SHADOW

PROGRAMMING GUIDE

NEON SYSTEMS, INC.

This document is published by the NEON Systems, Inc. Technical Publications Department and applies to Shadow® Server™, Version 4, Release 5, Shadow® OS/390 Web Server™, Version 4, Release 5, and Shadow® Enterprise Server™ , Version 2, Release 6.

Address inquiries to:

NEON Systems, Inc.
14100 SW Freeway, Suite 500
Sugar Land, Texas 77478

World Wide Web: http://www.neonsys.com

Phone: 1-800-505-6366
(281) 491-4200 (Corporate Sales, Technical Support)
Fax: (281) 242-3880

December 1999

Contents

About this Publication

This book contains programming information for Shadow Direct, Shadow OS/390 Web Server, and Shadow Enterprise.

How this Publication is Organized

This book contains the following chapters:

- Chapter 1, “Shadow RPC Direct,” provides information about Shadow RPC Direct, its product architecture, host data, and client API function definitions.
- Chapter 2, “ODBC CALL RPCs,” provides ODBC CALL RPC examples, sample ODBC CALL RPCs for VSAM, and other sample RPCs.
- Chapter 3, “Running DB2 Stored Procedures,” includes information about executing, preparing and troubleshooting a DB2 Stored Procedure.
- Chapter 4, “Shadow IMS Direct,” details Shadow IMS Direct, its product architecture, installation, programming, client API function definitions, and sample IMS batch message program code.
- Chapter 5, “Transaction Server for IMS,” provides information about SHADOW_IMS, including examples of using Shadow for IMS.
- Chapter 6, “Transaction Server for CICS,” includes information about SHADOW_CICS RPC, and examples of using Shadow for CICS.
- Chapter 7, “Host Application API Function Calls,” includes a listing and detailed description of the following:
  - ODBC CALL RPC APIs
  - IMS/APPC APIs
  - CICS APIs
  - Web Server Specific APIs
  - General APIs
  - Web Server REXX and SEF only APIs
  - RPC Direct APIs
- Chapter 8, “Shadow Enterprise Direct API Function Calls,” provides a listing and description of Shadow Enterprise Direct API calls.
- Chapter 9, “Transaction Level Security (TLS),” covers TLS, which was created to support the new and unique security requirements of Internet applications, while operating in the traditional enterprise computing environment.
- Chapter 10, “SQLProcedure and SQLProcedure Columns,” discusses how to create a pseudo DB2 stored procedure that contains necessary Meta data for

input and output fields, as well as other required parameters for accessing CICS and IMS transactions.

- Appendix A, “Shadow/REXX,” provides information about Shadow REXX, including its comparison to Standard REXX, its execution limits, elements, considerations, instructions, interfaces, and compiler error messages.
- Appendix B, “MVS Client Support,” covers the ODBC interface in a COBOL client program.
- “Glossary,” lists and defines terms and acronyms that appear in NEON Systems, Inc. publications.

Conventions

This book contains the following highlighting conventions:

BOLD CAPS
Identifies commands. For example:

Use the **KEYS** command to ...

Text enclosed in single quotes denotes library, data set, and DD names. For example:

```
'SLDSYSIN'    'PLUSIN'    'RESLIB'
```

`Monospace`
Identifies code examples, screen prompts, and messages, as well as directory paths. For example:

```
//STEP010    EXEC  PGM=NDBA2400
```

`Monospace Italics`
Identifies information you must provide at a screen prompt or in a text field. For example:

PARM='PARMLIB=*your.parmlib*'

<KEY> Identifies the key to press. For example:

<ENTER>

NEON Systems, Inc. uses *Release*.*Version* to identify software packages. For example, *Product 4.1*, denotes the fourth release, first revision of the software.

Reader's Comments

At NEON Systems, Inc. we are always looking for good ideas. If you have any comments or suggestions regarding any of our publications, please complete the Reader's Comment form (located at the back of this book) and return it to NEON, Attention: Technical Publications Department.

