's default value.

This InfoType& can return either numeric or string information. Your program should check for both.

If the Null value was specified as the default value, or if no default was specified, the string value `NULL` (i.e. the *word* "NULL") will be returned. If the default value cannot be represented without truncation, the word "TRUNCATED" will be returned.

This value can be used when generating a new column definition, except when it contains the word `TRUNCATED`.

`PROC_COLUMN_NULLABLE`

One of the following values:

`SQL_NO_NULLS` (The procedure column does not accept Null values.)

`SQL_NULLABLE` (The procedure column does accept Null values.)

`SQL_NULLABLE_UNKNOWN` (It is not known whether or not the procedure column accepts Null values.)

`PROC_COLUMN_NUM_PREC_RADIX`

The Num Prec Radix of the column.

PROC_COLUMN_ORDINAL_POSITION

ODBC 3.x+ ONLY: The column's number.

For input and output parameters, this is the ordinal position of the parameter in the procedure definition, in increasing order, starting at 1.

For result-set columns, this is the ordinal position of the column in the result set, with the first column in the result set being column number 1. If there are multiple result sets, the column positions are returned in different orders by different ODBC drivers, so you will need to determine the meaning of this value experimentally.

For a return value column, zero (0) is returned.

PROC_COLUMN_SIZE

The display size of the column.

PROC_COLUMN_SQL_DATA_TYPE

ODBC 3.x+ ONLY: This value is the same as PROC_COLUMN_DATA_TYPE except for datetime and interval data types. For datetimes and intervals, this value will be SQL_INTERVAL or SQL_DATETIME, and the PROC_COLUMN_SQL_DATETIME_SUB value will be the subcode for the specific interval or datetime data type.

PROC_COLUMN_SQL_DATETIME_SUB

ODBC 3.x+ ONLY: The subtype code for datetime and interval data types, such as ODBC2_INTERVAL_MINUTE.

PROC_COLUMN_TYPE

The column's type. This will always be one of the following values:

Input (Parameter) Columns: PROC_INPUT_PARAM, PROC_OUTPUT_PARAM, PROC_INPUT_OUTPUT_PARAM, or PROC_UNKNOWN_TYPE_PARAM.

Output (Result) Columns: PROC_RESULT_COLUMN

Return Values: PROC_RETURN_VALUE

Diagnostics

This function does not return Error Codes because an Error Code like SUCCESS_WITH_INFO (value 1) could be confused with a legitimate return value. This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Binding Statement Input Parameters
Execution of SQL Statements

SQL_ProcColInfoStr

Summary

Provides information about a column (result column, input parameter, etc.) of a Stored Procedure, in string form.

Twin

SQL_ProcedureColumnInfoStr

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_ProcColInfoStr(lProcedureNumber&, _  
                               lProcColumnNumber&, _  
                               lInfoType&)
```

Parameters

lProcedureNumber&

The number of a procedure, between one (1) and the value returned by SQL_ProcCount.

lProcColumnNumber&

The number of a column of a procedure, between one (1) and the value returned by SQL_ProcColCount.

lInfoType&

The type of string information being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, this function will return a string that contains the information that was requested. If an invalid parameter is used, an empty string will be returned.

Remarks

This function is used to obtain string information about a procedure's columns.

Keep in mind that procedures have three different kinds of columns:

- 1)** Input columns (i.e. parameters that must be defined before a procedure can be executed),
- 2)** Output columns (i.e. the columns of the result set that will be produced when the procedure is executed), and
- 3)** A "column" that contains the return value of the procedure.

Your program should use the PROC_COLUMN_TYPE value (see SQL_ProcColInfoSInt) to determine each column's type, to help put the rest of the column's information in context. See Stored Procedures for more information.

Please note that not *all* procedure column information is useful in string form. For a list of *InfoType*& values that can be used to obtain *numeric* information about a column, see `SQL_ProcColInfoSInt`.

The *InfoType*& parameter must be one of the following values when you are getting string information about a column:

`PROC_COLUMN_CATALOG`

The procedure's Catalog Name.

`PROC_COLUMN_DATA_TYPE_NAME`

The column's datasource-dependent data type name, such as "INTEGER" or "COUNTER".

`PROC_COLUMN_DEFAULT_VALUE`

ODBC 3.x+ ONLY: The column's default value.

This InfoType& can return either numeric or string information. Your program should check for both.

If the Null value was specified as the default value, or if no default was specified, the string value NULL (i.e. the *word* "NULL") will be returned. If the default value cannot be represented without truncation, the word "TRUNCATED" will be returned.

This value can be used when generating a new column definition, except when it contains the word TRUNCATED.

`PROC_COLUMN_IS_NULLABLE`

ODBC 3.x+ ONLY: The word "NO" if the column does not include nulls, "YES" if the column can include nulls, or an empty string if nullability is unknown. (Also see `SQL_ProcColInfoSInt (PROC_COLUMN_NULLABLE)`.)

`PROC_COLUMN_NAME`

The column's name.

`PROC_COLUMN_PROC_NAME`

The name of the procedure that uses this column.

`PROC_COLUMN_REMARKS`

An optional description field.

`PROC_COLUMN_SCHEMA`

The procedure's Schema Name.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Binding Statement Input Parameters
Execution of SQL Statements

SQL_ProcCount

Summary

Provides a count of the Stored Procedures that a database contains.

Twin

SQL_ProcedureCount

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcCount
```

Parameters

None.

Return Values

This function will return zero (0) if a database does not contain any stored procedures, or a positive number that indicates the total number of procedures.

Remarks

See Stored Procedures for more information.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a result like "this database contains one stored procedure". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Stored Procedures.

Driver Issues

None

Speed Issues

See Cached Information.

See Also

Binding Statement Input Parameters
Execution of SQL Statements

SQL_ProcedureColumnCount

Summary

Provides a count of the Stored Procedures that a database contains.

Twin

SQL_ProcColCount

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcedureColumnCount(lDatabaseNumber&, _  
                                     lProcedureNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ProcColCount for complete details.

Return Values

See twin function SQL_ProcColCount for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_ProcedureColumnCount is identical to SQL_ProcColCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ProcedureColumnCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ProcColCount for complete details.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Binding Statement Input Parameters

Execution of SQL Statements

SQL_ProcedureColumnInfoSInt

Summary

Provides information about a column (result column, input parameter, etc.) of a Stored Procedure, in numeric form.

Twin

SQL_ProcColInfoSInt

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcedureColumnInfoSInt(lDatabaseNumber&, _  
                                       lProcedureNumber&, _  
                                       lProcColumnNumber&, _  
                                       lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ProcColInfoSInt for complete details.

Return Values

See twin function SQL_ProcColInfoSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ProcedureColumnInfoSInt is identical to SQL_ProcColInfoSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ProcedureColumnInfoSInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ProcColInfoSInt for complete details.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See [Cached Information](#).

See Also

[Binding Statement Input Parameters](#)
[Execution of SQL Statements](#)

SQL_ProcedureColumnInfoStr

Summary

Provides information about a column (result column, input parameter, etc.) of a Stored Procedure, in string form.

Twin

SQL_ProcColInfoStr

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcedureColumnInfoStr(lDatabaseNumber&, _  
                                       lProcedureNumber&, _  
                                       lProcColumnNumber&, _  
                                       lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ProcColInfoStr for complete details.

Return Values

See twin function SQL_ProcColInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ProcedureColumnInfoStr is identical to SQL_ProcColInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ProcedureColumnInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ProcColInfoStr for complete details.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See [Cached Information](#).

See Also

[Binding Statement Input Parameters](#)
[Execution of SQL Statements](#)

SQL_ProcedureCount

Summary

Provides a count of the Stored Procedures that a database contains.

Twin

SQL_ProcCount

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcedureCount(lDatabaseNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

Return Values

See twin function `SQL_ProcCount` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_ProcedureCount` is identical to `SQL_ProcCount`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_ProcedureCount` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_ProcCount` for complete details.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Execution of SQL Statements

Binding Statement Input Parameters

SQL_ProcedureInfoSInt

Summary

Provides information about a Stored Procedure, in numeric form.

Twin

SQL_ProcInfoSInt

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcedureInfoSInt(lDatabaseNumber&, _  
                                lProcedureNumber&, _  
                                lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ProcInfoSInt for complete details.

Return Values

See twin function SQL_ProcInfoSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ProcedureInfoSInt is identical to SQL_ProcInfoSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ProcedureInfoSInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ProcInfoSInt for complete details.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also: Binding Statement Input Parameters Execution of SQL Statements

SQL_ProcedureInfoStr

Summary

Provides information about a Stored Procedure, in string form.

Twin

SQL_ProcInfoStr

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcedureInfoStr(lDatabaseNumber&, _  
                                lProcedureNumber&, _  
                                lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ProcInfoStr for complete details.

Return Values

See twin function SQL_ProcInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ProcedureInfoStr is identical to SQL_ProcInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ProcedureInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ProcInfoStr for complete details.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also: Execution of SQL Statements, Binding Statement Input Parameters

SQL_ProcInfoSInt

Summary

Provides information about a Stored Procedure, in numeric form.

Twin

SQL_ProcedureInfoSInt

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

See **Remarks** regarding `PROC_INPUT_PARAM_COUNT`, `PROC_OUTPUT_PARAM_COUNT`, and `PROC_RESULT_COLUMN_COUNT`.

Syntax

```
lResult& = SQL_ProcInfoSInt(lProcedureNumber&, _  
                             lInfoType&)
```

Parameters

lProcedureNumber&

The number of a stored procedure, between one (1) and the number of stored procedures that a database has, as returned by `SQL_ProcCount`.

lInfoType&

The type of numeric information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, this function will return the requested numeric information. Otherwise, zero (0) will be returned.

Remarks

Not *all* types of procedure information are useful in numeric form. For a list of *lInfoType&* values that can be used to obtain *string* information, see `SQL_ProcInfoStr`.

Here is the list of *lInfoType&* values that can be used to obtain numeric information:

`PROC_TYPE`

The procedure's type. This will always be one of the following values:

`SQL_PT_PROCEDURE` (The procedure does not have a return value.)

`SQL_PT_FUNCTION` (The procedure is a function, and therefore has a return value.)

`SQL_PT_UNKNOWN` (It is not known whether or not the procedure returns a value.)

PROC_INPUT_PARAM_COUNT,
PROC_OUTPUT_PARAM_COUNT, and
PROC_RESULT_COLUMN_COUNT

WARNING: Applications should not rely on these values.

Even though these values were defined in the ODBC 2.0 specification, they are still defined as "reserved for future use" by the ODBC 3.5 specification.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "procedure type 1". This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Execution of SQL Statements
Binding Statement Input Parameters

SQL_ProcInfoStr

Summary

Provides information about a Stored Procedure, in string form.

Twin

SQL_ProcedureInfoStr

Family

Stored Procedure Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ProcInfoStr(lProcedureNumber&, _  
                           lInfoType&)
```

Parameters

lProcedureNumber&

The number of a stored procedure, between one (1) and the number of stored procedures that a database has, as returned by SQL_ProcCount.

lInfoType&

The type of string information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, this function will return a string that contains the requested information. Otherwise, an empty string will be returned.

Remarks

Not *all* types of procedure information are useful in string form. For a list of *lInfoType&* values that can be used to obtain *numeric* information, see SQL_ProcInfoInt.

Here is the list of *lInfoType&* values that can be used to obtain string information:

PROC_CATALOG

The procedure's catalog name.

PROC_NAME

The procedure's name.

PROC_REMARKS

An optional description.

PROC_SCHEMA

The procedure's schema name.

Diagnostics

This function does not return Error Codes because it returns string values. This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Stored Procedures.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Execution of SQL Statements
Binding Statement Input Parameters

SQL_ResColBInt

Summary

Provides the value of one column of one row of a result set, in Big Integer (BInt) form. (Big Integers are returned as strings; see **Remarks** below.)

Twin

SQL_ResultColumnBInt

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_ResColBInt(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the SQL_ResColCount function.

Return Values

This function returns a string that represents a Big Integer numeric value in the SQL_BIGINT range.

Remarks

This function is usually used to obtain the value of a SQL_BIGINT column, but it can also be used to obtain other numeric values in string form.

Because most 32-bit computer languages (including Visual Basic and PowerBASIC) do not support a variable type that can handle the entire Big Integer range, the SQL_ResColBInt function returns a string that represents a numeric value. Your program is then responsible for converting the string into a numeric value, if that is what you need. (The VAL function and the BAS_QUAD variable type can be used if the SQL_BIGINT column contains a *signed* Big Integer value.)

This function can also be used to obtain string representations of other Integer-type numeric columns, such as SQL_INTEGER, SQL_SMALLINT, SQL_TINYINT, and SQL_BIT. **IMPORTANT NOTE:** The SQL_ResColBInt function treats all of these Integer-type columns as *signed* values, so if the column actually contains unsigned data you will receive erroneous results from this function. For example, if a column contains a BAS_BYTE value in the range 0 to 255, it will be interpreted as a string between "-128" and "127".

This function can also be used to obtain string representations of the integer *portions* of non-integer numeric columns, such as SQL_DECIMAL, SQL_NUMERIC, SQL_REAL, and SQL_DOUBLE.

If you attempt to use `SQL_ResColBInt` to obtain a value for any other type of result column, an empty string will be returned.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the contents of column  
'number 8 in Big Integer form:  
Print SQL_ResColBInt(8)
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColBufferPtr

Summary

Provides a pointer (ptr) to the memory buffer that SQL Tools uses for a result column.

Twin

SQL_ResultColumnBufferPtr

Family

Result Column Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ResColBufferPtr(lColumnNumber&)
```

Parameters

lColumnNumber&

The column number of the result column buffer that you need to access via a memory pointer, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function returns a `BAS_LONG` value that represents a memory pointer which points to the first byte of the memory buffer that SQL Tools is using for the specified column.

If you attempt to obtain a pointer to a buffer for a column that has not been autobound by SQL Tools, this function will return zero (0).

Remarks

This function, along with the `SQL_ResColSize` function, can be used to obtain information about the location and length of the "bind buffer" that SQL Tools is using for a particular column. It is possible for your program to use this information to access the buffer directly with Proxy Binding, instead of using a function like `SQL_ResColStr` or `SQL_ResColSInt`. This can be an acceptable (and attractive) alternative to Manual Binding or Direct Binding, especially if the column can *usually* be accessed "normally" and only *sometimes* needs to be accessed directly.

Diagnostics

This function does not return Error Codes because it returns a memory pointer (which can have any value in the `BAS_LONG` range) so it would not be possible for a program to distinguish between an Error Code and a valid pointer. This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

This function can be used to increase the speed with which your program is able to access the columns of a result set, without resorting to Manual Binding or Direct Binding.

See Also

Result Column Binding

SQL_ResColCount

Summary

Provides a value which indicates the number of columns in a result set.

Twin

SQL_ResultColumnCount

Family

Result Count Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColCount
```

Parameters

None.

Return Values

This function will return zero (0) if a SQL statement did not generate a result set, or a positive number that indicates the number of result columns that were generated.

Remarks

If speed is an important factor in your program's design, and if you suspect that a result set will be empty, it is usually faster to check this function's value than to attempt to use `SQL_Fetch` and `SQL_EOD`. See Detecting "No Data At All".

IMPORTANT NOTE: If bookmarks are being used, the return value of the `SQL_ResColCount` does *not* include the bookmark column (column zero). Strictly speaking, this function returns the number of the highest-numbered column, not the result column "count". But because "count" is the official ODBC terminology, it is used by SQL Tools.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with the result "this result set has one column". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

See **Remarks** above.

See Also: Result Column Family

SQL_ResColDate

Summary

Provides the value of one column of one row of a result set as a string that represents a formatted Date value.

Twin

SQL_ResultColumnNameDate

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColDate(lColumnName&. _  
                           sFormat$)
```

Parameters

lColumnName&

The number of a result column, between one (1) and the number that is returned by the SQL_ResColCount function. The column must be a SQL_TIMESTAMP, SQL_TYPE_TIMESTAMP, SQL_DATE, or SQL_TYPE_DATE column.

sFormat\$

PLEASE NOTE: The *sFormat\$* strings that are described here are *case-sensitive*. The *sFormat\$* parameter can be **1)** An empty string, for the system default date format or **2)** a date-formatting string that follows the guidelines that are described in **Remarks** below, or **3)** the string "long" for the system long-date format, or **4)** the string "short" for the system short-date-format or **4)** the string "default" for the date format that was specified with the SQL_SetOptionStr OPT_DATE_FORMAT setting. If you use "default" and you have not set the OPT_DATE_FORMAT setting, the system default date format will be used.

Return Values

This function returns a string that represents a Date value from a Date or Date-Time column of a result set. If the *lColumnName&* parameter is not the number of a Date or Date-Time column, an empty string will be returned.

Remarks

The *sFormat\$* parameter is typically a string that looks something like this:

yyyy-MM-dd

If a result set contained the date value January 1, 2000 that *sFormat\$* string would produce a date string that looked like this:

2000-01-01

Each character or group of characters in the formatting string represents either **1**) a different part of a date (month, year, etc.) or **2**) a literal character such as a hyphen (-).

You may use the following date-part codes in the formatting string:

d Day of month as one digit (days 1-9) or two digits (days 10-31).

dd Day of month as two digits, with leading zero for days 01-09.

ddd Day of week as a three-letter abbreviation. The abbreviation is defined by the runtime system's "locale" settings, including the native language setting.

dddd Day of week full name. The name is defined by the runtime system's "locale" settings, including the native language setting.

M Month as one digit (months 1-9) or two digits (months 10-12).

MM Month as two digits, with leading zero for months 01-09.

MMM Month as a three-letter abbreviation. The abbreviation is defined by the runtime system's "locale" settings, including the native language setting.

MMMM Month full name. The name is defined by the runtime system's "locale" settings, including the native language setting.

y Year as one digits (years 0-9) or two digits (years 10-99)

yy Year as two digits, with leading zero for years 01-09.

yyyy Year and century (always four digits).

gg Period/era string, if any, as defined by the runtime system's "locale" settings. (This string is often empty.)

Spaces are always treated as literal characters, i.e. they will appear in the final string exactly as you type them. Most other characters (punctuation, etc.) are also interpreted as literal characters, but to be certain you should enclose literal characters in single-quotes, such as this example where the comma (,) is quoted.

`ddd', ' MMM dd yy ...would produce...`

`Wed, Aug 31 94`

If you use a single-quote character in a formatting string, it will be interpreted as a delimiter even if it is not "paired" properly, i.e. even if a matching "close quote" is not used.

In order to include a *literal* single-quote character in a date-formatting string, you must use *four consecutive* single quotes, like this...

`ddd', ' ' ' ' ' MMM dd ' ' ' 'yy ...would produce...`

`Wed, Aug 31 '94`

Using four single-quotes produces an output string that actually *includes* one single-quote character.

Advanced Techniques

It is possible to affect the system-default Date formatting that is performed by the `SQL_ResColDate` function when an empty formatting string (" ") is used. (These options will have *no effect* if a formatting string is used.) Use the `SQL_SetOptionSInt` function to change these options:

`OPT_DATE_LOCALE`

Use `LOCALE_SYSTEM_DEFAULT`, `LOCALE_USER_DEFAULT`, or any other Standard Win32 `LOCALE_` constant. The default value is `LOCALE_USER_DEFAULT`.

`OPT_DATE_FLAGS`

Use the standard Win32 values `LOCALE_NOUSEROVERRIDE`, `DATE_SHORTDATE`, `DATE_LONGDATE`, and/or `DATE_USE_ALT_CALENDAR`. The default value is zero (0), i.e. no special flags are used.

The names of these standard Windows constants are reasonably self-explanatory. Please refer to the Win32 API documentation for complete information.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Get a date string from column 7  
  
sResult$ = SQL_ResColDate(7,"yyyy-MM-dd")
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColDateTime

Summary

Provides the value of one column of one row of a result set as a string that represents a formatted Date-Time value.

Twin

SQL_ResultColumnDateTime

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColDateTime(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the SQL_ResColCount function. The column must be a SQL_TIMESTAMP or SQL_TYPE_TIMESTAMP column.

Return Values

This function returns a string that represents the value in a Date-Time column of a result set. If the *lColumnNumber&* parameter is not the number of a Date-Time column, an empty string will be returned.

Remarks

The format of the date-time sting that is returned by this function is determined by three (3) different option settings. To change the default values, use the SQL_SetOptionStr function. (See **Examples** below.)

OPT_DATE_FORMAT

Determines the date format. The default value is an empty string (" ") which produces the system default date format. See the SQL_ResColDate function for a complete description of the strings that can be used for this option.

OPT_TIME_FORMAT

Determines the time format. The default value is an empty string (" ") which produces the system default time format. See the SQL_ResColTime function for a complete description of the strings that can be used for this option.

OPT_DATETIME_SEPARATOR

The character(s) that appear between the date and time. The default value is

a single space character.

So the default format for the string that is returned by the `SQL_ResColDateTime` function is the system default date format, followed by a single space, followed by the default system time format.

The default system date and time formats are determined by the runtime system's "locale" settings, including the native language setting.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Change the date format that is returned
'by the SQL_ResColDateTime function
SQL_SetOptionStr OPT_DATE_FORMAT, "dd-MM-yy"

'Change the time format that is returned
'by the SQL_ResColDatetime function
SQL_SetOptionStr OPT_TIME_FORMAT, "hh:mm tt"

'Change the separator characters
SQL_SetOptionStr OPT_DATETIME_SEPARATOR, " at "

'Get a date-time string from column 10
'using those formats...
sResult$ = SQL_ResColDateTime(10)

Typical result: 01-02-99 at 12:15 pm
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColDateTimePart

Summary

Provides the value of one column of one row of a result set as a "part" of a Date or Time, such as the Month or Hour, in numeric form.

Twin

SQL_ResultColumnDateTimePart

Family

Result Column Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ResColDateTimePart(lColumnNumber&, _  
                                   lPart&)
```

Parameters

lColumnNumber&

The column of the result set that contains a Date, Time, or DateTime value, between one (1) and the number that is returned by the SQL_ResColCount function. See **Remarks** below.

lPart&

A constant that indicates which part of the value should be returned. See **Remarks** below for a list of valid constants.

Return Values

This function returns either **1**) a number greater than or equal to zero (0) which indicates the value of the specified date/time part, or **2**) negative one (-1) if a Null Value or an error condition is detected.

Remarks

This function can be used to obtain a part of a Date, Time, or DateTime result column value, in numeric form. This is *much* more efficient than obtaining a formatted string from the SQL_ResColDate, SQL_ResColTime, or SQL_ResColDateTime function and then parsing the string to obtain a numeric value.

The *lColumnNumber&* parameter must contain the number of a result column that is one of the following types: SQL_TIMESTAMP, SQL_TYPE_TIMESTAMP, SQL_DATE, SQL_TYPE_DATE, SQL_TIME, or SQL_TYPE_TIME. If you attempt to use this function with a column of any other type, an ERROR_CANT_BE_DONE error message will be generated and the return value of this function will be negative one (-1). If the column contains a Null Value then negative one will also be returned, but no Error Message will be generated.

The *lPart&* parameter must be one of the following values: PART_YEAR, PART_MONTH, PART_DAY, PART_DOW, PART_HOUR, PART_MINUTE, PART_SECOND, or PART_NANOSEC.

IMPORTANT NOTE: The PART_DOW constant tells SQL_ResColDateTimePart

to return the Day Of Week as a number. The return value will be DOW_SUN, DOW_MON, DOW_TUE, DOW_WED, DOW_THU, DOW_FRI, or DOW_SAT.. **The value of DOW_SUN is one (1) and the value of DOW_SAT is seven (7).** This is the "core" numbering system that all ODBC databases use. Fortunately this system is also compatible with the Visual Basic system (vbSunday = 1 through vbSaturday = 7) and some Windows API functions. If you need to use a different system (such as the Windows standard Sunday = 0) you will need to adjust the SQL_ResColDateTimePart value using BASIC code.

If you specify an invalid *lPart&* for the column type, such as using PART_YEAR for a SQL_TIME column or PART_MINUTE for a SQL_DATE column, this function will return negative one (-1) and an ERROR_CANT_BE_DONE error message will be generated. Note that PART_NANOSEC is *not* part of a SQL_TIME structure. PART_NANOSEC can be used only with SQL_TIMESTAMP and SQL_TYPE_TIMESTAMP columns.

Diagnostics

This function does not return Error Codes because an Error Code like SUCCESS_WITH_INFO (value 1) could be confused with a legitimate return value like an Hour value of 1. This function does, however, return negative one (-1) to indicate an invalid date-time part value if it is unable to retrieve a valid value. (Negative one is never a valid value for any date-part or time-part.)

This function can also generate SQL Tools Error Messages.

Example

```
'Get the Month value from column 8
lResult& = SQL_ResColDateTimePart(8, PART_MONTH)
If lResult& = -1 Then
    'The column contains a Null Value
    'or an error was detected.
Else
    'lResult& now contains a Month
    'value between 1 and 12.
End If
```

Driver Issues

None.

Speed Issues

This function is significantly faster than using the SQL_ResColDate, SQL_ResColTime, or SQL_ResColDateTime function, but not as fast as the "Advance Technique" that is discussed in SQL_TIMESTAMP.

See Also

SQL_TIMESTAMP, SQL_DATE, SQL_TIME

SQL_ResColFloat

Summary

Provides the value of one column of one row of a result set, in Floating Point form.

Twin

SQL_ResultColumnFloat

Family

Result Column Family

Availability

Standard and Pro

Warning

See important notes about `SQL_FLOAT` data below.

Syntax

```
dpResult# = SQL_ResColFloat(lColumnNumber&)
```

Parameters

lColumnNumber&

The column of the result set that contains the Floating Point value, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function returns a double-precision floating point number (`BAS_DOUBLE`) that represents the value of a column of a result set.

Remarks

This function is usually used to obtain the numeric value of a floating point column (`SQL_REAL` or `SQL_DOUBLE`).

This function returns a double-precision (`BAS_DOUBLE`) value even if the column is single-precision (`SQL_REAL`), but that is not meant to imply that the accuracy of a single-precision column is *improved*. If a single-precision column is accessed with this function, your program should assume that any digits that are beyond the single-precision limit are unreliable.

For example, if a double-precision column is used to store the results of the equation "two divided by three", the result will be...

```
0.6666666666666667
```

If the same results are stored in a single precision column, the result will be...

```
0.6666667
```

Those are the exact values that will be returned by the `SQL_ResColFloat` functions. Keep in mind that the second (single-precision) value is equivalent to...

```
0.6666667000000000
```

...and that is clearly *not* the same as the first value. If your program were to obtain

that last value from a *double*-precision column, it should assume that all of the digits are accurate.

Because of the way computers perform floating-point math, it is also possible to obtain single-precision values similar to this:

```
0.6666667000000001
```

...so simply trimming the trailing zeros is not sufficient. Your program should *round* single-precision values to the appropriate number of digits.

Even though the `SQL_ResColFloat` function is *capable* of returning double-precision values, it can only return numbers that are as precise as the actual result column.

The `SQL_ResColFloat` function can also be used to access other types of result columns. For example, if a `SQL_DECIMAL`, `SQL_NUMERIC`, or `SQL_CHAR` column contains a value like the string "1.2345", the `SQL_ResColFloat` function can be used to obtain a floating-point numeric representation of the string.

IMPORTANT NOTE: If a `SQL_DECIMAL`, `SQL_NUMERIC`, or `SQL_CHAR` column contains a string representation of a value that is beyond double-precision (i.e. one that has too many digits to fit in a `BAS_DOUBLE` variable), the `SQL_ResColFloat` function will return a result that is *rounded* to double-precision.

The `SQL_ResColFloat` function can also be used to obtain floating point values for integer-type columns such as `SQL_INTEGER`, `SQL_SMALLINT`, `SQL_TINYINT`, and `SQL_BIT`. All of these column types are treated as *Signed Integers*, which are then returned by `SQL_ResColFloat` as a double-precision value.

Important Notes about SQL Float Data

It would seem logical to use the `SQL_ResColFloat` function to obtain results from a `SQL_FLOAT` column, but that is not always possible. It is very important to keep in mind that `SQL_FLOAT` is not a 100% standardized ODBC data type. A `SQL_FLOAT` is a *user-defined* (or DBMS-defined) data type which *may or may not* correspond to a double-precision numeric value. It often does, but not always. If `SQL_FLOAT` is defined in an unusual way in your database, wildly incorrect numeric values may be returned by `SQL_ResColFloat`. We therefore strongly recommend that you experiment with retrieving known values from a `SQL_FLOAT` column, to make sure you get the values you expect, before assuming that it can be accessed with `SQL_ResColFloat`. If you do *not* obtain the results that you expect, that means that the database is using a `SQL_FLOAT` bit-pattern which SQL Tools does not recognize. In that case you should manually re-bind the `SQL_FLOAT` column to a string buffer such as a `BAS_STRING`, so that your ODBC driver will interpret the bit pattern for you and return a number in string form. You can then use the `SQL_ResColStr` function to obtain a string value like "12.345". If you use `SQL_ResColStr` with a `SQL_FLOAT` column *without* re-binding, it will return the 8-byte binary representation of the numeric value, as stored in the database.

Diagnostics

This function does not return Error Codes because an Error Code like

SUCCESS_WITH_INFO (value 1) could be confused with a legitimate return value like 1.00000. This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the contents of column 33  
'as a floating point number  
Print SQL_ResColFloat(33)
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColInd

Summary

Provides the value of the Indicator that is associated with one column of a result set.

Twin

SQL_ResultColumnInd

Family

Result Column Binding Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColInd(lColumnNumber&)
```

Parameters

lColumnNumber&

The column of the result set for which you need the Indicator value, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function returns the Indicator value for a column. See **Remarks** below for more information about what the various Indicator values mean.

Remarks

This function will return zero (0) until `SQL_Fetch` or `SQL_FetchRel` is used to retrieve the first row of a result set. After that, it will return the Indicator value that is associated with the most-recently-fetched row.

Most programs will not use this function, because the `SQL_ResColNull` and `SQL_ResColMore` function are easier to use.

If the Indicator value is negative one (-1), the column contains a Null value.

If the Indicator value is negative four (-4), the column is a Long column and the ODBC driver does not know how long it is.

(No other negative values are defined for the ODBC functions that SQL Tools supports.)

If the indicator value is zero or a positive number, the column contains that number of bytes of data.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate Indicator value of one. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

It is usually faster to use the SQL Tools `SQL_ResColNull` and/or `SQL_ResColMore` function than to use `SQL_ResColInd`.

See Also

Result Column Family

SQL_ResColIndPtr

Summary

Provides a pointer (ptr) to the memory buffer that SQL Tools uses for a result column's Indicator.

Twin

SQL_ResultColumnIndPtr

Family

Result Column Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ResColIndPtr(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function returns a `BAS_LONG` value that represents a memory pointer which points to the first byte of the Indicator buffer that SQL Tools is using for the specified column.

If you attempt to obtain a pointer to an Indicator buffer for a column that has not been autobound or direct-bound by SQL Tools, this function will return zero (0).

Remarks

This function, plus the knowledge that all Indicator buffers are four (4) bytes long, make it possible for your program to use Proxy Binding to access an Indicator directly instead of using a function like `SQL_ResColInd` or `SQL_ResColNull`. This can be an acceptable (and attractive) alternative to Manual Binding, especially if the Indicator can *usually* be accessed "normally" and only *sometimes* needs to be accessed directly.

Diagnostics

This function does not return Error Codes because it returns a memory pointer (which can have any value in the `BAS_LONG` range) so it would not be possible for a program to distinguish between an Error Code and a valid pointer. This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

See Result Column Binding for a discussion of your options, and how they affect the execution speed of your program.

See Also

Indicators

SQL_ResColInfoSInt

Summary

Provides information about a column of a result set, in numeric form.

Twin

SQL_ResultColumnInfoSInt

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColInfoSInt(lColumnNumber&, _  
                               lInfoType&)
```

Parameters

lColumnNumber&

The column of the result set about which you want information, between one (1) and the number that is returned by the `SQL_ResColCount` function.

lInfoType&

The type of information that you are requesting. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, this function will return the requested information. If an invalid parameter is used, this function will return zero (0).

Remarks

Only certain *lInfoType&* values will produce useful information in numeric form. For a list of *lInfoType&* values that produce information in *string* form, see `SQL_ResColInfoStr`.

For numeric information, *lInfoType&* must be one of the following values:

RESCOL_AUTO_UNIQUE_VALUE

If the column is auto-incrementing, this value will be one (1). Otherwise, it will be zero (0).

RESCOL_CASE_SENSITIVE

If the result column is a string column (like a `SQL_CHAR` column) which is treated as *case-sensitive* for collations and comparisons, this value will be one (1). Otherwise, it will be zero (0).

RESCOL_CONCISE_TYPE

The SQL Data Type of the result column, such as `SQL_INTEGER` or `SQL_CHAR`.

RESCOL_COUNT

The number of columns that the result set has.

RESCOL_DISPLAY_SIZE

The display size of the result column.

RESCOL_FIXED_PREC_SCALE

If the result column has a fixed precision and a non-zero scale that are datasource-specific, this value will be one (1). Otherwise, it will be zero (0).

RESCOL_LENGTH

ODBC 3.x+ ONLY: The maximum length of a fixed-length data type, or the actual length of a variable-length data type. (This value always excludes the null-termination byte at the end of an ASCII character string.)

RESCOL_NULLABLE

ODBC 3.x+ ONLY: One of the following values:

SQL_NULLABLE (The result column can contain Null values.)

SQL_NO_NULLS (The result column cannot contain Null values.)

SQL_NULLABLE_UNKNOWN (It is not known whether or not the result column can contain Null values.)

RESCOL_NUM_PREX_RADIX

The Num Prec Radix of the result column.

RESCOL_OCTET_LENGTH

ODBC 3.x+ ONLY: For fixed-length character or binary columns, this is the *actual* length of the column, in bytes. For variable-length character or binary columns, this is the *maximum* length of the column, in bytes. This value *includes* the null terminator that is used to mark the end of variable-length strings.

RESCOL_PRECISION

ODBC 3.x+ ONLY: This value indicates the precision of a numeric data type. For timestamp and interval data types which represent a time interval, this value is the precision of the fractional seconds.

RESCOL_SCALE

ODBC 3.x+ ONLY: This value indicates the scale of a numeric data type. For SQL_DECIMAL and SQL_NUMERIC data types, this is the defined scale. For all other data types, this value will be zero.

RESCOL_SEARCHABLE

This column will return one of the following values:

SQL_PRED_NONE (The column cannot be used in a *WHERE* clause.)

SQL_PRED_CHAR (The column can be used in a *WHERE* clause, but *only* with the *LIKE* predicate. SQL_LONGVARCHAR and SQL_LONGVARBINARY columns usually return SQL_PRED_CHAR.)

SQL_PRED_BASIC (The column can be used in a *WHERE* clause with all the comparison operators *except* *LIKE*.)

SQL_PRED_SEARCHABLE (The column can be used in a *WHERE* clause with any comparison operator.

RESCOL_TYPE

ODBC 3.x+ ONLY: The SQL Data Type of the result column.

When *IColumnNumber* is zero (0), the constant value SQL_BINARY is returned for variable-length bookmarks, and SQL_INTEGER is returned for fixed-length bookmarks.

For the datetime and interval data types, this field returns SQL_DATETIME or SQL_INTERVAL.

RESCOL_UNNAMED

ODBC 3.x+ ONLY: This value will be one (1) if the result column is named, or zero (0) if it is not named. See SQL_ResColInfoStr(RESCOL_NAME) for more information.

RESCOL_UNSIGNED

If the result column contains *signed* numeric values or *non-numeric* values (such as strings) this value will be zero (0). If the result column contains *unsigned* numeric values, this value will be one (1).

RESCOL_UPDATABLE

This value describes the "updatability" of the column in the result set, not the column in the table from which the result set was created. (The updatability of the column from which the result column was generated may be different from this value.) This function will return one of these values:

SQL_ATTR_READONLY
SQL_ATTR_WRITE
SQL_ATTR_READWRITE_UNKNOWN

Whether or not a result column is updatable can be based on the data type, user privileges, and the definition of the result set itself. If it is unclear whether or not a result column is updatable, SQL_ATTR_READWRITE_UNKNOWN will be returned.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value. This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
If SQL_ResColInfoSInt(12, RESCOL_AUTO_UNIQUE_VALUE) = 1 Then  
    Print "Column 12 is Auto-Incrementing"  
End If
```

Driver Issues

None.

Speed Issues

This information is *not* cached by SQL Tools. If your program needs to use one of these values repeatedly, you may be able to speed up your program by reading the value once and storing it in a variable, instead of using the `SQL_ResColInfoSInt` function over and over.

See Also

Result Column Family

SQL_ResColInfoStr

Summary

Provides information about a column of a result set, in string form.

Twin

SQL_ResultColumnInfoStr

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_ResColInfoStr(lColumnNumber&, _  
                             lInfoType&)
```

Parameters

lColumnNumber&

The column of the result set about which you want information, between one (1) and the number that is returned by the `SQL_ResColCount` function.

lInfoType&

The type of information that you are requesting. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, this function will return the requested information. If an invalid parameter is used, this function will return an empty string.

Remarks

Only certain *lInfoType&* values will produce useful information in string form. For a list of *lInfoType&* values that produce information in *numeric* form, see `SQL_ResColInfoInt`.

For string information, *lInfoType&* must be one of the following values:

RESCOL_BASE_COLUMN_NAME

ODBC 3.x+ ONLY: The "base name" for the result set column, i.e. the name of the column in the table from which the result set was created. If a base name doesn't exist (such as when a result column is generated by an expression), this value will be an empty string.

If this ODBC 3.x *lInfoType&* does not return a value, try `RESCOL_LABEL` (below).

RESCOL_BASE_TABLE_NAME

ODBC 3.x+ ONLY: The name of the *table* from which the column of the result set was generated.

RESCOL_CATALOG_NAME

The name of the catalog that contains the table from which the column of the result set was generated.

RESCOL_LABEL

The result column's label, which is usually used only for display purposes. For example, a column named `EmpName` might be labeled "Employee Name" or "This Employee's Name". If a result column does not have a label, the original column name is returned. If a column does not have a label or a name, an empty string is returned.

RESCOL_LITERAL_PREFIX and
RESCOL_LITERAL_SUFFIX

ODBC 3.x+ ONLY: The character(s) that the ODBC driver recognizes as a prefix/suffix for a literal value of this data type. This will be an empty string for data types that do not have a literal prefix/suffix.

RESCOL_LOCAL_TYPE_NAME

ODBC 3.x+ ONLY: A "local native language" name for the data type. If there is no localized name, an empty string is returned. This field is provided for display purposes only.

RESCOL_NAME

ODBC 3.x+ ONLY: An optional column alias. If no alias is specified, the column name is returned. In either case, `RESCOL_UNNAMED` (see `SQL_ResColInfoSInt`) is set to the numeric value `SQL_NAMED`.

If there is no column name or alias, an empty string is returned and `RESCOL_DESC_UNNAMED` is set to the numeric value `SQL_UNNAMED`.

RESCOL_SCHEMA_NAME

The name of the schema of the table that contains the column from which the result column was generated. If the column is an expression or if the column is part of a view, this value is defined differently by different ODBC drivers.

RESCOL_TABLE_NAME

The name of the table that contains the column from which the result column was generated. If the column is an expression or if the column is part of a view, this value is defined differently by different ODBC drivers.

RESCOL_TYPE_NAME

The datasource-dependent data type name, such as "INTEGER" or "COUNTER".

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the label of result column 1  
Print SQL_ResColInfoStr(RESCOL_LABEL)
```

Driver Issues

None.

Speed Issues

This information is *not* cached by SQL Tools. If your program needs to use one of these values repeatedly, you may be able to speed up your program by reading the value once and storing it in a variable, instead of using the `SQL_ResColInfoStr` function over and over.

See Also

Result Column Family

SQL_ResColLen

Summary

Provides the actual length of the data in a variable-length column such as a `SQL_VARCHAR`, `SQL_LONGVARCHAR`, `SQL_VARBINARY`, or `SQL_LONGVARBINARY` column. (Compare this to the `SQL_ResColSize` function, which returns the *maximum* length of data that *can* be returned by a column.)

Twin

`SQL_ResultColumnLen`

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColLen(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function returns the number of bytes of result column data that were retrieved for the specified column by the most recent `SQL_Fetch` or `SQL_FetchRel` operation.

If a fetch function has not yet been used to retrieve a row of a result set, or if the specified result column has not been bound, or if it has been unbound, or if it has been Manually Bound or Direct Bound, this function will return zero (0).

Remarks

For variable-length columns (`SQL_VARCHAR`, `SQL_VARBINARY`, etc.) this function will return the *actual* length of the data that was retrieved for a given column.

For fixed-length-string columns and numeric columns (`SQL_CHAR`, `SQL_INTEGER`, etc.) this function will return the size of the buffer that is required for the data type. For example, a `SQL_INTEGER` column will always return four (4), because `SQL_INTEGER` values require 4 bytes of memory.

This function is conceptually similar to the BASIC `LEN` function. Compare it to `SQL_ResColSize`, which is conceptually similar to the BASIC `SIZEOF` function.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "the string is one byte long". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the length of the value  
'that is currently in column 14.  
Print SQL_ResColLen(14)
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColMore

Summary

Indicates whether or not a Long result column which has already been accessed with SQL_LongResCol contains *more* data.

Twin

SQL_ResultColumnMore

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColMore(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the SQL_ResColCount function.

Return Values

This function returns Logical True (-1) if there is more data to be read from a Long column, and False (zero) if all of the data has been read.

Remarks

After the SQL_LongResCol function has been used to obtain some (or all) of the data from a Long result column, this function can be used to determine whether or not the SQL_LongResCol function must be used *again* to obtain a continuation of the data (another "chunk").

IMPORTANT NOTE: A return value of True means "the last SQL_LongResCol operation only retrieved part of the available data". If you attempt to use this function before the SQL_LongResCol function has been used to obtain *any* data from a Long Column, this function will return False.

See Long Result Columns for information about using this function.

Diagnostics

This function does not return Error Codes, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Long Columns.

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColNull

Summary

Indicates whether or not one column of one row of a result set contains a Null value (see).

Twin

SQL_ResultColumnNull

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColNull(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function returns Logical True (-1) if the specified column contains a Null value, or False (zero) if it does not.

Remarks

See Null Values for more information about this function.

Diagnostics

This function does not return Error Codes, but it can generate ODBC Error Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColNumber

Summary

Returns the result column number that corresponds to a column name.

Twin

SQL_ResultColumnNumber

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColNumber(sColumnName$)
```

Parameters

sColumnName\$

A string that contains a column name.

Return Values

If a result column with the name *sColumnName\$* is found, this function will return the corresponding result column number. If no match is found, negative one (-1) will be returned.

Remarks

This function is *not* case-sensitive. If a result column named "COLNAME" exists, it can be found by using "COLNAME", "colname", "ColName", etc.

Diagnostics

This function does not return Error Codes because an Error Code like SUCCESS_WITH_INFO (value 1) could be confused with a legitimate return value, such as "that string matches column number 1". This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
lResult& = SQL_ResColNumber("ZIPCODE")
```

Driver Issues

None.

Speed Issues

Whenever this function is used, SQL Tools scans the names of a statement's result columns until it finds a match. If your program uses this function repeatedly for a certain column, it would be faster to use this function once and store the column number in a variable, and then repeatedly use the *variable* instead of this function.

See Also

SQL_ResColumnInfoStr

Result Column Family

SQL_ResColSInt

Summary

Provides the value of one column of one row of a result set in the form of a Signed Integer (SInt).

Twin

SQL_ResultColumnSInt

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColSInt(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the SQL_ResColCount function.

Return Values

This function returns a signed numeric value that corresponds to the data in a column of a result set.

Remarks

This function is usually used to obtain the value of a *Signed* SQL_INTEGER, SQL_SMALLINT, or SQL_TINYINT column of a result set.

IMPORTANT NOTE: Your choice of the variable type that is used to store the return value of this function is usually important. A BAS_LONG variable (as shown in the **Syntax** above) can always be used safely, because this function can only return values in the BAS_LONG range. But if a result column contains a SQL_SMALLINT, SQL_TINYINT, or other numeric value, it may be safe (and desirable) to use a BAS_WORD, BAS_INTEGER, BAS_BYTE, or other type of variable. You should probably consult your BASIC language's "Data Type" documentation before you decide which type of variable is appropriate for a given column. When in doubt, use a BAS_LONG variable. It is always safe, and BASIC generates extremely efficient executable code when BAS_LONG variables are used.

IMPORTANT NOTE: Your choice of SQL_ResColSInt (Signed Integer) and SQL_ResColUInt (Unsigned Integer) is usually very important. Numeric values are stored in databases in a binary format which does not have an explicit "signed or unsigned" property, so *you* are responsible for telling SQL Tools how to interpret a given column. For example, an unsigned SQL_TINYINT column can contain a value between zero (0) and 255, and a signed SQL_TINYINT column can contain a value between -128 and +127. If a result column contains an *unsigned* SQL_TINYINT and you use SQL_ResColSInt (Signed Integer) to access it, values between 0 and

127 will be returned correctly but the values between 128 and 255 will be returned as the "two's complement" negative numbers between -128 and -1. (In some cases this automatic transformation can be very useful, but if it is unintentional it can be disastrous.)

If a result column contains an unsigned value like a `BAS_DWORD`, `BAS_WORD`, or `BAS_BYTE` value, you should *probably* use `SQL_ResColUInt` instead of `SQL_ResColSInt`.

When it is used to obtain the value of a `SQL_BIT` column, `SQL_ResColSInt` returns Logical True and False values. For most programs, this is the preferred method of accessing `SQL_BIT` columns, which are usually used to store True/False values.

`SQL_ResColSInt` can also be used to obtain the integer portion of floating-point column data (`SQL_REAL` and `SQL_DOUBLE`).

`SQL_ResColSInt` can also be used to obtain the integer value of a string from a `SQL_CHAR`, `SQL_VARCHAR`, `SQL_DECIMAL`, or `SQL_NUMERIC` column,

If you attempt to use `SQL_ResColSInt` to obtain values for other types of columns, the return value will be zero.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value, such as "this column contains the value 1". This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the contents of result column 1  
'as a signed integer.  
Print SQL_ResColSInt(1)
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColSize

Summary

Provides the size of the buffer that is used for one column of a result set, i.e. the maximum length of the data that a column can return. (Compare this function to `SQL_ResColLen`, which returns the actual length of the data that was retrieved by the most recent `SQL_Fetch` or `SQL_FetchRel` operation.)

Twin

`SQL_ResultColumnSize`

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColSize(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function returns the size of the buffer that is used for the specified column of a result set.

If a column has not been bound, or if it has been unbound, or if it has been Manually Bound or Direct Bound, this function will return zero (0). Otherwise, it will always return a minimum value of one (1).

Remarks

Compare this function to the `SQL_ResColLen` function for more information.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value, such as "this column uses a 1-byte buffer". This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues: None.

Speed Issues: None.

See Also: `SQL_ResColBufferPtr`
Result Column Family

SQL_ResColStr

Summary

Provides the value of one column of one row of a result set in the form of a string (Str).

Twin

SQL_ResultColumnStr

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_ResColStr(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function returns a string that contains the contents of a result column.

If a column has not been bound, or if it has been unbound, or if it has been Manually Bound or Direct Bound, this function will return an empty string.

See **Remarks** below for more information.

Remarks

When used to access a `SQL_CHAR`, `SQL_VARCHAR`, `SQL_LONGVARCHAR`, `SQL_BINARY`, `SQL_VARBINARY`, `SQL_LONGVARBINARY`, `SQL_DECIMAL`, or `SQL_NUMERIC` result column, or a Unicode column, this function returns a string. The length of the string is determined by the contents of the result column.

When it is used to access *any* other type of column, this function returns a string that contains the binary representation of the data in the column.

For example, if a result column contains a `SQL_INTEGER` numeric value, the string that is returned by `SQL_ResColStr` would be the 4-byte binary representation of the integer value. It would be compatible with the PowerBASIC (signed) `MKL$` and `CVL` functions, and the (unsigned) `MKDWD$` and `CVDWD` functions. (See the PowerBASIC documentation for information about those functions.)

Another example: If the `SQL_ResColStr` function is used to access a column that contains a User Defined Type, it will return a string that can be used to assign a value to a UDT. See `SQL_TIMESTAMP` for an example of this technique.

Here is a more specific UDT example: If the `SQL_ResColStr` function is used to access a `SQL_TIMESTAMP` column, it will return a string that is compatible with the `TIMESTAMP_STRUCT` structure that is defined in the SQL Tools Declaration Files. So you could `Dim` a UDT as a `TIMESTAMP_STRUCT` and then use `LSet` to assign the column's string contents to the UDT. (See `SQL_TIMESTAMP` for an actual code example.)