Mailing Address: **NEON Systems, Inc.**
14100 SW Freeway, Suite 500
Sugar Land, Texas 77478

Fax Number: (281) 242-3880

You can also send comments to directly to our Technical Publications department via the following e-mail address: **documentation@neonsys.com**.

Thank you!

NEON Systems, Inc. Products

For a comprehensive list of the products currently marketed by NEON Systems, Inc., visit our World Wide Web site at: **http://www.neonsys.com**.

Note:
You can also access and download all of the current NEON publications from this Web site.

Year 2000 Compliancy Statement

The following products from NEON Systems, Inc., are Year 2000 ready.

- **Enterprise Security Management Products**
- **Enterprise Subsystem Management Product Family**
- **Shadow® Product Family and Add-On Components**

The mainframe code for the Shadow Product Family, Version 3.1 and all subsequent versions, are Y2K ready.

All versions of the client code associated with Shadow® Direct™ and Shadow Enterprise Direct® are Y2K ready.

These products use four-digit year values both internally and externally (although, in a few cases, two-digit year values are displayed while four-digit year values are maintained internally).

Note:
While Shadow Direct, Shadow® OS/390 Web Server™, and Shadow Enterprise Direct are Y2K ready, customers should be aware that these products can provide access to data sources that may not be Y2K ready.

Working with Technical Support

NEON Systems, Inc. provides a number of ways for you to obtain assistance for our products. All product support inquiries are handled by the same support group, regardless if you are a trial or a licensed customer. The following are available support options:

Support Option	How to Access	How it Works	This Option is Best for:
E-mail	To contact Technical Support via e-mail: **support@neonsys.com** Email is available for receipt 24 hours a day, 7 days a week and is answered between 9AM-7PM CST Monday through Friday.	Email goes to the support queue, which is continuously monitored by a staff of cross-functional technical experts. It is answered in the order it is received. It is logged in the support database and assigned a trouble ticket number for tracking purposes.	This type of support is excellent for low to medium priority requests. It is a proven method for providing further information on critical problems that may have been phoned in. Email is a convenient way of sending us a list of lower priority items you have collected at a time that is convenient for you.
Phone	To contact Technical Support, please call: **1-800-505-6366** (U. S. and Canada) **1-281-491-4200** (outside North America)	During normal working hours you will be transferred to someone who can usually answer your question on the first call. You may be required to page a support person via our phone mail system after hours.	This type of support is best for high priority requests and initial installation questions. Use this option for any obvious system errors or anytime you need the most rapid reply to your question.
Internet	To access Internet support, please visit our Web site at: **www.neonsys.com**	Simply visit our Web site. NEON Systems works to keep current, relevant materials on our Web site to support our trial and licensed customers.	This option provides immediate access to documentation, updated client-side drivers, and our product Knowledge Base. The Knowledge Base is a collection of questions answered by support. Use this option to answer your own questions or to get a better understanding of what customers ask on an ongoing basis.
Account Manager	To contact your NEON Systems Sales Representative, please call: **1-800-505-6366** (U. S. and Canada) **1-281-491-4200** (outside North America)	Your Sales Representative is your account manager. This person is ultimately responsible for your complete satisfaction with NEON Systems, Inc.	Contact your Sales Representative for pricing information, contract details, password renewal or if you feel your needs are not being met.

CHAPTER 1:
Shadow RPC Direct

This chapter covers programming information for Shadow RPC Direct, a component of Shadow Direct which allows a client application to invoke and communicate with an RPC running on the host. Iinformation includes product architecture, using host data, and client API function definitions.

This chapter specifically applies to Shadow Direct.

Introduction

Shadow RPC Direct works with any client application supported by Shadow Direct. The currently supported platforms for client applications include:

- Windows
- Windows 95
- Windows NT
- OS/2
- UNIX.

Client applications can be written in C, or any other language that can call DLL entry points. Shadow RPC Direct provides both a client and host API. The client API includes entry points for the following:

- Establishing connections to the host.
- Initiating host RPCs.
- Synchronously and asynchronously sending data to and from the host.
- Terminating host connections.
- Data conversion and other support functions (for example, the ability to access and update fields in an IMS PCB).
- Sending data to and receiving data from a client program.
- Obtaining information about the current environment.
- Setting environmental information (DB2 plan name).
- Writing messages to the Trace Browse log.