Diagnostics

This function does not return Error Codes because it returns string values, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the contents of column 3  
'in string form:  
Print SQL_ResColStr(3)
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColText

Summary

Provides the value of one column (or all columns) of one row of a result set in "text" form.

Twin

SQL_ResultColumnText

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColText(lColumnNumber&)
```

Parameters

lColumnNumber&

Either **1**) The number of a result column, between one (1) and the number that is returned by the `SQL_ResColCount` function, or **2**) the constant `ALL_COLUMNS`, which causes the values of all of the columns in a result set to be returned as a single (usually delimited) string.

Return Values

This function returns a string that represents the contents of one column (or all columns) in human-readable form.

Remarks

The `SQL_ResColText` function is a "universal" result column function which can return the result of *any* result column in human-readable form. All column values are translated into "printable" strings, and truncated to a certain maximum length, for *generic* display purposes.

PLEASE NOTE: `SQL_ResColText` is intended to be a "general purpose" function, but it is not an "all purpose" function! You should use the other `SQL_ResCol` functions such as `SQL_ResColSInt` and `SQL_ResColStr` when it is appropriate to do so.

The default maximum length for strings returned by this function is 32 characters. That length can be changed with the `SQL_SetOptionSInt(OPT_TEXT_MAXLENGTH)` function. When a `SQL_ResColText` result value is longer than `OPT_TEXT_MAXLENGTH`, it is automatically truncated by the `SQL_LimitTextLength` function, so it will have an ellipsis (. . .) appended to the end of the string.

If a column contains a Null value, this function will return the string `[NULL]` instead of returning an empty string or a string containing "0". The default string `[NULL]` can be changed to any string that you choose (including an empty string) by using the

SQL_SetOptionStr(OPT_TEXT_NULL) function.

If an unbound column is accessed with this function, it will return the string [UNBOUND]. This string can be changed to any string that you choose (including an empty string) by using the SQL_SetOptionStr(OPT_TEXT_UNBOUND) function.

Assuming that a column is bound and does not contain a Null value...

If this function is used to access a SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR, SQL_DECIMAL, SQL_NUMERIC, or SQL_BIGINT column, a normal (but possibly truncated) string is returned.

If this function is used to access a SQL_WCHAR, SQL_WVARCHAR, or SQL_WLONGVARCHAR column, (see Unicode Data Types) the resulting string will contain CHR\$(0) markers. For example, the string HELLO would look like this:

```
H [CHR$(0)] E [CHR$(0)] L [CHR$(0)] L [CHR$(0)] O [CHR$(0)]
```

If the SQL_ResColText function is used to access a SQL_INTEGER or SQL_SMALLINT numeric column, it assumes that the value is a *Signed* integer (BAS_LONG or BAS_INTEGER, respectively) and creates a string representation of the number, similar to the results of the BASIC FORMAT\$ function.

If this function is used to access a SQL_TINYINT numeric column, it assumes that the value is an *Unsigned* integer (BAS_BYTE) and creates a string representation of the number.

If this function is used to access a SQL_REAL or SQL_DOUBLE floating-point numeric column, it creates a string representation of the number. (No assumptions are necessary.)

If this function is used to access a SQL_BIT column, it returns the string True or the string False. These strings can be change (to Yes and No, for example) by using the SQL_SetOptionStr(OPT_TEXT_TRUE) and (OPT_TEXT_FALSE) functions.

If this function is used to access a SQL_TIMESTAMP column, it automatically uses the SQL_ResColDateTime function to create a displayable string using the system default date and time formats, or the format(s) that your program specifies with the SQL_SetOptionStr OPT_DATE_FORMAT and/or OPT_TIME_FORMAT settings.

If this function is used to access a SQL_DATE column, it returns a string using the system default date format, or the format that your program specifies with the SQL_SetOptionStr OPT_DATE_FORMAT setting. For more information about the use of formatting strings, see the SQL_ResColDate function.

If this function is used to access a SQL_TIME column, it returns a string using the system default time format, or the format that your program specifies with the SQL_SetOptionStr OPT_TIME_FORMAT setting. For more information about the use of formatting strings, see the SQL_ResColTime function.

If this function is used to access a SQL_BINARY, SQL_VARBINARY, or SQL_LONGVARBINARY column, it uses the SQL_TexStrOLE function to convert nonprintable characters to the [CHR\$(x)] notation. Printable characters are not

changed. You can control which characters are considered printable by using the `SQL_SetOptionSInt(OPT_MIN_TEXTCHAR)` and `(OPT_MAX_TEXTCHAR)` functions.

If this function is used to access any other type of column, it is treated as a `SQL_VARBINARY` column and processed with the `SQL_TextStrOLE` function.

The ALL_COLS Option

Unlike all of the other `SQL_ResCol` functions, `SQL_ResColText` can accept the *IColumnNumber*& value `ALL_COLS` (for All Columns). If this value is used, the return value of the function will be a comma-and-quote delimited string that contains *all* of the columns in the result set.

You can change the delimiter by using the `SQL_SetOptionStr(OPT_TEXT_DELIMITER)` function.

Please note that the `OPT_TEXT_DELIMITER` strings **1)** "quote-*something*-quote" and **2)** "pipe-space-pipe", are special cases. (The default value is quote-comma-quote.) If you use any other values for `OPT_TEXT_DELIMITER`, the columns of the result set will be separated by the string that you specify, and no other processing will be performed.

If quote-*something*-quote is used, the `SQL_ResColText` return value will be a standard "quote delimited" string, with the *something* character(s) in between the quoted string. For example, the default value of quote-*comma*-quote produces quoted string which are separated by commas. In addition to putting the delimiter *between* the columns, **1)** the `SQL_ResColText` function will automatically add leading and trailing quotes to the final string, and **2)** any quote characters that are contained *in* a column value will be replaced with back-apostrophes, to avoid confusion with the quotes that are used for the column delimiters. (The back-apostrophe replacement character can be changed with the `SQL_SetOptionStr(OPT_TEXT_QUOTE)` function.)

For example, if a three-column result row from an `ADDRESSBOOK` database was used, it would look something like this...

```
"Public","John","123 Main Street"
```

If you changed the `OPT_TEXT_DELIMITER` string to quote-CarriageReturn-LineFeed-quote (i.e. `CHR$(34,13,10,34)`), and saved the resulting string to a text file, it would look like this:

```
"Public"
"John"
"123 Main Street"
```

If pipe-space-pipe (" | ") is used, the `SQL_ResColText` return value will be formatted so that every column has the width specified by the `SQL_SetOptionSInt(OPT_TEXT_MAXLENGTH)` function. The default value for `OPT_TEXT_MAXLENGTH` is 32, so a string with 32-character fixed-width columns would normally be created, with a pipe symbol at the beginning and end.

For example, if the `OPT_TEXT_MAXLENGTH` value was set to 12 and a three-column

result set from an ADDRESSBOOK database was used, it would look something like this...

Public	John	123 Main Str
--------	------	--------------

The real advantage of this type of formatting becomes clear when you display several rows at a time...

Public	John	123 Main Str
Doe	Jane	456 First Bl
Smith	Bob	789 Second A
Jones	Mary	321 Deebee R

You probably noticed that the last column's values are truncated. If you want the "pipe-space-pipe" columns to have *different* widths to accommodate different types of column data, we suggest that you use a relatively large value for OPT_TEXT_MAXLENGTH, and then use BASIC string functions to shorten certain columns. This technique can be used to provide a very easy-to-read string format with very little effort.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display all of the columns of  
'the current result set...  
Print SQL_ResColText(ALL_COLUMNS)
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColTime

Summary

Provides the value of one column of one row of a result set as a string that represents a formatted Time value.

Twin

SQL_ResultColumnTime

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColTime(lColumnNumber& . _  
                           sFormat$)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the SQL_ResColCount function. The column must be a SQL_TIMESTAMP, SQL_TYPE_TIMESTAMP, SQL_TIME, or SQL_TYPE_TIME column.

sFormat\$

PLEASE NOTE: The *sFormat\$* strings that are described here are case-sensitive. The *sFormat\$* parameter can be **1)** An empty string, for the system default time format or **2)** a time-formatting string that follows the guidelines that are described in **Remarks** below, or **3)** the string "default" for the format that was specified with the SQL_SetOptionStr OPT_TIME_FORMAT setting. If you use "default" and you have not set the OPT_TIME_FORMAT setting, the system default time format will be used.

Return Values

This function returns a string that represents a Time value from a Time or Date-Time column of a result set. If the *lColumnNumber&* parameter is not the number of a Time or Date-Time column, an empty string will be returned.

Remarks

The *sFormat\$* parameter is typically a string that looks something like this:

```
hh:mm tt
```

If a result set contained the time value 14:10:00 that *sFormat\$* string would produce a time string that looked like this:

```
02:10 pm
```

Each character or group of characters in the formatting string represents either **1)** a different part of a time (hour, minute, etc.) or **2)** a literal character such as the colon (:).

You may use the following time-part codes in the formatting string:

`h` Hours as single digits for hours 1-9 and two digits for hours 10-12, using a 12-hour clock.

`hh` Hours as two-digit numbers for hours 01-12, using a 12-hour clock.

`H` Hours as single digits for hours 0-9 and two digits for hours 10-23, using a 24-hour clock.

`HH` Hours as two-digit numbers for hours 00-23, using a 24-hour clock.

`m` Minutes as single digits for minutes 0-9 and two digits for minutes 10-59.

`mm` Minutes as two-digit numbers for minutes 00-59.

`s` Seconds as single digits for seconds 0-9 and two digits for seconds 10-59.

`ss` Seconds as two digit numbers for seconds 00-59

`f` through `fffffffffff` Fractional Seconds. One `f` produces tenths, `ff` produces hundreds, `fff` produces thousandths (milliseconds), and so on through `fffffffffff` which produces billionths of a second (nanoseconds). Using more than nine (9) `f`'s will result in values with trailing zeros. Using more than 16 `f`'s will produce unpredictable results. Note that some ODBC Drivers -- most notably certain versions of Microsoft SQL Server -- do not support high-resolution fractional seconds, so even if you specify microseconds (for example) there is no guarantee that the actual data will be that accurate.

`t` Time Marker as a single character, such as `A` or `P`.

`tt` Time Marker as multiple characters, such as `AM` or `PM`.

Spaces are always treated as literal characters, i.e. they will appear in the final string exactly as you type them. Most other characters (punctuation, etc.) are also interpreted as literal characters, but to be certain you should enclose literal characters in single-quotes, such as this example where the at-sign (`@`) is quoted.

`'@' h:mm tt` ...would produce...

`@ 4:00 pm`

If you use a single-quote character in a formatting string, it will be interpreted as a delimiter even if it is not "paired" properly, i.e. even if a matching "close-quote" is not used.

In order to include a *literal* single-quote character in a time-formatting string, you must use *four consecutive* single quotes, like this...

`h O''''clock` ...would produce...

`4 O'clock`

Using four single-quotes produces an output string that actually *includes* one single-

quote character.

But be careful! If the string "clock" had contained any of the letters `h`, `H`, `m`, `s`, or `f`, they would have been replaced by the corresponding time-part. So the letter `O` and the word "clock" really should be surrounded by single-quotes too, like this:

```
h 'O''''''clock' ...would reliably produce...
```

```
4 O'clock
```

The first and second single-quotes delimit the letter `O`. The next four single-quotes are interpreted as one *literal* quote. And the seventh and eighth single-quotes delimit the word "clock". (This is a complicated system, but that's the way all Windows Date/Time formatting works.)

Advanced Techniques

It is possible to affect the system-default Time formatting that is performed by the `SQL_ResColTime` function when an empty formatting string (" ") is used. (These options will have no effect if a formatting string is used.) Use the `SQL_SetOptionSInt` function to change these options:

`OPT_TIME_LOCALE`

Use `LOCALE_SYSTEM_DEFAULT`, `LOCALE_USER_DEFAULT`, or any other Standard Win32 `LOCALE_` constant. The default value is `LOCALE_USER_DEFAULT`.

`OPT_TIME_FLAGS`

Use the standard Win32 values `LOCALE_NOUSEROVERRIDE`, `TIME_NOMINUTESORSECONDS`, `TIME_NOSECONDS`, `TIME_NOTIMEMARKER`, and/or `TIME_FORCE24HOURFORMAT`. The default value is zero (0), i.e. no special flags are used.

The names of these standard Windows constants are reasonably self-explanatory. Please refer to the Win32 API documentation for complete information.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Get a string from column 8

sResult$ = SQL_ResColTime(8,"hh:mm tt")
```

Driver Issues

None.

Speed Issues

None.

See Also: Result Column Family

SQL_ResColType

Summary

Provides the SQL Data Type of one column of a result set.

Twin

SQL_ResultColumnType

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResColType(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function will return zero (0) if a column has not been autobound by SQL Tools. Otherwise it will return the SQL Data Type of the column (`SQL_INTEGER`, `SQL_CHAR`, etc.).

Remarks

For a complete list of the possible return values for this function, see SQL Data Types.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate returns value like "this column's data type is `SQL_CHAR` (value 1)". This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the data type of column 8  
Print SQL_ResColType(8)
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResColUInt

Summary

Provides the value of one column of one row of a result set in the form of an Unsigned Integer (UInt).

Twin

SQL_ResultColumnUInt

Family

Result Column Family

Availability

Standard and Pro

Warning

This function should not usually be used with programming languages that do not support 32-bit Unsigned Integers, such as Visual Basic. See Remarks below.

Syntax

```
dwResult??? = SQL_ResColUInt(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number that is returned by the `SQL_ResColCount` function.

Return Values

This function returns an unsigned numeric value that corresponds to the data in one column of a result set.

Remarks

This function is usually used to obtain the value of an *Unsigned* `SQL_INTEGER`, `SQL_SMALLINT`, or `SQL_TINYINT` column of a result set.

IMPORTANT NOTE: Your choice of the variable type that is used to store the return value of this function is usually important. A `BAS_DWORD` variable (as shown in the Syntax above) can always be used safely because this function can only return values in the `BAS_DWORD` range. But if a result column contains a `SQL_SMALLINT`, `SQL_TINYINT`, or other numeric value, it may be safe (and desirable) to use a `BAS_LONG`, `BAS_WORD`, `BAS_INTEGER`, `BAS_BYTE`, or other type of variable. You should probably consult your BASIC language's "Data Type" documentation before you decide which type of variable is appropriate for a given column. If at all possible - if the column's value will never exceed the maximum positive value of a `BAS_LONG` variable -- try to use a `BAS_LONG` variable. BASIC generates extremely efficient executable code when `BAS_LONG` variables are used. But keep in mind that it is possible for `SQL_ResColUInt` to return large positive values that cannot be stored in a `BAS_LONG` variable. `BAS_DWORD` is always a safe choice, even if it is less efficient than `BAS_LONG`, but not all programming languages support `BAS_DWORD` variables.

IMPORTANT NOTE: Your choice of `SQL_ResColSInt` (Signed Integer) and

`SQL_ResColUInt` (Unsigned Integer) is usually *very* important. Numeric values are stored in databases in a binary format which does not have an explicit "signed or unsigned" property, so *you* are responsible for telling SQL Tools how to interpret a given column. For example, an unsigned `SQL_TINYINT` column can contain a value between zero (0) and 255, and a signed `SQL_TINYINT` column can contain a value between -128 and +127. If a result column contains an *unsigned* `SQL_TINYINT` and you use `SQL_ResColSInt` (Signed Integer) to access it, values between 0 and 127 will be returned correctly but the values between 128 and 255 will be returned as the "two's complement" negative numbers between -128 and -1. (In some cases this automatic transformation can be very useful, and if it is unintentional it can be disastrous.)

If a result column contains a signed value like a `BAS_LONG` or `BAS_INTEGER` value, you should *probably* use `SQL_ResColSInt` instead of `SQL_ResColUInt`.

When it is used to obtain the value of a `SQL_BIT` column, `SQL_ResColUInt` returns SQL True (+1) and False (0) values, instead of Logical True (-1) and False (0). For most programs, using `SQL_ResColSInt` (rather than this function) is the preferred method of accessing `SQL_BIT` columns, which are usually used to store True/False values.

`SQL_ResColUInt` can also be used to obtain the integer portion of values in floating-point columns (`SQL_REAL` and `SQL_DOUBLE`), if the floating point values are positive.

`SQL_ResColUInt` can also be used to obtain the positive integer value of a string from a `SQL_CHAR`, `SQL_VARCHAR`, `SQL_DECIMAL`, or `SQL_NUMERIC` column.

If you attempt to use `SQL_ResColUInt` to obtain values for other types of columns, the return value will be zero.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value, such as "this column contains the value 1". This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the contents of result column 1
'as an unsigned integer.
Print SQL_ResColUInt(1)
```

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResetStatementMode

Summary

Resets a statement's mode settings to the SQL Tools default settings.

Twin

SQL_ResetStmtMode

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

This function cannot be used to change the mode of an active (i.e. open) statement. It only affects statements that are prepared and executed after this function is used.

Syntax

```
SQL_ResetStatementMode lDatabaseNumber&, _  
                        lStatementNumber&
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

This function always returns `SUCCESS`, so it is possible to ignore the return value of this function.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_ResetStatementMode` is identical to `SQL_ResetStmtMode`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_ResetStatementMode` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

This function does not return Error Codes, but it can return SQL Tools Error Messages.

Example

```
'Reset the mode of database 1,  
'statement 2...  
SQL_ResetStatementMode 1,2
```

Driver Issues: None.

Speed Issues: None.

See Also: SQL Statement Mode, `SQL_ResetStmtMode`

SQL_ResetStmtMode

Summary

Resets the current statement's mode settings to the SQL Tools default settings.

Twin

SQL_ResetStatementMode

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

This function cannot be used to change the mode of an active (i.e. open) statement. It only affects statements that are prepared and executed after this function is used.

Syntax

SQL_ResetStmtMode

Parameters

None.

Return Values

This function always returns `SUCCESS`, so it is possible to ignore the return value of this function.

Remarks

For a general discussion, see [SQL Statement Mode](#).

This function is used to reset all of the various Statement Mode settings to their SQL Tools default values. This function does not affect a currently-open statement. The default settings will be used the next time that a SQL statement is opened with `SQL_OpenStmt` or is prepared or executed with `SQL_Stmt`.

You can change the SQL Tools default statement mode settings by using the `SQL_SetOptionSInt` function with one of the constant values in the SQL Tools Declaration Files between value 70 (`OPT_CURSOR_SENSITIVITY`) and value 84 (`OPT_USE_BOOKMARKS`). For a complete list, see `SQL_SetOptionSInt`.

Diagnostics

This function does not return Error Codes, ODBC Error Messages, or SQL Tools Error Messages.

Example

SQL_ResetStmtMode

Driver Issues: None.

Speed Issues: None.

See Also: [SQL Statement Mode](#)

SQL_ResRowCount

Summary

Provides the number of rows that were affected by a SQL statement.

Twin

SQL_ResultRowCount

Family

Result Count Family

Availability

Standard and Pro

Warning

This function should *not* be used to obtain the number of rows that are contained in a result set that was generated by a SQL [SELECT](#) statement. It should only be used for non-[SELECT](#) statements. See [Why You CAN'T Use SQL_ResRowCount for SELECT Statements](#) for more information.

Syntax

```
lResult& = SQL_ResRowCount
```

Parameters

None.

Return Values

This function will return zero (0) if a SQL statement has not yet been executed, or if it has been executed and did not affect any rows. Otherwise, this function will return the number of rows that were affected by the statement.

Remarks

See [Results from non-SELECT Statements](#) and [Why You CAN'T Use SQL_ResRowCount for SELECT Statements](#) for a discussion of this function.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value, like "one row was affected by the SQL statement". This function can, however, generate ODBC Error Messages and SQL Tools Error Codes.

Example

```
'Display the number of rows affected by  
'the most-recently-executed statement:  
Print SQL_ResRowCount
```

Driver Issues

Many ODBC drivers do not return a value for `SQL_ResRowCount` for [SELECT](#) statements, and those that do are not always accurate. See [Why You CAN'T Use SQL_ResRowCount for SELECT Statements](#) for more information.

Speed Issues: None.

See Also: [Detecting "No Data At All"](#)

SQL_ResultColumnBInt

Summary

Provides the value of one column of one row of a result set, in Big Integer (BInt) form. (Big Integers are returned as strings; see **Remarks** below.)

Twin

SQL_ResColBInt

Family

Result Column Family

Availability

Standard and Pro.

Warning

None.

Syntax

```
sResult$ = SQL_ResultColumnBInt(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber*&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColBInt for complete details.

Return Values

See twin function SQL_ResColBInt for complete details.

Remarks

Except for the *lDatabaseNumber*& and *lStatementNumber*& parameters, SQL_ResultColumnBInt is identical to SQL_ResColBInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnBInt is shown below.

For more information about using *lDatabaseNumber*& and *lStatementNumber*& in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColBInt for complete details.

Example

See twin function SQL_ResColBInt.

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Family

SQL_ResultColumnBufferPtr

Summary

Provides a pointer (ptr) to the memory buffer that SQL Tools uses for a result column.

Twin

SQL_ResColBufferPtr

Family

Result Column Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnBufferPtr(lDatabaseNumber&, _  
                                     lStatementNumber&, _  
                                     lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColBufferPtr for complete details.

Return Values

See twin function SQL_ResColBufferPtr for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnBufferPtr is identical to SQL_ResColBufferPtr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnBufferPtr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColBufferPtr for complete details.

Example

See twin function SQL_ResColBufferPtr.

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Binding Family

SQL_ResultColumnCount

Summary

Provides a value which indicates the number of columns in a result set.

Twin

SQL_ResColCount

Family

Result Count Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnCount(lDatabaseNumber&, _  
                                lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

See twin function SQL_ResColCount for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnCount is identical to SQL_ResColCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColCount for complete details.

Example

See twin function SQL_ResColCount.

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResultColumnNameDate

Summary

Provides the value of one column of one row of a result set as a string that represents a formatted Date value.

Twin

SQL_ResColDate

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnNameDate(lDatabaseNumber&, _  
                                     lStatementNumber&, _  
                                     lColumnName&. _  
                                     sFormat$)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColDate for complete details.

Return Values

See twin function SQL_ResColDate for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnNameDate is identical to SQL_ResColDate. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnNameDate is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColDate for complete details.

Example

See twin function SQL_ResColDate

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Family

SQL_ResultColumnDateTime

Summary

Provides the value of one column of one row of a result set as a string that represents a formatted Date-Time value.

Twin

SQL_ResColDateTime

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnDateTime(lDatabaseNumber&, _  
                                     lStatementNumber&, _  
                                     lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColDateTime for complete details.

Return Values

See twin function SQL_ResColDateTime for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnTime is identical to SQL_ResColDateTime. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnTime is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColDateTime for complete details.

Example

See twin function SQL_ResColDateTime

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Family

SQL_ResultColumnDateTimePart

Summary

Provides the value of one column of one row of a result set as a "part" of a Date or Time, such as the Month or Hour, in numeric form.

Twin

SQL_ResColDateTimePart

Family

Result Column Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnDateTimePart(lDatabaseNumber&, _  
                                         lStatementNumber&, _  
                                         lColumnNumber&, _  
                                         lPart&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColDateTimePart for complete details.

Return Values

See twin function SQL_ResColDateTimePart for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnDateTimePart is identical to SQL_ResColDateTimePart . To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnDateTimePart is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColDateTimePart for complete details.

Example

See twin function SQL_ResColDateTimePart.

Driver Issues

None.

Speed Issues

None.

See Also

SQL_TIMESTAMP, SQL_DATE, SQL_TIME

SQL_ResultColumnFloat

Summary

Provides the value of one column of one row of a result set, in Floating Point form.

Twin

SQL_ResColFloat

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
dpResult# = SQL_ResultColumnFloat(lDatabaseNumber&, _  
                                   lStatementNumber&, _  
                                   lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColFloat for complete details.

Return Values

See twin function SQL_ResColFloat for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnFloat is identical to SQL_ResColFloat. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnFloat is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColFloat for complete details.

Example

See twin function SQL_ResColFloat.

Driver Issues

None.

Speed Issues

None.

See Also: Result Column Family

SQL_ResultColumnInd

Summary

Provides the value of the Indicator that is associated with one column of one row of a result set.

Twin

SQL_ResColInd

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnInd(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColInd for complete details.

Return Values

See twin function SQL_ResColInd for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnInd is identical to SQL_ResColInd. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnInd is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColInd for complete details.

Example

See twin function SQL_ResColInd.

Driver Issues: None.

Speed Issues: None.

See Also

Result Column Family

SQL_ResultColumnIndPtr

Summary

Provides a pointer (ptr) to the memory buffer that SQL Tools uses for a result column's Indicator.

Twin

SQL_ResColIndPtr

Family

Result Column Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnIndPtr(lDatabaseNumber&, _  
                                   lStatementNumber&, _  
                                   lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColIndPtr for complete details.

Return Values

See twin function SQL_ResColIndPtr for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnIndPtr is identical to SQL_ResColIndPtr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnIndPtr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColIndPtr for complete details.

Example

See twin function SQL_ResColIndPtr.

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Binding Family

SQL_ResultColumnInfoSInt

Summary

Provides information about one column of a result set, in numeric form.

Twin

SQL_ResColInfoSInt

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnInfoSInt(lDatabaseNumber&, _  
                                     lStatementNumber&, _  
                                     lColumnNumber&, _  
                                     lInfoType&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColInfoSInt for complete details.

Return Values

See twin function SQL_ResColInfoSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnInfoSInt is identical to SQL_ResColInfoSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnInfoSInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColInfoSInt for complete details.

Example

See twin function SQL_ResColInfoSInt.

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Family

SQL_ResultColumnInfoStr

Summary

Provides information about one column of a result set, in string form.

Twin

SQL_ResColInfoStr

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnInfoStr(lDatabaseNumber&, _  
                                   lStatementNumber&, _  
                                   lColumnNumber&, _  
                                   lInfoType&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColInfoStr for complete details.

Return Values

See twin function SQL_ResColInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnInfoStr is identical to SQL_ResColInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColInfoStr for complete details.

Example

See twin function SQL_ResColInfoStr.

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Family

SQL_ResultColumnLen

Summary

Provides the actual length of the data in a variable-length result column such as a SQL_VARCHAR, SQL_LONGVARCHAR, SQL_WVARCHAR, SQL_WLONGVARCHAR, SQL_VARBINARY, or SQL_LONGVARBINARY column. (Compare this to the SQL_ResColSize function, which returns the *maximum* length of data than *can* be returned by a column.)

Twin

SQL_ResColLen

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnLen(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColLen for complete details.

Return Values

See twin function SQL_ResColLen for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnLen is identical to SQL_ResColLen. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnLen is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColLen for complete details.

Example

See twin function SQL_ResColLen.

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResultColumnMore

Summary

Indicates whether or not a Long result column that has already been accessed with SQL_LongResCol contains *more* data.

Twin

SQL_ResColMore

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnMore(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColMore for complete details.

Return Values

See twin function SQL_ResColMore for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnMore is identical to SQL_ResColMore. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnMore is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColMore for complete details.

Example

See twin function SQL_ResColMore.

Driver Issues: None.

Speed Issues: None.

See Also

Result Column Family

SQL_ResultColumnNull

Summary

Indicates whether or not one column of one row of a result set contains a Null value.

Twin

SQL_ResColNull

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnNull(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColNull for complete details.

Return Values

See twin function SQL_ResColNull for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnNull is identical to SQL_ResColNull. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnNull is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColNull for complete details.

Example

See twin function SQL_ResColNull.

Driver Issues: None.

Speed Issues: None.

See Also

Result Column Family

SQL_ResultColumnNumber

Summary

Returns the result column number that corresponds to a column name.

Twin

SQL_ResColNumber

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnNumber(lDatabaseNumber&, _  
                                   lStatementNumber&, _  
                                   sColumnName$)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColNumber for complete details.

Return Values

See twin function SQL_ResColNumber for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnNumber is identical to SQL_ResColNumber. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnNumber is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColNumber for complete details.

Example

See twin function SQL_ResColNumber.

Driver Issues

None.

Speed Issues

See twin function SQL_ResColNumber for complete details.

See Also: SQL_ResColInfoStr

SQL_ResultColumnSInt

Summary

Provides the value of one column of one row of a result set in the form of a Signed Integer (SInt).

Twin

SQL_ResColSInt

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnSInt(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColSInt for complete details.

Return Values

See twin function SQL_ResColSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnSInt is identical to SQL_ResColSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnSInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColSInt for complete details.

Example

See twin function SQL_ResColSInt.

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Family

SQL_ResultColumnSize

Summary

Provides the size of the memory buffer that is used for one column of a result set, i.e. the maximum length of the data that a column can return. (Compare this function to `SQL_ResColLen`, which returns the actual length of the data that was retrieved by the most recent `SQL_Fetch` or `SQL_FetchRel` operation.)

Twin

`SQL_ResColSize`

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnSize(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function `SQL_ResColSize` for complete details.

Return Values

See twin function `SQL_ResColSize` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_ResultColumnSize` is identical to `SQL_ResColSize`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_ResultColumnSize` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_ResColSize` for complete details.

Example

See twin function `SQL_ResColSize`.

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResultColumnStr

Summary

Provides the value of one column of one row of a result set in the form of a string (Str).

Twin

SQL_ResColStr

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnStr(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColStr for complete details.

Return Values

See twin function SQL_ResColStr for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnStr is identical to SQL_ResColStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColStr for complete details.

Example

See twin function SQL_ResColStr.

Driver Issues

None.

Speed Issues

None.

See Also: Result Column Family

SQL_ResultColumnText

Summary

Provides the value of one column (or all columns) of one row of a result set in "text" form.

Twin

SQL_ResColText

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnText(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColText for complete details.

Return Values

See twin function SQL_ResColText for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnText is identical to SQL_ResColText. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnText is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColText for complete details.

Example

See twin function SQL_ResColText.

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Family

SQL_ResultColumnTime

Summary

Provides the value of one column of one row of a result set as a string that represents a formatted Time value.

Twin

SQL_ResColTime

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnTime(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&. _  
                                sFormat$)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin SQL_ResColTime function for complete details.

Return Values

See twin function SQL_ResColTime for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnTime is identical to SQL_ResColTime. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnTime is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColTime for complete details.

Example

See twin function SQL_ResColTime

Driver Issues

None.

Speed Issues

None.

See Also

Result Column Family

SQL_ResultColumnType

Summary

Provides the SQL Data Type of one column of a result set.

Twin

SQL_ResColType

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnType(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColType for complete details.

Return Values

See twin function SQL_ResColType for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnType is identical to SQL_ResColType. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnType is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColType for complete details.

Example

See twin function SQL_ResColType.

Driver Issues

None.

Speed Issues

None.

See Also: Result Column Family

SQL_ResultColumnUInt

Summary

Provides the value of one column of one row of a result set in the form of an Unsigned Integer (UInt).

Twin

SQL_ResColUInt

Family

Result Column Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ResultColumnUInt(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_ResColUInt for complete details.

Return Values

See twin function SQL_ResColUInt for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_ResultColumnUInt is identical to SQL_ResColUInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_ResultColumnUInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_ResColUInt for complete details.

Example

See twin function SQL_ResColUInt.

Driver Issues: None.

Speed Issues: None.

See Also: Result Column Family

SQL_ResultRowCount

Summary

Provides the number of rows that were affected by a SQL statement.

Twin

SQL_ResRowCount

Family

Result Count Family

Availability

Standard and Pro

Warning

This function should *not* be used to obtain the number of rows that are contained in a result set that was generated by a SQL [SELECT](#) statement. It should only be used for non-[SELECT](#) statements. See [Why You CAN'T Use SQL_ResRowCount for SELECT Statements](#) for more information.

Syntax

```
lResult& = SQL_ResultRowCount(lDatabaseNumber&, _  
                               lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See [Using Database Numbers and Statement Numbers](#).

All Other Parameters

See twin function [SQL_ResRowCount](#) for complete details.

Return Values

See twin function [SQL_ResRowCount](#) for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, [SQL_ResultRowCount](#) is identical to [SQL_ResRowCount](#). To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to [SQL_ResultRowCount](#) is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see [Using Database Numbers and Statement Numbers](#).

Diagnostics

See twin function [SQL_ResRowCount](#) for complete details.

Example

See twin function [SQL_ResRowCount](#).

Driver Issues

None.

Speed Issues

None.

See Also

Results From non-SELECT Statements

SQL_SelectFile

Summary

Displays a standard Windows "Select A File" dialog box.

Twin

None

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_SelectFile(sTitle$, _  
                           sFileSpec$, _  
                           sInitialDir$, _  
                           sFilter$, _  
                           sDefExtension$, _  
                           lFlags&)
```

Parameters

sTitle\$

A string that specifies the title that the dialog box should display. If an empty string is used for this parameter, the default title is "Open".

sFileSpec\$

VERY IMPORTANT NOTE: This is an input-output parameter. See **Remarks** below for complete information.

sInitialDir\$

The drive and/or directory that will be displayed in the "Look In" listbox when the dialog box is first displayed. This parameter also affects the initial file-list display. If an empty string is used for this parameter, the dialog box will start in the default directory.

sFilter\$

The description(s) and file filter(s), separated with pipe symbols, that the dialog box should display in the "Files Of Type" listbox. See **Remarks** below for more information.

sDefExtension\$

The default extension. If the user types a file name without an extension, this string is automatically appended to the file name when the function exits. Also see OFN_EXTENSIONDIFFERENT below, for another use of this parameter.

lFlags&

VERY IMPORTANT NOTE: This is an input-output parameter. A bitmasked value that can contain many different options. See **Remarks** below for a complete list.

Return Values

This function actually returns *three* values.

- 1) The numeric return value of the function indicates the dialog box *button* that the

user selected. If the Open button is selected, the value `OPEN_BUTTON` is returned. Otherwise, `CANCEL_BUTTON` is returned. (This value can also be accessed with the `SQL_MsgBoxButton` function.)

2) The `sFileSpec$` parameter, which is passed to the function, will contain the name of the selected file (if any) when the function returns

3) The `IFlags&` parameter, which is passed to the function, will contain additional information about the file that the user selected.

Remarks

`sFileSpec$` and `IFlags&` are unusual parameters (at least for SQL Tools parameters) because they are used for both input *and* output. You should therefore use variables (not literal values) for these parameters, so that your program can use the values that are returned.

`sFileSpec$` Input value: The string that you pass to this parameter is used as an initial file specification. The file name (like `MYFILE.TXT`) or file spec (like `*.DSN`) that you specify will appear in the "File Name" field of the dialog box. (It is possible to do so, but you should not normally specify a drive and/or directory with this parameter.) If an empty string is used for this parameter, the File Name field will be blank when the dialog is first displayed.

`sFileSpec$` Output value: When the `SQL_SelectFile` function returns, this parameter will contain the name of the file that was actually selected by the user. If no file was selected (as would be the case if the user selected the Cancel button), this will be an empty string.

The `sFilter$` parameter (input only) can be used to specify one or more *pairs* of strings that will be displayed in the "Files Of Type" listbox. Each string pair should represent a "description" and a matching "file spec", separated by the pipe (|) symbol. If you use more than one pair, they should also be separated by pipe symbols. For example, if you use the string...

```
DSN Files|.DSN
```

...the "Files Of Type" listbox will contain the string "DSN Files" and only files that match the filter `*.DSN` will be displayed. If you use the string...

```
DSN Files|.DSN|Text Files|.TXT|Batch Files|.BAT
```

...the initial display will be the same as if you had used the shorter string above, but the Files Of Type listbox will allow you to select "DSN Files", "Text Files", or "Batch Files", and whenever one of those items is selected, the corresponding filter will be used.

Note that the filter itself is not automatically displayed. If you want the listbox to say "DSN Files (*.DSN)" you must use a string with duplicate information like this:

```
DSN Files (*.DSN)|*.DSN
```

You can also specify multiple filters for a single description by using semicolons. For example, using...

```
Source Code Files|.BAS;*.INC;*.RES
```

...would display the files that match *all* of the filters shown.

If you use an empty string for *sFilter\$*, files of all types will be shown and the Files Of Type listbox will be empty.

The *IFlags&* value is an *input-output* parameter.

For input purposes, you can tell the `SQL_SelectFile` function how to perform certain operations by passing the flag values shown below to the function. You can add any of the flag values together (see **Example** below) to specify multiple options. Also see Using Bitmasked Values.

Input flag `OFN_ALLOWMULTISELECT`

Tells the `SQL_SelectFile` function to allow the selection of multiple files. If the user does in fact select more than one file, the *sFileSpec\$* parameter will return the path to the current directory, followed by the filenames of the selected files. All of the elements of the *sFileSpec\$* string (the directory and all of the file names) will be separated by `CHR$(0)`. Multiple files are selected by holding down a Shift or Ctrl key while clicking on file names.

Input Flag `OFN_CREATEPROMPT`

This flag has no effect unless the `OFN_FILEMUSTEXIST` flag is also used.

If the user types the name of a file that does not exist, the `OFN_CREATEPROMPT` option causes the dialog box to prompt the user for permission to create the file. If the user chooses to create the file, the dialog box will close and the *sFileSpec\$* parameter will contain the name of the file that was entered by the user. Otherwise, the file-selection dialog box will remain open and allow the user to make another selection. *In any case, the file will not actually be created automatically. Your program must do that.*

Input Flag `OFN_FILEMUSTEXIST`

If you use this flag, it specifies that the user can only select existing files. If the user types an invalid name and selects the Open button, the `SQL_SelectFile` function will display a warning message and refuse to close. (If this flag is specified, the `OFN_PATHMUSTEXIST` flag is also used automatically.)

Input Flag `OFN_HIDEREADONLY`

Hides the Read Only check box that is normally displayed on the dialog box.

Input Flag `OFN_NOCHANGEDIR`

If the user changes the directory while searching for files, this option restores the current directory to its original value when the Open or Cancel button is selected. It does not, however, keep your program's current directory from being changed *while* files are being selected. This can be important if you are creating a multi-threaded applications, because the current directory of *all threads* will be temporarily changed during the file-selection process. (This is the normal behavior of the Windows select-file dialog. It is not a bug in SQL

Tools.)

Input Flag OFN_NODEREFERENCELINKS

Affects the selection of Windows Shortcut files. If you use this option, and if the user selects a shortcut file, the `SQL_SelectFile` function will return the path and filename of the selected shortcut (.LNK) file. If this option is not used, the function will automatically return the path and filename of the file that is *referenced* by the shortcut.

Input Flag OFN_NONETWORKBUTTON

Hides and disables the Network button that is normally displayed on the dialog box.

Input Flag OFN_NOTESTFILECREATE

This description is from the official Microsoft Win32 documentation: "*Specifies that the file is not created before the dialog box is closed. This flag should be specified if the application saves the file on a create-nonmodify network sharepoint. When an application specifies this flag, the library does not check for write protection, a full disk, an open drive door, or network protection. Applications using this flag must perform file operations carefully, because a file cannot be reopened once it is closed.*"

Input Flag OFN_NOVALIDATE

Specifies that the `SQL_SelectFile` function should allow invalid characters in the returned filename.

Input Flag OFN_PATHMUSTEXIST

Specifies that the user can type only valid (i.e. existing) paths. If this flag is used and the user types an invalid path in the File Name field, the `SQL_SelectFile` function will display a message box.

This flag is used automatically whenever the `OFN_FILEMUSTEXIST` flag is used.

Input Flag OFN_READONLY

The use of this flag causes the Read Only check box to be checked when the dialog box is first displayed. Also see Output Flag `OFN_READONLY` below.

The following flags can be *returned* by the `SQL_SelectFile` function. You can test the return value of *IFlags* for the following values by using the `AND` syntax (see **Example** below). Also see Using Bitmasked Values.

Output Flag OFN_EXTENSIONDIFFERENT

If this flag is set when the `SQL_SelectFile` function returns, it means that the user typed a filename extension that was different from the default extension that you specified with the `sDefExtension$` parameter.

Output Flag OFN_NOREADONLYRETURN

If this flag is set when the `SQL_SelectFile` function returns, it means that the selected file does not have the Read Only check box checked, and that it is not in a write-protected directory.

Output Flag `OFN_READONLY`

If this flag is set when the `SQL_SelectFile` function returns, it means that the Read Only check box was checked when the dialog box was closed.

Other (Non-Flag) Values

Finally, there is one `SQL_SelectFile` option that cannot be set with a "parameter" value. By default, the `SQL_SelectFile` function will use the Windows Desktop as its parent window. If you want to specify a different window or form, use the `SQL_SetOptionSInt(OPT_h_PARENT_WINDOW)` function.

Diagnostics

This function does not return Error Codes, ODBC Error Messages, or SQL Tools Error Messages.

Example

```
'Display a "Select File" dialog that:
'1) has the title "Select a DSN File"
'2) starts with nothing in the File Name
'   field,
'3) initially displays the files in the
'   \SQLTools directory,
'4) limits the file display to *.DSN files,
'5) does not have a Read Only button or
'   a Network button,
'6) requires that an existing file be
'   selected by the user,
'7) returns the flag OFN_EXTENSIONDIFFERENT
'   if a file that does not have the
'   default extension DSN is selected, and
'8) automatically resets the default directory
'   if the user changes it while looking for
'   a file.
```

```
lFlags& = OFN_FILEMUSTEXIST OR _
          OFN_NOCHANGEDIR OR _
          OFN_HIDEREADONLY OR _
          OFN_NONETWORKBUTTON
```

```
sFileSpec$ = ""
```

```
lResult& = SQL_SelectFile("Select a DSN File:", _
                          sFileSpec$, _
                          "\SQLTOOLS", _
                          "DSN Files|*.DSN", _
                          "*.DSN", _
                          lFlags&)
```

```

'examine the three different return values:

If lResult& = CANCEL_BUTTON Then
    'User selected Cancel
End If

If (lFlags& And OFN_EXTENSIONDIFFERENT) Then
    'User selected a non-DSN file.
    'Note the REQUIRED parentheses, which
    'force a "bitwise" operation.
End If

'display the name of the selected file:
Print sFileSpec$

```

Driver Issues

None.

Speed Issues

None.

See Also

Utility Family

SQL_SetDatabaseAttribStr

Summary

Sets the value of a database attribute which requires a string parameter.

Twin

SQL_SetDBAttribStr

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

Most attributes can only be set at certain times. See `SQL_SetDBAttribStr` for details.

Syntax

```
lResult& = SQL_SetDatabaseAttribStr(lDatabaseNumber&, _  
                                   lAttribute&, _  
                                   sValue$)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function `SQL_SetDBAttribStr` for complete details.

Return Values

See twin function `SQL_SetDBAttribStr` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_SetDatabaseAttribStr` is identical to `SQL_SetDBAttribStr`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_SetDatabaseAttribStr` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_SetDBAttribStr` for complete details.

Example

See twin function `SQL_SetDBAttribStr`.

Driver Issues: See twin function `SQL_SetDBAttribStr` for complete details.

Speed Issues: None.

See Also

Database Information and Attributes

SQL_SetDatabaseAttribUInt

Summary

Sets the value of a database attribute which requires a numeric parameter.

Twin

SQL_SetDBAttribUInt

Family

Database Info/Attrib Family

Availability

Most sub-functions are limited to **SQL Tools Pro Only** (see)

Warning

Most attributes can only be set at certain times. See SQL_SetDBAttribUInt for details.

Syntax

```
lResult& = SQL_SetDatabaseAttribUInt(lDatabaseNumber&, _  
                                     lAttribute&, _  
                                     dwValue???) 'or lValue&
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_SetDBAttribUInt for complete details.

Return Values

See twin function SQL_SetDBAttribUInt for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_SetDatabaseAttribUInt is identical to SQL_SetDBAttribUInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_SetDatabaseAttribUInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_SetDBAttribUInt for complete details.

Example

See twin function SQL_SetDBAttribUInt.

Driver Issues: See twin function SQL_SetDBAttribUInt for complete details.

Speed Issues: None.

See Also: Database Information and Attributes

SQL_SetDBAttribStr

Summary

Sets the value of a database attribute which requires a string parameter.

Twin

SQL_SetDatabaseAttribStr

Family

Database Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

Most attributes can only be set at certain times. See **Remarks** below for details.

Syntax

```
lResult& = SQL_SetDBAttribStr(lAttribute&, _  
                             sValue$)
```

Parameters

lAttribute&

One of the constants described in **Remarks**, below.

sValue\$

A valid value for the specified *lAttribute&*.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the attribute is changed successfully, or an ODBC Error Code or SQL Tools Error Code if it is not.

Remarks

IMPORTANT NOTE: Some database attributes can only be set after a database has been opened with `SQL_OpenDB` or `SQL_OpenDatabase`. Other attributes can be set only after the `SQL_OpenDatabase1` step has been completed, but before `SQL_OpenDatabase2`.

Only certain database attributes can be set using string values. For a list of attributes that are set using *numeric* values, see `SQL_SetDBAttribUInt`.

If you are setting a string attribute, *lAttribute&* must be one of the following values:

`DB_ATTR_CURRENT_CATALOG`

A string that contains the name of the catalog that is to be used by the Datasource. For example, a SQL Server catalog is a database, so the driver sends a *USE database* statement to the Datasource, where *database* is the string that was specified with this function. For a single-tier driver, on the other hand, the catalog might be a directory, so the driver would change its current directory to the directory specified by this function.

IMPORTANT NOTE: If you need to set this attribute, you must keep in mind that some ODBC drivers require it to be set between `SQL_OpenDatabase1` and `SQL_OpenDatabase2`. All drivers *allow* it to be set then, so if you need

to set this attribute, you should use `SQL_OpenDatabase1` and 2 instead of `SQL_OpenDB`, and set this attribute between steps 1 and 2. (Also remember to set the `DB_ATTR_ODBC_CURSORS` attribute, which is normally set by SQL Tools when `SQL_OpenDB` is used.)

`DB_ATTR_TRACEFILE`

This attribute can be used to specify the name of the "trace file" that is created when `SQL_SetDBAttrUInt(DB_ATTR_TRACE)` is set to `SQL_TRACE_ON`.

WARNING: If you are using the SQL Tools Trace Mode *and* the ODBC Trace Mode, you should not attempt to have both functions use the same file. If you do, one or both of the trace functions will fail.

`DB_ATTR_TRANSLATE_LIB`

A string that contains the name of a library containing the ODBC API functions called `SQLDriverToDataSource` and `SQLDataSourceToDriver`, which the ODBC driver uses (internally) to perform tasks such as character set translation.

IMPORTANT NOTE: This attribute *cannot* be set between `SQL_OpenDatabase1` and 2. It can only be set after a connection has been fully established, i.e. after the entire `SQL_OpenDB` process has been completed.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_DBAttrStr(DB_ATTR_CURRENTCATALOG)
```

Driver Issues

See **Remarks** above.

Speed Issues

None.

See Also

Database Information and Attributes

SQL_SetDBAttribUInt

Summary

Sets the value of a database attribute which requires a numeric parameter.

Twin

SQL_SetDatabaseAttribUInt

Family

Database Info/Attrib Family

Availability

Most sub-functions are limited to **SQL Tools Pro Only**.

Warning

Most attributes can only be set at certain times. See Remarks below for details.

Syntax

```
lResult& = SQL_SetDBAttribUInt(lAttribute&, _  
                                dwValue???)  
...or...  
lResult& = SQL_SetDBAttribUInt(lAttribute&, _  
                                lValue&)
```

Parameters

lAttribute&

One of the constants described in **Remarks**, below.

dwValue??? or lValue&

A valid value for the specified *lAttribute&*.

Return Values

This function returns SUCCESS or SUCCESS_WITH_INFO if the attribute is changed successfully, or an ODBC Error Code or SQL Tools Error Code if it is not.

Remarks

IMPORTANT NOTE: SQL Tools Standard can use this function, but *only* to set the *DB_ATTR_ODBC_CURSORS* and *DB_ATTR_TRACE* Attribute. SQL Tools Pro can be used to set any attribute listed below.

IMPORTANT NOTE: Some database attributes can only be set after a database has been opened with *SQL_OpenDB* or *SQL_OpenDatabase*. Other attributes can be set only after the *SQL_OpenDatabase1* step has been completed, but before *SQL_OpenDatabase2*.

Only certain database attributes can be set using numeric values. For a list of attributes that are set using *string* values, see *SQL_SetDBAttribStr*.

If you are setting a numeric attribute, *lAttribute&* must be one of the following values:

DB_ATTR_ACCESS_MODE **SQL Tools Pro Only**

ODBC 3.x+ ONLY: This attribute can be set to *SQL_MODE_READ_WRITE* (the default) or *SQL_MODE_READ_ONLY*.

IMPORTANT NOTE: `SQL_MODE_READ_ONLY` is only used as an indicator that the database is not *required* to support SQL statements that cause database updates. The ODBC driver is not required to *prevent* update statements from being submitted or executed. The behavior of the driver when asked to process SQL statements that are not read-only during a read-only connection is defined differently by different ODBC drivers.

IMPORTANT NOTE: If you need to set this attribute, you must keep in mind that some ODBC drivers only allow it to be set *between* `SQL_OpenDatabase1` and `SQL_OpenDatabase2`. All drivers allow it to be set then, so if you need to set this attribute, you should use `SQL_OpenDatabase1` and `2` instead of `SQL_OpenDB` or `SQL_OpenDatabase`, and set this attribute between steps 1 and 2. (Also remember to set the `DB_ATTR_ODBC_CURSORS` attribute, which is normally set by SQL Tools when `SQL_OpenDatabase` is used.)

`DB_ATTR_AUTOCOMMIT` **SQL Tools Pro Only**

This attribute can be set to `SQL_AUTOCOMMIT_OFF` or `SQL_AUTOCOMMIT_ON` (the default). The `SQL_DBAutoCommit` function is usually used to set this attribute value.

If you use `SQL_AUTOCOMMIT_OFF`, your program must use the `SQL_EndTrans` function to either commit or roll back each transaction.

IMPORTANT NOTE: Some Datasources delete all prepared statements and close all open statements each time a statement is committed. The AutoCommit mode can cause this to happen after each non-query statement is executed, or when the cursor is closed for a query.

IMPORTANT NOTE: When a batch is executed in the AutoCommit mode, two different behaviors are possible: **1)** The *entire* batch can be treated as an autocommittable unit, or **2)** each *statement* in a batch can be treated as an autocommittable unit. Each ODBC driver defines for itself which behavior it will support. (Some Datasources support both behaviors and provide a way of choosing between them.)

`DB_ATTR_CONNECTION_TIMEOUT` **SQL Tools Pro Only**

ODBC 3.x+ ONLY : This attribute can be used to tell the ODBC driver how long it should wait for a request to be completed before returning control to your program. The default value is zero (0), meaning "no timeout", i.e. "wait forever".

IMPORTANT NOTE: This attribute can only be set *between* `SQL_OpenDatabase1` and `SQL_OpenDatabase2`. If you need to set this attribute, you should use `SQL_OpenDatabase1` and `2` instead of `SQL_OpenDB` or `SQL_OpenDatabase`, and set this attribute between steps 1 and 2. (Also remember to set the `DB_ATTR_ODBC_CURSORS` attribute, which is normally set by SQL Tools when `SQL_OpenDatabase` is used.)

`DB_ATTR_DISCONNECT_BEHAVIOR` **SQL Tools Pro Only**

ODBC 3.x+ ONLY: *This attribute is not fully documented by the Microsoft*

ODBC Software Developer Kit. It appears to be related to connection pooling. This attribute will always be SQL_DB_RETURN_TO_POOL (zero) or SQL_DB_DISCONNECT (one) but the official documentation does not define what that means.

DB_ATTR_LOGIN_TIMEOUT **SQL Tools Pro Only**

The number of seconds that the driver should wait for a login request to be completed before returning to your program. The default value is driver-dependent, but it is often zero (0), which means "no timeout", i.e. "wait forever".

DB_ATTR_ODBC_CURSORS

This attribute can be set to one of the following values:

SQL_CUR_USE_IF_NEEDED (The ODBC cursor library is used only if the ODBC driver does not support the requested behavior. This is the SQL Tools default setting for this parameter. It is used automatically whenever SQL_OpenDB or SQL_OpenDatabase is used.)

SQL_CUR_USE_ODBC (The ODBC cursor library is always used, even if the ODBC driver supports the requested behavior.)

SQL_CUR_USE_DRIVER (The ODBC cursor library is never used. This is the default setting if you use SQL_OpenDatabase1 and SQL_OpenDatabase2 instead of using SQL_OpenDB or SQL_OpenDatabase. If you use SQL_OpenDatabase1 and 2 because you need to set a different attribute, you will probably *also* need to set the DB_ATTR_ODBC_CURSORS attribute.)

IMPORTANT NOTE: This attribute can only be set *between* SQL_OpenDatabase1 and SQL_OpenDatabase2. If you need to set this attribute, you should use SQL_OpenDatabase1 and 2 instead of SQL_OpenDB or SQL_OpenDatabase, and set this attribute between steps 1 and 2.

DB_ATTR_PACKET_SIZE **SQL Tools Pro Only**

This value specifies the network packet size, in bytes. *Many Datasources do not allow this attribute to be set.*

IMPORTANT NOTE: If a datasource allows it to be set, this attribute can *only* be set between SQL_OpenDatabase1 and SQL_OpenDatabase2. If you need to set this attribute, you should use SQL_OpenDatabase1 and 2 instead of SQL_OpenDB or SQL_OpenDatabase, and set this attribute between steps 1 and 2. (Also remember to set the DB_ATTR_ODBC_CURSORS attribute, which is normally set by SQL Tools when SQL_OpenDatabase is used.)

DB_ATTR_QUIET_MODE **SQL Tools Pro Only**

You can use this option to specify a 32-bit window handle value that will be used as the parent window when dialog boxes are displayed by the ODBC driver. If the value of this attribute is zero (the default), the ODBC driver will

not display any dialog boxes.

IMPORTANT NOTE: The `DB_ATTR_QUIET_MODE` attribute does not affect dialog boxes that are displayed by SQL Tools, such as those provided by `SQL_OpenDB`, `SQL_SelectFile`, `SQL_MsgBox`, etc. Those dialogs use the Windows desktop as the default parent window. See `SQL_hParentWindow` for more information.

`DB_ATTR_TRACE`

This attribute can be set to `SQL_TRACE_OFF` (the default) or `SQL_TRACE_ON`. If it is set to `ON`, the ODBC driver will create a "trace file" that contains all of the ODBC API function calls that SQL Tools makes. (Also see `SQL_SetDBAttrStr(DB_ATTR_TRACEFILE)`.)

Please note that this ODBC API Trace Mode is not the same thing as the SQL Tools Trace Mode. See `SQL_Trace` for more information.

WARNING: Because it involves the creation of a large text file, the use of the ODBC Trace Mode can *greatly* slow down a program. One of our very small test programs took 40.50 seconds to execute when the ODBC Trace Mode was turned on, but less than 0.05 seconds with tracing turned off. And the slowdown can be made worse if the SQL Tools Trace Mode is used at the same time, or if an existing Trace File is being appended (which is the default behavior). Instead of activating the ODBC Trace Mode at the very beginning of your program, we suggest that you attempt to isolate a small section of code that is likely to be causing a problem, and turn the ODBC Trace Mode on then off again as quickly as possible.

`DB_ATTR_TRANSLATE_OPTION` **SQL Tools Pro Only**

A 32-bit bitmasked value that is passed to the translation DLL.

IMPORTANT NOTE: This attribute *cannot* be set between `SQL_OpenDatabase1` and `SQL_OpenDatabase2`. It can only be set after a connection has been established, i.e. after the entire `SQL_OpenDatabase` process has been completed.

`DB_ATTR_TXN_ISOLATION` **SQL Tools Pro Only**

A 32-bit bitmasked value that sets the transaction isolation level for the current connection. A program must call `SQL_EndTrans` to commit or roll back all open transactions *before* setting this attribute. The valid values for this function can be determined by using the `SQL_DBInfoUInt(DB_TXN_ISOLATION_OPTION)` function.

IMPORTANT NOTE: This attribute can only be set when there are no open transactions on the database.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_DBAttrUInt(DB_ATTR_ODBCCURSORS)
```

Driver Issues

See **Remarks** above.

Speed Issues

None.

See Also

Database Information and Attributes

SQL_SetEnvironAttribSInt

Summary

Sets the value of an ODBC environment attribute, which affects all databases. While this function *can* be used, it is not usually necessary. Most of the important ODBC environment attributes should usually be set with the `SQL_Initialize` function.

Twin

None. (Please also note that there is no corresponding Str (string) function because all of the Environment Attributes are numeric.)

Family

Environment Family

Availability

Some sub-functions are limited to the **SQL Tools Pro Only** (see)

Warning

None.

Syntax

```
lResult& = SQL_SetEnvironAttribSInt(lAttribute&, _  
                                   lValue&)
```

Parameters

lAttribute&

One of the constants described in **Remarks**, below.

lValue&

A valid value for the specified *lAttribute&*.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the attribute is changed successfully, or an ODBC Error Code SQL Tools Error code if it is not changed.

Remarks

Most of the important ODBC environment attributes should normally be set with the `SQL_Initialize` function. While the `SQL_SetEnvironAttribSInt` function *can* be used, it is not usually necessary.

The *lAttribute&* parameter must have one of the following values:

`ENV_CONNECTION_POOLING` **SQL Tools Pro Only** (see)

This attribute can be set to one of the following values:

`SQL_CP_OFF` (Connection pooling is turned off. This is the default.)

`SQL_CP_ONE_PER_DRIVER` (A single connection pool is supported for each driver. Every database connection in a pool is associated with one driver.)

`SQL_CP_ONE_PER_HENV` (A single connection pool is supported for each environment. Every database connection in a pool is associated with one environment, i.e. one program.)

See `SQL_Initialize` for more information.

ENV_CP_MATCH

This attribute is ignored unless ENV_CONNECTION_POOLING has been set to a value other than SQL_CP_OFF.

This attribute can be set to one of the following values:

SQL_CP_STRICT_MATCH (Only connections that *exactly* match the connection options and connection attributes specified by your program are reused. This is the default value.)

SQL_CP_RELAXED_MATCH (Connections with matching connection string keywords can be used. Keywords must match, but not all connection attributes must match.)

See SQL_Initialize for more information.

ENV_ODBC_VERSION

This attribute can be set to either two (2) or three (3), to indicate the ODBC Version behavior that should be emulated by the environment. If an ODBC function (and therefore a SQL Tools function) behaves differently if ODBC 2 or 3 is used, this function can be used specify which behavior should be emulated.

By default, SQL Tools sets this attribute to 3 because most drivers can support at least *some* ODBC 3.x behavior.

See SQL_Initialize for more information.

ENV_OUTPUT_NTS

This attribute is "read-only" and cannot be set. See SQL_EnvironAttribSInt for more information.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages. Furthermore, the use of this function can cause many *other* SQL Tools functions to generate ODBC Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Database Information and Attributes
Statement Information and Attributes

SQL_SetOptionSInt

Summary

Sets the value of a SQL Tools numeric option.

Twin

None.

Family

Configuration Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_SetOptionSInt(lOption&, _  
                             lValue&)
```

Parameters

lOption& and lValue&

An option, and the value to which that option should be set. See **Remarks** below.

Return Values

This function returns `SUCCESS` if valid parameters are used, or `ERROR_BAD_PARAM_VALUE` if they are not.

Remarks

Not *all* SQL Tools Option values are useful in numeric form. For example, the `OPT_MY_PROGRAM` option is usually used to specify the name of your program, so using the `SQL_SetOptionSInt` function to set this option to a numeric value would not usually be desirable. It is possible, however, to assign a value like the string "2000" to the `OPT_MY_PROGRAM` string, in which case the `SQL_SetOptionSInt` function *could* be used. (There are other examples of this, which should become clear later.)

For that reason, SQL Tools allows all options to be changed *and* read with both string and numeric functions. In order to avoid errors when this document is updated in the future, a single list of all of the various SQL Tools Options is provided in the Reference Guide's `SQL_SetOptionStr` entry.

Diagnostics

This function returns Error Codes and can generate SQL Tools Error Messages.

Example

```
SQL_SetOptionSInt OPT_MAX_ERRORS, 32
```

Driver Issues: None.

Speed Issues: None.

See Also: Configuration Family

SQL_SetOptionStr

Summary

Sets the value of a SQL Tools string option. (Information about numeric options is also provided below.)

Twin

None.

Family

Configuration Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_SetOptionStr(lOption&, _  
                             sValue$)
```

Parameters

lOption&

See **Remarks** below for a complete list of valid values.

sValue\$

A valid value for the specified *lOption&*.

Return Values

This function returns `SUCCESS` if the option is successfully changed, or an Error Code if it is not.

Remarks

IMPORTANT NOTE: Not *all* SQL Tools Option values are useful in string form. For example, the `OPT_ERROR_SOUNDTYPE` option is used to specify a numeric value which tells SQL Tools what kind of sound it should make (if any) when an error is detected. Normally, you would use the `SQL_SetOptionSInt` function to specify a numeric value like 2. It is also possible, however, to use the `SQL_SetOptionStr` function to assign a value like the string "2" to the `OPT_ERROR_SOUNDTYPE` option, which would accomplish exactly the same thing. (There are other examples of this, which should become clear later.)

For that reason, SQL Tools allows all options to be changed *and* read with both string and numeric functions. In order to avoid errors when this document is updated in the future, a single list of all of the various SQL Tools Options is provided here. You should use this list whenever you are using the `SQL_OptionStr`, `SQL_OptionSInt`, `SQL_SetOptionStr`, or `SQL_SetOptionSInt` function.

The list is presented in alphabetical order, and the *normal* method of using the option (**Str** or **SInt**) is listed after the constant name.

`OPT_AUTOAUTO_BIND` (**SInt**)

This option tells SQL Tools whether or not it should automatically autobind all

of the columns in a result set whenever the `SQL_Stmt (SQL_STMT_EXECUTE)` or `SQL_Stmt (SQL_STMT_IMMEDIATE)` function is used to execute a SQL statement.

The default value for this option is Logical True (-1). Set this value to zero (0) to turn off `AutoAutoBinding`, or any nonzero value to turn it back on. *If you turn it off, your program is responsible for the binding of all result columns.*

`OPT_AUTOCLOSE_DB (SInt)`

This option tells SQL Tools whether or not it should automatically close an open *database* if your program attempts to open another database using the same database number.

The default value for this option is Logical True (-1). Set this value to zero (0) to turn off the Database AutoClose feature, or any nonzero value to turn it on. If you turn it off, your program is responsible for closing a database (with the `SQL_CloseDB` function) before opening another database using the same database number.

Even if this option is turned on, your program can use the `SQL_CloseDB` function to explicitly close a database.

If you turn this option off and fail to use the `SQL_CloseDB` function properly, a SQL Tools Error Message (`ERROR_DB_NOT_CLOSED`) will be generated by the `SQL_OpenDB` statement.

`OPT_AUTOCLOSE_STMT (SInt)`

This option tells SQL Tools whether or not it should automatically close an open *statement* if your program attempts to open another statement using the same statement number.

The default value for this option is Logical True (-1). Set this value to zero (0) to turn off the Statement AutoClose feature, or any nonzero value to turn it on. If you turn it off, your program is responsible for closing an open statement (with the `SQL_CloseStmt` function) before opening another statement using the same statement number.

Even if this option is turned on, your program can use the `SQL_CloseStmt` function to explicitly close a statement. In fact, this is recommended practice if you are going to change a statement's mode. (Using the `SQL_StmtMode` function while a statement is open will generate an `ERROR_ADVISORY` message to warn you that the mode change will not take effect until the next time a statement is opened.)

If you turn off this option and fail to use the `SQL_CloseStmt` properly, a SQL Tools Error Message (`ERROR_STMT_NOT_CLOSED`) will be generated by the `SQL_Stmt` or `SQL_OpenStmt` function.

`OPT_AUTOOPEN_DB (SInt)`

This option tells SQL Tools whether or not it should automatically use the

SQL_OpenDB function to prompt the user for a database connection if your program attempts to use the SQL_Stmt function with a database number that is not currently open.

The default value for this option is Logical True (-1). Set this value to zero (0) to turn off the Database AutoOpen feature, or any nonzero value to turn it on. If you turn it off, your program is responsible for opening a database *before* using the SQL_Stmt function.

Even if this option is turned on, your program can still use SQL_OpenDB to manually open a database. In fact, this is the recommended procedure. The Database AutoOpen feature is provided as a programming convenience, for those time when you are writing "quick and dirty" programs.

If you turn this option off and fail to use SQL_OpenDB before you use SQL_Stmt or SQL_OpenStmt, a SQL Tools Error Message (ERROR_DB_NOT_OPEN) will be generated.

OPT_AUTOOPEN_STMT (**SInt**)

This option tells SQL Tools whether or not it should automatically use the SQL_OpenStmt function to open a statement if your program attempts to use the SQL_Stmt function with a statement number that is not currently open.

The default value for this option is Logical True (-1). Set this value to zero (0) to turn off the Statement AutoOpen feature, or any nonzero value to turn it on. If you turn it off, your program is responsible for using the SQL_OpenStmt function to explicitly open a statement before using the SQL_Stmt function.

Even if this option is turned on, your program can still use the SQL_OpenStmt function to manually open a statement.

If you turn this option off and then fail to use SQL_OpenStmt properly, a SQL Tools Error Message (ERROR_STMT_NOT_OPEN) will be generated by the SQL_Stmt function.

OPT_BIND_TYPE, OPT_BLOCK_SIZE (both **SInt**)

These options specify default Statement Modes. See SQL Statement Mode.

OPT_COL_DELIMITER (**Str**)

This option is used to specify the Column Delimiter string that SQL Tools uses to delimit multiple return values from the SQL_Diagnostic function.

The default value for this option is one comma and one space (", ").

OPT_CONCURRENCY, OPT_CURSOR_SCROLLABLE, OPT_CURSOR_SENSITIVITY, OPT_CURSOR_TYPE (All **SInt**)

These options specify default Statement Modes. See SQL Statement Mode.

OPT_DATALEN_BINARY, OPT_DATALEN_CHAR, OPT_DATALEN_CHUNK,
OPT_DATALEN_LONGVARBINARY, OPT_DATALEN_LONGVARCHAR,
OPT_DATALEN_UNKNOWN, OPT_DATALEN_VARBINARY, OPT_DATALEN_VARCHAR,
OPT_DATALEN_wCHAR, OPT_DATALEN_wVARCHAR,
OPT_DATALEN_wLONGVARCHAR (All **SInt**)

The `_DATALEN_` or "data length" options are used to tell SQL Tools the size of the "bind buffer" that it should create for the various SQL data types.

SQL Tools uses a default value of 256 for all of these data types, except for **1)** the three Unicode Data Types, which use a length of 512 bytes (256 *characters*) and **2)** `OPT_DATALEN_CHUNK`, which is used by the `SQL_LongResCol` function. The `OPT_DATALEN_CHUNK` default value is 32,768 (32k) bytes.

For example, SQL Tools uses a default buffer size of 256 bytes for all `SQL_BINARY` result columns, unless you use the `SQL_SetOptionSInt(OPT_DATALEN_BINARY)` function to change the default value. And it uses a default "chunk" size of 32k bytes for the `SQL_LongResCol` function unless you use the `SQL_SetOptionSInt(OPT_DATALEN_CHUNK)` function to change the default value.

VERY IMPORTANT NOTE: You should only change these values *before* you use the `SQL_Stmt` function. If you execute a SQL statement and these values are used to create buffers, and then you change these values, SQL Tools may or may not be able to maintain the statement's buffers correctly.

`OPT_DATE_FORMAT`

Determines the date format that will be returned by certain date-time functions. The default value is an empty string ("") which produces the system default date format. See the `SQL_ResColDate` function for a complete description of the strings that can be used for this option.

`OPT_DATETIME_SEPARATOR`

The character(s) that will appear between the date and time that are returned by the `SQL_ResultColumnDateTime` function. The default value is a single space character.

`OPT_ERR999000030` through `OPT_ERR999000049` (all **Str**)

These options are used to specify the text messages that are associated with the various SQL Tools Error Codes.

For example, the default value for `OPT_ERR999000030` is "Bad Parameter Value". If you prefer different wording (or a different language), you can change these values.

OPT_ERRMSGBOX1 (**Str**)
OPT_ERRMSGBOX2 (**Str**)
OPT_ERROR_MSGBOXTYPE (**SInt**)

These options are used to control the message box that SQL Tools can display whenever an error is detected.

The default value for OPT_ERROR_MSGBOXTYPE is MSGBOX_NONE, so SQL Tools does not display this message box unless you change this option to one of the following values: MSGBOX_OK, MSGBOX_OKCANCEL, MSGBOX_ABORTRETRYIGNORE, MSGBOX_YESNOCANCEL, MSGBOX_YESNO, or MSGBOX_RETRYCANCEL. The names of the constants indicate which buttons will appear on the message box.

If the OPT_ERROR_MSGBOXTYPE option has been set to a value other than MSGBOX_NONE, the OPT_ERRMSGBOX1 and OPT_ERRMSGBOX2 options can be used to specify the wording that is used in the message box.

The default value of OPT_ERRMSGBOX1 is "ERROR: ". This string is automatically added to the beginning of every error-message box. You can change this value to use different wording, such as "ERROR DETECTED! PLEASE REPORT THE FOLLOWING INFORMATION TO THE TECHNICAL SUPPORT DEPARTMENT: ". You could also use a phrase in a different language.

The default value of OPT_ERRMSGBOX2 is an empty string. If you specify a value for this option, the string will be added to the *end* of the text in the error-message-box.

Keep in mind that it is possible to use SQL_IString "shorthand" strings (see) to add text formatting (such as NewLine characters) to these strings.

See OPT_ICON_ID and OPT_MY_PROGRAM (below) for other ways to customize the error-message box.

Also see OPT_ERROR_SOUNDTYPE (just below).

OPT_ERROR_SOUNDTYPE (**SInt**)

SQL Tools can optionally play a Windows Event Sound whenever an error is detected.

The default value for this option is SOUND_NONE, so no sound is normally played when an error is detected. You can change this option to SOUND_OK, SOUND_HAND, SOUND_QUESTION, SOUND_EXCLAMATION, or SOUND_ASTERISK. (The constant names correspond to the standard Windows Event Sound names. The actual sound that is produced for each value will depend on your computer's configuration at runtime.)

OPT_EXIT_CHECK (**SInt**)

Normally, your program should handle and clear all errors from the SQL Tools Error Stack before it exits. During development it is useful to know whether or not there are any "unhandled errors" when the SQL_Shutdown

function is used, so you may want to set this option to Logical True. If there are any unhandled errors in the Error Stack when SQL Tools unloads from memory (the final shutdown step), a message box will be displayed.

When it comes time you distribute your program you will probably want to set this option to False (zero), so that your users will not see the message box.

OPT_FETCH_BOOKMARK_PTR (**SInt**)

This option specifies a default Statement Mode. See SQL Statement Mode.

OPT_h_EXE_INSTANCE (**SInt**)

This option can be used to specify the "EXE Instance Handle" of your program. It is common to specify this value as a parameter of the `SQL_Initialize` function (see), but if you prefer to use `SQL_Init` you can use this option to pass the Instance Handle to SQL Tools.

SQL Tools only needs to know the instance handle of your EXE program if you want to use the `OPT_ICON_ID` option (see below).

OPT_h_PARENT_WINDOW (**SInt**)

This option can be used to tell SQL Tools to use a specific window as the parent window of the dialog boxes that it displays. See `SQL_hParentWindow` for more details.

OPT_ICON_ID (**SInt**)

By default, SQL Tools uses a Perfect Sync S logo as the icon that is displayed in all message boxes. You can use this option to specify a different icon.

You may use any of the following values: `ICON_SQLTOOLS`, `ICON_APPLICATION`, `ICON_HAND`, `ICON_ERROR`, `ICON_QUESTION`, `ICON_EXCLAMATION`, `ICON_WARNING`, `ICON_ASTERISK`, `ICON_INFORMATION`, or `ICON_WINLOGO`. The names of the constants correspond to the Windows names of the various standard icons. The actual images that are displayed will depend on the runtime configuration of your computer, including the Windows version.

If you have already set the `OPT_h_EXE_INSTANCE` option (see above), you can also specify the Resource ID Number of an icon that is embedded in your EXE or DLL file. (Icons are embedded by using the PowerBASIC `$RESOURCE` metastatement, or by adding a Resource File to a Visual Basic project.)

`OPT_ISTRING_ASCII`, `OPT_ISTRING_CR`, `OPT_ISTRING_ENTER`,
`OPT_ISTRING_LF`, `OPT_ISTRING_PREFIX`, `OPT_ISTRING_QUOTE`,
`OPT_ISTRING_REPLACE`, `OPT_ISTRING_SEARCH`, `OPT_ISTRING_SUFFIX`,
`OPT_ISTRING_TAB` (All **Str**)

These options are used to control the way the `SQL_IString` function works.

OPT_KEYSET_SIZE (**SInt**)

This option specifies a default Statement Mode. See SQL Statement Mode.

OPT_LONGRES_COLTYPE

See SQL_LongResCol for complete information.

OPT_MAXCONNSTRINGLEN (**SInt**)

The default value for this option is 1024 bytes (1kB). That means that SQL Tools can use connection strings that are up to 1024 bytes long when opening a database. It should only be necessary to increase this value if your program uses *very* complex connection strings. For more information, see SQL_OpenDB.

OPT_MAX_DB_NUMBER (**SInt**)

Under normal circumstances, this value is set with the *IMaxDatabaseNumber&* parameter of the SQL_Initialize function.

However, if your program uses a value for SQL_Initialize that turns out to be too large once the program is running, you can *reduce* the *IMaxDatabaseNumber&* value by using this option. For example, if your program determines that it has connected to a database that does not support multiple connections, you might reduce this number to one (1).

IMPORTANT NOTE: The use of this option does *not* reclaim the memory that SQL Tools reserved for the original *IMaxDatabaseNumber&* value. It will, however, affect the operation of functions like SQL_NewDBNumber. In fact, all SQL Tools functions will generate an ERROR_BAD_PARAM_VALUE error message if you use a database number that is larger than the new value that you specify with this option.

You must be careful not to reduce the *IMaxDatabaseNumber&* value while a database with a larger number is open. For example, if database number 2 is open and you reduce the *IMaxDatabaseNumber&* value to 1, it will be impossible for your program to access (or even close) database number 2.

It is not possible to *increase* the *IMaxDatabaseNumber&* value by using this option.

Also see OPT_MAX_STMT_NUMBER below.

OPT_MAX_ERRORS (**SInt**)

If multiple runtime errors are detected, SQL Tools stores the errors in the Error Stack. The performance of SQL Tools can be affected if too many errors accumulate in the stack, so the OPT_MAX_ERRORS option is used to specify the maximum number of errors that can be stored in the stack at any one time.

The default value for this option is 64. If 64 error messages are in the error stack and a 65th error is detected, the oldest error in the stack will be discarded.

In practice, your program should handle and clear errors long before the `OPT_MAX_ERRORS` value is reached. This feature is provided primarily as an aid during program development. You can increase or decrease the `OPT_MAX_ERRORS` value during development, but we recommend a value of 64 (or a *smaller* value) for distribution programs.

`OPT_MAX_ITEM_NUMBER` (**SInt**)

Basically, this option controls the maximum number of tables and columns that SQL Tools can handle. Specifically, it controls the point at which the various `SQL_Get (Info)` functions return `ERROR_TOO_MANY`, so it affects nearly all of the Info functions.

The default value is 9,999. If your database contains an unusually large number of tables, columns, privileges, stored procedures, etc., the maximum value for this option is 65,535. If you are certain that your program will be used with relatively small numbers of tables and columns, you can save some memory and speed up your program (very slightly) by using a number like 100. The minimum legal value for this option is 64.

`OPT_MAX_STMT_NUMBER` (**SInt**)

Under normal circumstances, this value is set with the *MaxStatementNumber*& parameter of the `SQL_Initialize` function.

However, if your program uses a value for `SQL_Initialize` that turns out to be too large once the program is running, you can *reduce* the *MaxStatementNumber*& value by using this option. For example, if your program determines that it has connected to a database that does not support multiple concurrent statements, you might reduce this number to zero (0). (See Statement Zero Operation for an explanation of this unusual value.)

IMPORTANT NOTE: The use of this option does *not* reclaim the memory that SQL Tools reserved for the original *MaxStatementNumber*& value. It will, however, affect the operation of functions like `SQL_NewStmtNumber`. In fact, all SQL Tools functions will generate an `ERROR_BAD_PARAM_VALUE` error message if you use a statement number that is larger than the new value that you specify with this option.

You must be careful not to reduce the *MaxStatementNumber*& value while a statement with a larger number is open. For example, if statement number 2 is open and you reduce the *MaxStatementNumber*& value to 1, it will be impossible for your program to access (or even close) statement number 2.

It is not possible to *increase* the *MaxStatementNumber*& value by using this option.

Also see `OPT_MAX_DB_NUMBER` above.

`OPT_MAX_TEXTCHAR` (**SInt**)

The `OPT_MIN_TEXTCHAR` and `OPT_MAX_TEXTCHAR` options are used to tell the `SQL_TextStr` and `SQL_TextStrOLE` functions which characters

should be considered "printable". The default values for these options are 32 and 126, which means that characters from the space character `CHR$(32)` to the tilde character (~) are printable. (Virtually all Windows fonts support that character range.)

If a string that contains a character less than `CHR$(32)` or greater than `CHR$(126)` is submitted to the `SQL_TextStr` or `SQL_TextStrOLE` function, it will be converted to the `[CHR$(X)]` notation.

If you are using a font (such as Terminal) that supports a different range of characters, you can change the range of printable characters by using these options.

If you change these values so that the `MIN` value is larger than the `MAX` value, an Error Message will be generated whenever the `SQL_TextStr` or `SQL_TextStrOLE` function is used. Since those functions are used *internally* by SQL Tools, this can result in a large number of error messages.

`OPT_MAX_COLUMN_LENGTH`,
`OPT_MAX_RESULT_ROWS` (both **SInt**)

These options specify default Statement Modes. See SQL Statement Mode.

`OPT_MIN_TEXTCHAR` (**SInt**)

See `OPT_MAX_TEXTCHAR` above.

`OPT_MY_COMPANY`,
`OPT_MY_FUNCTION` (both **Str**)

These string values are not currently used by SQL Tools. They are provided as a programmer convenience, to complement the `OPT_MY_PROGRAM` and `OPT_MY_MODULE` options (see below).

`OPT_MY_MODULE`,
`OPT_MY_PROGRAM` (both **Str**)

These string values are used by SQL Tools in various error-related message boxes. The default values for these options are "My Module" and "My Program". You will therefore see message boxes that say things like...

ERROR: My Program / My Module / SQL_OpenDB

ERR #999000030 (and so on)

You can change these values to provide useful information to the person that sees the message. For example, your program could display message boxes that look like this:

ERROR: AddressBook 2000 / Main / SQL_OpenDB

ERR #999000030 (and so on)

(If you use the `SQL_ErrorSimulate` function in your program, the name of

one of your program's functions may be displayed instead of a SQL Tools function like `SQL_OpenDB`.)

`OPT_ODBC_CURSORS` (**SInt**)

This option specifies a default Statement Mode. See SQL Statement Mode.

`OPT_OLE_STRING_PARAMS`

This option provides an alternate method of telling SQL Tools that your program is using OLE strings for function parameters. (PowerBASIC programs use OLE strings; Visual Basic programs do not.) This setting is usually handled automatically by the presence of a `SQL_OleStringParams` function in the SQL Tools declaration file (`SQLT_*.INC`). If you prefer, you can remove that function from the declaration file and add `SQL_SetOptionSInt OPT_OLE_STRING_PARAMS` immediately after your program calls `SQL_Init` or `SQL_Initialize`.

`OPT_OPENDB_PROMPT` (**SInt**)

This option tells the `SQL_OpenDB` function which type of prompting it should use if the connection string that you provide is not sufficient to make a connection to a database. See the `SQL_OpenDatabase` function for more information, including a list of valid values. The default value for this option is `PROMPT_TYPE_COMPLETE`.

`OPT_PARAMSET_SIZE`, `OPT_PARAMS_PROCESSED_PTR`,
`OPT_PARAM_BIND_OFFSET_PTR`, `OPT_PARAM_BIND_TYPE`,
`OPT_QUERY_TIMEOUT`, `OPT_RETRIEVE_DATA` (All **SInt**)

These options specify default Statement Modes. See SQL Statement Mode.

`OPT_ROW_DELIMITER` (**Str**)

This option is used to specify a Row Delimiter that is used by the `SQL_ErrorQuickAll` function, to separate multiple errors.

The default value is a pipe symbol (|) with one space on either side.

`OPT_ROWS_FETCHED_PTR`, `OPT_ROW_BIND_OFFSET_PTR`, `OPT_ROW_NUMBER`,
`OPT_SCANFORESCAPES`, `OPT_SIMULATE_CURSOR` (All **SInt**)

These options specify default Statement Modes. See SQL Statement Mode.

`OPT_SOFT_SUCCESS` (**SInt**)

Most ODBC function can produce the `SUCCESS` (value 0) and `SUCCESS_WITH_INFO` (value 1) error codes. Your program can use code like this...

```
IF lResult& = SUCCESS OR _  
   lResult& = SUCCESS_WITH_INFO THEN...
```

...or it can use the `SQL_Okay` function, or it can change the value of the

OPT_SOFT_SUCCESS option to Logical True, in which case all SQL Tools functions will automatically return SUCCESS whenever SUCCESS_WITH_INFO is detected. If your program relies on Error Messages instead of Error Codes (which we recommend), you can then simply use...

```
IF lResult& = SUCCESS THEN...
```

The default value for this option is False.

OPT_SQLSTATE_PREFIX (**SInt**)

By default, SQL Tools uses the "number" symbol ("#", known in the United States as a Pound Sign), as the first character of SQL State values that it produces. This is intended to make it easy to differentiate SQL Tools Error Messages from Error Messages that are produced by ODBC drivers.

If you are using an ODBC driver that uses the # prefix, you can use this option to change the prefix that SQL Tools uses.

IMPORTANT NOTE: You must specify an ASCII value, not a string, for this option. The default value is 35, which is the ASCII value of #.

OPT_STAT_ENSURE (**SInt**)

See the SQL_TblStatSInt for complete information.

OPT_TABLE_CATALOG, OPT_TABLE_NAME, OPT_TABLE_SCHEMA, and OPT_TABLE_TYPES (**All SInt**)

These options control the types of tables about which the SQL_GetTblInfo function retrieves information. They therefore affect nearly *all* of the Database Info functions. The default value for all of these options is an empty string, which tells SQL_GetTblInfo to retrieve information about all types of tables. See SQL_GetTblInfo for more information.

OPT_TEXT_DELIMITER (**Str**)

See the SQL_ResColText function for complete information.

OPT_TEXT_FALSE (**Str**)

This option specifies the string that is returned by the SQL_ResColText function when a SQL_BIT column contains a False value. The default value is "False".

OPT_TEXT_MAXLENGTH (**SInt**)

This option is used to specify the length of the longest string that the SQL_LimitTextLength function will return. Since the SQL_LimitTextLength function is automatically used by the SQL_ResColText function, this option also controls the strings that are returned by that function.

OPT_TEXT_NULL (**Str**)

This option specifies the string that is returned by the `SQL_ResColText` function when a column contains a Null value. The default value is "[NULL]".

OPT_TEXT_QUOTE (**Str**)

Specifies the character that is inserted by the `SQL_ResColText` and `SQL_ResultColumnText` functions when a quote-delimited string contains a quote symbol. (Quoted strings cannot contain quotes or the delimiting would not be clear.) The default replacement character is the back-apostrophe (```).

OPT_TEXT_TRUE (**Str**)

This option specifies the string that is returned by the `SQL_ResColText` function when a `SQL_BIT` column contains a True value. The default value is "True".

OPT_TEXT_UNBOUND (**Str**)

This option specifies the string that is returned by the `SQL_ResColText` function when a column has not been bound by SQL Tools. The default value is "[UNBOUND]".

OPT_TIME_FORMAT

Determines the time format that will be returned by certain date-time functions. The default value is an empty string (" ") which produces the system default time format. See the `SQL_ResColTime` function for a complete description of the strings that can be used for this option.

OPT_TRACE_APPEND (**SInt**)

By default, the SQL Tools Trace Mode (see `SQL_Trace`) automatically appends an existing trace file (if any) when the trace mode is activated. You can change this option to False (zero) if you want a new trace file to overwrite an old file.

OPT_TRACE_FILE (**Str**)

This option is used to specify the name of the file that is created by the SQL Tools Trace Mode (see `SQL_Trace`). The default value is "`\\SQLTOOLS.LOG`".

WARNING: If you are using the SQL Tools Trace Mode *and* the ODBC Trace Mode, you should not attempt to have both functions use the same file. If you do, one or both of the trace functions will fail.

OPT_TRACE_INDENT (**SInt**)

By default, the SQL Tools Trace Mode (see `SQL_Trace`) automatically indents trace file entries to indicate the internal nesting of functions. You can

change this option to False (zero) if you do not want the entries to be indented. This makes the Trace Files harder to read, but somewhat smaller, and it therefore makes Tracing somewhat *faster*.

OPT_TRACE_TIMES (SInt)

By default, the SQL Tools Trace Mode (see `SQL_Trace`) automatically adds a timer value in the form `[123456.789]` to the beginning of every trace file entry. You can change this option to False (zero) if you do not want the time values to be included in the file. This makes the Trace Files somewhat smaller, and it therefore makes Tracing somewhat *faster*.

OPT_UNIQUE_SCOPE (SInt)

See `SQL_UniqColInfoSInt(UNIQCOL_SCOPE)` for complete information.

OPT_USE_FETCHSCROLL (SInt)

See `SQL_OpenDB` for information about this option.

OPT_USE_BOOKMARKS (SInt)

This option specifies a default Statement Mode. See `SQL Statement Mode`.

Diagnostics

This function returns Error Codes, and can generate SQL Tools Error Messages.

Example

These two lines of code would do exactly the same thing...

```
SQL_SetOptionSInt OPT_MAX_ERRORS, 32
```

```
SQL_SetOptionStr OPT_MAX_ERRORS, "32"
```

Driver Issues

None.

Speed Issues

None.

See Also

Statement Information and Attributes

SQL_SetPos

Summary

Sets the cursor position within a MultiRow Cursor, and optionally performs delete, update, refresh, and row-locking operations. *This function cannot be used to position a single-row cursor within a result set. For that, use `SQL_Fetch`.*

Twin

`SQL_SetPosition`

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

Some drivers *simulate* positioned update and delete statements, and may not be able to guarantee that the operation will not affect other rows. This problem can be minimized by the correct construction of your result set. For more information, see the `STMT_ATTR_SIMULATE_CURSOR` attribute of the `SQL_StmtMode` function.

Syntax

```
lResult& = SQL_SetPos(lOperation&, _  
                     lRowNumber&, _  
                     lLockType&)
```

Parameters

lOperation&

One of the following values: `SET_POSITION`, `SET_REFRESH`, `SET_UPDATE`, or `SET_DELETE`. See **Remarks** below for details.

lRowNumber&

The position of the row in the rowset on which *lOperation&* is to be performed. If *lRowNumber&* is zero (0), the operation will be performed on all of the rows in the rowset.

lLockType&

One of the following values: `SQL_LOCK_NO_CHANGE`, `SQL_LOCK_EXCLUSIVE`, or `SQL_LOCK_UNLOCK`. See **Remarks** below.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the operation is successful, or an ODBC Error Code or SQL Tools Error Code if it is not.

Remarks

This function works only with MultiRow Cursors, i.e. it will not work unless you have configured SQL Tools to return the results of a SQL statement in "blocks" that contain multiple rows, instead of one row at a time.

The `SQL_SetPos` function can be used to perform a number of different operations on a MultiRow cursor.

If you use an *lOperation&* value of `SET_POSITION`, the cursor is simply moved to *lRowNumber&* within the rowset. In other words, an *lRowNumber&* value of 1 would move the cursor to the first row of the current rowset, as retrieved by `SQL_Fetch`. (It would *not* move the cursor to the first row of the result set, unless the rowset "block"

started with the first row of the result set.)

If you use an *IOperation*& value of `SET_DELETE`, `SET_UPDATE`, or `SET_REFRESH`, the cursor is moved ("set") to row *IRowNumber*&, and that row is immediately deleted, updated, or refreshed.

`SET_DELETE` deletes data from the rowset buffers *and* the database. Whether or not the row still remains visible to `SQL_Fetch` operations depends on the type of cursor (static, dynamic, etc.) that is being used.

`SET_UPDATE` effectively moves data from the rowset buffers (which have presumably been modified by your program) into the database.

`SET_REFRESH` simply refreshes the data in the rowset buffers, in the event that your program has changed them and you want to abandon the changes. It does not re-fetch the data from the database. `SET_REFRESH` cannot be used to undo a `SET_DELETE` or `SET_UPDATE` operation.

Note that `SET_ADD` has been deprecated (i.e. it is not supported) in the ODBC 3.x specification and *should not be used*. You should use the `SQL_BulkOp(BULK_ADD)` function instead.

For more information about the various `SET` options, we suggest that you consult the Microsoft ODBC Software Developer Kit.

If your ODBC driver supports it, the *ILockType*& parameter can be used to specify how the row should be locked after the *IOperation*& is performed. To determine which types of locking are supported by a database, you can use the `SQL_DBInfoUInt(DB_type_CURSOR_ATTRIBUTES1)` function, where *type* is the type of cursor (dynamic, static, etc.) that is being used.

If the lock status should remain unchanged (or if your driver does not support locking), use `SQL_LOCK_NO_CHANGE`.

To lock or unlock a row, use `SQL_LOCK_EXCLUSIVE` or `SQL_LOCK_UNLOCK`, respectively.

A row that is locked with `SQL_LOCK_EXCLUSIVE` will remain locked until **1)** `SQL_LOCK_UNLOCK` is used on the row, or **2)** all of the statements that can access the rowset are closed, or **3)** the database is closed, or **4)** `SQL_EndTrans` is used to commit or roll back a database transaction.

Locking operations are not specific to one SQL statement or result set. In other words, one statement can use `SQL_LOCK_EXCLUSIVE` to lock a row, and another (concurrent) statement can use `SQL_LOCK_UNLOCK` to unlock it.

For more information about locking, we suggest that you consult the Microsoft ODBC Software Developer Kit.

For information about using Long data values with `SQL_SetPos`, see Using Long Values with Bulk and Positioned Operations.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL

Tools Error Messages.

Example

None.

Driver Issues

Some ODBC drivers do not support locking. Also, there are some minor differences in the ways that some drivers respond to the various `SET` options.

Speed Issues

None.

See Also

Bulk Operations
Positioned Operations

SQL_SetPosition

Summary

Sets the cursor position within a MultiRow Cursor, and optionally performs delete, update, refresh, and row-locking operations. *This function cannot be used to position a single-row cursor within a result set. For that, use `SQL_FetchPosition`.*

Twin

`SQL_SetPos`

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

See twin function `SQL_SetPos` for complete details.

Syntax

```
lResult& = SQL_SetPosition(lDatabaseNumber&, _  
                           lStatementNumber&, _  
                           lOperation&, _  
                           lRowNumber&, _  
                           lLockType&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function `SQL_SetPos` for complete details.

Return Values

See twin function `SQL_SetPos` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_SetPosition` is identical to `SQL_SetPos`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_SetPosition` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_SetPos` for complete details.

Example

None.

Driver Issues

See twin function `SQL_SetPos` for complete details.

Speed Issues

None.

See Also

Bulk Operations
Positioned Operations

SQL_SetStatementAttrib

Summary

Changes one attribute of a currently-open statement. (Compare this to the SQL_StmtMode function, which *pre*-sets certain statement attributes and should be used in most cases.)

Twin

SQL_SetStmtAttrib

Family

Statement Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_SetStatementAttrib(lDatabaseNumber&, _  
                                   lStatementNumber&, _  
                                   lAttribute&, _  
                                   dwValue???) 'or lValue&
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_SetStmtAttrib for complete details.

Return Values

See twin function SQL_SetStmtAttrib for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_SetStatementAttrib is identical to SQL_SetStmtAttrib. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_SetStatementAttrib is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_SetStmtAttrib for complete details.

Example

See twin function SQL_SetStmtAttrib.

Driver Issues

See twin function SQL_SetStmtAttrib for complete details.

Speed Issues

None.

See Also

Statement Information and Attributes

SQL_SetStmtAttrib

Summary

Changes one attribute of a currently-open statement. (Compare this to the SQL_StmtMode function, which *pre*-sets certain statement attributes and should be used in most cases.)

Twin

SQL_SetStatementAttrib

Family

Statement Info/Attrib Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_SetStmtAttrib(lAttribute&, _  
                                dwValue???)  
  
...Or...  
  
lResult& = SQL_SetStmtAttrib(lAttribute&, _  
                                lValue&)
```

Parameters

lAttribute&

One of the constants described in **Remarks** below.

dwValue??? or lValue&

A valid value for the specified *lAttribute&*.

Return Values

This function returns SUCCESS or SUCCESS_WITH_INFO if the attribute is changed, or an Error Code if it is not.

Remarks

IMPORTANT NOTE: It is usually best to use the SQL_StmtMode function to pre-set most of the statement attributes, instead of using SQL_SetStmtAttrib to set them "manually", after a statement has been opened or executed.

If you choose to use SQL_SetStmtAttrib instead of SQL_StmtMode, there are two different groups of STMT_ATTR constants that you can use:

1) All of the attributes and values that are described under SQL_StmtMode can also be set with SQL_SetStmtAttrib. The attribute values that these two functions share are identical, so To avoid errors when this document is updated, information that is common to both functions is not duplicated here. You should refer to the SQL_StmtMode entry of this document for a list of valid Statement Attributes and their values.

2) In addition to the SQL_StmtMode attributes, you can use the following constants with the SQL_SetStmtAttrib function. The functions are divided into two groups of related functions. If you set the first attribute in a group, you will usually need to

set others.

Multi-Row Cursors (six *related* attributes)

STMT_ATTR_ROW_ARRAY_SIZE

ODBC 3.x+ ONLY: This mode setting is used to specify the number of rows that will be returned by each `SQL_Fetch` or `SQL_FetchRel` operation. In other words, this attribute sets the number of rows in a multirow cursor, which is also known as a "block cursor" or a "row array". This attribute is sometimes called the "block size".

The default value is one (1), which indicates that only one row at a time will be retrieved by `SQL_Fetch`, i.e. a MultiRow Cursor is *not* being used. If you specify a value larger than 1 for this attribute, your program will be responsible for handling all aspects (including binding) of the MultiRow Cursor.

If you specify a value that is too large for the ODBC driver that you are using, an error message will be generated when the statement is opened and the driver will use the largest value that it can. (The value of this attribute will be changed automatically, so you can then use the `SQL_StmtAttrib(STMT_ATTR_ROW_ARRAY_SIZE)` function to find out the actual block size that the ODBC driver used.)

STMT_ATTR_ROWS_FETCHED_PTR

This attribute is a memory pointer which points to a *variable* into which the ODBC driver will place **1)** the total number of rows that are retrieved by each `SQL_SetPos(SET_REFRESH)` or multi-row-cursor `SQL_Fetch` operation, or **2)** the total number of rows that are affected by a `SQL_BulkOp` operation.

The value of the variable will include error rows, if any.

STMT_ATTR_ROW_BIND_OFFSET_PTR

ODBC 3.x+ ONLY: This attribute is a memory pointer which points to a *variable* which contains an offset value that is added to pointers, to change the binding of column data.

Bind offsets allow a program to change an existing result column binding without using the `SQL_ManualBindCol` function. Using `SQL_ManualBindCol` to rebind a column changes the buffer pointer and the Indicator pointer. Rebinding with an offset, on the other hand, simply adds an offset to the *existing* pointer values. It does *not* represent an offset from the *previous* offset.

STMT_ATTR_ROW_NUMBER

IMPORTANT NOTE: This is a READ-ONLY attribute, which can be read with `SQL_StmtAttrib` but cannot be set with `SQL_SetStmtAttrib`.

IMPORTANT NOTE: Some ODBC drivers support this attribute only when a *multi-row* cursor is being used.

The row number of the current row, in the context of entire result set. If the row number cannot be determined, or if there is no current row, this value will be zero (0).

STMT_ATTR_ROW_OPERATION_PTR

ODBC 3.x+ ONLY: This attribute is a memory pointer which points to an *array* of `BAS_WORD` values. The array is used to ignore one or more rows during the execution of a `SQL_SetPos` operation. Each element of the array is set to either zero (0) if the corresponding row is to be executed, or one (1) if the row is to be ignored.

The array must have as many elements as there are rows in the rowset, i.e. the value of the `STMT_ATTR_BLOCK_SIZE` attribute.

STMT_ATTR_ROW_STATUS_PTR

ODBC 3.x+ ONLY: This attribute is a memory pointer which points to an *array* of `BAS_WORD` values. After a `SQL_Fetch` or `SQL_FetchRel` operation, the array will contain row status values.