The host RPCs invoked by the client are standard MVS programs that can access and update DB2 tables, IMS databases, VSAM files, partitioned data sets, etc. Host RPCs can be written in any high level language or in assembly language. In all cases, host RPCs use the same host API to communicate with the Shadow Direct environment.

Host RPCs execute as tasks or threads in the main product address space. A separate thread is created for every client session. This approach provides the highest degree of flexibility for the host RPCs and guarantees the maximum degree of independence between the threads. Each thread can perform whatever work it needs without any conflicts between itself and other threads.

Product Architecture

Using Shadow RPC Direct always involves two separate programs:

- A client application.
- An MVS RPC.

The client application initiates the host RPC. After the host RPC has been started, the client application and the host RPC can send messages back and forth, either synchronously or asynchronously. These messages or data buffers are defined entirely by the client application and the host RPC. The format, content, and sequence of these buffers is determined by the application programmers who create the client and host components. The client and host component must agree exactly on all aspects of the data flow between them.

Note:

There is no requirement that any data actually flow back and forth between the client application and host RPC, and zero length messages are supported.

No conversions are performed on the data buffers sent between the client application and the host RPC. Either the client, the host RPC, or both, must take responsibility for any conversions that are required. The Shadow RPC Direct client API includes entry points for performing some of these conversions. However some conversions (of non-message data) are performed on behalf of the client. For example, the optional OS parameter string passed to the host RPC is automatically converted from ASCII to EBCDIC (but is not converted to uppercase).

Client Applications

Shadow RPC Direct client applications are programs written in any one of several languages that use the Shadow Direct API to invoke and communicate with host RPCs. These applications are normally written in C or C++, however, these applications can be written in any language that can call DLL entry points including Visual Basic (VB), PowerScript, Pascal, COBOL, etc. In practice, almost any client application programming language can be used to invoke the Shadow RPC Direct API.

The Shadow RPC Direct API is implemented as:

- A DLL for Windows, Windows NT and OS/2.
- A shared library for UNIX environments (SunOS, etc.) supporting shared libraries.
- An archive file for other UNIX environments.

Shadow RPC Direct applications must be linked using one of the two following import libraries supplied with Shadow RPC Direct:

- SCODBC.LIB, used with SCODBC.DLL.
- SCODBCTS.LIB, used with SCODBCTS.DLL.

Since the SCODBCTS.DLL contains numerous diagnostic, debugging, and support tools, it should be used for all application development purposes. However, because the SCODBCTS.DLL is substantially larger and slower than its production counterpart SCODBC.DLL, production applications that have been fully debugged should be switched to the SCODBC.DLL for improved performance. .

Note:
ODBC.LIB must not be used with Shadow RPC Direct applications. The architecture of Shadow RPC Direct does not support passing calls from a Shadow RPC Direct application to the Shadow RPC Direct DLL via the Microsoft driver manager (ODBC.DLL).

Shadow RPC Direct applications written in C must include the scpghd.h header file. This file declares all Shadow RPC Direct structures and API entry points, and must be included in all Shadow RPC Direct client application functions. It can be used with both ANSI and non-ANSI C compilers, however, it is strongly recommended that ANSI C be used for compiling and building Shadow RPC client application programs. This header file will also work in all client environments including Windows, OS/2, and UNIX.

Host Applications

Host RPCs can be written in any high-level language or 370 assembler. PL/I, COBOL, FORTRAN and C are all supported. Host RPCs can be any AMODE and/or any RMODE. To conserve 24-bit memory, RMODE ANY and AMODE 31 are strongly preferred, however, RMODE 24 and AMODE 24 are supported. Data areas passed to and from the host RPCs can, in all cases, be either above or below the 16 MB line.

Note:
There are certain special considerations for some languages, which will be discussed below in detail.