The array must have as many elements as there are rows in the rowset, i.e. the value of the `STMT_ATTR_BLOCK_SIZE` attribute.

Bound SQL Statement Parameter Arrays (six *related* attributes)

STMT_ATTR_PARAMSET_SIZE

ODBC 3.x+ ONLY: This attribute specifies the number of elements that each bound-parameter array has. (See Bound Parameters for more information.)

The default value for this attribute is zero (0), which means that bound parameter *arrays* are not being used. (It does not mean that bound *parameters* are not being used.)

If this attribute has a value greater than 1, your program is responsible for creating and maintaining an array of values for each bound parameter in a SQL statement.

STMT_ATTR_PARAMS_PROCESSED_PTR

ODBC 3.x+ ONLY: This attribute is a memory pointer which points to a *variable* in which the ODBC driver will return the number of sets of parameters that have been processed, including error sets. In other words, if you set this attribute to a `VarPtr` value which points to a `BAS_LONG` variable, then when the `SQL_Stmt (SQL_STMT_IMMEDIATE)` or `SQL_Stmt (SQL_STMT_EXECUTE)` function is used, the ODBC driver will set the value of the *variable* to indicate the number of parameters that were processed. (If the `SQL_Stmt` function returns an error, this value should not be trusted.)

STMT_ATTR_PARAM_BIND_OFFSET_PTR

ODBC 3.x+ ONLY: This attribute is a memory pointer which points to a *variable* which contains an offset value that is added to pointers, to change the binding of parameters.

Bind offsets allow a program to *change* an existing parameter binding without using the `SQL_BindParam` function. Using `SQL_BindParam` to rebind a parameter changes the buffer pointer and Indicator pointer to new values. Rebinding with an offset, on the other hand, simply adds an offset to the *existing* pointers. A new offset can be specified at any time by changing the value of the variable (not of this attribute). **IMPORTANT NOTE:** The new offset is always added to the *original* pointer values. It does *not* represent an offset from the *previous* offset.

STMT_ATTR_PARAM_BIND_TYPE

ODBC 3.x+ ONLY: This attribute contains a value that indicates the "bind type" that is to be used for bound parameters. The default value is `COLUMN_WISE`.

To select row-wise parameter binding, this attribute is set to the length of the structure that will be bound to a set of dynamic parameters. We recommend that you consult the Microsoft ODBC Software Developer Kit for more information about row-wise parameter binding.

STMT_ATTR_PARAM_OPERATION_PTR

ODBC 3.x+ ONLY: This attribute is a memory pointer which points to an *array* of `BAS_WORD` or `BAS_LONG` values. The array is used to ignore one or more rows of parameters during the execution of a SQL statement. Each element of the array is set to either zero (0) if the corresponding row of parameters is to be executed, or one (1) if the row of parameter is to be ignored.

The array must have a number of elements equal to the `STMT_ATTR_PARAMSET_SIZE` attribute.

STMT_ATTR_PARAM_STATUS_PTR

ODBC 3.x+ ONLY: This attribute is a memory pointer which points to an *array* of `BAS_WORD` values. After a `SQL_Stmt(SQL_STMT_IMMEDIATE)` or `SQL_Stmt(SQL_STMT_EXECUTE)` operation, the array will contain status information about each row of parameter values.

This attribute must be set if (and only if) `STMT_ATTR_PARAMSET_SIZE` (see above) is greater than 1. The array must have a number of elements equal to the `STMT_ATTR_PARAMSET_SIZE` attribute.

Each of the elements of the array will contain one of the following values. You should note that the numeric values of these constants *do not* correspond to the normal SQL Tools Error Code values, so they are not interchangeable. For example, `SQL_PARAM_SUCCESS_WITH_INFO` has a value of 6, and `SUCCESS_WITH_INFO` has a value of 1, so you must be careful to use the *only* following constants when dealing with a status array:

SQL_PARAM_SUCCESS (The SQL statement was successfully executed for this set of parameters.)

SQL_PARAM_SUCCESS_WITH_INFO (The SQL statement was successfully executed for this set of parameters, however an Error Message was generated)

SQL_PARAM_ERROR (There was an error in processing this set of parameters. Additional error information is provided by an Error Message.)

SQL_PARAM_UNUSED (This parameter set was unused, possibly because a previous parameter set causing an error that aborted further processing, or because the parameter was ignored (see STMT_ATTR_PARAM_OPERATION_PTR above).

SQL_PARAM_DIAG_UNAVAILABLE (The driver does not provide parameter status information.)

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

See SQL_StmtMode.

Speed Issues

None.

See Also

Statement Information and Attributes

SQL_Shutdown

Summary

Closes all open statements and databases, and shuts down SQL Tools.

Twin

None.

Family

Configuration Family

Availability

Standard and Pro

Warning

Your program must use this function to properly shut down SQL Tools when your program is finished using SQL Tools functions. Failure to do so can result in a number of different problems, including Application Errors. See Four Critical Steps For Every SQL Tools Program for more information.

Syntax

```
lResult& = SQL_Shutdown
```

Parameters

None.

Return Values

This function returns `SUCCESS` if it is able to perform the final shutdown step (the freeing of the ODBC environment handle), or an Error Code if it is not able to free the handle.

Remarks

See Four Critical Steps For Every SQL Tools Program for more information about this function.

Diagnostics

If this function fails to shut down SQL Tools properly, please contact Perfect Sync Technical Support (support@perfectsync.com) with information about your program.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Configuration Family

SQL_State

Summary

Provides the SQL State value that is associated with the oldest error message in the SQL Tools Error Stack.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_State
```

Parameters

None.

Return Values

This function will return an empty string if there are no error messages in the SQL Tools Error Stack. Otherwise, it will return a five-character string that represents the SQL State value that was provided by the program (the ODBC driver, SQL Tools, etc.) which generated the Error Message.

Remarks

See ODBC Error Messages for more information about SQL States, including a partial list of the values that this function can return.

Diagnostics

None.

Example

```
'Display the SQLState of the oldest error  
'in the SQL Tools Error Stack...  
Print SQL_State
```

Driver Issues

Many SQL State values are driver-specific. In other words, a certain error condition may cause a given ODBC driver to generate a SQL State value, and a different driver may generate a different SQL State value.

Speed Issues

None.

See Also

Error Handling in SQL Tools Programs

SQL_Statement

Summary

Prepares and/or executes a SQL statement.

Twin

SQL_Stmt

Family

Statement Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_Statement(lDatabaseNumber&, _  
                        lStatementNumber&, _  
                        lAction&, _  
                        sStatement$)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Database Numbers and Statement Numbers for a detailed discussion.

All Other Parameters

See twin function *SQL_Stmt* for complete details.

Return Values

See twin function *SQL_Stmt* for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, *SQL_Statement* is identical to *SQL_Stmt*. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to *SQL_Statement* is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function *SQL_Stmt* for complete details.

Example

See twin function *SQL_Stmt*.

Driver Issues

None.

Speed Issues

See twin function *SQL_Stmt* for complete details.

See Also: Execution of SQL Statements

SQL_StatementAttrib

Summary

Provides the current value of a statement attribute.

Twin

SQL_StmtAttrib

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_StatementAttrib(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lAttribute&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_StmtAttrib for complete details.

Return Values

See twin function SQL_StmtAttrib for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_StatementAttrib is identical to SQL_StmtAttrib. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_StatementAttrib is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_StmtAttrib for complete details.

Example

See twin function SQL_StmtAttrib.

Driver Issues: See twin function SQL_StmtAttrib for complete details.

Speed Issues: See twin function SQL_StmtAttrib.

See Also

Statement Information and Attributes

SQL_StatementCancel

Summary

Cancels the execution of a SQL statement that is running asynchronously, running in another thread, or a Bulk Operation or Positioned Operation. that has not finished executing.

Twin

SQL_StmtCancel

Family

Statement Family

Availability

Standard and Pro

Warning

See twin function `SQL_StmtCancel` for complete details.

Syntax

```
lResult& = SQL_StatementCancel(lDatabaseNumber&, _  
                               lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

See twin function `SQL_StmtCancel` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_StatementCancel` is identical to `SQL_StmtCancel`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_StatementCancel` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_StmtCancel` for complete details.

Example

See twin function `SQL_StmtCancel`.

Driver Issues

See twin function `SQL_StmtCancel` for complete details.

Speed Issues

None.

See Also: Bulk Operation, Positioned Operation

SQL_StatementInfoStr

Summary

Provides information about a SQL statement, in string form.

Twin

SQL_StmtInfoStr

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_StatementInfoStr(lDatabaseNumber&, _  
                                lStatementNumber&, _  
                                lInfoType&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_StmtInfoStr for complete details.

Return Values

See twin function SQL_StmtInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_StatementInfoStr is identical to SQL_StmtInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_StatementInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_StmtInfoStr for complete details.

Example

See twin function SQL_StmtInfoStr.

Driver Issues: See twin function SQL_StmtInfoStr for complete details.

Speed Issues: None.

See Also

Statement Information and Attributes

SQL_StatementIsOpen

Summary

Indicates whether or not a SQL Statement is open.

Twin

SQL_StmtIsOpen

Family

Statement Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_StatementIsOpen(lDatabaseNumber&, _  
                                lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

See twin function `SQL_StmtIsOpen` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_StatementIsOpen` is identical to `SQL_StmtIsOpen`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_StatementIsOpen` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_StmtIsOpen` for complete details.

Example

See twin function `SQL_StmtIsOpen`.

Driver Issues

None.

Speed Issues

None.

See Also

Manually Opening and Closing Statements

SQL_StatementMode

Summary

Pre-sets the value of a SQL statement attribute, so that the specified attribute will be used in the future.

Twin

SQL_StmtMode

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_StatementMode(lDatabaseNumber&, _  
                             lStatementNumber&, _  
                             lMode&, _  
                             dwValue???) 'or lValue&
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_StmtMode for complete details.

Return Values

See twin function SQL_StmtMode for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_StatementMode is identical to SQL_StmtMode. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_StatementMode is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_StmtMode for complete details.

Example

See twin function SQL_StmtMode.

Driver Issues: See twin function SQL_StmtMode for complete details.

Speed Issues: See twin function SQL_StmtMode for complete details.

See Also

SQL Statement Modes

SQL_StatementNativeSyntax

Summary

Translates a SQL statement into the syntax that the ODBC driver will actually use if the statement is prepared or executed with `SQL_Stmt`.

Twin

`SQL_StmtNativeSyntax`

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

This function does not actually prepare or execute SQL statements. It is primarily a diagnostic tool.

Syntax

```
lResult& = SQL_StatementNativeSyntax(lDatabaseNumber&, _  
                                     sStatement$)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function `SQL_StmtNativeSyntax` for complete details.

Return Values

See twin function `SQL_StmtNativeSyntax` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_StatementNativeSyntax` is identical to `SQL_StmtNativeSyntax`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_StatementNativeSyntax` is shown below.

For more information about using *lDatabaseNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_StmtNativeSyntax` for complete details.

Example

See twin function `SQL_StmtNativeSyntax` for complete details.

Driver Issues: None.

Speed Issues: None.

See Also

Execution of SQL Statements

SQL_StatementParameterCount

Summary

Indicates how many bound parameters a SQL statement has.

Twin

SQL_StmtParamCount

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_StatementParameterCount(lDatabaseNumber&, _  
                                       lStatementNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

Return Values

See twin function `SQL_StmtParamCount` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_StatementParameterCount` is identical to `SQL_StmtParamCount`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_StatementParameterCount` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_StmtParamCount` for complete details.

Example

See twin function `SQL_StmtParamCount`.

Driver Issues

None.

Speed Issues

None.

See Also

Binding Statement Input Parameters

SQL_Stmt

Summary

Prepares and/or executes a SQL statement.

Twin

SQL_Statement

Family

Statement Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_Stmt(lAction&, _  
                    sStatement$)
```

Parameters

lAction&

One of the following constants: `SQL_STMT_PREPARE`, `SQL_STMT_EXECUTE`, or `SQL_STMT_IMMEDIATE`. (A number of aliases for these values are also recognized.) See **Remarks** below.

sStatement\$

The SQL statement to be prepared and/or executed. The exact syntax that you use will depend on the capabilities of the ODBC driver that your program uses. For a summary of the basic syntax that is recognized by all ODBC-compliant drivers, see Appendix A: SQL Statement Syntax.

Return Values

If the SQL statement is prepared and/or executed without errors, the return value of this function will be `SUCCESS` or `SUCCESS_WITH_INFO`.

Please note that "without errors" does *not* mean "the way you expect". As with all programming languages, SQL is very literal. A return value of `SUCCESS` indicates that the ODBC driver did precisely *what you asked it to do*.

If the preparation or execution is not successful, an ODBC Error Code or SQL Tools Error Code will be returned.

Remarks

The processing of most SQL statements is basically an "interpreted" operation. The ODBC driver must first analyze the string that contains the SQL statement and then "compile" the statement into an executable form. This step is called "preparation" and is roughly equivalent to the steps that are taken by a BASIC interpreter like Microsoft QBASIC to convert source code into executable code at run time. The actual "execution" of a SQL statement is a separate process.

`SQL_STMT_PREPARE` tells the `SQL_Stmt` function to prepare the SQL statement in *sStatement\$* but not to execute it. The alias `PREP` is also recognized.

`SQL_STMT_EXECUTE` tells the `SQL_Stmt` function to execute a SQL statement that

was previously prepared. The alias `EXEC` is also recognized.

`SQL_STMT_IMMEDIATE` tells the `SQL_Stmt` function to prepare *and* execute a SQL statement, as if it was a one-step process. The alias `IMMED` is also recognized, as are `SQL_STMT_EXECDIRECT` and `DIRECT`, which are based on the original ODBC terminology. (Be careful not to confuse `EXEC` and `SQL_STMT_EXECDIRECT`, which are *not* the same thing.)

Most programs will use `SQL_STMT_IMMEDIATE` most of the time.

The major advantage of using `SQL_STMT_PREPARE` and `SQL_STMT_EXECUTE` as separate steps is that it allows statement parameters to be bound to the SQL statement between the two steps. A SQL statement can be prepared once, bound to one or more parameter variables, and then executed many times with different parameter values. If a SQL statement is to be executed repeatedly with different parameter values it is much more efficient to use this procedure than to use `SQL_STMT_IMMEDIATE` to prepare and execute the statement over and over.

Databases can also contain pre-prepared SQL statements called Stored Procedures. They are stored in the database in compiled form. Creating Stored Procedures can be a complex process, but they are the fastest, most efficient way to execute most SQL statements because the process of preparing the statement is performed *before* runtime.

If you use the `SQL_STMT_EXECUTE` or `SQL_STMT_IMMEDIATE` option with a [SELECT](#) statement, SQL Tools will automatically bind all of the columns in the SQL statement's result set, so that your program can access the resulting data. (See Result Column Binding for more information.)

If you use the `SQL_STMT_PREPARE` or `SQL_STMT_IMMEDIATE` option, the `sStatement$` parameter must contain a valid SQL statement.

If you use the `SQL_STMT_EXECUTE` option, the `sStatement$` string is optional. If you use an empty string for `sStatement$`, SQL Tools will assume that you mean "execute the statement that was just prepared". If you have not previously prepared a statement, an error (`ERROR_STMT_NOT_PREPARED`) will be generated. If you do pass a `sStatement$` string to the `SQL_Stmt` function when the `SQL_STMT_EXECUTE` option is used, SQL Tools will check to make sure that it is *exactly* the same statement string that was previously prepared. If you are writing complex programs with many different statements that can be prepared and executed, this can be a valuable double-check that makes sure that your program is executing the statement that you think it is. If the strings do not match, an error (`ERROR_BAD_PARAM_VALUE`) will be generated.

If you attempt to use the `SQL_Stmt` function before you have used `SQL_OpenDB` to open a database, and if the SQL Tools Database AutoOpen feature has not been disabled, the `SQL_Stmt` function will automatically call the `SQL_OpenDB` function for you. An empty string will be used for the `sConnectionString$` parameter, to allow the user to specify a database. This is rarely necessary, however, since most SQL statements only have meaning in the context of a database connection. In other words, you are unlikely to need to execute a SQL statement like [SELECT * FROM MYTABLE](#) unless your program has already opened a database that contains a table called [MYTABLE](#). The auto-open feature is primarily provided as a programming convenience, for those times that you are writing quick-and-dirty test programs.

If you attempt to use the `SQL_Stmt` function before a statement from a previous `SQL_Stmt` function has been closed (with `SQL_CloseStmt`), and if you have not disabled the SQL Tools Statement AutoClose feature, SQL Tools will automatically close the previous SQL statement for you. **WARNING:** If you are *not* operating in the default AutoCommit mode, and if you have also *not* used the `SQL_EndTrans` function to explicitly commit or roll back a transaction, the auto-closing of a statement will result in an abandoned transaction.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
lResult& = SQL_Stmt(IMMED, "SELECT * FROM MYTABLE")
```

Driver Issues

None.

Speed Issues

If a statement is to be executed repeatedly with different parameter values, it is best to `SQL_STMT_PREPARE` the statement, bind the parameters to variables, then `SQL_STMT_EXECUTE` the statement repeatedly, changing only the parameters.

Stored Procedures are usually the fastest way to execute a SQL statement.

See Also

Execution of SQL Statements

SQL_StmtAttrib

Summary

Provides the current value of a statement attribute.

Twin

SQL_StatementAttrib

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_StmtAttrib(lAttribute&)
```

Parameters

lAttribute&

A constant that represents a statement attribute. See **Remarks** below.

Return Values

If a valid *lAttribute&* value is used, this function will return the value of the attribute. Otherwise, it will return zero (0)

Remarks

This function can be used to determine the current setting of a statement attribute.

Statement attributes can be *set* with the **SQL_SetStmtAttrib** and **SQL_StmtMode** functions. The *lAttribute&* values that are used by all of these functions are identical. To avoid errors when this document is updated, information that is common to all functions is not duplicated here. Only information that is unique to **SQL_StmtAttrib** is shown below.

For a list of *lAttribute&* values, see **SQL_StmtMode** and **SQL_SetStmtAttrib**.

Diagnostics

This function does not return Error Codes, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

None.

See Also

Statement Information and Attributes

SQL_StmtCancel

Summary

Cancels the execution of a SQL statement that is running asynchronously, running in another thread, or a Bulk Operation or Positioned Operation. that has not finished executing.

Twin

SQL_StatementCancel

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

Depending on the ODBC driver, the use of the `SQL_StmtCancel` function will not *necessarily* stop the processing of a SQL statement. The return value of the `SQL_StmtCancel` function simply indicates whether or not the driver acknowledged the cancellation request. If `SQL_StmtCancel` is used to cancel a statement that is being executed asynchronously or in another thread, it is possible for the execution to succeed and return `SUCCESS`, while the cancellation is also considered to be successful. In any event, once `SQL_StmtCancel` has been used **1)** the thread that originated the statement must continue to wait for the `SQL_Stmt` function to exit, and **2)** you should not attempt to access the results of the affected SQL statement.

Syntax

```
lResult& = SQL_StmtCancel
```

Parameters

None.

Return Values

This function returns `SUCCESS` or `SUCCESS_WITH_INFO` if the ODBC driver acknowledges the request, or an ODBC Error Code if it does not. This function can also return SQL Tools Error Codes.

Remarks

Once the `SQL_Stmt` function has been used to execute a SQL statement, your program "pauses" until the execution is complete. Depending on the size of the database and the complexity/size of the result set, this can cause your program to appear to be locked up for an extended period of time.

To solve this problem you can use asynchronous execution or multiple threads. A SQL statement can be executed in one thread, and a second thread can be used to display a "please wait" message with a time display, check for a timeout condition, check for a "user cancel" signal, check for Windows Message Loop (GUI) activity, and many other things.

If one thread detects a timeout condition or a user-cancel signal, it can use the `SQL_StmtCancel` function to cancel the SQL statement that is running in the other thread.

See **Warning** above.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

See **Warning** above.

Speed Issues

None.

See Also

Execution of SQL Statements

SQL_StmtInfoStr

Summary

Provides information about a SQL statement. (Generally speaking, "information" is a value that cannot be changed. "Attributes" are values that can be changed by your program.)

Twin

SQL_StatementInfoStr

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_StmtInfoStr(lInfoType&)
```

Parameters

lInfoType&

One of the constants described in **Remarks** below.

Return Values

If an invalid *lInfoType&* is used, or if a statement is not open, this function will return an empty string. Otherwise, it will return the requested information in string form.

Remarks

The *lInfoType&* parameter must have one of the following values:

STMT_SUBMITTED

The most recent *sStatement\$* value that was submitted to the `SQL_Stmt` function.

STMT_TRANSLATED

The "Native Syntax" version of the most recent *sStatement\$* value that was submitted to the `SQL_Stmt` function. In other words, the *actual* syntax that was executed by the ODBC driver, based on the SQL statement that you submitted. See `SQL_StmtNativeSyntax` for more information.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Visually compare the submitted and  
'translated SQL statements...  
Print SQL_StmtInfoStr(STMT_SUBMITTED)  
Print SQL_StmtInfoStr(STMT_TRANSLATED)
```

Driver Issues

None.

Speed Issues

None.

See Also

Statement Information and Attributes

SQL_StmtIsOpen

Summary

Indicates whether or not a SQL statement is open.

Twin

SQL_StatementIsOpen

Family

Statement Open/Close Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_StmtIsOpen
```

Parameters

None.

Return Values

This function will return Logical True (-1) if the statement is open, or False (zero) if it is not.

Remarks

This function can be used to determine whether or not the current statement number (the default value of 1, or the statement number specified with `SQL_UseStmt`) is currently open.

Since it returns a Logical True/False value, you can use either syntax that is shown here...

```
If SQL_StmtIsOpen Then
```

...or...

```
If NOT SQL_StmtIsOpen Then
```

Diagnostics

This function does not return Error Codes, but it can return SQL Tools Error Messages.

Example

```
If SQL_StmtIsOpen Then  
    'the current statement is open  
End If
```

Driver Issues

None.

Speed Issues

None.

See Also

Manually Opening and Closing Statements

SQL_StmtMode

Summary

Pre-sets a statement attribute value, for future use by the `SQL_Stmt` or `SQL_OpenStmt` function. (Attributes can also be set after a statement is open, by using the `SQL_SetStmtAttrib` function, but the `SQL_StmtMode` function should normally be used instead of `SQL_SetStmtAttrib`.)

Twin

`SQL_StatementMode`

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_StmtMode(lMode&, _  
                        dwValue???)
```

...Or...

```
lResult& = SQL_StmtMode(lMode&, _  
                        lValue&)
```

Parameters

lMode&

One of the constants described in **Remarks**, below.

dwValue??? or lValue&

A valid value for the specified *lMode&*.

Return Values

This function can return `ERROR_ADVISORY` to warn you that you have used this function while a statement is open (see below).

It will return `ERROR_BAD_PARAM_VALUE` if you specify an invalid *lMode&* value.

Otherwise it will return `SUCCESS`, regardless of whether or not the value of *dwValue??? or lValue&* is valid. (This is because the valid values for each mode are ODBC driver-dependent, so errors cannot be detected until a statement is actually opened *using* a value. Some errors may not be detected until a statement is executed.)

Remarks

This function is used to *pre-set* the attribute values that will be used the next time the `SQL_Stmt` or `SQL_OpenStmt` function is used. That is a very important distinction: *this function cannot be used to change the attributes of a currently-open statement*. If you use this function while a statement is open, the mode value *will* be changed for *future* use by `SQL_Stmt` and `SQL_OpenStmt`, and an `ERROR_ADVISORY` message will be generated to remind you that the new setting will *not* affect the currently-open statement. If you have already executed one or more SQL statements and need to change the statement mode for future statements, you should use `SQL_CloseStmt`

to make sure that the statement is closed before you use this function to change the mode. Otherwise you will receive the `ERROR_ADVISORY` described above.

The `Mode` parameter must be one of the following values:

`STMT_ATTR_CONCURRENCY`

This mode can be set to any one of the following values, as long as the value is supported by your ODBC driver. If the specified value is not supported, the ODBC driver will substitute a different value and an ODBC Error Message will be generated. (Older versions of ODBC provided a method of determining the level of concurrency that is supported, but it has been deprecated in ODBC 3.x.)

`CONC_READONLY` (The cursor is read-only. If you attempt to use a SQL statement to modify a database when `STMT_CONCURRENCY` is set to `SQL_CONCUR_READ_ONLY`, an ODBC Error Message will be generated when the statement is executed.)

`CONC_LOCK` (The cursor will use the lowest level of locking that is sufficient to *ensure* that the row can be updated. This option is not supported by all ODBC drivers.)

The remaining two options use "optimistic concurrency control", which are usually reliable but do not *ensure* that a row can always be updated. If a row is not updated properly, an ODBC Error Message will be generated and your program should try again. If your program is update-intensive, and if multiple applications and/or statements will be accessing the database at the same time, you should probably try to use `SQL_CONCUR_LOCK` to improve reliability.

`CONC_ROWVER` (The cursor will use optimistic concurrency control, comparing "row versions" such as SQLBase ROWID or Sybase TIMESTAMP. This option is not supported by many ODBC drivers.)

`CONC_VALUES` (The cursor will use optimistic concurrency control, comparing values. This is the SQL Tools default value, because it allows updates and is supported by almost all ODBC drivers.)

Please note the following interactions between this mode setting and `STMT_CURSOR_TYPE` (which is described in its own section, below):

If you specify a value for `STMT_CONCURRENCY` that does not support the current value of `STMT_CURSOR_TYPE`, the value of `STMT_CURSOR_TYPE` will be changed by the ODBC driver.

If you specify a value for `STMT_CURSOR_TYPE` that does not support the current value of `STMT_CONCURRENCY`, the value of `STMT_CONCURRENCY` will be changed by the ODBC driver.

`STMT_ATTR_CURSOR_SCROLLABLE`

IMPORTANT NOTE: This attribute should not be set if the ODBC Cursor Library is being used. SQL Tools uses the ODBC Cursor Library by default, so unless your program uses `SQL_OpenDatabase1` and

SQL_OpenDatabase2 to bypass this default, you should not attempt to set this attribute. Use `STMT_ATTR_CURSOR_TYPE` (below) and/or `STMT_ATTR_CONCURRENCY` (above) instead.

ODBC 3.x+ ONLY: This mode can be set to one of the following values:

`SCRL_OFF` (Scrollable cursors are not required. If you use `SQL_Fetch`, the only valid value for the *lRowNumber*& parameter is `NEXT_ROW`.)

`SCRL_ON` (Scrollable cursors are required.)

NOTE: If the ODBC Cursor Library is being used (which is the SQL Tools default mode) it is not usually necessary to change this attribute.

`STMT_ATTR_CURSOR_SENSITIVITY`

IMPORTANT NOTE: This attribute should not be set if the ODBC Cursor Library is being used. SQL Tools uses the ODBC Cursor Library by default, so unless your program uses `SQL_OpenDatabase1` and `SQL_OpenDatabase2` to bypass this default, you should not attempt to set this attribute. Use `STMT_ATTR_CURSOR_TYPE` (below) and/or `STMT_ATTR_CONCURRENCY` (above) instead.

ODBC 3.x+ ONLY: Specifies whether or not the statement's cursor "sees" the changes that are made to a result set by another cursor.

`SENS_NONE` (It is unspecified what the cursor type is and whether or not the cursor sees the changes that are made to a result set by another cursor. Cursors may see none, some, or all such changes. This is the default setting.)

`SENS_INSENSITIVE` (The cursor does not see any changes that are made by other cursors. Insensitive cursors are read-only. This corresponds to a static cursor, which has a concurrency that is read-only.)

`SENS_SENSITIVE` (The cursor sees all changes made to a result set by other cursors.)

`STMT_ATTR_CURSOR_TYPE`

`CUR_FORWARDONLY` (The cursor can only scroll forward.)

`CUR_STATIC` (The data in the result set is static. This is the SQL Tools default value.)

`CUR_KEYSET` (The driver saves and uses the keys for the number of rows specified in the `STMT_ATTR_KEYSET_SIZE` setting (see below).)

`CUR_DYNAMIC` (The driver only saves and uses the keys for the rows in the rowset.)

Please note the following interactions between this mode setting and `STMT_CONCURRENCY` (which is described in its own section, above):

If you specify a value for `STMT_CURSOR_TYPE` that does not support the current value of `STMT_CONCURRENCY`, the value of `STMT_CONCURRENCY` will be changed by the ODBC driver.

If you specify a value for `STMT_CONCURRENCY` that does not support the current value of `STMT_CURSOR_TYPE`, the value of `STMT_CURSOR_TYPE` will be changed by the ODBC driver.

`STMT_ATTR_KEYSET_SIZE`

Specifies the number of rows in the keyset for a keyset-driven cursor. (See `STMT_CURSOR_TYPE` just above.)

If the value of `STMT_CURSOR_TYPE` (see above) is `SQL_CURSOR_KEYSET_DRIVEN`, and if the keyset size is `KEYSET_FULL` (zero, the default), the cursor is fully keyset-driven.

If the keyset size is greater than 0, the cursor is keyset-driven within the keyset and dynamic outside of the keyset. This is called a "mixed" cursor.

`STMT_ATTR_MAX_COLUMN_LENGTH`

Specifies the maximum amount of data that the driver will return from a character string) or binary column. **IMPORTANT NOTE:** This setting is intended to reduce network traffic and should be used only when the Datasource (as opposed to the driver) in a multiple-tier driver can implement it. *This setting should not be used as a way to truncate data.*

If the value of this setting is zero (0, the default), the driver will attempt to return all of the available data. If the length of the available data is greater than the length of the memory buffer that is supplied by SQL Tools, the `SQL_Fetch` and `SQL_FetchRel` functions will truncate the data and `SUCCESS_WITH_INFO` will be returned, along with an ODBC Error Message indicating that the data was truncated.

If the value of this setting is changed to a nonzero value, and if that value is less than the length of the available data in a column, `SQL_Fetch` and `SQL_FetchRel` will truncate the data and return `SUCCESS`.

If the value of this setting is **1**) less than the minimum amount of data that the Datasource can return, or **2**) greater than the maximum amount of data that the Datasource can return, the driver will substitute a value that it can handle, and an ODBC Error Message will be generated when the statement is opened.

This setting can also affect the `SQL_LongResCol` function, depending on the behavior of the ODBC driver.

`STMT_ATTR_MAX_RESULT_ROWS`

The maximum number of rows that the ODBC driver should return for a [*SELECT*](#) statement.

The default value is zero (0), which tells the driver to return *all* rows.

This setting is intended to reduce network traffic. If the number of rows in the result set is greater than this setting's value, the result set will be truncated.

STMT_ATTR_QUERY_TIMEOUT

The number of seconds that the driver should wait for a SQL statement to execute before returning to your program.

The default value is zero (0), which means "no timeout", i.e. "wait forever".

STMT_ATTR_RETRIEVE_DATA

This setting can be used to tell the `SQL_Fetch` and `SQL_FetchRel` functions to not actually retrieve any data. It can be used when all you need to do is confirm that a row exists, and you don't care what the row contains.

You can use either `RD_SEEKONLY` or `RD_NORMAL` (the default) for this setting.

STMT_ATTR_ROW_BIND_TYPE

This mode setting determines whether or not Row-wise binding will be used.

The default value is `COLUMN_WISE` (zero). If you specify a positive integer value for this mode, it represents the number of bytes that will be used for the "row bind buffer". If you use row-wise binding, your program is responsible for managing all aspects (including binding) of the row buffer.

STMT_ATTR_SCANFORESCAPES

You can use either `DO_SCAN` (the default) or `DONT_SCAN` for this setting, which tells the ODBC driver whether or not it should scan SQL statements for escape sequences.

STMT_ATTR_SIMULATE_CURSOR

Specifies whether or not ODBC drivers which simulate positioned update and delete statements guarantee that those statements will affect only one row.

To simulate positioned update and delete statements, most ODBC drivers construct an `UPDATE` or `DELETE` statement that contains a `WHERE` clause which specifies the value of each column in the current row. Unless these columns make up a unique key, the constructed statement may affect more than one row. To guarantee that such statements will affect only one row, the driver determines the columns in a unique key and adds these columns to the result set.

If *your program* guarantees that the columns in the result set make up a unique key, the driver is not required to do so, so changing the value of this attribute may reduce execution time.

You may use any one of the following values, as long as your ODBC driver supports the value:

`SIMC_NONUNIQUE` (The driver does not guarantee that simulated positioned

update or delete statements will affect only one row. It is your program's responsibility to do so. If a statement affects more than one row, an ODBC Error Message will be generated.)

`SIMC_TRYUNIQUE` (The driver attempts to guarantee that simulated positioned update or delete statements affect only one row. The driver always executes such statements, even if they might affect more than one row, such as when there is no unique key. If a statement affects more than one row, an ODBC Error Message will be generated.)

`SIMC_UNIQUE` (The driver guarantees that simulated positioned update or delete statements affect only one row. If the driver cannot guarantee this for a given statement, an ODBC Error Message will be generated.)

If the Datasource provides native support for positioned update and delete statements, and the driver does not simulate cursors, `SUCCESS` is returned when `SQL_SC_UNIQUE` is specified.

A `SUCCESS_WITH_INFO` message is usually generated if `SQL_SC_TRY_UNIQUE` or `SQL_SC_NON_UNIQUE` is requested.

If the Datasource provides the `SQL_SC_TRY_UNIQUE` level of support, and the driver does not, `SUCCESS` is returned for `SQL_SC_TRY_UNIQUE` and `SUCCESS_WITH_INFO` is returned for `SQL_SC_NON_UNIQUE`.

If the specified cursor simulation type is not supported by the Datasource, the driver will substitute a value that it can handle and an ODBC Error Message will be generated.

`STMT_ATTR_USE_BOOKMARKS`

Specifies whether or not a program will use bookmarks with a cursor:

`BOOKMARKS_OFF` (The default value. Bookmarks are not used.)

`BOOKMARKS_ON` (For ODBC 2.0 applications *only*. See bookmarks for more information.)

`BOOKMARKS_VARIABLE` (A program will use bookmarks with a cursor, and the driver will provide variable-length bookmarks if they are supported.)

Please note that `BOOKMARKS_ON` (also known as `SQL_UB_FIXED`) is deprecated in ODBC 3.x. All programs should always use variable-length bookmarks, even when working with ODBC 2.x drivers.

Diagnostics

This function returns Error Codes and can generate SQL Tools Error Messages, but it does not generate ODBC Error Messages. If you specify an invalid mode value, however, it is very likely that an ODBC Error Message will be generated when a statement is opened or executed.

Example

See SQL Statement Modes for examples.

Driver Issues

See individual comments about each mode setting, in **Remarks** above.

Speed Issues

None.

See Also

Statement Information and Attributes

SQL_StmtNativeSyntax

Summary

Translates a SQL statement into the syntax that the ODBC driver will actually use if the statement is prepared or executed with `SQL_Stmt`.

Twin

`SQL_StatementNativeSyntax`

Family

Statement Info/Attrib Family

Availability

Standard and Pro

Warning

This function does not actually prepare or execute SQL statements. It is primarily a diagnostic tool.

Syntax

```
sResult$ = SQL_StmtNativeSyntax(sStatement$)
```

Parameters

sStatement\$
A string that contains a SQL statement.

Return Values

sResult\$ will be the ODBC driver's translation of the SQL statement.

Remarks

Different databases and ODBC drivers can actually implement and execute SQL statements somewhat differently. In addition to minor differences in delimiters, different databases may make other changes in SQL syntax in order to optimize the execution of a statement, or to implement otherwise-unsupported syntax.

Diagnostics

This function does not return Error Codes, but it can generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Different ODBC drivers will interpret  
'the CONVERT function differently...
```

```
sStmt$ = "SELECT { fn CONVERT (ZIPCODE, SQL_INTEGER) } FROM  
ADDRESSBOOK"
```

```
PRINT SQL_StmtNativeSyntax(sStmt$)
```

If the Oracle ODBC Driver was being used, that code might produce...

```
SELECT to_number (ZIPCODE) FROM ADDRESSBOOK
```

If the Microsoft SQL Server ODBC driver was being used...

```
SELECT convert (integer, ZIPCODE) FROM ADDRESSBOOK
```

And if Microsoft Access was being used...

```
SELECT { fn CONVERT (ZIPCODE, SQL_INTEGER) } FROM ADDRESSBOOK
```

Different databases and ODBC drivers -- and even different versions of the same driver -- may produce different results. Note that Microsoft Access returned exactly the same string that was submitted.

Driver Issues

None.

Speed Issues

None.

See Also

SQL Statements

SQL_StmtParamCount

Summary

Indicates how many bound parameters a prepared SQL statement has.

Twin

SQL_StatementParameterCount

Family

Statement Binding Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_StmtParamCount
```

Parameters

None.

Return Values

If a SQL statement has not been prepared, this function will return zero. Otherwise, it will return a number that indicates how many bound parameters the statement has. (This number can also be zero, or a positive integer value.)

Remarks

In most cases you will already know how many parameters a SQL statement has, because you will have designed the statement. In some cases, however, it may be necessary to determine this value programmatically, by using this function.

IMPORTANT NOTE: This function cannot be used to determine the number of parameters that a Stored Procedure requires. For that, you will need to use the value that is returned by the `SQL_ProcColCount` function *with* the `SQL_ProcColInfoSInt` function, to examine the procedure's "columns" and determine which of the columns are "input columns". For more information, see `SQL_ProcColInfoSInt`.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "this statement has one bound parameter". This function can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues: None.

Speed Issues: None.

See Also: Binding Statement Input Parameters

SQL_StringToType

Summary

Assigns the value of a string to a User Defined Type. (See your BASIC documentation for general information about User Defined Types.)

Twin

None.

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_StringToType(sString$, lPointer&, lLength&)
```

...or...

```
lResult& = SQL_StringToType(sString$, dwPointer???, lLength&)
```

Parameters

sString\$

The string value that should be assigned to the User Defined Type. This string must be at least one byte long or `ERROR_BAD_PARAM_VALUE` will be generated.

lPointer& or dwPointer???

A pointer (from the `VarPtr` function) which points to the User Defined Type. See **Remarks** below. (Note that Visual Basic has an *undocumented* `VarPtr` function. See MSDN article Q199824.)

lLength&

Either **1)** the length of the User Defined Type, or **2)** a smaller value, indicating the *portion* of the UDT that should be affected.

Return Values

This function returns `SUCCESS` if the string value is assigned to the User Defined Type, or `ERROR_BAD_PARAM_VALUE` if it is not.

Remarks

PowerBASIC programmers can use the `LSet` function to perform this type of operation. The `SQL_StringToType` function is provided primarily for Visual Basic programmers, but it can be used by any programming language that supports OLE strings.

Visual Basic's `LSet` function does not allow "mixed data types" such as the assignment of a string value to a User Defined Type. VB will generate a "Data Type Mismatch" error if you try to do something like this:

```
LSet MyType = sString$
```

However, this type of operation is a *very* convenient method of manipulating certain types of database data, so SQL Tools provides the `SQL_StringToType` function.

A detailed (and useful) example is provided in the section of this document that is titled `SQL_TIMESTAMP`.

This function performs a "direct assignment" of the string value to the User Defined Type (UDT), so the `sString$` parameter must contain a string that is compatible with the UDT. In other words, the bytes of the string must align properly with the bytes of the UDT. This is not usually a problem if you are using a string that is returned by SQL Tools for a date-time column, because the string data is *designed* to be compatible with UDTs. But databases are allowed to contain *any* type of UDT, so if you are using a nonstandard type, it is up to you to make sure that the string is compatible with the target UDT.

The `IPtr` or `dwPtr` parameter must be a pointer to one of the bytes (usually the first byte) of the UDT. To obtain a pointer to the first byte of a UDT, use:

```
VarPtr(MyType)
```

To obtain a pointer to the fifth byte (for example), use:

```
VarPtr(MyType) + 5
```

Note that Visual Basic has an *undocumented* `VarPtr` function. See MSDN article Q199824.

The `lLength` parameter must not, under any circumstances, be larger than the actual length of the User Defined Type. If you use a value that is too large, data corruption will take place and Application Errors are possible. A smaller value may be used if you want to assign a value to a *portion* of the UDT. For example:

```
SQL_StringToType sString$, VarPtr(MyType), 2
```

...would assign the values in the first 2 bytes of `sString$` to the first 2 bytes of `MyType`.

Diagnostics

This function returns Error Codes and can generate SQL Tools Error Messages.

Example

See `SQL_TIMESTAMP`.

Driver Issues

None.

Speed Issues

None.

See Also

`SQL_TIMESTAMP`

SQL_SyncFetchPos

Summary

Re-synchronizes the SQL Tools row-counting system. It is only necessary to use this function if your program performs a fetch operation that causes SQL Tools to lose track of the current row number. See `SQL_FetchPos` for more information.

Twin

`SQL_SyncFetchPosition`

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_SyncFetchPos(lPosition&))
```

Parameters

lPosition&

The correct row number for the current statement.

Return Values

This function will return `SUCCESS` if you use a value for *lPosition&* that is greater than or equal to zero (0). Otherwise, it will return `ERROR_BAD_PARAM_VALUE`. Please note that if you use an *incorrect* row number, this function will still return `SUCCESS`. Keep in mind that you are *telling* SQL Tools that *lPosition&* is the correct value.

Remarks

Certain types of fetch operations can cause SQL Tools to lose track of a [SELECT](#) statement's current row number, causing the `SQL_FetchPos` and `SQL_FetchPosition` functions to return negative two (-2). The `SQL_SyncFetchPos` function can be used to re-synchronize the row-counting system. For a much more detailed description of this process, see `SQL_FetchPos`.

Diagnostics

This function returns Error Codes and can generate SQL Tools Error Messages.

Example

See `SQL_FetchPos`.

Driver Issues

None.

Speed Issues

None.

See Also

Result Sets

SQL_SyncFetchPosition

Summary

Re-synchronizes the SQL Tools row-counting system. It is only necessary to use this function if your program performs a fetch operation that causes SQL Tools to lose track of the current row number. See `SQL_FetchPos` for more information.

Twin

`SQL_FetchPos`

Family

Statement Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_SyncFetchPosition(lDatabaseNumber&, _  
                                lStatementNumber&.  
                                lPosition&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

lPosition&

See twin function `SQL_SyncFetchPos` for complete details.

Return Values

See twin function `SQL_SyncFetchPos` for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, `SQL_SyncFetchPosition` is identical to `SQL_SyncFetchPos`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_SyncFetchPosition` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_SyncFetchPos` for complete details.

Example

See twin function `SQL_SyncFetchPos`.

Driver Issues

None.

Speed Issues

None.

See Also

Result Sets

SQL_TableColumnCount

Summary

Indicates how many columns a table contains.

Twin

SQL_TblColCount

Family

Table Info Family

Availability

Standard and Pro

Warning

Some ODBC drivers do not include *all* columns in this value. For example, an ODBC driver might not return any information about columns that are created by expressions, or about pseudo-columns such as Oracle ROWID columns. Your program can *use* any valid column, regardless of whether or not it is counted by `SQL_TableColumnCount`.

Syntax

```
lResult& = SQL_TableColumnCount(lDatabaseNumber&, _  
                                lTableNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function `SQL_TblColCount` for complete details.

Return Values

See twin function `SQL_TblColCount` for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, `SQL_TableColumnCount` is identical to `SQL_TblColCount`. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to `SQL_TableColumnCount` is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function `SQL_TblColCount` for complete details.

Example

See twin function `SQL_TblColCount`.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also: Tables, Rows, Columns, and Cells

SQL_TableColumnInfoSInt

Summary

Provides information about one column of a table, in numeric form.

Twin

SQL_TblColInfoSInt

Family

Table Column Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TableColumnInfoSInt(lDatabaseNumber&, _  
                                   lTableNumber&, _  
                                   lColumnNumber&, _  
                                   lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_TblColInfoSInt for complete details.

Return Values

See twin function SQL_TblColInfoSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_TableColumnInfoSInt is identical to SQL_TblColInfoSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TableColumnInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblColInfoSInt for complete details.

Example

See twin function SQL_TblColInfoSInt.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TableColumnInfoStr

Summary

Provides information about one column of a table, in string form.

Twin

SQL_TblColInfoStr

Family

Table Column Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_TableColumnInfoStr(lDatabaseNumber&, _  
                                   lTableNumber&, _  
                                   lColumnNumber&, _  
                                   lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_TblColInfoStr for complete details.

Return Values

See twin function SQL_TblColInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber*& parameter, SQL_TableColumnInfoStr is identical to SQL_TblColInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TableColumnInfoStr is shown below.

For more information about using *lDatabaseNumber*& and *lStatementNumber*& in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblColInfoStr for complete details.

Example

See twin function SQL_TblColInfoStr.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TableColumnName

Summary

Indicates which column number corresponds to a column name.

Twin

SQL_TblColNumber

Family

Table Column Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TableColumnName(lDatabaseNumber&, _  
                                lTableNumber&, _  
                                sColumnName$)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_TblColNumber for complete details.

Return Values

See twin function SQL_TblColNumber for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_TableColumnName is identical to SQL_TblColNumber. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TableColumnName is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblColNumber for complete details.

Example

See twin function SQL_TblColNumber.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also: Tables, Rows, Columns, and Cells

SQL_TableCount

Summary

Indicates the number of tables (of all kinds) that are contained by a database.

Twin

SQL_TblCount

Family

Table Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TableCount(lDatabaseNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

Return Values

See twin function SQL_TblCount for complete details.

Remarks

Except for the *lDatabaseNumber&* parameters, SQL_TableCount is identical to SQL_TblCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TableCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblCount for complete details.

Example

See twin function SQL_TblCount.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TableInfoInt

Summary

Provides information about a table, in numeric form.

Twin

SQL_TblInfoStr

Family

Table Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TableInfoInt(lDatabaseNumber&, _  
                             lTableNumber&, _  
                             lInfoType&)
```

Parameters

lDatabaseNumber

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_TblInfoStr for complete details.

Return Values

See twin function SQL_TblInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_TableInfoStr is identical to SQL_TblInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TableInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblInfoStr for complete details.

Example

See twin function SQL_TblInfoStr.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TableInfoStr

Summary

Provides information about a table, in string form.

Twin

SQL_TblInfoStr

Family

Table Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_TableInfoStr(lDatabaseNumber&, _  
                             lTableNumber&, _  
                             lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_TblInfoStr for complete details.

Return Values

See twin function SQL_TblInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber*& parameter, SQL_TableInfoStr is identical to SQL_TblInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TableInfoStr is shown below.

For more information about using *lDatabaseNumber*& and *lStatementNumber*& in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblInfoStr for complete details.

Example

See twin function SQL_TblInfoStr.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also: Tables, Rows, Columns, and Cells

SQL_TableNumber

Summary

Returns the table number that corresponds to a table name (and an optional table type).

Twin

SQL_TblNumber

Family

Table Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TableNumber(lDatabaseNumber&, _  
                           sTableName$, _  
                           sTableType$)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_TblNumber for complete details.

Return Values

See twin function SQL_TblNumber for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_TableNumber is identical to SQL_TblNumber. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TableNumber is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblNumber for complete details.

Example

See twin function SQL_TblNumber.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TablePrivilegeCount

Summary

Indicates how many Table Privileges a table has.

Twin

SQL_TblPrivCount

Family

Table Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_TablePrivilegeCount(lDatabaseNumber&, _  
                                   lTableNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_TblPrivCount for complete details.

Return Values

See twin function SQL_TblPrivCount for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_TablePrivilegeCount is identical to SQL_TblPrivCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TablePrivilegeCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblPrivCount for complete details.

Example

See twin function SQL_TblPrivCount.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Table Privileges

SQL_TablePrivilegeInfoStr

Summary

Provides information about a Table Privilege, in string form.

Twin

SQL_TblPrivInfoStr

Family

Table Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_TablePrivilegeInfoStr(lDatabaseNumber&, _  
                                     lTableNumber&, _  
                                     lPrivilegeNumber&, _  
                                     lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_TblPrivInfoStr for complete details.

Return Values

See twin function SQL_TblPrivInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_TablePrivilegeInfoStr is identical to SQL_TblPrivInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TablePrivilegeInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblPrivInfoStr for complete details.

Example

See twin function SQL_TblPrivInfoStr.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also

Table Privileges

SQL_TableStatisticSInt

Summary

Provides a Table Statistic, in numeric form.

Twin

SQL_TblStatSInt

Family

Table Info Family

Availability

SQL Tools Pro Only (see)

Warning

Because Table Statistics can change very rapidly (as rows are added and deleted from a table), this information is *not* cached by SQL Tools. The process that is required to access Table Statistic information is relatively time-consuming, so using this function repeatedly may cause your program to slow down significantly.

Syntax

```
lResult& = SQL_TableStatisticSInt(lDatabaseNumber&, _  
                                lTableNumber&, _  
                                lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_TblStatSInt for complete details.

Return Values

See twin function SQL_TblStatSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_TableStatisticSInt is identical to SQL_TblStatSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_TableStatisticSInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_TblStatSInt for complete details.

Example: See twin function SQL_TblStatSInt.

Driver Issues: None.

Speed Issues: See **Warning** above.

See Also: Tables, Rows, Columns, and Cells

SQL_TblColCount

Summary

Indicates how many columns a table has.

Twin

SQL_TableColumnCount

Family

Table Column Info Family

Availability

Standard and Pro

Warning

Some ODBC drivers do not include *all* columns in this value. For example, an ODBC driver might not return any information about columns that are created by expressions, or about pseudo-columns such as Oracle ROWID columns. Your program can *use* any valid column, regardless of whether or not it is counted by SQL_TblColCount.

Syntax

```
lResult& = SQL_TblColCount(lTableNumber&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

Return Values

If a valid *lTableNumber&* value is used, this function will return the number of columns that a table contains. Otherwise, it will return zero (0). Please note that certain types of tables do not have any columns, so this function may also return zero for a valid *lTableNumber&*.

Remarks

This function can be used to determine the number of columns that a table contains.

Diagnostics

This function does not return Error Codes because an Error Code like SUCCESS_WITH_INFO (value 1) could be confused with a legitimate return value like "the specified table has 1 column". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the names of the
'columns in table 2
For lCol& = 1 To SQL_TblColCount(2)
    Print SQL_TblColInfoStr(2,lCol&,TBLCOL_COLUMN_NAME)
Next
```

Driver Issues

None.

Speed Issues

See [Cached Information](#).

See Also

[Tables, Rows, Columns, and Cells](#)

SQL_TblColInfoSInt

Summary

Provides information about one column of a table, in numeric form.

Twin

SQL_TableColumnInfoSInt

Family

Table Column Info Family

Availability

Standard and Pro

Warning

Some ODBC drivers do not provide information about *all* of the columns in a table. For example, an ODBC driver might not return any information about columns that are created by expressions, or about pseudo-columns such as Oracle ROWID columns. Your program can *use* any valid column, regardless of whether or not this function returns any information about it.

Syntax

```
lResult& = SQL_TblColInfoSInt(lTableNumber&, _  
                               lColumnNumber&, _  
                               lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lColumnNumber&

The number of a column, between one (1) and the number returned by the SQL_TblColCount function.

lInfoType&

The type of numeric information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, this function will return the requested numeric information. Otherwise, zero (0) will be returned.

Remarks

Please note that not *all* of the information about a table's columns is useful in numeric form. For a list of *lInfoType&* values that can be used to obtain *string* information about a table's columns, see SQL_TblColInfoStr.

To obtain numeric information about a table's columns, use one of the following *lInfoType&* values:

TBLCOL_BUFFER_LENGTH

The column's buffer size.

In the case of SQL_CHAR, SQL_VARCHAR, and SQL_LONGVARCHAR columns the buffer length may be reported in bytes or it may be reported in

characters. (This behavior is driver-dependent.) That means that if the database uses Unicode *internally* -- even if the column itself does not *appear* to contain Unicode -- the value that is reported for `TBLCOL_BUFFER_LENGTH` may be twice as large as the actual column size. In practice it is not necessary to use a buffer which is that large.

The `TBLCOL_DISPLAY_SIZE` value (see below) returns values that are generally more useful when dealing with `SQL_CHAR`, `SQL_VARCHAR`, and `SQL_LONGVARCHAR` columns.

`TBLCOL_CHAR_OCTET_LENGTH`

ODBC 3.x+ ONLY: The maximum length of a character or binary column, in bytes. For all other data types, this *InfoType*& returns zero (0).

`TBLCOL_DATA_TYPE`

The column's SQL Data Type (`SQL_CHAR`, `SQL_INTEGER`, etc.)

`TBLCOL_DATETIME_SUB`

ODBC 3.x+ ONLY: The sub-type code for datetime and interval data types. For all other data types, this *InfoType*& returns zero (0).

See `TBLCOL_SQL_DATA_TYPE` (below) for more information.

`TBLCOL_DECIMAL_DIGITS`

The number of decimal digits that the column has.

`TBLCOL_DEFAULT_VALUE`

ODBC 3.x+ ONLY: The default value of the column.

Please note that Microsoft Access 97 has been observed returning erroneous values for this *InfoType*&.

Please also note that *InfoType*& can return *both* string and numeric data, depending on the column type, so your program will need to use this function *and/or* `SQL_TblColInfoStr` with `TBLCOL_DEFAULT_VALUE` to obtain a value.

The value in this column should be interpreted as a string if it is enclosed in quotation marks. Otherwise, it should be interpreted as a numeric or binary value.

If the Null value was specified as the default value, this *InfoType*& will return the word `NULL`, *not* enclosed in quotation marks.

If the default value cannot be represented without truncation, this *InfoType*& will return the word `TRUNCATED`, not enclosed in quotation marks. (The value of `TBLCOL_DEFAULT_VALUE` can be used when you are generating a new column definition, except when it is `TRUNCATED`.)

If no default value was specified, then this *InfoType*& will return an empty

string or the number zero.

TBLCOL_DISPLAY_SIZE

The column's display size.

TBLCOL_NULLABLE

This value indicates the column's nullability. It will always return one of the following values:

SQL_NO_NULLS (The column can not include Null values.)

SQL_NULLABLE (The column accepts Null values.)

SQL_NULLABLE_UNKNOWN (It is not known whether or not the column accepts Null values.)

Please note that the value that is returned for TBLCOL_NULLABLE is different from the value returned by TBLCOL_IS_NULLABLE (see SQL_TblColumnInfoStr). The TBLCOL_NULLABLE value indicates with certainty that a column *can* accept Null values, but cannot indicate with certainty that a column does *not* accept Null values. The TBLCOL_IS_NULLABLE value, on the other hand, indicates with certainty that a column does *not* accept Null values, but cannot indicate with certainty that a column *can* accept Null values.

TBLCOL_NUM_PREC_RADIX

The column's Num Prec Radix value.

TBLCOL_ORDINAL_POSITION

ODBC 3.x+ ONLY: The first column in a table will return 1, the second will return 2, and so on.

TBLCOL_SQL_DATA_TYPE

ODBC 3.x+ ONLY: This value is the same as the TBLCOL_DATA_TYPE value, except for datetime and interval columns. For datetime and interval data types, this *InfoType* returns the non-concise data type (such as SQL_DATETIME or SQL_INTERVAL), rather than the concise data type (such as SQL_INTERVAL_YEAR_TO_MONTH).

If this column returns SQL_DATETIME or SQL_INTERVAL, the specific data type can be determined from the TBLCOL_DATETIME_SUB function (see above).

DRIVER-DEFINED COLUMNS

In addition to the standard ODBC values, SQL Tools also supports up to 14 driver-defined information types. You can use the *InfoType* values TBLCOL_DRIVERDEF_FIRST and TBLCOL_DRIVERDEF_LAST, or literal numeric *InfoType* values, to access this information.

If your driver's documentation lists a driver-specific info by number, add 102 to get the SQL Tools info number. For example, if a certain driver provides one driver-specific type of numeric info, and documents it as #19, you would access it with SQL Tools #121, like this...

```
SQL_TblColInfoSInt(TableNumber&, ColumnNumber&, 121)
```

...or, if the information was documented as a string value...

```
SQL_TblColInfoStr(TableNumber&, ColumnNumber&, 121)
```

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "the specified column has a data type of `SQL_CHAR` (value 1)". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the data type of  
'table 1, column 7:  
Print SQL_TblColInfoSInt(1,7,TBLCOL_DATA_TYPE)
```

Driver Issues

See note regarding Microsoft Access and `TBLCOL_DEFAULT_VALUE`, above.

Speed Issues

See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TblColInfoStr

Summary

Provides information about one column of a table, in string form.

Twin

SQL_TableColumnInfoStr

Family

Table Column Info Family

Availability

Standard and Pro

Warning

Some ODBC drivers do not provide information about *all* of the columns in a table. For example, an ODBC driver might not return any information about columns that are created by expressions, or about pseudo-columns such as Oracle ROWID columns. Your program can *use* any valid column, regardless of whether or not this function returns any information about it.

Syntax

```
sResult$ = SQL_TblColInfoStr(lTableNumber&, _  
                             lColumnNumber&, _  
                             lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lColumnNumber&

The number of a column, between one (1) and the number returned by the SQL_TblColCount function.

lInfoType&

The type of string information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, this function will return the requested information. Otherwise, an empty string will be returned.

Remarks

Please note that not *all* of the information about a table's columns is useful in string form. For a list of *lInfoType&* values that can be used to obtain *numeric* information about a table's columns, see SQL_TblColInfoSInt.

To obtain string information about a table's columns, use one of the following *lInfoType&* values:

TBLCOL_CATALOG_NAME, TBLCOL_SCHEMA_NAME, and TBLCOL_TABLE_NAME

The name of the catalog, schema, and table that contain the column about which information is being requested. If a database does not support catalog and/or schema names, these values may be empty strings.

TBLCOL_COLUMN_NAME

The name of the column. This can be (but is not usually) an empty string.

TBLCOL_DEFAULT_VALUE

ODBC 3.x+ ONLY: The default value of the column.

Please note that Microsoft Access 97 has been observed returning erroneous values for this *InfoType*.

Please also note that *InfoType* can return *both* string and numeric data, depending on the column type, so your program will need to use this function or `SQL_TblColInfoStr` with `TBLCOL_DEFAULT_VALUE` to obtain a value.

The value in this column should be interpreted as a string if it is enclosed in quotation marks. Otherwise, it should be interpreted as a numeric or binary value.

If the Null value was specified as the default value, this *InfoType* will return the word `NULL`, not enclosed in quotation marks.

If the default value cannot be represented without truncation, this *InfoType* will return the word `TRUNCATED`, not enclosed in quotation marks. (The value of `TBLCOL_DEFAULT_VALUE` can be used when you are generating a new column definition, except if it contains `TRUNCATED`.)

If no default value was specified, then this *InfoType* will return an empty string or the number zero.

TBLCOL_IS_NULLABLE

This function will return one of the following values:

The string "NO" if the column does not allow Null values.

The string "YES" if the column does allow Null values.

An empty string if the column's nullability is not known.

See `SQL_TblColInfoInt (TBLCOL_NULLABLE)` (as opposed to this string *InfoType*, `TBLCOL_IS_NULLABLE`) for more information.

TBLCOL_REMARKS

An optional column description.

TBLCOL_TYPE_NAME

The datasource-dependent name of the column's data type, such as "INTEGER" or "COUNTER".

DRIVER-DEFINED COLUMNS

In addition to the standard ODBC values, SQL Tools also supports up to 14

driver-defined information types. You can use the *InfoType* values `TBLCOL_DRIVERDEF_FIRST` and `TBLCOL_DRIVERDEF_LAST`, or literal numeric *InfoType* values, to access this information.

If your driver's documentation lists a driver-specific info by number, add 102 to get the SQL Tools info number. For example, if a certain driver provides one driver-specific type of string info, and documents it as #20, you would access it with SQL Tools #122, like this...

```
SQL_TblColInfoStr ( ITableNumber&, IColumnNumber&, 122 )
```

...or, if the information was documented as a numeric value...

```
SQL_TblColInfoSInt ( ITableNumber&, IColumnNumber&, 122 )
```

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the name of  
'table 1, column 7:  
Print SQL_TblColInfoStr(1,7,TBLCOL_COLUMN_NAME)
```

Driver Issues

See note regarding Microsoft Access and `TBLCOL_DEFAULT_VALUE`, above.

Speed Issues

See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TblColNumber

Summary

Returns the column number that corresponds to a column name.

Twin

SQL_TableColumnNumber

Family

Table Column Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TblColNumber(lTableNumber&, _  
                             sColumnName$)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

sColumnName\$

A string that contains the name of a column.

Return Values

If *sColumnName\$* contains a string that matches the name of a column in the specified table number, the corresponding column number is returned.

If no match is found, negative one (-1) will be returned.

Remarks

This function is not case sensitive. If Column 4 is named "ADDRESS", then using a *sColumnName\$* value of "ADDRESS", "address", or "Address" (etc.) would return the number 4.

Diagnostics

This function does not return Error Codes because an Error Code like SUCCESS_WITH_INFO (value 1) could be confused with a legitimate return value like "the specified column name matches column 1". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the column number of the  
'table 1 ADDRESS column...  
Print SQL_TblColNumber(1, "ADDRESS")
```

Driver Issues: None.

Speed Issues: See Cached Information.

See Also: Tables, Rows, Columns, and Cells

SQL_TblCount

Summary

Indicates the number of tables (of all types) that a database contains.

Twin

SQL_TableCount

Family

Table Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TblCount
```

Parameters

None.

Return Values

This function will return the total number of tables (tables, system tables, views, etc.) that are contained by a database. If a database has not yet been opened, or if a database contains no tables, the return value of this function will be zero (0).

Remarks

Virtually all databases contain tables, unless **1)** no tables have yet been added to a new database, or **2)** all of the tables have been deleted from a database.

Keep in mind that this function returns the *total* number of tables in a database, including tables, system tables, and views. Other types of tables can include "global temporary", "local temporary", "alias", and "synonym". Databases can also contain datasource-specific table types.

See `SQL_TblInfoStr(TABLE_TYPE)` for more information.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "the table has 1 column". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
'Display the number of tables  
'in the current database:  
Print SQL_TblCount
```

Driver Issues

None.

Speed Issues

See [Cached Information](#).

See Also

[Tables, Rows, Columns, and Cells](#)

SQL_TblInfoSInt

Summary

Provides information about a table, in numeric form.

Twin

SQL_TableInfoSInt

Family

Table Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TblInfoSInt(lTableNumber&, _  
                           lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

lInfoType&

The type of numeric information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, this function will return the requested information. Otherwise, it will return zero (0).

Remarks

Please note that not *all* of the information that is available about a table is useful in numeric form. For a list of *lInfoType&* values that can be used to obtain *string* information about a table, see `SQL_TblInfoStr`.

In order to obtain numeric information about a table, the *lInfoType&* parameter must be one of the following values:

```
TABLE_AUTOCOLUMN_COUNT  
TABLE_COLUMN_COUNT  
TABLE_FOREIGNKEY_COUNT  
TABLE_INDEX_COUNT  
TABLE_PRIMARYKEY_COUNT  
TABLE_PRIVILEGE_COUNT  
TABLE_UNIQUECOLUMN_COUNT
```

These *lInfoType&* values can be used to obtain exactly the same information that is available from the `SQL_TblColCount`, `SQL_TblPrivCount`, `SQL_FornKeyCount`, `SQL_PrimKeyCount`, `SQL_IndxCount`, `SQL_UniqColCount`, and `SQL_AutoColCount` functions, respectively.

But the `SQL_TblInfoSInt` function obtains that information using a

technique that is *different* from those functions. The values that are provided by `SQL_TblInfoSInt` are not cached.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_TblInfoSInt(1, TABLE_INDEX_COUNT)
```

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TblInfoStr

Summary

Provides information about a table, in string form.

Twin

SQL_TableInfoStr

Family

Table Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_TblInfoStr(lTableNumber&, _  
                           lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

lInfoType&

The type of string information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, this function will return the requested information. Otherwise, it will return an empty string.

Remarks

Please note that not *all* of the information that is available about a table is useful in string form. For a list of *lInfoType&* values that can be used to obtain *numeric* information about a table, see `SQL_TblInfoInt`.

In order to obtain string information about a table, the *lInfoType&* parameter must be one of the following values:

TABLE_CATALOG_NAME, and TABLE_SCHEMA_NAME

The catalog and schema names that are associated with the table.

TABLE_NAME

The table's name.

TABLE_REMARKS

An optional comment field.

TABLE_TYPE

This *InfoType* will return a string like "TABLE", "SYSTEM TABLE", "VIEW", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", or "SYNONYM", or a datasource-specific type name.

A "TABLE" is *usually* a "normal" database table that is completely accessible to your program.

A "SYSTEM TABLE" is an "internal" database table that is created by a DBMS program. For example, when you use Microsoft Access to create a "Form" or a "Report", you are really creating a System Table which contains the information that Access needs to build the form or table.

A "VIEW" is a "virtual table" that is created from the columns of one or more "real" tables.

The other, less common table types cannot usually be accessed, so they are not covered here.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_TblInfoStr(1, TABLE_NAME)
```

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TblNumber

Summary

Returns the table number that corresponds to a table name (and an optional table type).

Twin

SQL_TableNumber

Family

Table Info Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TblNumber(sTableName$, _  
                        sTableType$)
```

Parameters

sTableName\$

A string that contains the name of a table, or an empty string.

sTableType\$

A string that contains the type-name of a table, or an empty string. Common table types include "TABLE", "SYSTEM TABLE", and "VIEW". For a reasonably complete list, see `SQL_TblInfoStr(TABLE_TYPE)`.

Return Values

If a table is found that matches the specified parameters, the table's number (between one and the number returned by the `SQL_TblCount` function) will be returned. Otherwise, negative one (-1) will be returned.

Remarks

This function is *not* case-sensitive. If a table named ADDRESSBOOK exists, it can be found by using the *sTableName\$* parameter "ADDRESSBOOK", "addressbook", "AddressBook", etc. If a table with the type "TABLE" exists, it can be found with the *sTableType\$* parameter "TABLE", "table", "Table", etc. If you need to perform a case-sensitive search, you should use the `SQL_TblInfoStr` function to examine the table names/types directly.

If the *sTableType\$* parameter is an empty string, the first table number that matches the *sTableName\$* parameter (if any) will be returned. Generally speaking, well-designed databases do not use duplicate table names, so it is not usually necessary to specify a table type.

If the *sTableName\$* parameter is an empty string, the first table number that matches the *sTableType\$* parameter (if any) will be returned. (Keep in mind that databases often contain more than one "TABLE", more than one "SYSTEM TABLE", etc. and this function does not provide a method for retrieving subsequent matches.)

If neither parameter is an empty string, the first (and presumably only) table number that matches *both* parameters will be returned.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value such as "table number 1". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

```
Print SQL_TblNumber("MYTABLE", "")
```

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Tables, Rows, Columns, and Cells

SQL_TblPrivCount

Summary

Provides the number of Table Privileges that a table has.

Twin

SQL_TablePrivilegeCount

Family

Table Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_TblPrivCount(lTableNumber&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

Return Values

If the *lTableNumber&* parameter is valid, and if the database is open, this function will return the number of Table Privileges that are associated with a table.

Remarks

A Table Privilege is an "access right" that is granted to a user, called the Grantee, by another user, called the Grantor. For example, if Table Privileges have been specified for a certain table like `PAYROLL`, a certain user may have a `SELECT` privilege (the right to use the [SELECT](#) statement to retrieve data from the table) but not an `UPDATE` privilege (the right to change the values in the table). Other users might not have any rights to access the `PAYROLL` table in any way.

This function returns the total number of Table Privileges that have been defined for a table.

See Table Privileges for more information. Also compare Column Privileges.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "this table has one Table Privilege". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Table Info Family

SQL_TblPrivInfoStr

Summary

Provides information about a Table Privilege, in string form.

Twin

SQL_TablePrivilegeInfoStr

Family

Table Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_TblPrivInfoStr(lTableNumber&, _  
                               lPrivilegeNumber&, _  
                               lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lPrivilegeNumber&

The number of a Table Privilege, between one (1) and the number returned by the SQL_TblPrivCount function.

lInfoType&

The type of information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, this function will return the requested information. Otherwise, it will return an empty string.

Remarks

A Table Privilege is an "access right" that is granted to a user, called the Grantee, by another user, called the Grantor. For example, if Table Privileges have been specified for a certain table like `PAYROLL`, a certain user may have a `SELECT` privilege (the right to use the `SELECT` statement to retrieve data from the table) but not an `UPDATE` privilege (the right to change the values in the table). Other users might not have any rights to access the `PAYROLL` table in any way.

See Table Privileges for more information. Also compare Column Privileges.

To obtain information about a Table Privilege, use one of the following *lInfoType&* values:

TBLPRIV_GRANTEE

The name of the user to whom the privilege has been granted.

TBLPRIV_GRANTOR

The name of the user that granted the privilege. If the value of TBLPRIV GRANTEE (just above) is the owner of the table, the TBLPRIV GRANTOR value will be "_SYSTEM".

TBLPRIV_IS_GRANTABLE

Indicates whether or not the grantee is permitted to grant the privilege to other users.

This *InfoType* will return "YES" or "NO", or an empty string if the grantability is unknown or is not applicable to the Datasource.

TBLPRIV_PRIVILEGE

Identifies the privilege that is granted. May be one of the following values, or other values that are supported by the Datasource. Please note that the quotation marks that are shown below are *not* part of the value that will be returned by this *InfoType*.

"SELECT" (The grantee is permitted to retrieve data from the table)

"INSERT" (The grantee is permitted to insert new rows into the table.)

"UPDATE" (The grantee is permitted to update data in the table.)

"REFERENCES" (The grantee is permitted to refer to the table within a constraint (for example, a unique, referential, or table-check constraint).

The scope of action that is given to the grantee by a given Table Privilege is datasource-dependent. For example, an UPDATE privilege might permit the grantee to update all of the columns in a table on one Datasource, but only those columns for which the grantor has the UPDATE privilege on another Datasource.

TBLPRIV_TABLE_CATALOG, TBLPRIV_TABLE_SCHEMA, and TBLPRIV_TABLE_NAME

The catalog, schema, and table name to which the privilege applies.

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

None

See Also: Table Info Family

SQL_TblStatSInt

Summary

Provides a Table Statistic, in numeric form.

Twin

SQL_TableStatisticSInt

Family

Table Info Family

Availability

SQL Tools Pro Only (see)

Warnings

Because Table Statistics can change very rapidly (as rows are added and deleted from a table), this information is *not* cached by SQL Tools. The process that is required to access Table Statistic information is relatively time-consuming, so using this function repeatedly may cause your program to slow down significantly.

Also, ODBC drivers that conform only to the X/Open standard and do not support ODBC extensions may not be able to use this function unless a particular SQL Tools Option is set. See **Remarks** below for more information.

Syntax

```
lResult& = SQL_TblStatSInt(lTableNumber&, _  
                           lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

lInfoType&

The type of information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, and if the ODBC driver can supply Table Statistics, this function will return the requested information. Otherwise, it will return zero (0).

Remarks

A *Statistic* is an ODBC data structure that contains very basic information about a table. Specifically, this term refers to a single structure that contains both **1**) the number of rows in a table, and **2**) the number of pages that are used to store the table.

You can use one of the following *lInfoType&* values to obtain Statistic information about a table:

TBLSTAT_PAGECOUNT

The number of pages that are used to store the table. Zero (0) may be returned if the value is not available from the datasource (see `SQL_ENSURE`

below), or if "pages" are not applicable to the Datasource.

TBLSTAT_ROWCOUNT

The number of rows that are currently in the table. Zero (0) may be returned if the value is not available from the datasource; see `SQL_ENSURE` below. (The row count is also known as the "cardinality" of the table.)

When SQL Tools requests a statistic from the ODBC driver, it can ask for a certain level of "confidence" about the values that are returned. The default confidence level is `SQL_ENSURE`, which tells the ODBC driver to retrieve the statistic "unconditionally". You can also use `SQL_QUICK`, which tells the driver to retrieve the statistic only if it is readily available. In this case, the ODBC driver does not ensure that the values are current, and zero (0) may be returned. To change the confidence level for this function, use the `SQL_SetOptionSInt(OPT_STAT_ENSURE)` function.

IMPORTANT NOTE: ODBC drivers that conform only to the X/Open standard and do not support ODBC extensions will not be able to use `SQL_ENSURE`, which is the default confidence level. Applications that are written for drivers which use the X/Open standard will always get `SQL_QUICK` behavior from ODBC 3.x-compliant drivers.

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "this table has one row". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

See **Warnings** and **Remarks** above.

See Also

Table Info Family

SQL_TextDate

Summary

Converts a binary `SQL_DATE` structure (also known as `DATE_STRUCT`) into a printable date string.

Twin

None

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TextDate(sString$)
```

Parameters

sString\$

A string with the appropriate `DATE_STRUCT` format (see the SQL Tools Declaration File or `SQL_DATE`).

Return Values

This function always returns a printable date string.

Remarks

The format of the return value of this function is:

CCYY-MM-DD

The individual components are Century (CC), Year (YY), Month (MM), and Day (DD).

Diagnostics

None.

Example

See **Remarks** above.

Driver Issues

None.

Speed Issues

None.

See Also

Utility Family

SQL_TextDateTime

Summary

Converts a binary SQL_TIMESTAMP structure (also known as TIMESTAMP_STRUCT) into a printable date/time string.

Twin

None

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TextDateTime(sString$, _  
                             lFrom&, _  
                             lTo&)
```

Parameters

sString\$

A string with the appropriate TIMESTAMP_STRUCT format (see the SQL Tools Declaration File or SQL_TIMESTAMP).

lTo& and lFrom&

The starting and ending character positions. See **Remarks** below for more information.

Return Values

This function returns a printable date/time string, based on the parameters that you provide.

Remarks

The default format for the return value of this function is:

```
CCYY-MM-DD @ HH:MM:SS.ffffffffff
```

The individual components are Century (CC), Year (YY), Month (MM), Day (DD), Hour (HH), Minute (MM), Second (SS) and Fractional Seconds (ffffffffff). Your program can specify that the entire string be returned by using *lFrom&* and *lTo&* values of zero (0), or it can use those parameters to specify the starting and ending characters of the string.

For example, if the TIMESTAMP_STRUCT that was used for *sString\$* represented the last nanosecond of the 20th century, the SQL_TextDateTime function could be used as follows, to return the results shown:

```
Print SQL_TextDateTime(sString$,0,0)
```

```
Result: 1999-12-31 @ 23:59:59.999999999
```

```
Print SQL_TextDateTime(sString$,1,10)
```

Result: 1999-12-31

```
Print SQL_TextDateTime(sString$,3,10)
```

Result: 99-12-31

```
Print SQL_TextDateTime(sString$,14,21)
```

Result: 23:59:59

```
Print SQL_TextDateTime(sString$,6,18)
```

Result: 12-31 @ 23:59

Diagnostics

None.

Example

See **Remarks** above.

Driver Issues

None.

Speed Issues

None.

See Also

Utility Family

SQL_TextStr and SQL_TextStrOLE

Summary

These functions can be used to convert a string to an all-text format, replacing non-printable characters with printable strings.

Twin

None

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
sResult$ = SQL_TextStr(sString$)
```

...Or...

```
sResult$ = SQL_TextStrOLE(sString$)
```

Parameters

sText\$

Any string. (If the `SQL_TextStr` function is used, *sText\$* cannot contain `CHR$(0)`. See **Remarks** for details.)

Return Values

These functions will return a copy of *sString\$* with the non-printable characters replaced by the `[CHR$(x)]` notation. See **Remarks** below for more information.

Remarks

`SQL_TextStr` and `SQL_TextStrOLE` are identical, except that the `SQL_TextStr` function *cannot* handle strings that contain `CHR$(0)`, and `SQL_TextStrOLE` *can*. `SQL_TextStrOLE` has the additional advantage of being very slightly faster, so it should be used whenever possible. `SQL_TextStr` is provided primarily for languages that do not support Windows OLE strings. (Both Visual Basic and PowerBASIC do support OLE strings.)

Unless you change the default options that affect these functions (see below), they will make a copy of the *sString\$* parameter, and replace any characters with ASCII values below 32 (the space character) or above 126 (the tilde character) with the following notation:

```
[ CHR$(x) ]
```

The resulting string will then become the return value of the function.

Please note that while this document often refers to the `[CHR$(x)]` notation, that is done in order to make sure that the string is not displayed or printed incorrectly. The actual notation contains one space character after the "[" character and one before

the "]" character. The actual value of (x) will be the string representation of an ASCII value, from 0 to 255. Examples:

```
[ CHR$(0) ]  
[ CHR$(255) ]
```

You can change the range of characters that are considered to be non-printable by using the `SQL_SetOptionSInt(OPT_TEXT_MINCHAR)` and `SQL_SetOptionSInt(OPT_TEXT_MAXCHAR)` functions.

The `SQL_ResColText` function uses this function to convert non-printable column data into a printable text string.

The `SQL_BinaryStr` function can be used to re-convert a text string into a string that contains all of the original characters.

Diagnostics

These functions will return a SQL Tools Error Message only if the `OPT_TEXT_MINCHAR` value is larger than the `OPT_TEXT_MAXCHAR` value.

Example

```
sString$ = "HELLO"+CHR$(10,13)+"WORLD"  
Print sString$  
Print "-----"  
Print SQL_TextStrOLE(sString$)
```

Results:

```
HELLO  
WORLD  
-----  
HELLO[ CHR$(10) ][CHR$(13 )]WORLD
```

Driver Issues

None.

Speed Issues

None.

See Also

Utility Family

SQL_TextTime

Summary

Converts a binary `SQL_TIME` structure (also known as `TIME_STRUCT`) into a printable time string.

Twin

None

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_TextTime(sString$)
```

Parameters

sString\$

A string with the appropriate `TIME_STRUCT` format (see the SQL Tools Declaration Files or `SQL_TIME.`).

Return Values

This function always returns a printable time string, which may be an empty string.

Remarks

The format of the return value of this function is:

HH:MM:SS

The individual components are Hour (HH), Minute (MM), and Second (SS).

Diagnostics

None.

Example

See **Remarks** above.

Driver Issues

None.

Speed Issues

None.

See Also

Utility Family

SQL_Thread

Summary

Tells SQL Tools about your multithreaded program. Also used for the Asynchronous Execution of SQL Statements.

Twin

None.

Family

Configuration Family

Availability

SQL Tools Pro Only (see)

Warning

See Multithreaded Programs.

Not all ODBC Drivers support multi-threading.

Syntax

```
lResult& = SQL_Thread(lOperation&, lThreadNumber&)
```

Parameters

lOperation&

One of the following constants: `THREAD_MAX`, `THREAD_START`, or `THREAD_STOP`. See **Remarks** below for details.

lThreadNumber&

Depending on the value of *lOperation&*, either a thread number, or the maximum thread number that will be used. Under no circumstances can this value be less than zero (0) or greater than 9,999. See **Remarks** below.

Return Values

If the requested operation is successful, `SUCCESS` will be returned. Otherwise, a SQL Tools Error Code will be returned.

Remarks

For background information, see Multithreaded Programs.

`THREAD_MAX`

If *lOperation&* is `THREAD_MAX`, then *lThreadNumber&* tells SQL Tools the maximum thread number that your program will be using. The `THREAD_MAX` function can only be used by your program's *primary* thread (thread zero), i.e. the thread that is created by the `WinMain` or `PBMain` function. If you attempt to use `THREAD_MAX` from a secondary thread (one that was launched with `THREAD_CREATE`), an Error Message will be generated and the operation will not be performed.

The `THREAD_MAX` function may be used more than once by your program, under the following conditions:

You may use `THREAD_MAX` to increase the maximum thread number at any time.

You may use `THREAD_MAX` to decrease the maximum thread number, as long as there are no threads currently running which have thread numbers that are greater than the new maximum. For example, if thread number 5 is running you may not specify a new maximum thread number that is less than 5. If you attempt to do so, the operation will be ignored and an Error Message will be generated.

If your program does not use threads, it is *not* necessary to specify a `THREAD_MAX` value of zero (0).

`THREAD_START`

If *Operation* is `THREAD_START`, SQL Tools will create a new Error Stack for thread number *ThreadNumber*. The `THREAD_START` function can only be used from within a secondary thread, i.e. from within a thread that has been launched with the `THREAD CREATE` statement. If your program's primary thread (the thread that is started by WinMain or PMain) attempts to use `THREAD_START`, an Error Message will be generated and the operation will be ignored.

`THREAD_START` must be used once and only once by each thread, preferably as the first statement that is executed by a new thread. It *must* be used before the new thread uses any other SQL Tools functions. (Threads which do not use any SQL Tools functions at all should *not* use `SQL_Thread THREAD_START`.)

`THREAD_STOP`

If *Operation* is `THREAD_STOP`, SQL Tools will destroy the Error Stack for thread number *ThreadNumber*. The `THREAD_STOP` function can only be used from within a secondary thread, i.e. from within a thread that has been launched with the `THREAD CREATE` statement. If your program's primary thread (the thread that is started by WinMain or PMain) attempts to use `THREAD_STOP`, an Error Message will be generated and the operation will be ignored.

`THREAD_STOP` must be used once and only once by each thread, preferably as the last statement that is executed by a thread before it terminates. A thread must not use any other SQL Tools function after a `THREAD_STOP`.

Diagnostics

This function can return Error Codes, and it can generate SQL Tools Error Messages.

Example

See Multithreaded Programs.

Driver Issues: Not all ODBC drivers support multithreading. See Multithreaded Programs for more information.

Speed Issues: Using a `THREAD_MAX` value that is significantly larger than necessary can, under certain circumstances, slow down a program very slightly. This is especially true if low-numbered threads are left unused.

See Also: Multithreaded Programs

SQL_ToolsVersion

Summary

Returns the version number which is embedded in the SQL Tools Runtime File that has been loaded by your program.

Twin

None.

Family

Utility Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_ToolsVersion
```

Parameters

None.

Return Values

This function will return a positive number greater than 100 if SQL Tools Pro is loaded, or a negative number less than -100 if SQL Tools Standard is loaded.

An absolute value of 200 represents SQL Tools Version 2.00, an absolute value of 201 represents version 2.01, and so on.

Remarks

If your program relies on features that are present only in SQL Tools Pro, or if it relies on a certain revision level of the Runtime Files being installed, you should use this function to check the version number when your program is first started.

Keep in mind that the Runtime Files that you provided with your program may have been overwritten by another application's installation program. Or an old version of the SQL Tools Runtime Files may have been restored from a backup tape. Or more than one SQL Tools Runtime File may exist (in different directories, of course) and your program may be using the wrong one.

Diagnostics

None.

Example

```
Print SQL_ToolsVersion
```

Driver Issues: None.

Speed Issues: None.

See Also: Utility Family

SQL_Trace

Summary

Turns program tracing on and off.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

See **Speed Issues** below.

Syntax

```
lResult& = SQL_Trace(lOnOff&)
```

Since it is usually safe to ignore the return value of this function, you can also use the simplified syntax...

```
SQL_Trace lOnOff&
```

Parameters

lOnOff&

TRACE_ON activates the Trace Mode.

TRACE_OFF deactivates the Trace Mode.

TRACE_SINGLE activates the Trace Mode for the *next* SQL Tools function that is used. Tracing is automatically deactivated after the function exits.

You can also use the alternate syntax `SQL_Trace 1` instead of TRACE_SINGLE.

Return Values

This function returns `SUCCESS` if a legal value is specified for *lOnOff&*, or `ERROR_BAD_PARAM_VALUE` if an illegal value is used.

As long as you use one of the predefined constants shown above, it is safe to ignore the return value of this function.

Remarks

The Trace Mode creates a trace file that records the use of SQL Tools functions, for troubleshooting purposes.

SQL Tools Functions are logged by name, and trace file entries include all of the parameters that were passed to a function, all errors that were detected, all return values, and certain other information that can be valuable during troubleshooting.

While tracing is turned on, your program can optionally add strings and numbers to the trace file by using the `SQL_TraceStr`, `SQL_TraceStrOLE`, and

SQL_TraceSInt functions. Using these functions while tracing is turned off has no effect.

When you examine a trace file you will probably notice that "abbreviated" functions (see) are always logged as "verbose" functions. For example, if you use SQL_**Tbl**Count in your program you will see something like this in the trace file:

```
SQL_TableCount DB 1
```

This is done so that you can see exactly what your program is doing when it uses an abbreviated function.

Please note that all of the following options must be set while the Trace Mode is turned off. You should set all of the desired options, and then activate the Trace Mode with SQL_Trace.

The default trace file name is \SQLTOOLS.LOG. You can change the default setting by using the SQL_SetOptionStr(OPT_TRACE_FILE, sFileName\$) function, where sFileName\$ is the name and optional drive/path of a file. If you change this value but do not specify a *valid* filename, the SQL_Trace function will fail to activate the trace mode.

By default, the Trace Mode records the time of every trace file entry in the format [12345.678] where the numbers represent a timer value, in seconds past midnight. If you do not need this information, you can increase the speed of trace-file creation by using the SQL_SetOptionSInt(OPT_TRACE_TIMES, 0) function.

By default, the Trace Mode appends an existing trace file. You can instruct SQL Tools to overwrite a trace file each time the Trace Mode is turned on, by using the SQL_SetOptionSInt(OPT_TRACE_APPEND, 0) function. If your program switches tracing on and off repeatedly, keep in mind that this option overwrites the old trace file *every time that tracing is switched on*.

By default, the Trace Mode automatically indents trace file entries to indicate internal function nesting (see example). It is possible to disable this function by using the SQL_SetOptionSInt(OPT_TRACE_INDENT, 0) function, in order to speed up the creation of trace files, but the files become significantly less readable if this is done.

Diagnostics

None.

Example

```
SQL_Trace TRACE_SINGLE
PRINT SQL_ResColText(2)
```

'sample trace file...

```
[36938.800] ----- START TRACE
[36938.990] >SQL_ResultColumnText DB 1, Stmt 1, Col 2
[36938.990] |      Type 4
[36938.990] |      >SQL_ResultColumnSInt DB 1, Stmt 1, Col 2
[36939.000] |      |      Type 4
[36939.000] |      <8
[36939.000] <8<
[36939.010] ----- END TRACE
```

Important note: For best results, trace files should always be viewed with a program that displays text in a fixed-width ("monospace") font such as Courier.

The > symbol indicates a point where a SQL Tools function was used, and the < symbol indicates a return value. String return values have a second < symbol at the end, to make trailing spaces easier to see.

If one SQL Tools function calls another (internally), the functions are indented and lines are drawn to make the nesting easier to interpret. The vertical lines are shown to make it clear which return values correspond to which functions.

In the example trace file above, the calling program used the `SQL_ResultColumnText` function (or the abbreviated `SQL_ResColText` function) to get a value for Database 1, Statement 1, Column 2. SQL Tools identified the column as SQL Data Type 4 (which corresponds to `SQL_INTEGER`), then internally used the `SQL_ResultColumnSInt` function to get the numeric value of the column. The `SQL_ResultColumnSInt` function returned a value of 8. You can tell that it was a numeric return value because there is only one < symbol. Then the return value of the `SQL_ResultColumnText` function is shown as <8< because it is a string value.

Driver Issues

None.

Speed Issues

Because it involves the creation of a large text file, the use of the SQL Tools Trace Mode can *greatly* slow down a program. One of our very small test programs took 7.26 seconds to execute when the Trace Mode was turned on, but less than 0.05 seconds with tracing turned off. And the slowdown can be made *much* worse if the ODBC Trace Mode is used at the same time, or if an existing Trace File is being appended (which is the default behavior). Instead of activating the Trace Mode at the very beginning of your program, we suggest that you attempt to isolate a small section of code that is likely to be causing a problem, and turn the Trace Mode on then off again as quickly as possible.

Also please note that SQL Tools *often* uses its own functions internally, as shown in the example, so it is possible for a single function to create a *huge* trace file. The SQL Tools Info functions, in particular, can create very large volumes of text for a simple function like `SQL_TableCount`.

See Also

Error/Trace Family

SQL_TraceSInt

Summary

Adds a numeric value (of your choosing) to a Trace File.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

See SQL_Trace.

Syntax

```
SQL_TraceSInt lNumber&
```

Parameters

lNumber&

Any number in the BAS_LONG range.

Return Values

This function always returns SUCCESS, so it is safe to ignore the function's return value.

Remarks

When the trace mode is turned on with the SQL_Trace function, SQL Tools creates a trace file which contains the names of all of the SQL Tools functions that are used, including the parameters that are passed to them, the values that they return, and any errors that are detected. It is possible, however, that it will still be difficult to troubleshoot a problem because you can't "see" the values in *your* program. You can add numeric values to the trace file by using the SQL_TraceSInt function. *If the trace mode is turned on* when the function is executed, the numeric value will be added to the trace file. (You can also use the SQL_TraceStr and SQL_TraceStrOLE functions to add strings to the trace file, including strings that represent floating point numbers.)

Diagnostics

None.

Example

```
SQL_TraceSInt 12345
```

Driver Issues

None.

Speed Issues

Using the Trace Mode can *significantly* slow down your program.

See Also

Error/Trace Family

SQL_TraceStr and SQL_TraceStrOLE

Summary

These functions are used to add a string value to a Trace File.

Twin

None.

Family

Error/Trace Family

Availability

Standard and Pro

Warning

See SQL_Trace.

Syntax

```
SQL_TraceStr sString$
```

...or...

```
SQL_TraceStrOLE sString$
```

Parameters

sString\$

Any string. (When using SQL_TraceStr, *sString\$* may not contain CHR\$(0). See **Remarks** for details.)

Return Values

These functions always returns SUCCESS, so it is safe to ignore their return values.

Remarks

SQL_TraceStr and SQL_TraceStrOLE are identical, except that the SQL_TraceStr function *cannot* handle strings that contain CHR\$(0), and SQL_TraceStrOLE *can*. SQL_TraceStrOLE has the additional advantage of being very slightly faster, so it should be used whenever possible. SQL_TraceStr is provided primarily for languages that do not support Windows OLE strings. (Both Visual Basic and PowerBASIC do support OLE strings.)

When the trace mode is turned on with the SQL_Trace function, SQL Tools creates a trace file which contains the names all of the SQL Tools functions that are used, including the parameters that are passed to them, the values that they return, and any errors that are detected. It's possible, however, that it will still be difficult to troubleshoot a problem because you can't "see" the variables in *your* program.

You can add strings like "PROBLEM HERE?" (or anything else) to the trace file by using the SQL_TraceStr and SQL_TraceStrOLE functions. *If the trace mode is turned on* when the function is executed, the string will be added to the trace file. (You can also use the SQL_TraceSInt function to add *numeric* values to the trace file.)

The SQL_TraceStr and SQL_TraceStrOLE functions automatically use the

`SQL_TextStr` function to convert the `sString$` parameter into a printable form, so you don't have to worry about accidentally adding a character like `CHR$(26)`, which many editors recognize as an end-of-file marker, to your trace file.

Diagnostics

None.

Example

```
SQL_TraceStrOLE "X& VALUE =" + Str$(X&)
```

Driver Issues

None.

Speed Issues

Using the Trace Mode can *significantly* slow down your program.

See Also

Error/Trace Family

SQL_UnbindCol

Summary

Unbinds one column (or all columns, or all Long columns) of a result set.

Twin

SQL_UnbindColumn

Family

Result Column Binding Family

Availability

Standard and Pro

Warning

You should not attempt to use this function with the Microsoft Visual FoxPro ODBC driver. See **Driver Issues** below.

Syntax

```
lResult& = SQL_UnbindCol(lColumnNumber&)
```

Parameters

lColumnNumber&

The number of a result column, between one (1) and the number returned by the SQL_ResColCount function. You can also use the value ALL_COLS or ALL_LONG_COLS. See **Remarks** below for more information.

Return Values

This function will return SUCCESS or SUCCESS_WITH_INFO if the requested unbind operation is successful, or an Error Code if it is not.

Remarks

See Result Column Binding for background information.

It is not *usually* necessary, but it is possible to un-bind a column of a result set, i.e. to eliminate the relationship between a result column and its SQL Tools memory buffer.

If you use a number for *lColumnNumber&*, that column will be unbound. If the column is not bound when the function is used, ERROR_COL_NOT_BOUND will be returned.

If you use ALL_COLS, all *currently-bound* columns will be unbound. No Error Code will be returned if some result columns are not bound when the function is used.

If you use ALL_LONG_COLS, all *currently-bound* SQL_LONGVARCHAR, SQL_LONGVARBINARY, and SQL_wLONGVARCHAR columns will be unbound. No Error Code will be returned if some Long result columns are not bound when the function is used.

It is not necessary for your programs to use the SQL_UnbindCol function to "manually" unbind the columns of a result set before using the SQL_CloseStmt or SQL_CloseDB function. SQL Tools automatically unbinds result columns as necessary.

Diagnostics

This function returns Error Codes, and can generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

The Microsoft Visual FoxPro ODBC Driver has been observed refusing to unbind individual result columns when the standard ODBC technique is used. SQL Tools uses the standard technique, so the `SQL_UnbindCol` function may fail unexpectedly when it is used with a FoxPro database. (The ODBC specification does not provide an alternate technique for unbinding individual columns.) An error message that says "Restricted data type attribute violation" is usually generated, with SQL State 07006.

This FoxPro ODBC driver restriction is a serious problem only if your program needs to unbind *individual* result columns. The Visual FoxPro ODBC driver is apparently not capable of doing that. In all other circumstances, SQL Tools automatically uses a "backup" technique that is guaranteed to unbind *all* of a statement's result columns at the same time, so this error message can usually be safely ignored.

Speed Issues

It is very slightly faster to avoid binding a column than to bind it (using the AutoAutoBind feature) and then to unbind it with this function.

See Also

Result Column Binding (Basic), Result Column Binding (Advanced)

SQL_UnbindColumn

Summary

Unbinds one column (or all columns, or all Long columns) of a result set.

Twin

SQL_UnbindCol

Family

Result Column Binding Family

Availability

Standard and Pro

Warning

See SQL_UnbindCol **Driver Issues**.

Syntax

```
lResult& = SQL_UnbindColumn(lDatabaseNumber&, _  
                             lStatementNumber&, _  
                             lColumnNumber&)
```

Parameters

lDatabaseNumber& and *lStatementNumber&*

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_UnbindCol for complete details.

Return Values

See twin function SQL_UnbindCol for complete details.

Remarks

Except for the *lDatabaseNumber&* and *lStatementNumber&* parameters, SQL_UnbindColumn is identical to SQL_UnbindCol. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_UnbindColumn is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_UnbindCol for complete details.

Example

None.

Driver Issues

See twin function SQL_UnbindCol.

Speed Issues

See twin function SQL_UnbindCol for complete details.

See Also

Result Column Binding (Basic), Result Column Binding (Advanced)

SQL_UniqColCount

Summary

Returns the number of columns that are used to create a unique key for a table.

Twin

SQL_UniqueColumnCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_UniqColCount(lTableNumber&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the `SQL_TblCount` function.

Return Values

If a valid *lTableNumber&* is used, and if the database is open, this function will return the number of columns that are used to create a unique key for the table.

Remarks

A *Unique Column* is a column that is used in the construction of a Unique Key. A Unique Key can be used to identify a certain row of a database, without ambiguity.

For more information, see Unique Columns.

This function returns the number of columns that you must use when constructing a Unique Key for a table. (The names of the columns can be identified with the `SQL_UniqColInfoStr` function.)

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value like "this table has one unique column". It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

See Unique Columns.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also: Table Column Info Family

SQL_UniqColInfoSInt

Summary

Provides information about a Unique Column, in numeric form.

Twin

SQL_UniqueColumnInfoSInt

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_UniqColInfoSInt(lTableNumber&, _  
                                lColumnNumber&, _  
                                lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lColumnNumber&

The number of a unique column, between one (1) and the number returned by the SQL_UniqColCount function (*not* the SQL_TblColCount function).

lInfoType&

The type of numeric information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, this function will return the requested information. Otherwise, zero (0) will be returned.

Remarks

A *Unique Column* is a column that is used in the construction of a Unique Key, which is also called a *Row ID*. A Row ID can be used to identify a certain row of a database, without ambiguity.

For more information, see Unique Columns.

Please note that not *all* of the information about a table's Unique Columns is useful in numeric form. For a list of *lInfoType&* values that can be used to obtain *string* information about a Unique Column, see SQL_UniqColInfoStr.

In order to obtain numeric information about an Index, the *lInfoType&* parameter must be one of the following values:

UNIQCOL_BUFFER_LENGTH

The buffer size of the column.

UNIQCOL_COLUMN_SIZE

The display size of the column.

UNIQCOL_DATA_TYPE

The SQL data type of the column, such as `SQL_INTEGER` or `SQL_CHAR`.
(See `SQL_UniqColInfoStr` for the datasource-dependent data type name, such as "COUNTER".)

UNIQCOL_DECIMAL_DIGITS

The decimal digits of the column.

UNIQCOL_PSEUDO_COLUMN

Indicates whether or not the column is a pseudo-column, such as an Oracle ROWID column. One of the following values will be returned:

`SQL_PC_PSEUDO`
`SQL_PC_NOT_PSEUDO`
`SQL_PC_UNKNOWN`

UNIQCOL_SCOPE

When SQL Tools requests Unique Column information for your program, it uses the default "scope" of `SQL_SCOPE_SESSION` (see below). Your program can use the `SQL_SetOptionSInt(OPT_UNIQUE_SCOPE)` function to change this default. The Unique Column information that is returned by the ODBC driver is always of equal-or-greater scope than the request. Since `SQL_SCOPE_SESSION` is the greatest scope value, unless you change the default scope, this *InfoType*& will always return `SQL_SCOPE_SESSION`.

If you change the default scope by using `SQL_SetOptionSInt`, this *InfoType*& will return the actual scope of the Unique Column. In that case, this *InfoType*& can return any of the following values:

`SQL_SCOPE_CURROW` (The Row ID is guaranteed to be valid only while positioned on that row. A later `SELECT . . . WHERE` using the Row ID may not return the row if it was updated or deleted by another transaction.)

`SQL_SCOPE_TRANSACTION` (The Row ID is guaranteed to be valid for the duration of the current transaction.)

`SQL_SCOPE_SESSION` (The Row ID is guaranteed to be valid for the duration of the session, i.e. across transaction boundaries.)

Diagnostics

This function does not return Error Codes because an Error Code like `SUCCESS_WITH_INFO` (value 1) could be confused with a legitimate return value. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Table Column Info Family

SQL_UniqColInfoStr

Summary

Provides information about a Unique Column, in string form.

Twin

SQL_UniqueColumnInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_UniqColInfoStr(lTableNumber&, _  
                               lColumnNumber&, _  
                               lInfoType&)
```

Parameters

lTableNumber&

The number of a table, between one (1) and the number returned by the SQL_TblCount function.

lColumnNumber&

The number of a unique column, between one (1) and the number returned by the SQL_UniqColCount function (*not* the SQL_TblColCount function).

lInfoType&

The type of string information that is being requested. See **Remarks** below for a complete list of valid values.