Host RPCs

Host RPCs execute in the main Shadow address space as ordinary load modules. In other words, host RPCs can use normal programming procedures to access and update VSAM data sets, flat files, PDSs, etc. Host RPCs are not subject to any of the restrictions that are normally associated with the CICS or IMS environments.

As previously mentioned, host RPCs can use standard programming constructs to access and update host data. In addition, host RPCs can use APIs provided by Shadow to perform certain additional functions.

These APIs include facilities for:

- Sending and receiving data buffers as large as 30 kilobytes.
- Inserting messages into Shadow Server's Trace Browse log.
- Obtaining information about the current execution environment.
- Updating information about the current environment.

Host RPCs can also access and update IMS databases using the DBCTL interface. This interface allows host RPCs to communicate with either an IMS DB/DC control region or a DBCTL limited function control region. In either case, the same API is used to execute DL/I calls. This API is almost identical to the API used to execute IMS calls in the CICS environment. A host RPC starts by scheduling a PSB and then uses the PCB list returned by the PSB schedule operation to access and update IMS data.

A separate PRB is always created for each host RPC. This approach provides an additional degree of isolation for the RPC. Specifically, the host RPC can use ESTAE and ESPIE, as need be, to intercept and recover from abends and program checks. There is no requirement that a host RPC establish an ESTAE or an ESPIE of its own. Shadow Server will always establish an ESTAE to catch all abend errors that occur while host RPC is executing. If Shadow Server does intercept an abend while the host RPC is executing, the host RPC will be terminated and all database changes (DB2, IMS) will be rolled back.

Client-RPC Interaction

Figure 1–1, Shadow RPC Direct Product Architecture, shows how client applications interact with the MVS-based RPCs.

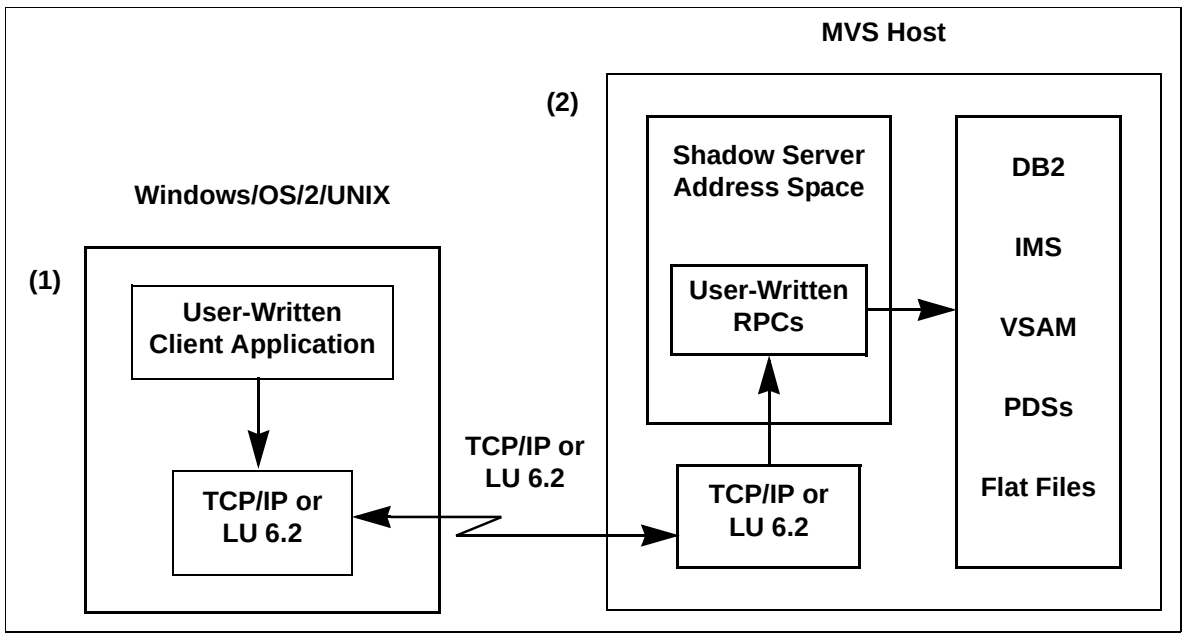


Figure 1–1. Shadow RPC Direct Product Architecture

1. The client application uses the Shadow RPC Direct API to establish either a TCP/IP or LU 6.2 session with the host. The host RPC is initiated as part of the session establishment process. A new host thread (or TCB) is always created for each new session started using the Shadow RPC Direct API.