Return Values

If valid parameters are used, and if the database is open, this function will return the requested information. Otherwise, an empty string is returned.

Remarks

A *Unique Column* is a column that is used in the construction of a Unique Key, which is also called a *Row ID*. A Row ID can be used to identify a certain row of a database, without ambiguity.

For more information, see Unique Columns.

Please note that not *all* of the information about a table's Unique Columns is useful in string form. For a list of *lInfoType&* values that can be used to obtain *numeric* information about a Unique Column, see SQL_UniqColInfoSInt.

In order to obtain string information about an Index, the *lInfoType&* parameter must be one of the following values:

UNIQCOL_COLUMN_NAME

The column's name.

UNIQCOL_TYPE_NAME

The datasource-dependent data type name, such as "INTEGER" or "COUNTER".

Diagnostics

This function does not return Error Codes because it returns string values. It can, however, generate ODBC Error Messages and SQL Tools Error Messages.

Example

None.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Table Column Info Family

SQL_UniqueColumnCount

Summary

Indicates the number of Unique Columns that a table has.

Twin

SQL_UniqColCount

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_UniqueColumnCount(lDatabaseNumber&, _  
                                lTableNumber&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_UniqColCount for complete details.

Return Values

See twin function SQL_UniqColCount for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_UniqueColumnCount is identical to SQL_UniqColCount. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_UniqueColumnCount is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_UniqColCount for complete details.

Example

See twin function SQL_UniqColCount.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also

Table Column Info Family

SQL_UniqueColumnInfoSInt

Summary

Provides information about a Unique Column, in numeric form.

Twin

SQL_UniqColInfoSInt

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
lResult& = SQL_UniqueColumnInfoSInt(lDatabaseNumber&, _  
                                     lTableNumber&, _  
                                     lColumnNumber&, _  
                                     lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_UniqColInfoSInt for complete details.

Return Values

See twin function SQL_UniqColInfoSInt for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_UniqueColumnInfoSInt is identical to SQL_UniqColInfoSInt. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_UniqueColumnInfoSInt is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_UniqColInfoSInt for complete details.

Example

See twin function SQL_UniqColInfoSInt.

Driver Issues

None.

Speed Issues

See Cached Information.

See Also: Table Column Info Family

SQL_UniqueColumnInfoStr

Summary

Provides information about a Unique Column, in string form.

Twin

SQL_UniqColInfoStr

Family

Table Column Info Family

Availability

SQL Tools Pro Only (see)

Warning

None.

Syntax

```
sResult$ = SQL_UniqueColumnInfoStr(lDatabaseNumber&, _  
                                   lTableNumber&, _  
                                   lColumnNumber&, _  
                                   lInfoType&)
```

Parameters

lDatabaseNumber&

See Using Database Numbers and Statement Numbers.

All Other Parameters

See twin function SQL_UniqColInfoStr for complete details.

Return Values

See twin function SQL_UniqColInfoStr for complete details.

Remarks

Except for the *lDatabaseNumber&* parameter, SQL_UniqueColumnInfoStr is identical to SQL_UniqColInfoStr. To avoid errors when this document is updated, information that is common to both functions is not duplicated here. Only information that is unique to SQL_UniqueColumnInfoStr is shown below.

For more information about using *lDatabaseNumber&* and *lStatementNumber&* in various SQL Tools functions, please see Using Database Numbers and Statement Numbers.

Diagnostics

See twin function SQL_UniqColInfoStr for complete details.

Example

See twin function SQL_UniqColInfoStr.

Driver Issues: None.

Speed Issues: See Cached Information.

See Also

Table Column Info Family

SQL_UseDB

Summary

Specifies the Database Number that SQL Tools should use for all "abbreviated" functions.

Twin

None.

Family

Use Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_UseDB(lDBNumber&)
```

Parameters

lDBNumber&

A database number between one (1) and the *lMaxDatabaseNumber&* value that was specified with the `SQL_Initialize` function.

Return Values

If a valid Database Number is specified, `SUCCESS` will be returned. Otherwise, `ERROR_BAD_PARAM_VALUE` will be returned. If you are certain that only valid values will be used, it is safe to ignore the return value of this function.

Remarks

Please see Using Database Numbers and Statement Numbers for a complete discussion of this function.

Diagnostics

This function returns Error Codes and can generate SQL Tools Error Messages.

Example

```
SQL_UseDB 2
```

Driver Issues

None.

Speed Issues

None.

See Also

`SQL_UseStmt`
`SQL_UseDBStmt`

SQL_UseDBStmt

Summary

Specifies the Database Number and Statement Number that SQL Tools should use for all "abbreviated" functions.

Twin

None.

Family

Use Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_UseDBStmt(lDBNumber&, _  
                        lStmtNumber&)
```

Parameters

See `SQL_UseDB` and `SQL_UseStmt` for complete information.

Return Values

If a valid parameters are specified, `SUCCESS` will be returned. Otherwise, `ERROR_BAD_PARAM_VALUE` will be returned. If you are certain that only valid values will be used, it is safe to ignore the return value of this function.

Remarks

This function is simply a combination of the `SQL_UseDB` and `SQL_UseStmt` functions. Please refer to those Reference Guide entries for complete information.

Diagnostics

This function returns Error Codes and can generate SQL Tools Error Messages.

Example

```
SQL_UseDBStmt 1,2
```

Driver Issues

None.

Speed Issues

None.

See Also

Using Database Numbers and Statement Numbers

SQL_UseStmt

Summary

Specifies the Statement Number that SQL Tools should use for all "abbreviated" functions.

Twin

None.

Family

Use Family

Availability

Standard and Pro

Warning

None.

Syntax

```
lResult& = SQL_UseStmt(lStmtNumber&)
```

Parameters

lStatementNumber&

A statement number between one (1) and the *lMaxDatabaseNumber&* value that was specified with the `SQL_Initialize` function. Under some circumstances, you may use statement number zero (0). (See Statement Zero Operation.)

Return Values

If a valid Statement Number is specified, `SUCCESS` will be returned. Otherwise, `ERROR_BAD_PARAM_VALUE` will be returned. If you are certain that only valid values will be used, it is safe to ignore the return value of this function.

Remarks

Please see Using Database Numbers and Statement Numbers for a complete discussion of this function.

Diagnostics

This function returns Error Codes and can generate SQL Tools Error Messages.

Example

```
SQL_UseStmt 2
```

Driver Issues

None.

Speed Issues

None.

See Also

Abbreviated Functions

Appendix A: SQL Statement Syntax

VERY IMPORTANT NOTE: This Appendix is intended to describe only the *minimum* SQL syntax that *all* ODBC drivers support. If you are writing an Interoperable Application, you should limit yourself to the use of this syntax, plus the additional syntax (if any) that is common to *all* of the ODBC drivers that you will be using. For complete information about the syntax that your ODBC driver accepts, you will need to acquire additional, driver-specific reference materials.

VERY IMPORTANT NOTE: This Appendix is not intended to be a comprehensive SQL tutorial. Many, many fine books have been written on this topic, and a detailed discussion of the SQL language is well beyond the scope of this document.

There are six basic SQL statements. Before using any of them, please see Basic SQL Syntax Rules, which apply to *all* of the following statement types.

CREATE TABLE (see) is used to add a new table to a database.

DROP TABLE (see) is used to delete an existing table from a database.

INSERT INTO (see) is used to add rows to a table.

DELETE FROM (see) is used to delete rows from a table.

UPDATE (see) is used to change values in existing rows.

SELECT (see) is used to retrieve data from a database.

In addition, *CALL* (see) can be used to execute Stored Procedures that contain any of the six basic statement types.

Also see Appendix C: ODBC Scalar and Aggregate (Set) Functions.

You should note that other SQL statements *may* be supported by your ODBC driver, such as *ALTER TABLE*, *CREATE INDEX* and *DROP INDEX*, *CREATE VIEW* and *DROP VIEW*, and *GRANT* and *REVOKE*. But these statements are not part of the minimum ODBC syntax, and some ODBC drivers may not support them, so they are not covered here.

You may also notice that certain relatively common syntax elements are not included here, such as the *SELECT* statement's *GROUP BY*, *HAVING*, *UNION* and *JOIN* clauses. Again, these keywords are not part of the minimum ODBC syntax, and some ODBC drivers *may* not support them, so they are not covered here.

For complete information about the syntax that your ODBC driver accepts, you will need to acquire additional, driver-specific reference materials.

Basic SQL Syntax Rules

Whenever you use *any* SQL statement, in addition to using the correct statement syntax you must always remember to adhere to certain general rules.

Special Characters

Some variations of the SQL language use a semicolon (;) at the end of each SQL statement. Semicolons are *not* a part of the ODBC specification for single statements, and we do not recommend their use except when they are *required* to separate the individual statements in a batch of SQL statements.

The * abbreviation (the "star" or "asterisk" character) can be used in some SQL statements, to mean "all". For example, `SELECT * FROM MYTABLE` means "select all of the columns in the table called MYTABLE". This practice is discouraged, however, because not all ODBC drivers recognize it, and some drivers use the star character for other purposes.

PLEASE NOTE: The star character is used in many of the `SELECT` examples in this document, to make them more concise and easier to understand. This should not be interpreted as a "recommended practice".

If an identifier such as a column name or table name contains a space (or any other "special" character), you must place the appropriate delimiters around the identifier. For example, if a column has the name...

ZIP CODE

...with a space between the words, you must place the appropriate quotes around it, like this...

```
SELECT * FROM MYTABLE WHERE 'ZIP CODE' = 48070
```

Otherwise, if the statement looked like this...

```
SELECT * FROM MYTABLE WHERE ZIP CODE = 48070
```

...the ODBC driver might not be able to interpret the statement correctly. *As a general rule, it is considered to be bad practice to use spaces in identifier names.* And don't use underscores either! (See Wildcards below).

You might have noticed that a single quote (') was used instead of a double quote ("). You should *always* use the quotation character that is returned by the `SQL_DBInfoStr(DB_IDENTIFIER_QUOTE_CHAR)` function. The character can vary from driver to driver, but is *usually* the single quote.

It is very likely that you will also need to use "literal values" in your SQL statements. Consider the following statement...

```
SELECT * FROM MYTABLE WHERE NAME = YOUR NAME HERE AND COUNTER <> 0
```

You must, of course, include the appropriate quotes around the string value, like this...

```
SELECT * FROM MYTABLE WHERE NAME = 'YOUR NAME HERE' AND COUNTER <> 0
```

If you don't, the ODBC driver may reject the statement or look for a row where the NAME column's value is the string "YOUR NAME HERE AND COUNTER <> 0".

You will probably also need to use literal values that actually *contain* the single quote character. For example, consider this statement:

```
SELECT * FROM MYTABLE WHERE NAME = 'O'Malley'
```

If you use a statement like that, the ODBC driver will get confused about where the literal string starts and stops. The standard solution for this is to use *two* single quotes, like this...

```
SELECT * FROM MYTABLE WHERE NAME = 'O''Malley'
```

... to tell the ODBC driver that that is a *literal* single-quote character, not a delimiter:

IMPORTANT NOTE: That's not a double-quote character (ASCII 34) that's two single quote characters (ASCII 39).

(Certain Windows function -- and therefore certain SQL Tools functions -- actually require the use of *four* single-quote characters to denote a literal character, but *this is not one of them*. Using four would result in *two* literal characters being used, because each *pair* would be interpreted as a literal character.)

The second single-quote character is only temporary. For example, if you use something like 'O''Malley' in an *UPDATE* statement, only one single-quote will actually be inserted into your database.

PowerBASIC programmers can use the *REPLACE* function to perform this operation, but be sure that you don't replace *every* single-quote in your SQL statement with two. Use two single quotes only when a single quote needs to appear *inside* a string that is quoted with single quotes.

Certain numeric values may need special delimiters as well. For example...

```
SELECT * FROM MYTABLE WHERE OFFSET = 12
```

If you intend "12" to be a decimal (base ten) value, then you do not need a delimiter. But if you intend 12 to be a hex value (base sixteen, like the BASIC notation &h12), you would need to add the appropriate prefix:

```
SELECT * FROM MYTABLE WHERE OFFSET = 0x12
```

The string "0x" (zero-x) is a common numeric prefix, but each *data type* can have its own literal prefix *and* suffix. You can determine which delimiters to use for each column *type* by using the *SQL_DBDataTypeInfoStr(DTYPE_LITERAL_PREFIX)* and *DTYPE_LITERAL_SUFFIX* functions.

You must also be careful when using certain characters in identifier names, such as these characters:

~ @ # \$ % ^ & * _ - + = \ } { " ' ; : ? / > < ,

The *SQL_DBInfoStr(DB_SPECIAL_CHARACTERS)* function can be used to obtain a string that contains the special characters that a database uses. (The string above was generated by Microsoft Access 97. Extra spaces were added to make it more readable here.)

Certain characters (such as quotes and question marks) should *never* be used in identifier names, and certain others have special meanings when they are used in identifier names.

Wildcard Characters

Most databases recognize certain "wildcard characters" or "search pattern" strings.

The % character (the percent sign) is often used as an "any string" wildcard, so if you used the string x% for an identifier, it would be interpreted as "any identifier that starts with x". A SQL statement like this...

```
SELECT * FROM MYTABLE WHERE MYCOLUMN = ABC%
```

...would mean "select all rows where the MYCOLUMN column contains a value that starts with the letters ABC". Using %ABC would mean "ends with ABC", and %ABC% would mean "contains ABC anywhere in the data". (Remember that % can be satisfied by an empty string, so ABC could be the first or last characters, as well as characters in the middle.) The SQL percent-sign wildcard is very similar to the DOS command-line star (*) wildcard.

The _ character (the underscore) is often used as an "any single character" wildcard, so if you used the identifier MY_TABLE it would be recognized as "any identifier that starts with MY and ends with TABLE, with one character in between". So if you happened to have tables called MY1TABLE and MY2TABLE, the SQL statement would apply to *both* of them. Fortunately for many less-than-careful programmers, it would also apply to a table called MY_TABLE, with a literal underscore character. *As a general rule, it is considered to be bad practice to use underscores in identifier names.* The SQL underscore wildcard is very similar to the DOS command-line question-mark (?) wildcard.

The Escape Character

If you *must* use a special character in an identifier name, you can use a value called the "search pattern escape string". This might be necessary, for example, if you are using a database that somebody else designed, or if you are using an Excel database (see below). You can determine the value of the escape string (which is usually a single character) by using the SQL_DBInfoStr(DB_SEARCH_PATTERN_ESCAPE) function. If, for example the backslash character (\) is returned, that means that you can use the backslash as an escape character that means "the character that follows is a literal character". If you were to use...

```
SELECT * FROM MY\_TABLE
```

...it would mean the literal value "MY_TABLE" where the underscore is *not* treated as a wildcard.

(It is possible to globally disable the wildcard functions by using a database attribute called "metadata ID", but doing so will interfere with the SQL Tools Info functions, which rely on the default attribute setting.)

Date Delimiters

Some DBMSs, such as Microsoft Access, require the use of the number-sign (#) delimiter for literal date/time values.

```
SELECT * FROM AddressBook WHERE Birthdate = #1950-01-01#
```

It is not usually required, but we recommend the use of the "descending" date format (YYYY-MM-DD) because it is unambiguous and is not affected by the runtime computer's Locale settings.

Special Microsoft Excel Characters

Microsoft Excel treats each "sheet" in a spreadsheet file as a separate table. By default, they are named SHEET1\$, SHEET2\$, and so on. This creates a problem because the dollar sign (\$) is an ODBC reserved character (see above). Most versions of Excel also require an unusual character, the *left*-apostrophe or "back-tick" character, to delimit table names which contain dollar signs. So a very simple `SELECT` statement for an Excel spreadsheet might look like this:

```
SELECT * FROM `SHEET1$`
```

Note the use of the *left*-apostrophe, not the normal right-apostrophe, around the table name.

Oddly, you must use the *normal* apostrophe for all other quoted strings, like this:

```
SELECT * FROM `SHEET1$` WHERE MYCOLUMN = 'HELLO'
```

Special Words

There are also certain *words* that can never be used as column identifiers. For example, imagine the confusion that would be caused if you named a table "SELECT"...

```
SELECT * FROM SELECT
```

You must avoid *all* of the words that are used by the SQL syntax that your ODBC driver accepts. For a list of reserved words which all ODBC drivers recognize, see Appendix B. The `SQL_DBInfoStr(DB_KEYWORDS)` function can be used to obtain a list of words that you must avoid. Here is the list that is returned by Microsoft Access 97:

```
ALPHANUMERIC, AUTOINCREMENT, BINARY, BYTE, COUNTER, CURRENCY,  
DATABASE, DATABASENAME, DATETIME, DISALLOW, DISTINCTROW, DOUBLEFLOAT,  
FLOAT4, FLOAT8, GENERAL, IEEEDOUBLE, IEEE SINGLE, IGNORE, INT,  
INTEGER1, INTEGER2, INTEGER4, LEVEL, LOGICAL, LOGICAL1, LONG,  
LONGBINARY, LONGCHAR, LONGTEXT, MEMO, MONEY, NOTE, NUMBER, OLEOBJECT,  
OPTION, OWNERACCESS, PARAMETERS, PERCENT, PIVOT, SHORT, SINGLE,  
SINGLEFLOAT, SMALLINT, STDEV, STDEVP, STRING, TABLEID, TEXT, TOP,  
TRANSFORM, UNSIGNEDBYTE, VALUES, VAR, VARBINARY, VARP, YESNO
```

Note that words like `SELECT` and `UPDATE` are not included on the list. Those words are part of the "universal SQL syntax" (see Appendix B) and may not be used as identifiers under any circumstances, so the `SQL_DBInfoStr(DB_KEYWORDS)` does not bother to return them.

CREATE TABLE

VERY IMPORTANT NOTE: This Appendix is intended to describe only the *minimum* SQL syntax that *all* ODBC drivers support. If you are writing an Interoperable Application, you should limit yourself to the use of this syntax, plus the additional syntax (if any) that is common to *all* of the ODBC drivers that you will be using. For complete information about the syntax that your ODBC driver accepts, you will need to acquire additional reference materials.

CREATE TABLE is used to add a new table to a database.

Minimum Syntax:

```
CREATE TABLE table-name (column-name data-type [,column-name data-type]...)
```

The *table-name* parameter is the name that will be used for the new table. You must then specify a *column-name* and *data-type* value for at least one column. The square [brackets] around the second set of parameters, and the ellipsis (. . .) indicate that additional columns may be specified, separated by commas.

IMPORTANT NOTE: When you are creating a table, the *data-type* string must always be one of the data type names that is returned by the `SQL_DBDataTypeInfoStr(DTYPE_NAME)` function. The ODBC driver will reject all other values.

DROP TABLE

VERY IMPORTANT NOTE: This Appendix is intended to describe only the *minimum* SQL syntax that *all* ODBC drivers support. If you are writing an Interoperable Application, you should limit yourself to the use of this syntax, plus the additional syntax (if any) that is common to *all* of the ODBC drivers that you will be using. For complete information about the syntax that your ODBC driver accepts, you will need to acquire additional reference materials.

DROP TABLE is used to delete an existing table from a database.

Minimum Syntax:

DROP TABLE table-name

The *table-name* parameter is the name of the table that is to be deleted.

WARNING: Once a table has been dropped it cannot be restored.

INSERT INTO

VERY IMPORTANT NOTE: This Appendix is intended to describe only the *minimum* SQL syntax that *all* ODBC drivers support. If you are writing an Interoperable Application, you should limit yourself to the use of this syntax, plus the additional syntax (if any) that is common to *all* of the ODBC drivers that you will be using. For complete information about the syntax that your ODBC driver accepts, you will need to acquire additional reference materials.

`INSERT INTO` is used to add rows to a table.

Minimum Syntax:

```
INSERT INTO table [(column [, column]...)] VALUES (value[, value]... )
```

The *table* parameter is the name of the table into which the row is to be inserted.

After the `INTO` keyword, you should use a list of one or more column names, and after the `VALUES` keyword, a list with an equal number of values. In other words, the column name immediately after `INTO` will be given the first value after the word `VALUES`, the second column name after the word `INTO` will be given the second value after `VALUES`, and so on.

You may have noticed the square [brackets] around the column-identifier list. If you omit the column list and simply use...

```
INSERT INTO table VALUES (value[, value]... )
```

...and if you are careful to specify the values in the "natural" order of the table, the statement will be accepted. However this is usually considered to be bad practice because if the table's layout is changed, it will break your program.

If a value is not assigned to a column, the column's default value (if any) will be used.

If a column does not have a default value, and if the column allows Null values, and either **1)** the value list contains a blank entry for the column (two commas with no value in between), or **2)** a "natural order" value list does not contain an entry for the column because the list is too short, or **3)** a list of columns does not contain the name of the column, then the Null value will be assigned to the column.

DELETE FROM

VERY IMPORTANT NOTE: This Appendix is intended to describe only the *minimum* SQL syntax that *all* ODBC drivers support. If you are writing an Interoperable Application, you should limit yourself to the use of this syntax, plus the additional syntax (if any) that is common to *all* of the ODBC drivers that you will be using. For complete information about the syntax that your ODBC driver accepts, you will need to acquire additional reference materials.

`DELETE FROM` is used to delete rows from a table.

Minimum Syntax:

```
DELETE FROM table [WHERE search-condition]
```

The *table* parameter is the name of the table from which the row(s) should be deleted.

WARNING: If no `WHERE` clause is specified, all of the table's rows will be deleted.

If a `WHERE` clause is specified, only the rows that match the *search-condition* will be deleted. For example...

```
DELETE FROM MYTABLE WHERE MYCOLUMN = 'DELETE ME' AND OURCOLUMN <>
'SAVE ME'
```


UPDATE

VERY IMPORTANT NOTE: This Appendix is intended to describe only the *minimum* SQL syntax that *all* ODBC drivers support. If you are writing an Interoperable Application, you should limit yourself to the use of this syntax, plus the additional syntax (if any) that is common to *all* of the ODBC drivers that you will be using. For complete information about the syntax that your ODBC driver accepts, you will need to acquire additional reference materials.

UPDATE is used to change the values in existing rows.

Minimum Syntax:

```
UPDATE table SET column = {expression/NULL} [, column =  
{expression/NULL}]... [WHERE search-condition]
```

The *table* parameter is the table which contains the rows that are to be updated. You must specify at least one *SET*, where *column* is the name of the column to be updated and *expression* is the value that the column should be given. If the column accepts Null values, in place of *expression* you may use the keyword *NULL* (without quotes).

As indicated by the square [brackets] and the ellipsis (. . .), you may optionally use more than *SET* expression, in order to change more than one column's value. If you use more than one, you should only use the word *SET* once, followed by a comma-separated list of *column = expression/NULL* strings.

The optional *WHERE* clause can be used to specify the row(s) that should be updated. If it is not used, all rows will be updated. If *WHERE* is used, you can specify a list of conditions to specify a single row or a group of rows. For example...

```
UPDATE MYTABLE SET MYCOLUMN = 'OK' WHERE OURCOLUMN = 'UNKNOWN' AND  
THEIRCOLUMN = '' AND YOURCOLUMN = 17
```

It is very common to use a single, unique column (or a small set of columns that make up a Unique Key) in the *WHERE* clause. For example...

```
UPDATE MYTABLE SET MYCOLUMN = 'OK' WHERE COUNTER = 12345
```

SELECT

VERY IMPORTANT NOTE: This Appendix is intended to describe only the *minimum* SQL syntax that *all* ODBC drivers support. If you are writing an Interoperable Application, you should limit yourself to the use of this syntax, plus the additional syntax (if any) that is common to *all* of the ODBC drivers that you will be using. For complete information about the syntax that your ODBC driver accepts, you will need to acquire additional reference materials.

SELECT is used to retrieve data from a database. Unlike other SQL statements, the *SELECT* statement produces a "result set" that contains zero or more rows of data.

Minimum syntax:

```
SELECT [DISTINCT] column-list FROM table-list [WHERE search-condition] [ORDER BY sort-spec [, sort-spec]]
```

The shortest possible *SELECT* statement has this form:

```
SELECT column-list FROM table-list
```

The *column-list* parameter specifies the names of the columns that you want the result set to include. After the *SELECT* keyword, you may optionally use the * (star) wildcard character if you want the result set to contain *all* of the columns in the table. This practice is discouraged because if the table's design is changed, it will probably break your program. It is usually better to specify the exact list of columns that you want to be included in the result set, in the order that you want them, separated by commas.

The *table-list* parameter specifies the names of the tables that contain the columns in *column-list*. If more than one table is listed, their names should be separated by commas.

DISTINCT

The optional *DISTINCT* keyword is used to eliminate duplicate rows from a result set.

You can include the *DISTINCT* keyword after the word *SELECT* if you want the result set to contain only "distinct" values. In other words, if a result set created by a SQL statement *without* *DISTINCT* contained the following rows...

```
SMITH  
JONES  
PUBLIC  
SMITH  
SMITH  
DOE  
SMITH
```

...then adding the *DISTINCT* keyword would produce this:

```
SMITH  
JONES  
PUBLIC  
DOE
```

If you do *not* want a *DISTINCT* result set you may use the optional keyword *ALL*, but we do not recommend it because **1)** *ALL* is the default behavior so using the keyword doesn't really do anything, and **2)** using the *ALL* keyword can cause confusion with the * (star) wildcard, which stands for "all columns". The SQL statement *SELECT ALL * FROM MYTABLE* could be read aloud as "select all...all from MYTABLE".

WHERE

The optional *WHERE* clause can be used to specify that only rows that contain certain values should be included in the result set. (You can think of the *column-list* parameter as controlling the "width" of the result set, and the *WHERE* clause as controlling the "height".) For example...

```
SELECT MYCOLUMN, YOURCOLUMN FROM MYTABLE WHERE MYCOLUMN <> YOURCOLUMN  
AND THEIRCOLUMN = 1
```

You can use eleven different types of comparisons in *WHERE* clauses (see below) but *not all data types support all types of comparisons*. The exact types of comparisons that can be performed on a given column are determined by the column's data type, and can be checked with the `SQL_DBDataTypeInfoInt(DTYPE_SEARCHABLE)` function.

Another factor that must be considered is whether or not a *WHERE* comparison is case-sensitive. Again, this is determined by the column's data type, and can be checked with the `SQL_DBDataTypeInfoInt(DTYPE_CASE_SENSITIVE)` function.

Please note that you will *usually* be able to use the *NOT* operator when specifying *WHERE* comparisons, but *NOT* is *not* part of the official ODBC minimum syntax.

Here are the eleven basic types of comparisons that you can use in *WHERE* clauses:

Relational: = <> < > => <=

BETWEEN Example: *...WHERE MYCOLUMN BETWEEN 1 AND 10*

LIKE Example: *...WHERE LASTNAME LIKE 'Jo%'* would return the rows where the LASTNAME column contains values that start with "Jo", like Jones, Johnson, Joker, and so on. The percent-sign (%) wildcard means "any string", and the underscore (_) wildcard means "any single character". *Some ODBC drivers may use alternate wildcard characters.*
IMPORTANT NOTE: *LIKE* comparisons are usually the slowest types of *WHERE* clauses, and the most resource-intensive.

IN Example: *...WHERE LASTNAME IN ('Smith','Jones','Public')* would return the rows where the LASTNAME column matched one of the values in the comma-separated list.

NULL Example: *...WHERE MIDDLENAME = NULL*

EXISTS This keyword is used to determine whether or not a particular row exists in a table. It returns a True or False value, and it is generally used with *AND* and another condition.

ORDER BY

The optional *ORDER BY* clause can be used to sort the rows of the result set into a certain

order. One or more *sort-spec* parameters can be specified, separated by commas. A *sort-spec* consists of either a column name or a result column number, and it can be followed by an optional *ASC* or *DESC* keyword to specify an Ascending or Descending sort.

CALL

If your ODBC driver supports Stored Procedures, the *call* keyword is *usually* used to execute them. (Not all drivers use *call*; see **IMPORTANT NOTE** below.)

The basic ODBC syntax is:

```
[?=] call procname([parameter],[parameter]...)]
```

The *?=* at the beginning of the syntax represents an optional Bound Statement Output Parameter. (If you use this option, you must use the `SQL_BindParam` function to bind the placeholder to one of your program's variables.)

The *call* keyword (which must be in lower case letters) is followed by *procname*, which is the name of the Stored Procedure.

If the Stored Procedure requires one or more parameters, they should follow the name of the procedure. If there are two or more, they must be separated by commas. The standard *?* marker may be used, if you wish to use a Bound Statement Input Parameter. (In fact, some ODBC drivers *require* that you use bound parameters with Stored Procedures.)

IMPORTANT NOTE: Some ODBC drivers require you to use a datasource-dependent syntax to execute Stored Procedures. For example, Oracle databases require something like this...

```
sStmt$ = "BEGIN" + _  
        CHR$(13) + _  
        "procname(param,param,etc.);" + _  
        CHR$(13) + _  
        "END;"  
  
SQL_Stmt SQL_STMT_IMMEDIATE, sStmt$
```

Other DBMSs require the use of the words `EXECUTE` or `RUN`.

You should consult your database and/or ODBC driver documentation for the Stored Procedure syntax that your driver requires. Because the syntax for Stored Procedures has not been standardized, it is beyond the scope of this document.

Appendix B: ODBC Reserved Words

The following words have special meaning to all ODBC drivers and should not be used as identifiers (table names, column names, etc.). For additional words that a specific ODBC driver reserves, use the `SQL_DBInfoStr(DB_KEYWORDS)` function.

ABSOLUTE, ACTION, ADA, ADD, ALL, ALLOCATE, ALTER, AND, ANY, ARE, AS, ASC, ASSERTION, AT, AUTHORIZATION, AVG

BEGIN, BETWEEN, BIT, BIT_LENGTH, BOTH, BY

CASCADE, CASCADED, CASE, CAST, CATALOG, CHAR, CHARACTER, CHARACTER_LENGTH, CHAR_LENGTH, CHECK, CLOSE, COALESCE, COLLATE, COLLATION, COLUMN, COMMIT, CONNECT, CONNECTION, CONSTRAINT, CONSTRAINTS, CONTINUE, CONVERT, CORRESPONDING, COUNT, CREATE, CROSS, CURRENT, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER, CURSOR

DATE, DAY, DEALLOCATE, DEC, DECIMAL, DECLARE, DEFAULT, DEFERRABLE, DEFERRED, DELETE, DESC, DESCRIBE, DESCRIPTOR, DIAGNOSTICS, DISCONNECT, DISTINCT, DOMAIN, DOUBLE, DROP

ELSE, END, END-EXEC, ESCAPE, EXCEPT, EXCEPTION, EXEC, EXECUTE, EXISTS, EXTERNAL, EXTRACT, FALSE

FETCH, FIRST, FLOAT, FOR, FOREIGN, FORTRAN, FOUND, FROM, FULL

GET, GLOBAL, GO, GOTO, GRANT, GROUP

HAVING, HOUR

IDENTITY, IMMEDIATE, IN, INCLUDE, INDEX, INDICATOR, INITIALLY, INNER, INPUT, INSENSITIVE, INSERT, INT, INTEGER, INTERSECT, INTERVAL, INTO, IS, ISOLATION

JOIN

KEY

LANGUAGE, LAST, LEADING, LEFT, LEVEL, LIKE, LOCAL, LOWER

MATCH, MAX, MIN, MINUTE, MODULE, MONTH

NAMES, NATIONAL, NATURAL, NCHAR, NEXT, NO, NONE, NOT, NULL, NULLIF, NUMERIC

OCTET_LENGTH, OF, ON, ONLY, OPEN, OPTION, OR, ORDER, OUTER, OUTPUT, OVERLAPS

PAD, PARTIAL, PASCAL, POSITION, PRECISION, PREPARE, PRESERVE, PRIMARY, PRIOR, PRIVILEGES, PROCEDURE, PUBLIC

READ, REAL, REFERENCES, RELATIVE, RESTRICT, REVOKE, RIGHT, ROLLBACK, ROWS

SCHEMA, SCROLL, SECOND, SECTION, SELECT, SESSION, SESSION_USER, SET,
SIZE, SMALLINT, SOME, SPACE, SQL, SQLCA, SQLCODE, SQLERROR, SQLSTATE,
SQLWARNING, SUBSTRING, SUM, SYSTEM_USER

TABLE, TEMPORARY, THEN, TIME, TIMESTAMP, TIMEZONE_HOUR,
TIMEZONE_MINUTE, TO, TRAILING, TRANSACTION, TRANSLATE, TRANSLATION,
TRIM, TRUE

UNION, UNIQUE, UNKNOWN, UPDATE, UPPER, USAGE, USER, USING

VALUE, VALUES, VARCHAR, VARYING, VIEW

WHEN, WHENEVER, WHERE, WITH, WORK, WRITE

YEAR

ZONE

Appendix C: ODBC Scalar And Aggregate (Set) Functions

Depending on the ODBC version (2.0 or 3.x) that they support, various ODBC drivers support many different built-in functions.

IMPORTANT NOTE: These are functions that can be used *in SQL statements*. They are not SQL Tools functions that you can use in BASIC source code.

A *scalar* function operates much like a BASIC function. For example, the `LCASE` string function can be used in a SQL statement to convert a string to lower case, and the `ROUND` function can be used to round a numeric value. (BASIC programmers should note that the parameters that the various functions require are not necessarily the same as the functions that you're used to, and the some functions have different names. Instead of `INSTR`, for instance, you will have to use the `LOCATE` string function in SQL statements.)

An *aggregate* function is very different from a string or numeric scalar function. It is a function that can be used in a SQL statement to force it to return a single value that represents the entire result set. For example, the `AVG` function can be used in a SQL statement to return a single value that represents the average value in a particular column of a result set.

The various ODBC functions are divided into categories based on their types, and the parameters that they require, so you may need to check two or more different lists to find the function that you are looking for.

Aggregate Functions

String Functions

Numeric Functions

Time/Date/Interval Functions

System Functions

Explicit Data Type Conversion

ODBC Aggregate Functions

You can determine which aggregate functions are supported by your ODBC driver **1)** experimentally, or **2)** by examining the return value of `SQL_DBInfoUInt(DB_AGGREGATE_FUNCTIONS)`.

`AVG()`

Returns the average of all of the values in a numeric column, i.e. the sum of all of the values, divided by the number of values. Rows with Null values are ignored completely. They are not added to the sum, and they are not counted in the number of values. (This function cannot be used with string columns.)

Example:

```
SELECT AVG(SALARY) FROM PAYROLL
```

...would produce a single-row result set containing a number that represents the average SALARY value in the PAYROLL table.

Example with subquery:

```
SELECT NAME FROM PAYROLL WHERE SALARY > (SELECT AVG(SALARY) FROM PAYROLL)
```

The `SELECT AVG...` statement would be executed first, and an average salary value would be obtained. Then the main statement would be executed using that value, and a result set would be produced that contained the NAME values of all employees that have a SALARY value greater than the average.

`COUNT()`

Returns the number of rows in a result set.

Example:

```
SELECT COUNT(*) FROM EMPLOYEES WHERE AGE > 18
```

...would produce a single-row result set containing a numeric value that indicates the number of rows in the EMPLOYEES table where the AGE column has a value greater than 18.

`MAX()`

Returns the maximum value in a column of a result set.

Example:

```
SELECT MAX(AGE) FROM EMPLOYEES
```

...would return the largest value in the AGE column of the EMPLOYEES table, i.e. the age of the oldest employee (or employees).

Example with subquery:

```
SELECT NAME FROM EMPLOYEES WHERE AGE = (SELECT MAX(AGE) FROM  
EMPLOYEES)
```

The *SELECT AGE . . .* subquery would be executed first, and the age of the oldest employee(s) would be determined. Then the main query would be executed using that value, and a result set containing the *NAME* value of all of the employees with that age value would be produced.

MIN()

Works just like *MAX()*, except that it produces a minimum value instead of a maximum value.

Example:

```
SELECT MIN(NAME) FROM EMPLOYEES
```

...would produce a single-row result set that contained the "minimum name" in the *EMPLOYEES* table, i.e. the name with the lowest alphabetical-sorting value.

SUM()

Returns the sum of the values in a numeric column. (This function cannot be used with string columns.)

Example:

```
SELECT SUM(SALARY) FROM PAYROLL
```

...would produce a single-row result set containing the sum of the values in the *SALARY* column of the *PAYROLL* table, i.e. the "total payroll" value.

ODBC String Functions

You can determine which string functions are supported by your ODBC driver **1)** experimentally, or **2)** by examining the return value of `SQL_DBInfoUInt(DB_STRING_FUNCTIONS)` and `SQL_DBInfoUInt(DB_SQL92_STRING_FUNCTIONS)`.

`ASCII(string_exp)`

Returns the ASCII code value of the leftmost character of `string_exp` as an integer.

`BIT_LENGTH(string_exp)`

ODBC 3.x+ ONLY: Returns the length of `string_exp` in bits.

`CHAR(code)`

Returns the character that has the ASCII code value specified by `code`. The value of `code` must be between 0 and 255; otherwise, the return value is datasource-dependent.

`CHAR_LENGTH(string_exp)` and
`CHARACTER_LENGTH (string_exp)`

ODBC 3.x+ ONLY: If `string_exp` is of a character data type, these functions both return the length of `string_exp` in characters. Otherwise, they return the length in bytes of the string expression (i.e. the smallest integer that is not less than the number of bits divided by 8). Also see `LENGTH` below.

`CONCAT(string_exp1, string_exp2)`

Returns a character string that is the result of adding `string_exp2` to the end of `string_exp1`. If Null values are involved, the resulting string is driver-dependent. See `SQL_DBInfoUInt(DB_CONCAT_NULL_BEHAVIOR)` for more information.

`DIFFERENCE(string_exp1, string_exp2)`

Returns an integer value that indicates the difference between the `SOUNDEX` values for `string_exp1` and `string_exp2`. See `SOUNDEX` below.

`INSERT(string_exp1, start, length, string_exp2)`

Returns a character string where `length` characters have been deleted from `string_exp1` beginning at `start`, and where `string_exp2` has been inserted into `string_exp1`, beginning at `start`.

`LCASE(string_exp)`

Returns a copy of `string_exp` with all uppercase characters converted to lowercase.

LEFT(string_exp, count)

Returns the left-most count characters of string_exp.

LENGTH(string_exp)

Returns the number of characters in string_exp, excluding trailing blanks. Also see CHAR_LENGTH above.

LOCATE(string_exp1, string_exp2 [, start])

Returns the starting position of the first occurrence of string_exp1 within string_exp2. The search begins with the first character of string_exp2 unless the optional start argument is specified. If start is specified, the search begins with the specified character position. If string_exp1 is not found within string_exp2, the value zero (0) is returned.

If an ODBC driver is only capable of using the LOCATE function with the string_exp1, string_exp2, and start parameters, the SQL_DBInfoUInt (SQL_STRING_FUNCTIONS) function will return SQL_FN_STR_LOCATE.

If an ODBC driver cannot use the start parameter, SQL_FN_STR_LOCATE_2 will be returned.

If an ODBC driver is capable of using the LOCATE function with or without the start parameter, both SQL_FN_STR_LOCATE and SQL_FN_STR_LOCATE_2 will be returned.

LTRIM(string_exp)

Returns the characters of string_exp, with leading blanks removed.

OCTET_LENGTH(string_exp)

ODBC 3.x+ ONLY: Returns the length of string_exp in bytes. The result is the smallest integer not less than the number of bits divided by 8.

POSITION(character_exp1 IN character_exp2)

ODBC 3.x+ ONLY: Returns the position of character_exp1 in the character_exp1.

REPEAT(string_exp, count)

Returns a character string that is composed of string_exp repeated count times.

REPLACE(string_exp1, string_exp2, string_exp3)

Searches string_exp1 for occurrences of string_exp2, and replaces them with string_exp3.

`RIGHT(string_exp, count)`

Returns the right-most `count` characters of `string_exp`.

`RTRIM(string_exp)`

Returns `string_exp` with trailing blanks removed.

`SOUNDEX(string_exp)`

Returns a datasource-dependent string that represents the sound of the words in `string_exp`. For example, SQL Server returns a 4-digit `SOUNDEX` code; Oracle returns a phonetic representation of each word.

`SPACE(count)`

Returns a string consisting of `count` spaces.

`SUBSTRING(string_exp, start, length)`

Returns a string that is derived from `string_exp` beginning at the character position specified by `start`, for `length` characters.

`UCASE(string_exp)`

Returns a copy of `string_exp` with all lowercase characters converted to uppercase.

ODBC Numeric Functions

`ABS(numeric_exp)`

Returns the absolute value of `numeric_exp`.

`ACOS(float_exp)`

Returns the arccosine of `float_exp` as an angle (in radians).

`ASIN(float_exp)`

Returns the arcsine of `float_exp` as an angle (in radians).

`ATAN(float_exp)`

Returns the arctangent of `float_exp` as an angle (in radians).

`ATAN2(float_exp1, float_exp2)`

Returns the arctangent of the `x` and `y` coordinates, specified by `float_exp1` and `float_exp2`, as an angle (in radians).

`CEILING(numeric_exp)`

Returns the smallest integer greater than or equal to `numeric_exp`.

`COS(float_exp)`

Returns the cosine of `float_exp`, where `float_exp` is an angle (in radians).

`COT(float_exp)`

Returns the cotangent of `float_exp`, where `float_exp` is an angle (in radians).

`DEGREES(numeric_exp)`

Returns the number of degrees converted from `numeric_exp` radians.

`EXP(float_exp)`

Returns the exponential value of `float_exp`.

`FLOOR(numeric_exp)`

Returns the largest integer less than or equal to `numeric_exp`.

`LOG(float_exp)`

Returns the natural logarithm of `float_exp`.

`LOG10(float_exp)`

Returns the base 10 logarithm of `float_exp`.

MOD(integer_exp1, integer_exp2)

Returns the remainder (modulus) of integer_exp1 divided by integer_exp2.

PI()

Returns the constant value of *pi* as a floating point value.

POWER(numeric_exp, integer_exp)

Returns the value of numeric_exp to the power of integer_exp.

RADIANS(numeric_exp)

Returns the number of radians in numeric_exp degrees.

RAND([integer_exp])

Returns a random floating point value, using the optional integer_exp as the seed value.

ROUND(numeric_exp, integer_exp)

Returns numeric_exp rounded to integer_exp places right of the decimal point. If integer_exp is negative, numeric_exp is rounded to integer_exp places to the left of the decimal point.

SIGN(numeric_exp)

Returns the sign of numeric_exp. If numeric_exp is less than zero, negative one (-1) will be returned. If numeric_exp equals zero, 0 will be returned. If numeric_exp is greater than zero, positive one (1) will be returned.

SIN(float_exp)

Returns the sine of float_exp, where float_exp is an angle (in radians).

SQRT(float_exp)

Returns the square root of float_exp.

TAN(float_exp)

Returns the tangent of float_exp, where float_exp is an angle (in radians).

TRUNCATE(numeric_exp, integer_exp)

Returns numeric_exp truncated to integer_exp places right of the decimal point. If integer_exp is negative, numeric_exp is truncated to integer_exp places to the left of the decimal point.

ODBC Time/Date/Interval Functions

`CURRENT_DATE`

ODBC 3.x+ only: Returns the current date.

`CURRENT_TIME[(precision)]`

ODBC 3.x+ ONLY: Returns the current local time. The `precision` argument determines the precision of the fractional seconds of the returned value.

`CURRENT_TIMESTAMP[(precision)]`

ODBC 3.x+ ONLY: Returns the current local date and local time as a timestamp value. The `precision` argument determines the precision of the fractional seconds of the timestamp.

`CURDATE()`

Returns the current date.

`CURTIME()`

Returns the current local time.

`DAYNAME(date_exp)`

Returns a character string containing the datasource-specific day-name for the day portion of `date_exp`. For example, Sunday through Saturday for a Datasource that uses English, or Domingo through Sabado for a Datasource that uses Spanish.

`DAYOFMONTH(date_exp)`

Returns the day of the month based on the month field in `date_exp` as an integer value in the range of 1 to 31.

`DAYOFWEEK(date_exp)`

Returns the day of the week based on the week field in `date_exp` as an integer value in the range of 1 to 7, where 1 represents Sunday.

`DAYOFYEAR(date_exp)`

Returns the day of the year based on the year field in `date_exp` as an integer value in the range of 1 to 366.

`EXTRACT(extract-field FROM extract-source)`

ODBC 3.x+ ONLY: Returns the `extract-field` portion of the `extract-source` value. The `extract-source` argument is a datetime or interval expression. The `extract-field` argument can be one of the following keywords: YEAR, MONTH, DAY, HOUR, MINUTE, or SECOND.

The precision of the value returned by `EXTRACT` is driver-defined. The scale is 0 unless `SECOND` is specified, in which case the scale is not less than the fractional seconds precision of the `extract-source` field.

`HOUR(time_exp)`

Returns the hour based on the hour field in `time_exp` as an integer value in the range of 0 to 23.

`MINUTE(time_exp)`

Returns the minute based on the minute field in `time_exp` as an integer value in the range of 0 to 59.

`MONTH(date_exp)`

Returns the month based on the month field in `date_exp` as an integer value in the range of 1 to 12.

`MONTHNAME(date_exp)`

Returns a character string containing the datasource-specific month-name for the month portion of `date_exp`. (For example, January through December for a Datasource that uses English, or Enero through Diciembre for a Datasource that uses Spanish.)

`NOW()`

Returns the current date and time as a timestamp value.

`QUARTER(date_exp)`

Returns the quarter in `date_exp` as an integer value in the range of 1 to 4, where 1 represents January 1 through March 31.

`SECOND(time_exp)`

Returns the second based on the seconds field in `time_exp` as an integer value in the range of 0 to 59.

`TIMESTAMPADD(interval, timestamp_exp1, timestamp_exp2)` and
`TIMESTAMPDIFF(interval, timestamp_exp1, timestamp_exp2)`

Returns the timestamp calculated by adding or subtracting `integer_exp` intervals of type `interval` to `timestamp_exp`. Valid values of `interval` are the following keywords...

`SQL_TSI_FRAC_SECOND`, `SQL_TSI_SECOND`, `SQL_TSI_MINUTE`,
`SQL_TSI_HOUR`, `SQL_TSI_DAY`, `SQL_TSI_WEEK`, `SQL_TSI_MONTH`,
`SQL_TSI_QUARTER`, `SQL_TSI_YEAR`

If `timestamp_exp` is a time value and `interval` specifies days, weeks, months, quarters, or years, the date portion of `timestamp_exp` is set to the current date before calculating the resulting timestamp.

If `timestamp_exp` is a date value and `interval` specifies fractional seconds, seconds, minutes, or hours, the time portion of `timestamp_exp` is set to 0 before calculating the resulting timestamp.

You can determine which intervals are supported by a database by using the `SQL_DBInfoUInt (SQL_TIMEDATE_ADD_INTERVALS)` function.

`WEEK (date_exp)`

Returns the week of the year based on the week field in `date_exp` as an integer value in the range of 1 to 53.

`YEAR (date_exp)`

Returns the year based on the year field in `date_exp` as an integer value. The range is datasource-dependent.

ODBC System Functions

`DATABASE ()`

Returns the name of the database.

`IFNULL(exp, value)`

If `exp` is null, `value` is returned. If `exp` is not null, `exp` is returned. The data type of `value` must be compatible with the data type of `exp`.

`USER ()`

Returns the user name.

Explicit Data Type Conversion

The syntax of the `CONVERT` function, which is used for all data-type conversions, is:

```
CONVERT(value_exp, data_type)
```

The `data_type` parameter must be a valid SQL data type, such as `SQL_INTEGER` or `SQL_CHAR`.

The ODBC driver will reject any conversion which, although legal in the ODBC syntax, is not supported by the Datasource. You can use the various `SQL_DBInfoUInt (DB_CONVERT_)` functions to determine whether or not a particular conversion is supported by a database.

Appendix D: SQL Tools Error Codes

An Error Code is a numeric value that correspond to a type of error.

In addition to the ODBC Error Codes that can be generated by your ODBC driver, SQL Tools can generate its own set of Error Codes. Here is an alphabetical list of all of the SQL Tools Error Codes, and their general meanings. The exact meaning of an Error Code is determined by the function that returns it.

(If you're curious why the various numeric values are so large, read the last portion of this Appendix.)

PLEASE NOTE THAT THESE ERROR CODES ARE LISTED IN ALPHABETICAL ORDER, NOT NUMERIC ORDER.

ERROR_ADVISORY (value 999000049)

This Error Code is very similar to the ODBC Error Code `SUCCESS_WITH_INFO`. It means that SQL Tools was able to perform the requested function, but that you may need to know about (and act upon) a certain detail.

If this message is generated by the `SQL_OpenDatabase` function, it means that either **1)** the value that your program is using for `OPT_MAXCONNSTRINGLEN` is too small, and that the final connection string was truncated, or **2)** the `SQL_OpenDatabase` function determined that your ODBC driver cannot perform "Fetch Scroll" operations, so your use of `SQL_Fetch` will be limited to `NEXT_ROW` operations.

If this message is generated by the `SQL_OpenStatement` function, it means that one of the statement attributes (modes) that your program attempted to set was rejected by the ODBC driver. This condition will almost always be accompanied by an Error Message from the ODBC driver, describing the exact error.

If this message is generated by the `SQL_StatementMode` function, it means that your program used the function to set a statement mode *while a statement was open*. This message is intended to remind you that mode changes do not take effect until a statement is opened, so the changes will not take effect until the current statement is closed.

If this message is generated by the `SQL_Diagnostic` function, it means that the database or statement for which you requested diagnostic information is no longer open, or that you used an incorrect database number or statement number. In effect, this message means "no diagnostic information is available", which *may or may not* mean that diagnostic information was *lost* because a database or statement was closed before this function was used.

If this message is generated by the `SQL_Bookmark` function, it means that the function did everything that it was supposed to do, but it was not able to retrieve a bookmark. Many different things can cause this "general failure"; please see Bookmarks for more information.

ERROR_BAD_PARAM_VALUE (value 999000030)

Many different SQL Tools functions can return this Error Code. It simply means that

a parameter with an invalid value was passed to the function.

ERROR_CANT_BE_DONE (value 999000048)

Your program attempted to do something that is not possible, such as use the `SQL_Initialize` function to re-initialize SQL Tools, or change the name of the SQL Tools Trace File while the Trace Mode was turned on.

The most common reason for this Error Message is the use of a SQL Tools Info function that is not supported by the ODBC driver. For example, if you attempt to use one of the `SQL_TablePrivilege` functions with the Microsoft Access 97 ODBC driver, you will receive this error because the driver does not support privilege functions. You can avoid these errors by using the `SQL_FuncAvail` function before attempting to use an Info function that you are not certain is supported.

ERROR_COL_NOT_BOUND (value 999000038)

Your program attempted to access a column that has not been bound, using a function which requires a bound column. For example, if you use the `SQL_Init` function and (thereby) use the default *MaxColNumber* value of 32, and if a table contains more than 32 columns, SQL Tools will be unable to bind all of the columns in your result set. (The solution is to either **1**) reduce the number of columns that are produced by your SQL statement, or **2**) use `SQL_Initialize` instead of `SQL_Init`, and use a sufficiently large value for *MaxColNumber*.) Another example: If you attempt to use the `SQL_UnbindCol` function twice on the same column, the second use of the function will return this Error Code because the first use will unbind the column, and the second will not be able to unbind it.

ERROR_DB_NOT_CLOSED (value 999000032)

Your program attempted to open a database number that was already open, and the `OPT_AUTOCLOSE_DB` option was disabled.

ERROR_DB_NOT_OPEN (value 999000031)

Your program attempted to use a database number that was not open.

ERROR_DLL_NOT_AUTHORIZED (value 999000000)

This Error Code is returned by the `SQL_Init` and `SQL_Initialize` functions if you attempt to use them before you have used the `SQL_Authorize` function. See *Four Critical Steps For Every SQL Tools Program* for more information.

ERROR_FIRST_BAS_ERROR (value 999001001 to 999001999)

A runtime error occurred inside the SQL Tools DLL. You can obtain a BASIC Error Code by subtracting 999001000 from the Error Code value. For example, if you specify an invalid file name for the SQL Tools Trace File and then attempt to turn the Trace Mode on, you will receive an Error Code that is equal to 999001000 plus the BASIC ERR value of the error.

ERROR_STMT_NOT_CLOSED (value 999000035)

Your program attempted to open a statement number that was already open, and the `OPT_AUTOCLOSE_STMT` option was disabled.

ERROR_STMT_NOT_OPEN (value 999000034)

Your program attempted to use a statement number that was not open, and the OPT_AUTOOPEN_STMT option was disabled.

ERROR_STMT_NOT_PREPARED (value 999000036)

1) Your program attempted to use `SQL_Stmt(SQL_STMT_EXECUTE)` before it used `SQL_Stmt(SQL_STMT_PREPARE)` to prepare a statement, *or* **2)** it used `SQL_Stmt(SQL_STMT_PREPARE)` but then *closed* the statement before using `SQL_Stmt(SQL_STMT_EXECUTE)`. This is very similar to an `ERROR_STMT_NOT_OPEN` Error Code.

ERROR_TOO_MANY (value 999000047)

A SQL Tools function encountered a number that was too large for it to handle, such as a situation where more than 9,999 tables are found by the `SQL_TblCount` function, or more than 9,999 datasources are found by the `SQL_DataSourceCount` function. Because many different SQL Tools functions use the various "get info:" functions (internally), this error code can be returned by a wide variety of functions. If you encounter this error, you may need to use a different value for `SQL_SetOptionSInt(OPT_MAX_ITEM_NUMBER)`.

ERROR_UNKNOWN_ERROR (value 999999999)

A SQL Tools function encountered an error that it could not identify. This can happen when Windows or the ODBC subsystem reports an error but does not provide any details.

ERROR_USER_CANCEL (value 999000045)

An operation failed because the user selected a Cancel button. For example, the `SQL_OpenDB` function can fail if it displays a dialog box to allow the user to select a database, and the user selects the Cancel button.

A Frequently Asked Question:

*Whoa! Why are these Error Code numbers so **LARGE**?*

The Answer:

Microsoft made us do it.

Well, they didn't actually write us a letter or anything. They just made rules for 32-bit Windows programs that require the use of certain number ranges. Basically, Microsoft has reserved all of the "reasonable" numbers for itself, so that Windows can report a wide variety of error numbers when it has problems.

There are well over 4,000,000,000 (4 *billion*) possible Error Codes. Microsoft has reserved 50% of those for non-Microsoft use. Any Error Code that has Bit 29 *set* is defined as an "Application-Defined Error Code", and if Bit 29 is *not* set, it's a Microsoft Error Code.

The lowest-value range of numbers that has Bit 29 set is...

536,870,912 to 1,073,721,824

SQL Tools could have easily used the numbers that start with 1,000,000,000 so that they'd be easy to read, but we figured that you'd rather use that range for *your* programs, since it's the "best" range of number that the Microsoft rules have to offer.

So we chose the range 999,000,000 to 999,999,999. All SQL Tools Error Codes -- in fact the Error Codes from *all* Perfect Sync software development products -- fall into that range.

If a SQL Tools function reports an Error Code that is not in that range, you can count on the fact that it came from an ODBC driver (or that Windows reported a Windows Error) and that SQL Tools is simply "passing the number along".

Appendix E: ODBC Error Codes

In addition to the SQL Tools Error Codes (see) that can be generated by SQL Tools function, your ODBC driver can generate its own set of Error Codes. Here is a list of all of the ODBC Error Codes, and their general meanings. The exact meaning of an Error Code is determined by the function that returns it.

`SUCCESS` (value 0)

This Error Code means "zero errors". It is returned by functions that do not encounter any errors.

`SUCCESS_WITH_INFO` (value 1)

"Success With Info" means that the requested operation was performed, but that a condition was detected that your program may or may not need to address.

For example, if you use the `SQL_Fetch` function to retrieve a row of data from a result set, and if one of the columns contains data that is too long to fit in the buffer that is provided, a `SUCCESS_WITH_INFO` message will be generated. The Error Text that is associated with this error will contain a string like "Data right-truncated". In other words, the Fetch operation was successful and the data in the buffer is valid, but it is not complete.

That's typical of a `SUCCESS_WITH_INFO` message. They all mean "It worked, but..."

`SQL_STILL_EXECUTING` (value 2)

This value, which can be returned by the `SQL_AsyncStatus` function, indicates that an asynchronous SQL statement has not yet finished executing.

`SQL_ERROR` (value *negative* 1)

This is the Error Code that corresponds to a generic "something went wrong and the function failed" condition. The Error Text that is associated with the error will contain specific information about the failure.

`SQL_INVALID_HANDLE` (value *negative* 2)

This Error Code indicates that an invalid handle value was passed to an ODBC function. Unless your program is using ODBC handles directly (via the `SQL_h` functions), this Error Code indicates a serious error inside the SQL Tools DLL. Please contact Perfect Sync Technical Support if this Error Code is reported and your program is *not* using the `SQL_h` functions.

`SQL_NEED_DATA` (value 99)

This Error Code indicates that more data is needed, such as when parameter data is required before a SQL statement can be processed.

`SQL_NO_DATA` (value 100)

This Error Code is returned by the `SQL_Fetch` and `SQL_FetchRel` functions when

they fail because there was no data (or no *more* data) to be retrieved from a result set. This is a perfectly normal condition and does not represent a serious error (at least in *most* cases).

Appendix F: SQL States (ODBC Error Messages)

A "SQL State" value is a five-character string that corresponds to a specific condition. Most SQL States represent error conditions, but some are simply "advisory" messages that are associated with the `SUCCESS_WITH_INFO` Error Code.

While Error Text strings can vary from ODBC driver to ODBC driver, SQL State strings are *supposed* to be highly consistent. **But these numbers and strings are not strictly required by the ODBC specification, and not all ODBC drivers use them in this way. ODBC drivers are free to define their own SQL States, so if your program returns a SQL State value that is not on the list below, you should consult your driver and/or DBMS documentation.**

SQL Tools generates SQL State strings that start with the # symbol, to help you distinguish between SQL Tools Error Messages and ODBC Error Messages. (The # prefix can be changed with the `SQL_SetOptionSInt (OPT_SQLSTATE_PREFIX)` function.

Here is an alphabetical list of fairly common SQL State strings, and their basic meanings. If a description says X **or** Y, then the ODBC documentation lists two different descriptions. If a description says "ODBC 2.0 terminology: see 3.x State", that means that the SQL State value has been "mapped" to a new value in ODBC 3.x.

01000

General warning. This SQL State is usually associated with a `SUCCESS_WITH_INFO` Error Code.

01001

Cursor operation conflict. A positioned update or delete operation was performed, and either **1)** no rows or **2)** more than one row was affected. See Positioned Operations.

01002

Disconnect error. An error occurred during the `SQL_CloseDB` process, but the database-disconnect operation was successful.

01003

NULL value eliminated in set function. A SQL statement contained an Aggregate Function (such as `AVG` or `MAX`, but not `COUNT`), and Null values were eliminated before the function was applied.

01004

String data, right truncated. The right-most character(s) of a string value were cut off by the ODBC driver, usually because a memory buffer was not large enough to hold the entire value. Also see 22001. If a fetch operation generates this error, you may need to use the `SQL_LongResCol` function to retrieve the data from one or more columns.

01006

Privilege not revoked. A SQL statement contained a `REVOKE` statement, but the user

did not have the specified privilege.

01007

Privilege not granted. A SQL statement contained a [GRANT](#) statement, but the user could not be granted the specified privilege.

01S00

Invalid connection string attribute. The SQL_OpenDB function detected that a connection string (or DSN file) contained an invalid keyword, or a keyword without a value, but the driver was able to connect to the data source.

01S01

Error in row. An error occurred while fetching one or more rows from the database.

01S02

Option value changed. An invalid value (or a valid value which conflicted with another value) was submitted to the ODBC driver, and it automatically substituted a valid, non-conflicting value.

01S03

ODBC 2.0 terminology: see 3.x State 01001

01S04

ODBC 2.0 terminology: see 3.x State 01001

01S06

Attempt to fetch before result set returned first rowset. The rowset requested with SQL_Fetch or SQL_FetchRel overlapped the start of the result set, *and* one of the following four things was true: **1)** FETCH_PREV was used, the current position was beyond the first row, and the number of the current row was less than or equal to the rowset size, or **2)** FETCH_PREV was used, the current position was beyond the end of the result set, and the rowset size was greater than the result set size, or **3)** a Relative Fetch with a negative offset was performed, and the absolute value of the offset was less than or equal to the rowset size or **4)** a fetch-by-row-number was performed, the row number was negative, and the absolute value of the row number was greater than the result set size but less than or equal to the rowset size. Please refer to the Microsoft ODBC Software Developer Kit for more information.

01S07

Fractional truncation. The fractional part of a value (such as a SQL_DECIMAL, SQL_NUMERIC, or SQL_TIMESTAMP value) was truncated.

01S08

Error saving File DSN. A connection string contained the SAVEFILE keyword, but the file was not saved. (The Microsoft ODBC Software Developer Kit says "the FILEDSN keyword", but we believe that to be incorrect.)

01S09

Invalid keyword. A connection string contained `SAVEFILE` but not `DRIVER` or `FILEDSN`.

07002

COUNT field incorrect. A SQL statement contained one or more bound parameters, and the `SQL_BindParam` function was not used correctly. For example, this error could be generated if a statement contained one "?" placeholder but `SQL_BindParam` was used to bind two parameters.

07005

Prepared statement not a cursor-specification. A SQL statement did not return a result set, so there were no columns for the `SQL_ResColInfoStr` or `SQL_ResColInfoSInt` function to provide information about.

07006

Restricted data type attribute violation. Two incompatible data types were specified for an ODBC operation. For example, this error might be generated if you attempted to bind a bookmark column to a data buffer with a SQL Data Type that was not compatible with bookmarks. (Also see `SQL_UnbindCol` **Driver Issues**.)

07009

Invalid descriptor index. An invalid column number or parameter number was used. For example, you may have used a column number that is larger than the number of columns in a result set, or you may have specified column zero when the `STMT_ATTR_USE_BOOKMARKS` attribute was not set to the correct value.

07S01

Invalid use of default parameter. A parameter value which was set with `SQL_BindParam` was `SQL_DEFAULT_PARAM`, and the corresponding parameter **1)** did not have a default value or **2)** was not a parameter for an ODBC procedure invocation. See the Microsoft ODBC Software Developer Kit for more information.

08001

Client unable to establish connection. The ODBC driver was unable to establish a connection with the data source.

08002

Connection name in use. You attempted to set the `DB_ATTR_ODBC_CURSORS` attribute, but the driver was already fully connected to the data source. See `SQL_OpenDatabase1`.

08003

Connection does not exist. A SQL Tools function used a database handle that was not open. Please report this problem to Perfect Sync Technical Support.

08004

Server rejected the connection. The datasource rejected the requested connection.

08007

Connection failure during transaction. A database connection failed during the execution of the `SQL_EndTrans` function, and it can't be determined whether or not the requested `TRANS_COMMIT` or `TRANS_ROLLBACK` occurred before the failure.

08S01

Communication link failure. The communication link between the driver and the datasource failed before a SQL Tools function finished the requested operation.

21S01

Insert value list does not match column list. The number of parameters in an `INSERT` statement did not match the number of columns in the table that was named in the statement.

21S02

Degree of derived table does not match column list. Either **1)** a `BULK_UPDATE` or `SET_UPDATE` operation was requested, but no columns were updatable because all columns were unbound, read-only, or the value of the Indicator was `SQL_COLUMN_IGNORE`, or **2)** a SQL statement contained a `CREATE VIEW` statement and the number of names that were specified was not the same degree as the derived table defined by the query specification, or **3)** a SQL statement contained a `CREATE VIEW` statement and the unqualified column list (the number of columns specified for the view in the column-identifier arguments of the SQL statement) contained more names than the number of columns in the derived table defined by the query-specification argument of the SQL statement.

22001

String data, right truncated The right-most character(s) of a string value were cut off by the ODBC driver, usually because a memory buffer was not large enough to hold the entire value. Also see 01004. If a fetch operation generates this error, you may need to use the `SQL_LongResCol` function to retrieve the data from one or more columns.

22002

Indicator variable required but not supplied. An Indicator variable that was required for an operation was set to a Null pointer value. This usually indicates the incorrect use of a column-binding or parameter-binding function.

22003

Numeric value out of range **or** ODBC 2.0 terminology. If the later, see 3.x State HY019

22005

ODBC 2.0 terminology: see 3.x State 22018

22007

Invalid datetime format. A timestamp, time, or date value had an invalid format or an illegal sub-value (such as an illegal seconds value like 99).

22008

Datetime field overflow **or** ODBC 2.0 terminology: If the later, see 3.x State 22007

22012

Division by zero.

22015

Interval field overflow. A Interval value contained an invalid value, or a valid value that could not be converted to the requested data type for some other reason.

22018

Invalid character value for cast specification. An invalid literal value was used, based on the value's data type.

22019

Invalid escape character. Escape characters must be exactly one character long.

22025

Invalid escape sequence. The character following an escape character was not a percent sign (%) or an underscore (_).

22026

String data, length mismatch. A string length was specified for an operation, and too few characters were supplied.

23000

Integrity constraint violation. A Null value was supplied for a column that was defined as NOT NULL, or a duplicate value was supplied for a column that must contain unique values, or some other integrity constraint was violated.

24000

Invalid cursor state **or** ODBC 2.0 terminology: If the later, see 3.x State 07005

25000

Invalid transaction state. There was a transaction in progress when the SQL_CloseDB function was used. When this happens, the transaction remains active.

25S01

Transaction state unknown. One or more transactions failed, and the outcome is unknown.

25S02

Transaction is still active. The ODBC driver was not able to guarantee that all work in a global transaction could be completed, and the transaction is still active.

25S03

Transaction is rolled back. The ODBC driver was not able to guarantee that all work in a global transaction could be completed, and the transaction active was rolled back.

28000

Invalid authorization specification. The user that was identifier in a connection string, or the authorization string, or both, violated restrictions defined by the Datasource

34000

Invalid cursor name. An invalid name was specified for a cursor (invalid characters, too long, etc.), or a cursor name was used which did not correspond to an open cursor.

37000

ODBC 2.0 terminology: see 3.x State 42000

3C000

Duplicate cursor name. The specified cursor name already exists. Cursor names must be unique.

3D000

Invalid catalog name. An invalid catalog name was used.

3F000

Invalid schema name. An invalid schema name was used.

40001

Serialization failure. A transaction was rolled back because of a resource deadlock with another transaction.

40002

Integrity constraint violation. A `TRANS_COMMIT` operation was requested, but the transaction was rolled back because the commitment of changes caused a violation of an integrity constraint.

40003

Statement completion unknown. A database connection failed during the execution of a function, and the state of the transaction cannot be determined.

42000

Syntax error or access violation. An operation was not performed because of invalid SQL statement syntax or a lack of the necessary permissions.

42S01

Base table or view already exists. A SQL statement contained a *CREATE TABLE* or *CREATE VIEW* statement, and the specified table or view already exists.

42S02

Base table or view not found. The specified table or view does not exist.

42S11

Index already exists. A SQL statement contained a *CREATE INDEX* statement and the specified index already existed.

42S12

Index not found. The specified index does not exist.

42S21

Column already exists. A SQL statement contained an *ALTER TABLE* statement and the column specified in the *ADD* clause is not unique, or it identifies a column that already exists in the table.

42S22

Column not found. The specified column does not exist.

44000

WITH CHECK OPTION violation. A SQL statement contained an *INSERT* or *UPDATE* statement which was supposed to be performed on a viewed table or a table derived from the viewed table which was created by specifying *WITH CHECK OPTION*, such that one or more rows affected by the statement will no longer be present in the viewed table.

70100

ODBC 2.0 terminology: see 3.x State HY018

HY000

General error. An error occurred for which no specific SQL State is defined.

HY001

Memory allocation error. The ODBC driver or Driver Manager was unable to allocate memory for the requested operation.

HY003

Invalid application buffer type. A data type that is invalid, or is invalid for the requested operation, was specified.

HY004

Invalid SQL data type. The data type that was specified is not a valid SQL Data Type or a valid datasource-dependent data type.

HY007

Associated statement is not prepared. This SQL State is related to descriptors and should never be reported by a SQL Tools application.

HY008

Operation canceled. `SQL_StmtCancel` was used to cancel an operation.

HY009

Invalid use of null pointer. A Null pointer was used in a situation where Null pointers are not allowed.

HY010

Function sequence error. This error message means "steps were performed in the wrong order". Since SQL Tools handles most sequence-oriented operations automatically, this error should usually not be reported by SQL Tools programs.

HY011

Attribute cannot be set now. Certain database, statement, and environment attributes can be set only before or after certain other operations have been performed. For example, many database attributes must be set between `SQL_OpenDatabase1` and `SQL_OpenDatabase2`. (These restrictions are often datasource-dependent.)

HY012

Invalid transaction operation code. This SQL State should never be reported by SQL Tools programs.

HY013

Memory management error. This error usually relates to low-available-memory conditions.

HY014

Limit on the number of handles exceeded. An ODBC-driver-defined limit was reached, such as the maximum number of databases or statements that can be open

at the same time.

HY015

No cursor name available. A cursor name was requested for a statement that did not have an open cursor.

HY016

Cannot modify an implementation row descriptor. This SQL State is related to descriptors and should never be reported by a SQL Tools application.

HY017

Invalid use of an automatically allocated descriptor handle. This SQL State is related to descriptors and should never be reported by a SQL Tools application.

HY018

Server declined cancel request. The server refused to perform a `SQL_StmtCancel` operation.

HY019

Non-character and non-binary data sent in pieces. The `SQL_LongParam` function was used incorrectly, to send data that was not in a character (string) or binary data format.

HY020

Attempt to concatenate a null value. The `SQL_LongParam` function was used to send data in pieces, and one of the pieces was a Null value.

HY021

Inconsistent descriptor information. This SQL State is related to descriptors and should never be reported by a SQL Tools application.

HY024

Invalid attribute value. An invalid attribute value was specified.

HY090

Invalid string or buffer length. An invalid string length or buffer length (such as zero, a negative number, or a value that is invalid for a certain circumstance) was specified.

HY091

Invalid descriptor field identifier. This SQL State is related to descriptors and should never be reported by a SQL Tools application.

HY092

Invalid attribute/option identifier. This is roughly equivalent to a SQL Tools `ERROR_BAD_PARAM_VALUE` message. It means that an invalid value was specified

for an ODBC function, and SQL Tools wasn't able to detect the error.

HY095

Function type out of range. An invalid parameter was used for the `SQL_FuncAvail` function.

HY096

Invalid information type. An invalid parameter was used for the `SQL_DBInfoStr` or `SQL_DBInfoUInt` function.

HY097

Column type out of range. This error should never be reported by a SQL Tools program.

HY098

Scope type out of range. This error should never be reported by a SQL Tools program.

HY099

Nullable type out of range. This error should never be reported by a SQL Tools program.

HY100

Uniqueness option type out of range. This error should never be reported by a SQL Tools program.

HY101

Accuracy option type out of range. This error should never be reported by a SQL Tools program.

HY103

Invalid retrieval code. This error should never be reported by a SQL Tools program.

HY104

Invalid precision or scale value. The value specified for the Column Size or Decimal Digits was outside the range of values supported by the data source for a column of the SQL data type that was specified.

HY105

Invalid parameter type. This error should never be reported by a SQL Tools program.

HY106

Fetch type out of range. This error should never be reported by a SQL Tools program.

HY107

Row value out of range. An invalid row value was specified.

HY109

Invalid cursor position. The requested operation could not be performed at the current cursor location.

HY110

Invalid driver completion. This error should never be reported by a SQL Tools program.

HY111

Invalid bookmark value. An invalid bookmark was used.

HYC00

Optional feature not implemented. This error message indicates that your ODBC driver does not support the requested operation.

HYT00

Timeout expired.

HYT01

Connection timeout expired.

IM001

Driver does not support this function.

IM002

Datasource name not found and no default driver specified.

IM003

Specified driver could not be loaded.

IM004

Driver's SQLAllocHandle on SQL_HANDLE_ENV failed

IM005

Driver's SQLAllocHandle on SQL_HANDLE_DBC failed

IM006

Driver's SQLSetConnectAttr failed.

IM007

No Datasource or driver specified; dialog prohibited.

IM008

Dialog failed.

IM009

Unable to load translation DLL.

IM010

Datasource name too long.

IM011

Driver name too long.

IM012

DRIVER keyword syntax error.

IM013

Trace file error.

IM014

Invalid name of File DSN.

IM015

Corrupt file Datasource.

S0001 ODBC 2.0 terminology: see 3.x State 42S01
S0002 ODBC 2.0 terminology: see 3.x State 42S02
S0011 ODBC 2.0 terminology: see 3.x State 42S11
S0012 ODBC 2.0 terminology: see 3.x State 42S12
S0021 ODBC 2.0 terminology: see 3.x State 42S21
S0022 ODBC 2.0 terminology: see 3.x State 42S22
S0023 ODBC 2.0 terminology: see 3.x State 42S23
S1000 ODBC 2.0 terminology: see 3.x State HY000
S1001 ODBC 2.0 terminology: see 3.x State HY001
S1002 ODBC 2.0 terminology: see 3.x State 07009
S1003 ODBC 2.0 terminology: see 3.x State HY003
S1004 ODBC 2.0 terminology: see 3.x State HY004
S1008 ODBC 2.0 terminology: see 3.x State HY008
S1009 ODBC 2.0 terminology: see 3.x State HY009
S1009 ODBC 2.0 terminology: see 3.x State HY024
S1009 ODBC 2.0 terminology: see 3.x State HY092
S1010 ODBC 2.0 terminology: see 3.x State HY007 *and* HY010
S1011 ODBC 2.0 terminology: see 3.x State HY011
S1012 ODBC 2.0 terminology: see 3.x State HY012

S1090 ODBC 2.0 terminology: see 3.x State HY090
S1091 ODBC 2.0 terminology: see 3.x State HY091
S1092 ODBC 2.0 terminology: see 3.x State HY092
S1093 ODBC 2.0 terminology: see 3.x State 07009
S1096 ODBC 2.0 terminology: see 3.x State HY096
S1097 ODBC 2.0 terminology: see 3.x State HY097
S1098 ODBC 2.0 terminology: see 3.x State HY098
S1099 ODBC 2.0 terminology: see 3.x State HY099
S1100 ODBC 2.0 terminology: see 3.x State HY100
S1101 ODBC 2.0 terminology: see 3.x State HY101
S1103 ODBC 2.0 terminology: see 3.x State HY103
S1104 ODBC 2.0 terminology: see 3.x State HY104
S1105 ODBC 2.0 terminology: see 3.x State HY105
S1106 ODBC 2.0 terminology: see 3.x State HY106
S1107 ODBC 2.0 terminology: see 3.x State HY107
S1108 ODBC 2.0 terminology: see 3.x State HY108
S1109 ODBC 2.0 terminology: see 3.x State HY109
S1110 ODBC 2.0 terminology: see 3.x State HY110
S1111 ODBC 2.0 terminology: see 3.x State HY111
S1C00 ODBC 2.0 terminology: see 3.x State HYC00
S1T00 ODBC 2.0 terminology: see 3.x State HYT00

Appendix G: Connection String Syntax

Connection strings are made up of keyword-value pairs. An equal-sign (=) is used to separate keywords and values, and semicolons (;) are used to separate pairs.

Example Connection String:

```
DSN=SYSTEM1; UID=JOHNSMITH; PWD=HELLOWORLD
```

A connection string may contain any of the following ODBC-defined keywords: DSN, FILEDSN, DRIVER, UID, PWD, and SAVEFILE. (See below for details.)

A connection string may also include *any number* of driver-defined keywords. Because the standard DRIVER keyword does not use system information, an ODBC driver must define enough keywords to allow it to connect to a datasource using *only* the information in the connection string. *Each ODBC driver defines which keywords it requires to connect to a Datasource.*

Standard Connection String Keywords

DSN=

The name of a Datasource as returned by the `SQL_DataSourceInfoStr` function, or by the "Datasources" dialog box that can be displayed by the `SQL_OpenDB` function. The DSN= value cannot be an empty string, and should not contain leading spaces.

FILEDSN=

The name of a .DSN file from which a connection string will be built for the Datasource, i.e. a text file with the filename-extension DSN that *contains* a connection string.

DRIVER=

The description of the driver as returned by the `SQL_DriverInfoStr` function. Programs do not have to add {curly braces} around the attribute value after the DRIVER keyword unless the attribute contains a semicolon (;), in which case the braces are required.

UID=

A User ID. (The UID keyword is optional.)

PWD=

The password that corresponds to the User ID, or an empty string if there is no password for the User ID. Examples: PWD=HELLO or PWD=;. (Note: In order to keep them secret, the PWD keyword and value are never stored in a .DSN file.)

SAVEFILE=

The file name of a .DSN file in which the final connection string should be saved, if the connection is successful. The `SAVEFILE` keyword must be used in conjunction with the `DRIVER` keyword or the `FILEDSN` keyword, or both. If this is not done, the `SQL_OpenDB` function will generate a `SUCCESS_WITH_INFO` Error Message with SQL State `01S09` (Invalid keyword). If both `SAVEFILE` and `DRIVER` are used, the `SAVEFILE` keyword must appear in the connection string *before* the `DRIVER` keyword.

If any keywords are repeated in the connection string, the driver will use the value that is associated with the first occurrence of the keyword.

If the `DRIVER` and `DSN` keywords are included in the same connection string, the one that appears first will be used.

If the `FILEDSN` and `DSN` keywords are included in the same connection string, the one that appears first will be used.

The `FILEDSN` and `DRIVER` keywords, on the other hand, can be used together.

If the `FILEDSN` keyword is used, the keywords that are specified in a .DSN file will be used to create a connection string. If any keyword appears in a connection string *with* `FILEDSN`, then the keyword's value in the connection string will be used in place of the value in the file.

The default directory for saving and loading a .DSN file is a combination of the path specified by

1) The CommonFileDir registry entry in...

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`, and

2) the subdirectory `ODBC\DataSources`.

For example, if the `CommonFileDir` value in the registry was...

`C:\Program Files\Common Files`

...the default DSN directory would be...

`C:\Program Files\Common Files\ODBC\Datasources`

Keywords and Datasource names cannot contain the backslash (\) character. Keywords and attribute values which contain the characters...

`[] { } () , ; ? * = ! @`

... should be avoided.

Appendix H: Logical True And False

VERY IMPORTANT NOTE: Most SQL Tools functions that return True/False values return *Logical True* and *Logical False* according to the descriptions in this Appendix. If you use SQL Tools True/False functions with BAS_DWORD variables (which cannot accept the Logical True value of negative one) then the functions will *appear* to malfunction.

There are two different ways to look at the values of True and False.

The technical definition of False is **Zero**, and the technical definition of True is **Nonzero**. In other words, all Microsoft API functions and virtually all programming languages recognize zero as False and *everything else* as True.

Since computers use binary numbers -- ones and zeros -- it is fairly common to use zero for False and one for True. This works fairly well when all you're trying to do is specify a simple True/False value. For example, consider the following BASIC code...

```
False = 0
True  = 1
Do
    lCount& = lCount& + 1
    If lCount& = 100 Then lComplete& = True
Loop Until lComplete& = True
```

This code is very straightforward. You could also use this code...

```
False = 0
True  = 1
Do
    lCount& = lCount& + 1
    If lCount& = 100 Then lComplete& = True
Loop Until lComplete&
```

...to accomplish exactly the same thing, because the simple expression lComplete& would be evaluated by BASIC and when the value was True (nonzero) the program would exit from the loop. And you could even do this...

```
False = 0
True  = 1
Do
    lCount& = lCount& + 1
    If lCount& = 100 Then lComplete& = True
Loop While lComplete& = False
```

... and it would work fine. But there is a significant problem with a True/False system that uses one (1) for the True value. The following code will not perform the way you might expect...

```
'"broken" code...
False = 0
True  = 1
Do
    lCount& = lCount& + 1
    If lCount& = 100 Then lComplete& = True
Loop While Not lComplete&
```

This code looks like it should work, but there's a serious problem. Most computer languages uses binary ("bitwise") operations for logical operators like AND, OR and NOT. The value one (1) is evaluated as True -- remember that *all* nonzero values evaluate as True -- but here's the problem: if you do this...

```
True = 1
Print Not True
```

... you will not see zero, the value that you probably expected, you will see *negative two*. Here's the reason. If you write out the value of one (1) in binary ones and zeros, you get this...

```
0000000000000001      'fifteen zeros and a one
```

The NOT operator reverses all of the bits (see your BASIC language documentation), so NOT 1 yields this...

```
1111111111111110      'fifteen ones and a zero
```

...which evaluates to --2. **So, if you use one (1) for your True value, "NOT True" evaluates to --2, which is nonzero and therefore *also* evaluates as True.**

In the "broken" example above, while lComplete& is zero, NOT lComplete& evaluates to a nonzero value so the loop continues running. But if lComplete& is set to one (1) then NOT lComplete& *still* evaluates to a nonzero value and the loop *still* continues running.

It is possible, fortunately, to use a value for True that works "logically" in virtually all cases. Consider this...

The binary representation of False (*always* defined as zero) is sixteen zeros..

```
0000000000000000
```

If you do this...

```
False = 0
Print Not False
```

...you will see that the value negative one (-1) is displayed. This is because the binary value 1111111111111111, which is the same as NOT 0000000000000000, evaluates to negative one. (The reason that this bit pattern evaluates to --1 is pretty complicated, but if you're interested you can read about it in your BASIC documentation. Take our word for it: 1111111111111111 evaluates to negative one in *all* computer languages that use "Signed Integers", as BASIC does.)

So if you modified the "broken" example code above to use negative one for True instead of one...

```
False = 0
True = -1
Do
    lCount& = lCount& + 1
    If lCount& = 100 Then lComplete& = True
Loop While Not lComplete&
```

...it would work "logically", and the program would exit from the loop when 100 was reached.

The bottom line is that the use of `--1` for True can make your code easier to write *and* read.

There's one final "glitch" that you have to keep in mind, however. Negative one is (of course) a negative number, and *not all variable types can be used to store negative numbers*. The 32-bit BASIC variable type "Long Integer" -- which is the fastest and most efficient BASIC variable type -- *can* use negative numbers, so this is not usually a problem. But some programs and some Windows API functions use 32-bit `BAS_DWORD` variables, which are unsigned variables and won't accept negative values, so you'll be forced to use `+1` and avoid `NOT` and other "bitwise" operations.

Suggested Reading

Look at the `ISFALSE` and `ISTRUE` functions in the PowerBASIC documentation, as well as the `NOT` operator and the `IF/THEN` statement.

Appendix I: Internet Resources

The internet provides a wide variety of free and low-cost resources for SQL and ODBC programmers. Unfortunately, internet file names and locations (URLs) change almost constantly, and items are added and deleted all the time, so the most we can do here is point you in the right direction.

When it comes to reference materials, perhaps the most valuable single item is the Microsoft **ODBC Software Development Kit** (also known as the ODBC SDK), which is now part of the larger **MDAC Software Development Kit**. It is a highly technical document, and it can be very difficult to read, but it contains a wealth of information. This remarkable package, which is available at no cost, contains the most comprehensive set of *definitive* ODBC reference materials that are available. The downloadable ODBC SDK includes the ODBC.HLP file, which is a large Windows Help File (1.5 megabytes) that details every aspect of the ODBC interface. As of this writing, the ODBC SDK can be downloaded from <http://msdn.microsoft.com/library/default.asp?url=/downloads/list/dataaccess.asp>.

Another valuable Microsoft file is MDAC_TYP.EXE, which contains a complete set of Microsoft ODBC drivers that can, under certain circumstances, be redistributed with your SQL Tools applications. (See the Microsoft End User License Agreement for details.) As of this writing, this file could be downloaded from <http://msdn.microsoft.com/library/default.asp?url=/downloads/list/dataaccess.asp>.

If you are developing Microsoft Access databases, you might want to visit the Access Developer forum <http://www.microsoft.com/accessdev/>. While it is heavily focused on Access and Visual Basic, it does contain some valuable items such as <http://www.microsoft.com/accessdev/articles/solution.htm>, which is a sample database that demonstrates some advanced Access techniques.

Of course, every major database vendor has a web site, such as <http://www.oracle.com> and <http://borland.com/>.

The internet can also lead you to non-electronic materials. For example, SQL Server Magazine can be found at <http://www.sqlmag.com/>, and Intelligent Enterprise (formerly DBMS Magazine and Database Programming & Design) can be found at <http://www.dbmsmag.com/index.shtml>.

Using the Alta Vista search engine to search the internet for "SQL" produced over 8 *million* pages, and "ODBC" produced well over 1.5 million.

Start surfing!

Newgroups

Several internet newsgroups are devoted to SQL and ODBC discussions. There are currently 33 groups with "SQL" in their name, 7 with "ODBC", 35 with "access", 16 with "dBase", 86 with "database"... and on and on.

We recommend the use of <http://groups.google.com/> because it allows you to search hundreds of internet newsgroups, all at the same time. It's even possible to search for things like *an entire ODBC Error Message*, and to read "threads" (groups of related newsgroup postings) from people who have received the same Error Messages. If you don't find an answer in the message archives, it's very easy to post a newsgroup question of your own

using Internet Explorer or Netscape.

PowerBASIC Resources

You can find a very helpful group of people that are doing exactly what you're doing -- using SQL Tools with a PowerBASIC compiler -- at <http://www.powerbasic.com/>. Select the "BBS" menu item, and you'll find SQL Tools being discussed in the "Third Party Addons" forum.

Appendix J: Using Bitmasked Values

A bitmasked value is an integer variable (such as a `SQL_INTEGER` or `BAS_DWORD`) that is used to store two or more different values at the same time. Each of the value's *bits* has a different meaning. For example, a 32-bit (4-byte) `BAS_DWORD` or `BAS_LONG` variable can be used to store 32 different true/false flags.

PowerBASIC programmers can use the `BIT` function to examine a bitmasked value, to find out the status (on or off) of any given certain bit.

But because it is usually just as easy (and fast) to use the bitwise `AND` operator, and because Visual Basic does not have a `BIT` function, we will focus on using `AND`. (PowerBASIC users may wish to consult the PB documentation for more information about `BIT`.)

Technically speaking, most bitmasked values are `BAS_DWORD` variables, but because not all computer languages support unsigned integers -- and because there is *no difference whatsoever* in the "bit patterns" of `BAS_LONG` and `BAS_DWORD` variables -- we will use `BAS_LONG` variables for all of the examples below.

Let's assume that a `BAS_LONG` variable called `lResult&` contains a bitmasked value, and that several "bitmask identifier" constants are provided for the bitmask. This would be the case if you were attempting to analyze the return value of a SQL Tools Info function that returns a bitmasked value. The return value of the function would contain the bitmask, and one or more constants would be described in this document.

Here's a specific example. The `SQL_DBInfoUInt` function (SQL Database Info, Unsigned Integer) can be used to obtain a value called `DB_NUMERIC_FUNCTIONS` which describes the built-in numeric functions that are supported by a database. Here are four of the twenty-four different constants that are provided in the Reference Guide for `DB_NUMERIC_FUNCTIONS`:

```
SQL_FN_NUM_ABS
SQL_FN_NUM_COS
SQL_FN_NUM_PI
SQL_FN_NUM_SIN
```

To find out whether or not a database supports the `ABS` (absolute value) function, you would open the database with `SQL_OpenDB` and then use this code:

```
'get the bitmasked value:
lResult& = SQL_DBInfoUInt(DB_NUMERIC_FUNCTIONS)

'check the SQL_FN_NUM_ABS bit:
If (lResult& AND SQL_FN_NUM_ABS) Then
    'The ABS function IS supported
Else
    'The ABS function is NOT supported
End If
```

You could also use `SQL_FN_NUM_COS` in place of `SQL_FN_NUM_ABS` to find out whether or not the `COS` (cosine) function was supported, and so on.

IMPORTANT NOTE: If you do not use the parentheses around the `AND` test, like this...

```
If (IResult& AND SQL_FN_NUM_ABS) Then
```

...*the test may not be performed correctly.* Different computer languages use slightly different notations to indicate that a bitwise comparison should be performed. Parentheses are the most common notation, but you should consult your language's documentation to make sure.

Visual Basic programmers: The VB `AND` operator is *always* a bitwise operator, so the parentheses are not necessary. (At least this is the case in Visual Basic versions 5 and 6.)

PowerBASIC programmers: You must *always* use parentheses around the `AND` test to tell PowerBASIC that you want to perform a "bitwise" operation. Without the parentheses, PowerBASIC will use "short circuit evaluation" to speed up the operation, and this will produce incorrect "bitwise" results. In fact, the line containing `SQL_FN_NUM_ABS` above will *always* return True if you forget the parentheses, regardless of the value of `IResult&`.

Appendix K: SQLSetEnvAttr

Certain Microsoft Error Messages refer to this low-level ODBC API function. For example, it is common to receive the following Error Message after you use `SQL_OpenDB` to open a database:

```
[Microsoft][ODBC Driver Manager] The driver doesn't support the
version of ODBC behavior that the application requested (see
SQLSetEnvAttr).
```

The SQL Tools equivalent of `SQLSetEnvAttr` is the `SQL_SetEnvironAttribSInt` but you will find more useful information under `SQL_Initialize` and `Error Messages After Opening a Database`.

Appendix L: Upgrading From SQL Tools Version 1 to Version 2

This Appendix covers the process of upgrading from SQL Tools Version 1 to Version 2 in two parts. The first part covers the things that you *must* do in order for your Version 1 program to work with Version 2, and the second part covers the new features that you will find in Version 2.

Upgrading Your Existing Programs

When we created SQL Tools Version 2 we tried to make it as easy as possible to upgrade your programs from SQL Tools Version 1. If you check the following items, you should find the upgrade process to be quick and painless. *Most of these items will not apply to most programs, but we strongly suggest that you review the entire list.*

1) ALL PROGRAMS: All SQL Tools Version 2 programs must use the new `SQL_Authorize` function *before* they use `SQL_Init` or `SQL_Initialize`. If you do not use `SQL_Authorize` first, the `SQL_Init` and `SQL_Initialize` functions will return the new Error Code `ERROR_DLL_NOT_AUTHORIZED`. For more information about this, see `SQL_Authorize`, SQL Tools Authorization Codes. and Four Critical Steps For Every SQL Tools Program.

2) ALL PROGRAMS: Instead of using `$INCLUDE SQL_STD.INC` or `SQL_PRO.INC`, your programs must now use `SQLT_STD.INC` or `SQLT_PRO.INC`. Note the addition of the letter T to the names. Note also that instead of several different, small INC files (separate files for different types of equates, TYPEs, declarations, etc.) everything has been consolidated into those two large files, one for SQL Tools Standard and one for SQL Tools Pro.

3) ALL PROGRAMS: SQL Tools Version 1 used a DLL file called `SQLTools.DLL` for both the Standard and Pro versions, but this sometimes caused problems when a Pro DLL was overwritten by a Standard DLL and programs unexpectedly stopped working. So Version 2 uses separate files for the Standard and Pro versions, called `SQLT_STD.DLL` and `SQLT_PRO.DLL`. When SQL Tools Version 2 was installed on your hard drive, only one of those files was installed, depending on whether you obtained a Standard or Pro license.

4) If your program uses them, you will need to change the following equate names. Some of them were changed to avoid naming conflicts with keywords that are used in various programming languages, such as `EXECUTE` and `PREPARE`. Others were changed to make them more descriptive, to make them more consistent with related values, and so on.

IMPORTANT NOTE: You really don't have to search your programs for these equates, because they will simply fail to compile if you use them. This list is provided so that when the compiler produces an "Undefined equate" error message, you will know which new equates to use.

OLD NAME	NEW NAME
-----	-----
%EXECUTE	%SQL_STMT_EXECUTE
%PREPARE	%SQL_STMT_PREPARE
%IMMEDIATE	%SQL_STMT_IMMEDIATE
%EXECDIRECT	%SQL_STMT_EXECDIRECT
%NOPROMPT	%PROMPT_TYPE_NOPROMPT
%COMPLETE	%PROMPT_TYPE_COMPLETE
%PROMPT	%PROMPT_TYPE_PROMPT
%REQUIRED	%PROMPT_TYPE_REQUIRED
%ERROR_FIRST_ST_ERROR	%ERROR_FIRST_SQLT_ERROR
%ERROR_LAST_ST_ERROR	%ERROR_LAST_SQLT_ERROR
%ERROR_FIRST_PB_ERROR	%ERROR_FIRST_BAS_ERROR
%ERROR_LAST_PB_ERROR	%ERROR_LAST_BAS_ERROR

Finally, the equates `%ON` and `%OFF` have been "retired" because **1)** they conflicted with keywords in several programming languages and **2)** they caused too much confusion with values like `%TRACE_ON`, `%LOCK_ON`, and so on, which do not always

have the same numeric values. You should now use True and False, or 1 and 0. As long as you use a value of zero (0) for %OFF and a nonzero value like one (1) or negative one (-1) for %ON, your programs will work properly. %ON and %OFF were formerly used for the SQL_DBAutoCommit and SQL_DatabaseAutoCommit functions, and for various option settings.

5) If your programs use Dates, Times, and Date-Times, the way that SQL Tools handles them has been greatly improved. Instead of returning all Dates in the format YYYY-MM-DD and all times in the format HH:MM:SS.fffffffffff, the SQL_ResColText and SQL_ResultColumnText functions now return dates and times in the Windows default format, which can vary from computer to computer depending on the system configuration. If you want the SQL_ResColText and SQL_ResultColumnText functions to use the "old" date and/or time formatting, simply add this code right after SQL_Init...

```
SQL_SetOptionStr %OPT_DATE_FORMAT, "yyyy-mm-dd"  
SQL_SetOptionStr %OPT_TIME_FORMAT, "hh:mm:ss.fffffffffff"
```

6) If your program uses the SQL_Open or DB SQL_OpenDatabase functions in a mode that displays a Connection Dialog, you should be aware that the Error Codes that are returned by those functions are now more consistent. In all cases, if the user selects Cancel or Quit, the SQL_OpenDB and SQL_OpenDatabase functions will now return %ERROR_USER_CANCEL. (Under certain circumstances, Version 1 returned %SQL_NO_DATA when the user selected Cancel.)

7) The function SQL_Error has been renamed SQL_ErrorPending. This was done to avoid a naming conflict with the %SQL_ERROR constant. (It also provides a more meaningful name for the SQL_Error function, which has caused some confusion in the past.)

8) The function SQL_Binary has been renamed SQL_BinaryStr. This was done to avoid a naming conflict with the %SQL_BINARY data type.

9) The function SQL_Text has been renamed SQL_TextStr, to keep it consistent with the newly renamed SQL_BinaryStr function. (Those two functions are often used together.)

10) If your programs use the SQL_Text function (now called SQL_TextStr), you should be aware of a new, minor limitation. The SQL_TextStr function cannot be used with strings that contain CHR\$(0). If you use a string that contains CHR\$(0) with the SQL_TextStr function, only the portion of the string before the CHR\$(0) will be included in the return value. A new function called SQL_TextStrOLE has been added to allow strings with CHR\$(0) to be used. (**Tip:** The SQL_TextStrOLE function is actually slightly *faster* than SQL_TextStr, so if you want to, simply replace *all* calls to SQL_Text with SQL_TextStrOLE.)

11) If your programs use the SQL_TraceStr function, you should be aware of a new, minor limitation. The SQL_TraceStr function can no longer be used with strings that contain CHR\$(0). If you use a string that contains CHR\$(0) with the SQL_TraceStr function, only the portion of the string before the CHR\$(0) will be placed in the Trace File. A new function called SQL_TraceStrOLE has been added to allow strings with CHR\$(0) to be added to a Trace File. (**Tip:** The SQL_TraceStrOLE function is actually slightly *faster* than SQL_TraceStr, so if you want to, simply replace all calls to SQL_TraceStr with SQL_TraceStrOLE.)

12) If your program uses the SQL_DriverInfoStr(%DRIVER_DESCRIPTION) function,

you should be aware that it now uses the Carriage Return character (`CHR$(13)`) as a delimiter instead of the "null" character `CHR$(0)`. This change does not affect `SQL_DriverInfoStr(%DRIVER_NAME)` or any of the "200+" options because they do not use delimiters.

13) If your program uses databases that contain Unicode, you will find that SQL Tools no longer auto-converts Unicode strings into ANSI strings. If you use a SQL Tools function like `SQL_ResColStr` to get data from a Unicode column of a result set, the function will now *return* a Unicode string. (This is not a concern if your program uses Access or Excel databases, because Access and Excel do not support any of the Unicode data types. But SQL Server, Oracle, and many other databases do support Unicode.)

New Features in SQL Tools Version 2

Unless otherwise noted, these features are available in both the Standard and Pro versions of SQL Tools Version 2.

1) SQL Tools Version 2 is now compatible with Microsoft Visual Basic as well as PowerBASIC. In fact, SQL Tools Version 2 can be used with C, C++, Delphi, Fortran, or any other 32-bit Windows programming language that can use standard DLLs. Now you can use SQL Tools for many more of your database projects, large and small!

2) Unlike the SQL Tools Version 1 License Agreement, the Version 2 License Agreement does not restrict your use of SQL Tools in free/demo/shareware projects. (This change does *not* affect programs that are still using SQL Tools Version 1. The new "relaxed" rules apply *only* to programs that use SQL Tools Version 2)

3) Greatly improved handling of Dates, Times, and Date-Time columns. The `SQL_ResColText` and `SQL_ResultColumnText` functions now return dates and times in a much more usable format, and you can change the default formats very easily. The new `SQL_ResColDateTime` function has been added to make accessing and formatting Date-Time columns much easier. New functions called `SQL_ResColDate`, and `SQL_ResColTime` (as well as their "verbose" counterparts) have also been added. Those functions allow you to specify the date and/or time format that is returned on a *column-by-column* basis, using standard-Windows formatting strings like "hh:mm:ss" and "ddd, MMM dd, yyyy".

4) The following new Options can be set with the `SQL_SetOptionSInt` and `SQL_SetOptionStr` functions. They affect the Date, Time, and Date-Time functions that are described above.

```
OPT_DATE_FORMAT
OPT_TIME_FORMAT
OPT_DATETIME_SEPARATOR
OPT_DATE_LOCALE
OPT_DATE_FLAGS
OPT_TIME_LOCALE
OPT_TIME_FLAGS
```

5) Improved (now fully automatic) handling of Signed and Unsigned Numeric columns. It is no longer necessary to use Manual Binding to handle tables that contain unusual numeric data types such as Signed Bytes (Tiny Integers), Unsigned Words, Unsigned Double Words, etc.

6) Improved (now fully automatic) handling of all three Unicode Data Types that are supported by ODBC-compliant databases. That means that SQL Tools is now compatible with ODBC Version 3.5, Level 2, which (as of this writing) is the current state of the art for ODBC databases.

7) The new `SQL_Test.EXE` program can be used to determine whether or not ODBC Drivers have been installed on a computer. If ODBC drivers *have* been installed, `SQL_Test` will simply launch your SQL Tools program. If they have not been installed, `SQL_Test` will display an error message (of your choice). This will result in a much "cleaner" failure if your SQL Tools program is run on an improperly configured system, because it will avoid displaying the intimidating Windows Error Message "*The dynamic link library... could not be found in the specified path...*", etc. For more information, see Appendix M: Using `SQL_Test.EXE`.

8) The new `SQL_StringToType` function can be used to assign the value of a string to a User Defined Type, such as a `SQL_TIMESTAMP`, `SQL_DATE`, or `SQL_TIME` value. (This function is provided primarily for Visual Basic programmers because PowerBASIC programmers can use the `LSet` function to perform this type of operation.)

9) SQL Tools Pro Only: In addition to the new Date-Time functions described above, SQL Tools Pro also includes new functions called `SQL_ResColDateTimePart` and `SQL_ResultColumnDateTimePart`. These functions can be used to obtain numeric values that correspond to *part* of a Date, Time, or DateTime column, such as the Month or Hour.

10) SQL Tools Pro Only: SQL Tools now supports the Asynchronous Execution of SQL Statements.

11) SQL Tools Pro Only: SQL Tools now automatically tracks the current row number of each `SELECT` statement, and your programs can obtain that value by using the `SQL_FetchPos` and `SQL_FetchPosition` functions. Two functions called `SQL_SyncFetchPos` and `SQL_SyncFetchPosition` have also been added, for special circumstances.

12) SQL Tools Pro Only: Three new function have been added to allow your SQL Tools programs to manage Data Sources much more effectively: `SQL_DataSourceAdd`, `SQL_DataSourceAdmin`, and `SQL_DataSourceModify`. When used with a Microsoft Access database, the `SQL_DataSourceModify` function can also be used to perform operations such compacting (repairing) databases and creating new databases.

13) SQL Tools Standard *and* Pro: This document has been thoroughly updated and revised, including new sections that cover the questions that Perfect Sync Technical Support receives most often.

Appendix M: Using SQL_Test.EXE

SQL_Test.EXE is a very small (12k) program that can be used to determine whether or not ODBC Drivers have been installed on a computer *before* your main program starts up.

If you have ever started a SQL Tools program on a computer where ODBC Drivers were *not* installed, then you have seen the Windows Error Message...

The dynamic link library ODBC32.DLL could not be found in the specified path..

...followed by a very long, complex-looking list of directories. To the average user, that message is very unfriendly and intimidating, but it is difficult to avoid because Windows *automatically* displays it whenever it can't find a DLL, such as those used by the ODBC system. (Visual Basic displays a similar message.)

The SQL_Test.EXE program can be used to eliminate that ugly message, and to display a message that *you* write, to explain to your users (presumably in plain, non-technical language) what they need to do. For example you could tell them to contact you, or to download and install the Microsoft MDAC package. (See Installing ODBC Drivers for more information.)

If your main program is called MyProg.EXE you would start it this way (using a Windows shortcut or a Batch File)...

```
SQL_Test MyProg.EXE
```

The SQL_Test program will start up and automatically figure out whether or not a key ODBC file called ODBC32.DLL is present on the system. (Note that SQL_Test doesn't check for a specific ODBC Driver, it simply checks to see if the "ODBC subsystem" has been installed.)

If SQL_Test determines that the ODBC subsystem *has* been installed, it will remain invisible and automatically launch MyProg.EXE. It will look as though your application had been launched directly.

If SQL_Test determines that the ODBC subsystem has *not* been installed, it will display a Message Box instead of launching your program. Here is the default message:

```
This program cannot operate unless ODBC DRIVERS are
installed on your computer. Contact the author of
this software for instructions for obtaining and
installing the necessary drivers.
```

It's very easy to change what the message box says, by placing a copy of the \SQLTOOLS\SQL_Test.TXT file (note the TXT extension) in the same directory as SQL_Test.EXE, and editing the file.

Here are the original contents of the `SQL_Test.TXT` file:

```
Message Box Title Goes Here
ODBC DRIVERS HAVE NOT BEEN INSTALLED.
```

```
Your message text starts on the second line of the file
and can fill several lines. The only limit is the maximum
size of the standard Windows Message Box.
```

```
See the SQL Tools Help File for more information.
```

The first line of the `SQL_Test.TXT` file determines the text that will be displayed in the Message Box's caption or "title bar.

The rest of the file will be displayed in the main message area of the Message Box.

Using SQL_Test in the Quiet Mode

If you launch `SQL_Test` like this:

```
SQL_Test QUIET
```

...it will not, as you might expect, attempt to launch a program called QUIET. The keyword QUIET tells `SQL_Test` that it should *not* attempt to launch a program or display a message box, it should set an ERRORLEVEL to indicate whether or not the ODBC subsystem has been installed, and then exit immediately.

If `SQL_Test QUIET` determines that the ODBC subsystem *has* been installed, it will return an ERRORLEVEL of zero (0).

If `SQL_Test QUIET` determines that the ODBC subsystem has *not* been installed, it will return an ERRORLEVEL of one (1).

The ERRORLEVEL value can be used to control how Batch Files (*.BAT files) operate.

A discussion of ERRORLEVELs and Batch Files is beyond the scope of this document. If you do not already know how to use them, we suggest that you consult the Windows documentation or a DOS manual (since Windows Batch Files are very similar to DOS Batch Files).

Appendix Z: Topics Not Covered

The following ODBC/SQL topics are supported by SQL Tools but are not thoroughly covered in this version of this document. We recommend that you consult the Microsoft ODBC Software Developer Kit for information about these topics:

- Connection Pooling

- DDL (Data Definition Language) Statements

- Outer Joins

- Interoperable Application Guidelines

The Microsoft ODBC Software Developer Kit is an excellent source of information for SQL and ODBC programmers.

A Simple Program, Step By Step

This section of this document will walk you through the basic steps that SQL Tools programs usually perform. Your programs, of course, will probably be much more complex than these simple examples.

When it is presented in electronic (Help File) form, all of the pages of this document are linked together so that you can use the >> button at the top of the screen to move from one page to the next.

Quick and Dirty: The SQL-DUMP Program

The goal of this simple program is to scan an entire SQL database table, and to create a text file that contains all of the data from the table. This is often called an "export" or "dump" operation.

The `SQL-Dump.MDB` file, which is a Microsoft Access 97 database, is provided with SQL Tools. If Microsoft Access is installed on your computer, you can use it to examine the sample database. You will see a single table called `AddressBook` with six columns called `UniqueID`, `FullName`, `StreetAddress`, `City`, `State` and `ZipCode`.

If you install the Microsoft Access ODBC driver (or if they are already installed on your system) you can compile and run the final `SQL-Dump` program. See [Installing ODBC Drivers](#) for more information.

IMPORTANT NOTE: These sample programs assume that you installed SQL Tools in the default `\SQLTOOLS\` directory. If you installed SQL Tools somewhere else, then you will be required to change *both* the sample source code files *and* the sample DSN files that are provided. For example, to compile and run the SQL-Dump program, you must change the `\SQLTOOLS\` paths in the SQL-Dump source code files (`BAS`, `FRM`, etc.) *and* in the `SQL-Dump.DSN` file. Specifically, these lines would need to be changed:

```
DefaultDir=\SQLTOOLS
DBQ=\SQLTOOLS\Samples\SQL-Dump.mdb
```

Failing to change the `DSN` file will result in a program that displays Error Messages when it is run.

[Visual Basic Users Start Here](#)

[PowerBASIC Users Start Here](#)

SQL-DUMP Visual Basic Step 1: Link SQL Tools to Your Project

This section describes the process of creating a SQL Tools program from scratch, using Microsoft Visual Basic. If you are adding SQL Tools to an existing Visual Basic program, the steps are basically the same. To make things easier, we recommend that you cut and paste the source code that is provided in the skeleton program (see below) into your existing program.

The steps below are covered in more detail in [Four Critical Steps For Every SQL Tools Program](#).

The easiest way to start writing a SQL Tools program with Visual Basic is to use the "skeleton" project that is provided here:

```
\\SQLTOOLS\\SAMPLES\\VB\\SKELETON\\SQL-SKEL.VBP
```

It contains all of the basic elements that you'll need to get started. The `SQL-Dump.FRM` file (in the same directory) includes code that looks like this:

```
Private Sub Form_Load()  
    'Skeleton for simple SQL Tools EXE programs.  
    SQL_Authorize %MY_SQLT_AUTHCODE  
    SQL_Init  
    'Add your code here...  
  
End Sub  
  
Private Sub Form_Unload(Cancel As Integer)  
    SQL_Shutdown  
End Sub
```

Let's start out by modifying the skeleton to use our sample program's name, `SQL-DUMP`.

```
Private Sub Form_Load()  
    'The SQL-DUMP sample program  
    'Add your Authorization Code here...
```

As you can see, as we modify the program's source code we will highlight the most recent changes in **bold** text. (This document does not cover things like changing the form's Caption property and other routine operations which are specific to Visual Basic programs. We assume that you already know how to do those things.)

IMPORTANT NOTE: In practice, you should always start with a *copy* of the skeleton project, so that the original skeleton will always be available when you want to start a new project. So at this point, you should use **File > Save Project As...** to save the project under a different file name. The rest of this section will assume that you saved the file as:

```
\\SQLTOOLS\\SAMPLES\\VB\\SQL-DUMP\\SQL-DUMP.VBP
```

...but keep in mind that *a file with that name is supplied with SQL Tools*. It is the final version of `SQL-Dump` as shown in Step 7. So you may want to choose a different name, to avoid over-writing the existing file.

You should also use **File > Save SQL-Dump.frm As...** to save the FRM file as `SQL-Dump.frm`, or another name of your choosing.

Next you will need to tell Visual Basic which version of SQL Tools you are using, either Standard or Pro. This is done by adding a SQL Tools "module" file to the project. The exact technique you use will vary, depending on the version of Visual Basic that you are using. For example, if you are using VB5 or VB6, open your project, select **Project > Add Module**, select the "Existing" tab, navigate to the `\SQLTOOLS\` directory, and select the appropriate `mod*.BAS` file. If you are using SQL Tools Standard, select `modSQLToolsStd.BAS`, and if you are using SQL Tools Pro, select `modSQLToolsPro.BAS`.

The next step is to add your SQL Tools Authorization Code. It is the eight-digit hexadecimal string (i.e. a string that contains numbers and the letters A-F) that was supplied with your SQL Tools Installation File. Enter your eight-digit Authorization Code where you see the eight dots in the source code. For example, if your Authorization Code is `1234ABCD` then you would change the source code like this:

```
'Add your Authorization Code here...
SQL_Authorize &H1234ABCD
SQL_Init
```

Please note that "1234ABCD" is *not* a valid Authorization Code! *You must use the eight-digit code that was supplied with your SQL Tools Installation file.*

By the way, if your eight-digit Authorization code starts with one or more zeros, Visual Basic will automatically remove them. For example, if you were to type `&H0034ABCD` and move to another line of source code, VB would remove the zeros and you would see `&H34ABCD`. This is perfectly normal, and it will not interfere with the operation of the `SQL_Authorize` function.

After those changes have been made, the basic program is ready to go. In fact, you could compile and run it at this point, but it really wouldn't do anything useful.

For more information about the structure of the skeleton project and the basic SQL Tools functions that it contains, you can read *Four Critical Steps for Every SQL Tools Program*.

Otherwise, it's time to make the SQL-Dump program do something useful...

Step 2: Open the Database

SQL-DUMP Visual Basic Step 2: Open the Database

Opening a database with SQL Tools is similar to using the Visual Basic `Open` statement to open a disk file that you want to access. It prepares the file for use, and assigns a number to it. For example...

```
Open "C:\MYDIR\MYFILE.TXT" For Input As #1
```

... tells Visual Basic to prepare the specified file and to use the file number `1` for all future operations (such as `Line Input #1`, etc.)

Similarly, the `SQL_OpenDB` (Open Database) function is used to tell SQL Tools that you want to open a database. For the purposes of this example we will specify a very specific type of file, called a DSN file, like this:

```
SQL_OpenDB "filename.DSN"
```

Instead of *filename*, of course, you will need to specify the name of a real DSN file. (By the way, the number `1` is used *automatically*, so you will not usually need to specify it. See Database Numbers for more information about using different numbers.)

A DSN or "DataSource Name" file is *not* a database. It is a text file that contains information *about* a database, such as where it is located, the ODBC driver that is required to access it, and so on.

In this example, to keep things simple, we are going to use an existing DSN file called...

```
\SQLTOOLS\SAMPLES\SQL-DUMP.DSN.
```

This file is supplied with SQL Tools so that you can actually compile and run the `SQL-DUMP` program exactly as it is described here.

IMPORTANT NOTE: If you installed SQL Tools someplace other than the default `\SQLTOOLS\` directory, you must use the appropriate path in place of `\SQLTOOLS\` in all sample source code. *You must also edit the `SQL-DUMP.DSN` file itself, because it contains references to the `\SQLTOOLS\` directory.* DSN files can be edited with any text editor. If you fail to change either the source code *or* the DSN file, the final program will display an Error Message.

Here is the actual syntax for opening the sample database...

```
SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
```

And this is where the `SQL_OpenDB` line should be added to the `SQL-Dump` program...

```
Private Sub Form_Load()  
    'The SQL-DUMP sample program  
    SQL_Authorize &%MY_SQLT_AUTHCODE  
    SQL_Init  
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"  
End Sub
```

If you were to compile and run this program, it would open the `SQL-Dump` database and then exit immediately.

SQL-DUMP Visual Basic Step 3: Tell the Database Which Data We Want

The closest Visual Basic equivalent to this step would be the `Seek` statement, which tells a VB program to jump to a particular location in a file. But `Seek` has to be performed line-by-line, and using it would require your program to know the locations of all of the lines of data that you want to retrieve.

SQL Statements make data retrieval much easier than that. Here is the syntax for telling SQL Tools to retrieve all of the data from the `AddressBook` table in the `SQL-Dump` database...

```
SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"
```

You'll find that the abbreviation `Stmt` is used extensively by SQL Tools. It stands for Statement.

The `SQL_Stmt` function can be used in many, many different ways. In this example, the `SQL_STMT_IMMEDIATE` parameter tells it that we want the results right away, and the `SELECT * FROM ADDRESSBOOK` parameter tells it that we want `*` (all) of the columns `FROM` the table called `ADDRESSBOOK`.

This is where the `SQL_Stmt` line should be added to our program:

```
Private Sub Form_Load()  
    'The SQL-DUMP sample program  
    SQL_Authorize %MY_SQLT_AUTHCODE  
    SQL_Init  
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"  
    SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"  
End Sub
```

Step 4: Retrieve the Data

For more information about the different kinds of SQL Statements that you can use, see Appendix A: SQL Statement Syntax.

SQL-DUMP Visual Basic Step 4: Retrieve the Data

Retrieving a row of data from a database is similar to using the Visual Basic `Line Input` statement on a disk file. It gets data from the database and places it in variables that your program can use.

For example, this VB code...

```
Line Input #1, sOneLine$
```

... would get one line of data from the file that was opened as #1, and place the data in the variable called `sOneLine$`. (To understand the variable-naming convention that is used in this document, see [Conventions](#).)

The equivalent SQL Tools syntax would be...

```
SQL_Fetch NEXT_ROW
```

The `SQL_Fetch` function can be used in a number of different ways. Using the `NEXT_ROW` parameter tells it to get the next row from the database. (Of course when the database is freshly opened, `NEXT_ROW` means the same thing as `FIRST_ROW`.)

The `SQL_Fetch` function should be added here...

```
Private Sub Form_Load()  
    'The SQL-DUMP sample program  
    SQL_Authorize %MY_SQLT_AUTHCODE  
    SQL_Init  
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"  
    SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"  
    SQL_Fetch NEXT_ROW  
End Sub
```

You probably noticed that no variable name like `sOneLine$` was specified. That's because each row of data is automatically broken down into columns by SQL Tools, and each column can be accessed individually. More about that in a minute.

Step 5: Detect the End of the Data

SQL-DUMP Visual Basic Step 5: Detect the End of the Data

Normally, a Visual Basic program would use the `Line Input` statement in a loop with the `Eof` (End Of File) function, like this...

```
Do
    If Eof(1) Then Exit Do
    Line Input #1, sOneLine$
    'do something with the data
Loop
```

The equivalent SQL Tools function is called `SQL_EOD`, which stands for End Of Data. You use it like this...

```
Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD Then Exit Do
Loop
```

You must keep in mind that there is a very important difference between `Eof` and `SQL_EOD`. The Visual Basic `Eof` function returns a True (nonzero) value when there are no more lines to be read. The `SQL_EOD` function returns a True value only *after* the `SQL_Fetch` function has *failed* to read a row of data.. That's a very important distinction if you write code that is structured like this:

```
Do Until Eof(1)
    Line Input #1, sOneLine$
    'do something with the data
Loop
```

That code will execute the way you would expect it to. It will read lines of data from the file until the end-of-file is encountered. Most importantly, the `Line Input` will always return a line of data and the program will always be able to do something with the data in `sOneLine$`.

However this SQL Tools code...

```
Do Until SQL_EOD
    SQL_Fetch NEXT_ROW
    'do something with the data
Loop
```

...will *not* work in the same way. Remember, the `SQL_EOD` function will not return a True value until *after* a `SQL_Fetch` has failed. That loop would eventually fetch the last row of data and process it. But then `SQL_EOD` would still return False (because a fetch has not yet *failed*) so the program would not exit from the loop, and the final `SQL_Fetch` operation would fail. At that point there would be no data for the program to "do something" with. Only then, after the fetch had failed and invalid data had been processed, would the program exit from the `DO/LOOP`. So...

Here is the *correct* way to structure a SQL Tools "read until end of data" loop:

```

Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD Then Exit Do
    'do something with the data
Loop

```

You must fetch a row, check for `SQL_EOD`, and *then* process the data.

Please note that *this is standard SQL behavior*. It is not a limitation of SQL Tools. ODBC drivers do not have a "look ahead" function that works like the Visual Basic `Eof` function.

Here is the addition that should be made to the example program:

```

Private Sub Form_Load()
    'The SQL-DUMP sample program
    SQL_Authorize %MY_SQLT_AUTHCODE
    SQL_Init
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
    SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"
    Do
        SQL_Fetch NEXT_ROW
        If SQL_EOD Then Exit Do
        'do something with the data
    Loop
End Sub

```

Step 6: Use the Data

SQL-DUMP Visual Basic Step 6: Use the Data

In a Visual Basic program, in order to use the data from a text file, you would probably do something like this...

```
'Open an input file...
Open "C:\MYDIR\OLDFILE.TXT" For Input As #1
'Open an output file...
Open "C:\MYDIR\NEWFILE.TXT" For Output #2

Do
    'Read a line from the input file...
    Line Input #1, sOneLine$
    'Put that line in the output file...
    Print #2, sOneLine$
    'Check for end of file...
    If Eof(1) Then Exit Do
Loop

Close #1
Close #2
```

In a SQL Tools program, if you want to use something like `Print #2` to save all of the data from all of the columns in a table, the easiest method is to use the `SQL_ResColText` function.

```
Do
    SQL_Fetch NEXT_ROW
    If SQL_EOD Then Exit Do
    Print #2, SQL_ResColText(ALL_COLS)
Loop
```

The `SQL_ResColText` function can take data in virtually any form (string, numeric, binary, etc.) and convert it to "text" that can be used with the `Print #` statement, or with any other function that requires a text string (MsgBox, etc.)

The `ALL_COLS` parameter tells `SQL_ResColText` to automatically count the number of columns that a result set contains, and to processes all of them. It also comma-quote delimits the data from all of the columns, and it limits the length of each column to a "reasonable" length, so it may not be practical for every program. But for quick and dirty programs like SQL-Dump it can be very useful.

```

Private Sub Form_Load()
    'The SQL-DUMP sample program
    SQL_Authorize &%MY_SQLT_AUTHCODE
    SQL_Init
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
    Open "\SQLTOOLS\SAMPLES\SQL-DUMP.TXT" For Output As #2
    SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"
    Do
        SQL_Fetch NEXT_ROW
        If SQL_EOD Then Exit Do
        Print #2,SQL_ResColText(ALL_COLS)
    Loop
    Close #2
End Sub

```

Step 7: Compile and Run

SQL-DUMP Visual Basic Step 7: Compile and Run

Here is the basic SQL Tools program that we have just written:

```
Private Sub Form_Load()  
    'The SQL-DUMP sample program  
    SQL_Authorize %MY_SQLT_AUTHCODE  
    SQL_Init  
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"  
    Open "\SQLTOOLS\SAMPLES\SQL-DUMP.TXT" For Output As #2  
    SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"  
    Do  
        SQL_Fetch NEXT_ROW  
        If SQL_EOD Then Exit Do  
        Print #2, SQL_ResColText(ALL_COLS)  
    Loop  
    Close #2  
End Sub
```

That's just a dozen lines of executable code, and it's a working --albeit very simple -- program!

All you have to do is compile it using Visual Basic, make sure that the Microsoft Access ODBC driver is installed, make sure that the SQL Tools DLL (SQLT_PRO.DLL or SQLT_PRO.DLL) is installed where your program can find it, and run the program.

When you run the sample program, the \SQLTOOLS\SQL-DUMP.TXT file will be created and it will contain the following data:

```
"1","John Q. Public","123 Main Street","Anytown","MI","48070"  
"2","Jane Doe","456 First Blvd.","Janestown","CA","90210"  
"3","Bob Smith","789 Second Ave.","Buffalo","NY","12345"  
"4","Mary Jones","321 Deebee Row","Jonesville","AK","98765"
```

BUT WAIT! That's a working program, but there is one more very important step that you need to take...

Step 8: Add Error Checking

SQL-DUMP Visual Basic Step 8: Add Error Checking

If you are writing a very simple "utility" program, then simple code like the program in Step 7 will probably be sufficient. But if you need to write a more robust program -- one that can react appropriately when something goes wrong -- you will need to add (at least) a few more lines of code.

The key to adding error-checking is to figure out exactly where errors *might* occur. For example, if your program will always be run on your development computer, it is probably safe to ignore the return value of the `SQL_Init` function. Very little can go wrong as long as the system is configured properly. But if you are writing an "industrial strength" program that will run on many different systems, you might want to change the program like this...

```
Private Sub Form_Load()  
    'The SQL-DUMP sample program  
    SQL_Authorize %MY_SQLT_AUTHCODE  
    If SQL_Init <> SUCCESS Then  
        SQL_Msgbox "SQL TOOLS INIT FAILURE", MSGBOX_OK  
        Exit Sub  
    End If
```

As we said, the `SQL_Init` function is quite reliable. A *much* more important place to add error checking is the `Do/Loop` block.

As an experiment, you can purposely "break" the SQL-Dump program by changing the `SQL_Stmt` line like this:

```
SQL_Stmt SQL_STMT_IMMEDIATE, "SELECT * FROM NONEXIST"  
Do  
    SQL_Fetch NEXT_ROW  
    If SQL_EOD Then Exit Do  
    Print #2,SQL_ResColText(ALL_COLS)  
Loop
```

If you re-compile and run the program, it will never exit from the `Do/Loop` block and will completely "lock up". That's because the `SQL_EOD` function will never return a True value.

`SQL_EOD` means "End Of Data" and that has a *very* specific meaning. It means that `SQL_Fetch` failed because the last row of data has been read from a result set. But the program did *not* reach the End Of Data point -- it never read *any data at all* -- and `SQL_EOD` will not return True if `SQL_Fetch` fails *for some other reason*, such as an error condition.

So it would be a *very* good idea to add some error checking code here too:

```
Do  
    SQL_Fetch NEXT_ROW  
    If SQL_EOD Then Exit Do  
    If SQL_ErrorPending Then  
        'Handle error here (display message,  
        'create error-log file, etc.)  
        Exit Sub 'to avoid locking up the system  
    End if  
    Print #2,SQL_ResColText(ALL_COLS)  
Loop
```

Here's another, more "compact" way to handle the same situation...

```
Do
    If SQL_Fetch(NEXT_ROW) <> SUCCESS Then
        'Either End Of Data or an Error, so...
        Exit Do
    End If
    Print #2,SQL_ResColText(ALL_COLS)
Loop
```

One more thing... The `SQL_Fetch` function can return either `SUCCESS` or `SUCCESS_WITH_INFO` when it works properly, so here is an even *better* error check which uses the `SQL_Okay` function to check for success:

```
Do
    If SQL_Okay(SQL_Fetch(NEXT_ROW)) = False Then
        'Either End Of Data or an Error, so...
        Exit Do
    End If
    Print #2,SQL_ResColText(ALL_COLS)
Loop
```

As you can see, there are lots of different ways in which a program can fail, and lots of different ways to handle the failures. You should familiarize yourself with all of the SQL Tools functions that start with `SQL_Error`, and read the section of this document titled Error Handling in SQL Tools Programs. It contains a *lot* of information about the various Error Handling techniques that are available to you.

The important thing is to try to anticipate the things that could possibly go wrong with your program, and to add code to handle those failures. To create a truly reliable program, you should examine the return value of every SQL Tools function and/or check the `SQL_ErrorPending` function after any SQL Tools function is used.

Suggested reading:

Error Handling in SQL Tools Programs.

Miscellaneous Error Handling Techniques

The PowerBASIC Version of SQL-Dump

If you are reading this document "in order" (for example, as a printed document), you have reached the end of the SQL-Dump Visual Basic sample program.

The following sections cover almost exactly the same material, but using the PowerBASIC For Windows compilers.

Step 1

SQL-DUMP PowerBASIC Step 1: Link SQL Tools to Your Program

This section describes the process of creating a SQL Tools program from scratch, using PowerBASIC. If you are adding SQL Tools to an existing PowerBASIC program, the steps are basically the same. To make things easier, we recommend that you cut and paste the source code that is provided in the skeleton program (see below) into your existing program.

The steps below are covered in more detail in [Four Critical Steps For Every SQL Tools Program](#).

The easiest way to start writing a SQL Tools program with PowerBASIC is to use the "skeleton" file that is provided here:

```
\\SQLTOOLS\\SAMPLES\\PB\\SKELETON\\SQL-SKEL.BAS
```

It contains all of the basic elements that you'll need to get started. This is what it looks like:

```
'===== SQL-SKEL.BAS
'Skeleton for simple SQL Tools programs.

$COMPILE EXE "SQL-SKEL.EXE"

'Remove the comment mark (')
'from ONE of these lines:
'$INCLUDE "\\SQLTOOLS\\SQLT_STD.INC"
'$INCLUDE "\\SQLTOOLS\\SQLT_PRO.INC"

DECLARE FUNCTION MainProgram AS LONG

FUNCTION PBMain AS LONG
    SQL_Authorize %MY_SQLT_AUTHCODE
    SQL_Init
    FUNCTION = MainProgram
    SQL_Shutdown
END FUNCTION

FUNCTION MainProgram AS LONG
    'your code goes here
END FUNCTION

'===== end of SQL-SKEL.BAS
```

IMPORTANT NOTE: In practice, you should always start with a *copy* of the skeleton program, so that the original skeleton will always be available when you want to start a new project.

Let's start out by modifying the skeleton to use our sample program's name, SQL-DUMP.

```

'===== SQL-DUMP.BAS
'The SQL-DUMP sample program

$COMPILE EXE "SQL-DUMP.EXE"

'Remove the comment mark (') from
'ONE of these lines:
'$INCLUDE "\SQLTOOLS\SQLT_STD.INC"
'$INCLUDE "\SQLTOOLS\SQLT_PRO.INC"

DECLARE FUNCTION SQLDump AS LONG

FUNCTION PMain AS LONG
    SQL_Authorize %MY_SQLT_AUTHCODE
    SQL_Init
    FUNCTION = SQLDump
    SQL_Shutdown
END FUNCTION

FUNCTION SQLDump AS LONG
    'your code goes here
END FUNCTION
'===== end of SQL-DUMP.BAS

```

As you can see, as we modify the program we will highlight the most recent changes in **bold** text.

IMPORTANT NOTE: In practice, you should always start with a *copy* of the skeleton program, so that the original skeleton will always be available when you want to start a new project. So at this point, you should use **File > Save As...** to save the skeleton under a different file name. The rest of this section will assume that you saved the file as:

```

\SQLTOOLS\SAMPLES\PB\SQL-DUMP\SQL-DUMP.BAS

```

...but keep in mind that *a file with that name is supplied with SQL Tools*. It is the final version of SQL-Dump as shown in Step 7. So you may want to choose a different name, to avoid over-writing the existing file.

Next you will need to tell the PowerBASIC compiler which version of SQL Tools you are using, either Standard or Pro. If you are using SQL Tools Standard you should remove the comment mark (') from the beginning of the line that says `SQLT_STD.INC`, like this:

```

'Remove the comment mark (')
'from ONE of these lines:
$INCLUDE "\SQLTOOLS\SQLT_STD.INC"
'$INCLUDE "\SQLTOOLS\SQLT_PRO.INC"

```

...and of course if you are using SQL Tools Pro it should look like this:

```

'Remove the comment mark (')
'from ONE of these lines:
'$INCLUDE "\SQLTOOLS\SQLT_STD.INC"
$INCLUDE "\SQLTOOLS\SQLT_PRO.INC"

```

The next step is to add your SQL Tools Authorization Code. It is the eight-digit hexadecimal string (i.e. a string that contains numbers and the letters A-F) that was supplied with your SQL

Tools Installation File. Enter your eight-digit Authorization Code where you see the eight dots in the source code. For example, if your Authorization Code is 1234ABCD then you would change the source code like this:

```
FUNCTION PBMain AS LONG
    'Add your Authorization Code here...
    SQL_Authorize &h1234ABCD
    SQL_Init
```

Please note that "1234ABCD" is *not* a valid Authorization Code! *You must use the eight-digit code that was supplied with your SQL Tools Installation file.*

After those changes have been made, the basic program is ready to go. In fact, you could compile and run it at this point, but it really wouldn't do anything useful.

For more information about the structure of the `SQL_SKELETON.BAS` file and the basic SQL Tools functions that it contains, you can read [Four Critical Steps for Every SQL Tools Program](#).

Otherwise, it's time to make the SQL-Dump program do something useful...

STEP 2: Open the Database

SQL-DUMP PowerBASIC Step 2: Open the Database

Opening a database with SQL Tools is similar to using the PowerBASIC `OPEN` statement to open a disk file that you want to access. It prepares the file for use, and assigns a number to it. For example...

```
Open "C:\MYDIR\MYFILE.TXT" For Input As #1
```

... tells PowerBASIC to prepare the specified file and to use the file number `1` for all future operations (such as `Line Input #1`, etc.)

Similarly, the `SQL_OpenDB` (Open Database) function is used to tell SQL Tools that you want to open a database. For the purposes of this example we will specify a very specific type of file, called a DSN file, like this:

```
SQL_OpenDB "filename.DSN"
```

Instead of *filename*, of course, you will need to specify the name of a real DSN file. (By the way, the number `1` is used *automatically*, so you will not usually need to specify it. See Database Numbers for more information about using different numbers.)

A DSN or "DataSource Name" file is *not* a database. It is a text file that contains information *about* a database, such as where it is located, the ODBC driver that is required to access it, and so on.

In this example, to keep things simple, we are going to use an existing DSN file called...

```
\SQLTOOLS\SAMPLES\SQL-DUMP.DSN.
```

This file is supplied with SQL Tools so that you can actually compile and run the `SQL-DUMP` program exactly as it is described here.

IMPORTANT NOTE: If you installed SQL Tools someplace other than the default `\SQLTOOLS\` directory, you must use the appropriate path in place of `\SQLTOOLS\` in all sample source code. *You must also edit the `SQL-DUMP.DSN` file itself, because it contains references to the `\SQLTOOLS\` directory.* DSN files can be edited with any text editor. If you fail to change either the source code *or* the DSN file, the final program will display an Error Message.

Here is the actual syntax for opening the sample database...

```
SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
```

And here is the `SQL-DUMP` program so far...

```

'===== SQL-DUMP.BAS
'The SQL-DUMP sample program

$COMPILE EXE "SQL-DUMP.EXE"

'Remove the comment mark (')
'from ONE of these lines:
'$INCLUDE "\SQLTOOLS\SQLT_STD.INC"
'$INCLUDE "\SQLTOOLS\SQLT_PRO.INC"

Declare Function SQLDump As Long

Function PBMain As Long
    SQL_Authorize %MY_SQLT_AUTHCODE
    SQL_Init
    Function = SQLDump
    SQL_Shutdown
End Function

Function SQLDump As Long
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
End Function
'===== end of SQL-DUMP.BAS

```

If you were to compile and run this program, it would open the SQL-Dump database and then exit immediately.

STEP 3: Tell the Database Which Data We Want

SQL-DUMP PowerBASIC Step 3: Tell the Database Which Data We Want

The closest PowerBASIC equivalent to this step would be the `Seek` statement, which tells a PowerBASIC program to jump to a particular location in a file. But `Seek` has to be performed line-by-line, and using it would require your program to know the locations of all of the lines of data that you want to retrieve.

SQL Statements make data retrieval much easier than that. Here is the syntax for telling SQL Tools to retrieve all of the data from the `AddressBook` table in the `SQL-Dump` database...

```
SQL_Stmt %SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"
```

You'll find that the abbreviation `Stmt` is used extensively by SQL Tools. It stands for Statement.

The `SQL_Stmt` function can be used in many, many different ways. In this example, the `%SQL_STMT_IMMEDIATE` parameter tells it that we want the results right away, and the `SELECT * FROM ADDRESSBOOK` parameter tells it that we want `*` (all) of the columns `FROM` the table called `ADDRESSBOOK`.

This is where the `SQL_Stmt` line should be added to our program:

```
Function SQLDump As Long
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
    SQL_Stmt %SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"
End Function
```

STEP 4: Retrieve the Data

For more information about the different kinds of SQL Statements that you can use, see Appendix A: SQL Statement Syntax.

SQL-DUMP PowerBASIC Step 4: Retrieve the Data

Retrieving a row of data from a database is similar to using the PowerBASIC `Line Input` statement on a disk file. It gets data from the database and places it in variables that your program can use. For example, this...

```
Line Input #1, sOneLine$
```

... would get one line of data from the file that was opened as #1, and place the data in the variable called `sOneLine$`. (To understand the variable-naming convention that is used in this Help File, see Conventions.)

The equivalent SQL Tools syntax would be...

```
SQL_Fetch %NEXT_ROW
```

The `SQL_Fetch` function can be used in a number of different ways. Using the `%NEXT_ROW` parameter tells it to get the next row from the database. (Of course when the database is freshly opened, `%NEXT_ROW` means the same thing as `%FIRST_ROW`.)

```
Function SQLDump As Long
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
    SQL_Stmt %SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"
    SQL_Fetch %NEXT_ROW
End Function
```

You probably noticed that no variable name like `sOneLine$` was specified. That's because each row of data is automatically broken down into columns by SQL Tools, and each column can be accessed individually. More about that in a minute.

STEP 5: Detect the End of the Data

SQL-DUMP PowerBASIC Step 5: Detect the End of the Data

Normally, a PowerBASIC program would use the `Line Input` statement in a loop with the `Eof` (End Of File) function, like this...

```
Do
    If Eof(1) Then Exit Do
    Line Input #1, sOneLine$
    'do something with the data
Loop
```

The equivalent SQL Tools function is called `SQL_EOD`, which stands for End Of Data. You use it like this...

```
Do
    SQL_Fetch %NEXT_ROW
    If SQL_EOD Then Exit Do
Loop
```

You must keep in mind that there is a very important difference between `Eof` and `SQL_EOD`. The PowerBASIC `Eof` function returns a True (nonzero) value when there are no more lines to be read. The `SQL_EOD` function returns a True value only *after* the `SQL_Fetch` function has *failed* to read a row of data.. That's a very important distinction if you write code that is structured like this:

```
Do Until Eof(1)
    Line Input #1, sOneLine$
    'do something with the data
Loop
```

That code will execute the way you would expect it to. It will read lines of data from the file until the end-of-file is encountered. Most importantly, the `Line Input` will always return a line of data and the program will always be able to do something with the data in `sOneLine$`.

However this SQL Tools code...

```
Do Until SQL_EOD
    SQL_Fetch %NEXT_ROW
    'do something with the data
Loop
```

...will *not* work in the same way. Remember, the `SQL_EOD` function will not return a True value until *after* a `SQL_Fetch` has *failed*. That loop would eventually fetch the last row of data and process it. But then `SQL_EOD` would still return False (because a fetch has not yet *failed*) so the program would not exit from the loop, and the final `SQL_Fetch` operation would fail. At that point there would be no data for the program to "do something" with. Only then, after the fetch had failed and invalid data had been processed, would the program exit from the `DO/LOOP`. So...

Here is the *correct* way to structure a SQL Tools "read until end of data" loop:

```

Do
    SQL_Fetch %NEXT_ROW
    If SQL_EOD Then Exit Do
    'do something with the data
Loop

```

You must fetch a row, check for `SQL_EOD`, and *then* process the data.

Please note that *this is standard SQL behavior*. It is not a limitation of SQL Tools. ODBC drivers do not have a "look ahead" function that works like the PowerBASIC `Eof` function.

(Incidentally, when a file is opened `For Binary` with PowerBASIC, the `Eof` function works exactly the same way as the `SQL_EOD` function. `Eof` does not return a True value until *after* a binary-read operation has failed.)

Here is the addition that should be made to the example program:

```

Function SQLDump As Long
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
    SQL_Stmt %SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"
    Do
        SQL_Fetch %NEXT_ROW
        If SQL_EOD Then Exit Do
        'do something with the data
    Loop
End Function

```

STEP 6: Use the Data

SQL-DUMP PowerBASIC Step 6: Use the Data

In a PowerBASIC program, in order to use the data from a text file, you would probably do something like this...

```
'Open an input file...
Open "C:\MYDIR\OLDFILE.TXT" For Input As #1
'Open an output file...
Open "C:\MYDIR\NEWFILE.TXT" For Output As #2

Do
    'Read a line from the input file...
    Line Input #1, sOneLine$
    'Put that line in the output file...
    Print #2, sOneLine$
    'Check for end of file...
    If Eof(1) Then Exit Do
Loop

Close #1
Close #2
```

In a SQL Tools program, if you want to use something like `Print #2` to save all of the data from all of the columns in a table, the easiest method is to use the `SQL_ResColText` function.

```
Do
    SQL_Fetch %NEXT_ROW
    If SQL_EOD Then Exit Do
    Print #2, SQL_ResColText(%ALL_COLS)
Loop
```

The `SQL_ResColText` function can take data in virtually any form (string, numeric, binary, etc.) and convert it to "text" that can be used with the `Print #` statement, or with any other function that requires a text string (`Print`, `MsgBox`, etc.)

The `%ALL_COLS` parameter tells `SQL_ResColText` to automatically count the number of columns that a result set contains, and to processes all of them. It also comma-quote delimits the data from all of the columns, and it limits the length of each column to a "reasonable" length, so it may not be practical for every program. But for quick and dirty programs like SQL-Dump it can be very useful.

```
Function SQLDump As Long
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
    Open "\SQLTOOLS\SAMPLES\SQL-DUMP.TXT" For Output As #2
    SQL_Stmt %SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"
    Do
        SQL_Fetch %NEXT_ROW
        If SQL_EOD Then Exit Do
        Print #2,SQL_ResColText(%ALL_COLS)
    Loop
    Close #2
End Function
```

SQL-DUMP PowerBASIC Step 7: Compile and Run

Here is the basic SQL Tools program that we have just written:

```
'===== SQL-DUMP.BAS
'The SQL-DUMP sample program

$COMPILE EXE "SQL-DUMP.EXE"

'Remove the comment mark (') from ONE of these lines:
'$INCLUDE "\SQLTOOLS\SQLT_STD.INC"
'$INCLUDE "\SQLTOOLS\SQLT_PRO.INC"

Declare Function SQLDump As Long

Function PBMain As Long
    SQL_Authorize %MY_SQLT_AUTHCODE
    SQL_Init
    Function = SQLDump
    SQL_Shutdown
End Function

Function SQLDump As Long
    SQL_OpenDB "\SQLTOOLS\SAMPLES\SQL-DUMP.DSN"
    Open "\SQLTOOLS\SAMPLES\SQL-DUMP.TXT" For Output As #2
    SQL_Stmt %SQL_STMT_IMMEDIATE, "SELECT * FROM ADDRESSBOOK"
    Do
        SQL_Fetch %NEXT_ROW
        If SQL_EOD Then Exit Do
        Print #2,SQL_ResColText(%ALL_COLS)
    Loop
    Close #2
End Function
'===== end of SQL-DUMP.BAS
```

That's just a dozen lines of executable code, and it's a working --albeit very simple -- program!

All you have to do is compile this program using one of the PowerBASIC For Windows compilers, make sure that the Microsoft Access ODBC driver is installed, make sure that the SQL Tools DLL (SQLT_PRO.DLL or SQLT_PRO.DLL) is installed where your program can find it, and run the program.

When you run the sample program, the \SQLTOOLS\SQL-DUMP.TXT file will be created and it will contain the following data:

```
"1","John Q. Public","123 Main Street","Anytown","MI","48070"
"2","Jane Doe","456 First Blvd.","Janestown","CA","90210"
"3","Bob Smith","789 Second Ave.","Buffalo","NY","12345"
"4","Mary Jones","321 Deebee Row","Jonesville","AK","98765"
```

BUT WAIT! That's a working program, but there is one more very important step that you need to take...

SQL-DUMP PowerBASIC Step 8: Add Error Checking

If you are writing a very simple "utility" program, then simple code like the program in Step 7 will probably be sufficient. But if you need to write a more robust program -- one that can react appropriately when something goes wrong -- you will need to add (at least) a few more lines of code.

The key to adding error-checking is to figure out exactly where errors *might* occur. For example, if your program will always be run on your development computer, it is probably safe to ignore the return value of the `SQL_Init` function. Very little can go wrong as long as the system is configured properly. But if you are writing an "industrial strength" program that will run on many different systems, you might want to change the `PBMain` function like this...

```
Function PBMain As Long
    SQL_Authorize %MY_SQLT_AUTHCODE
    If SQL_Init <> %SUCCESS Then
        SQL_Msgbox "SQL TOOLS INIT FAILURE", %MSGBOX_OK
        Exit Function
    End If
    Function = SQLDump
    SQL_Shutdown
End Function
```

As we said, the `SQL_Init` function is quite reliable. A *much* more important place to add error checking is the `DO/LOOP` block.

As an experiment, you can purposely "break" the SQL-Dump program by changing the `SQL_Stmt` line like this:

```
SQL_Stmt %SQL_STMT_IMMEDIATE, "SELECT * FROM NONEXIST"
Do
    SQL_Fetch %NEXT_ROW
    If SQL_EOD Then Exit Do
    Print #2,SQL_ResColText(%ALL_COLS)
Loop
```

If you re-compile and run the program, it will never exit from the `DO/LOOP` block and will completely "lock up". That's because the `SQL_EOD` function will never return a True value.

`SQL_EOD` means "End Of Data" and that has a *very* specific meaning. It means that `SQL_Fetch` failed because the last row of data has been read from a result set. But the program did *not* reach the End Of Data point -- it never read *any data at all* -- and `SQL_EOD` will not return True if `SQL_Fetch` fails *for some other reason*, such as an error condition.

So it would be a *very* good idea to add some error checking code here too:

```

Do
    SQL_Fetch %NEXT_ROW
    If SQL_EOD Then Exit Do
    If SQL_ErrorPending Then
        'Handle error here (display message,
        'create error-log file, etc.)
        Exit Function 'to avoid locking up the system
    End If
    Print #2,SQL_ResColText(%ALL_COLS)
Loop

```

Here's another, more "compact" way to handle the same situation...

```

Do
    If SQL_Fetch(%NEXT_ROW) <> %SUCCESS Then
        'Either End Of Data or an Error, so...
        Exit Do
    End If
    Print #2,SQL_ResColText(%ALL_COLS)
Loop

```

One more thing... The `SQL_Fetch` function can return either `SUCCESS` or `SUCCESS_WITH_INFO` when it works properly, so here is an even *better* error check which uses the `SQL_Okay` function to check for success:

```

Do
    If SQL_Okay(SQL_Fetch(NEXT_ROW)) = False Then
        'Either End Of Data or an Error, so...
        Exit Do
    End If
    Print #2,SQL_ResColText(ALL_COLS)
Loop

```

As you can see, there are lots of different ways in which a program can fail, and lots of different ways to handle the failures. You should familiarize yourself with all of the SQL Tools functions that start with `SQL_Error`, and read the section of this document titled Error Handling in SQL Tools Programs. It contains a *lot* of information about the various Error Handling techniques that are available to you.

The important thing is to try to anticipate the things that could possibly go wrong with your program, and to add code to handle those failures. To create a truly reliable program, you should examine the return value of every SQL Tools function and/or check the `SQL_ErrorPending` function after any SQL Tools function is used.

Suggested reading:

Error Handling in SQL Tools Programs.

Miscellaneous Error Handling Techniques