2. The host RPC starts execution in the Shadow Server address space. The host RPC and client application can then communicate back and forth either synchronously or asynchronously.

3. The host RPCs can access any number of different types of data including DB2, IMS, VSAM, PDSs, and flat files. DB2 data can be accessed using static or dynamic SQL, although static SQL is the preferred choice in most cases. User-specified plans and packages can be used to access DB2 data, and all DB2 security will be handled using the userid and password provided by the client. IMS data is accessed using standard DL/I calls. For example:

   - The PLITDLI function can be used in PL/I programs.
   - COBTDLI can be used in COBOL programs.
   - CTDLI can be used in C programs.
   - ASMTDLI can be used in 370 assembler programs.

   The supported DL/I calls include all DL/I calls that can normally be used in the CICS environment. This includes all DL/I calls for accessing and updating databases, but excludes all DL/I calls for accessing and updating the IMS message queue.

VSAM files, PDSs, and flat files can also be accessed by the host RPC using normal high-level language programming procedures.

As previously mentioned, the data flow between the client application and RPC is entirely under the control of the client application and the host RPC. These programs can be designed to send any combination of data buffers back and forth. For example, the client application might send one data buffer to the host RPC and get one response back. Alternatively, a great many data buffers might be sent back and forth constituting an extended “conversation”. In other cases, there may be no exchange of any data buffers between the client application and the host RPC.

Note:

Once the client application program sends a message to the host to initiate execution of the host RPC, Shadow Server responds to the client before starting execution of the host RPC. In other words, the client application program will resume execution before the host RPC starts execution. This means that the client application cannot determine if the initiation of the host RPC has been successful or not. This approach must be used to allow the client application to regain control so that the client application and the host RPC can send messages back and forth.

Host Execution Environment

Host RPCs execute in a very specific environment. All host RPCs run as separate TCBs in the main product address space. A separate TCB is created for each client session with the host. If a client application creates multiple sessions with one copy of the product running on the host, multiple TCBs will be created on behalf of that client. As a consequence of the TCB processing architecture, host RPCs are completely independent of each other and are executed in an environment very similar to a TSO application or batch program.

Host RPCs run in problem, not supervisor, state for the following reasons:

- Host RPCs allowed to run in supervisor state could compromise the integrity of the Shadow Server address space or perhaps even the system as a whole.
- Several high-level languages (PL/I, C) will not execute properly if they are invoked in supervisor state.

The restriction that all host RPCs execute in problem state should not restrict which applications can be implemented using host RPCs. Host RPCs will always execute in KEY 8. This is the standard protection key for all problem programs, such as TSO applications or batch programs. Once again, this design does not restrict which applications can be implemented using Shadow Server.

Although non-reentrant programs are supported, RPCs should be designed and implemented as reentrant programs, if at all possible. This will allow all users of an RPC application to share one copy of the object code. This approach will dramatically reduce memory utilization requirements in many cases. If the application runs in RMODE 24, reentrance is even more important due to the fact that the storage below the 16 MB line is scarce in many installations. The Shadow Server address space itself uses no storage below the 16 MB line.

The client application can optionally pass a parameter string to the host RPC. The parameter string length can range from 0 to 100 bytes. This string is passed to the host RPC using an OS parameter list. In other words, the first word of the parameter list points to a two-byte prefix followed by up to 100 characters of actual parameter data. The parameter string is placed in 24-bit storage so it can be accessed by all host RPCs.

Virtual Storage Utilization

RPCs running in the Shadow Server address space can acquire and free 24- and/or 31-bit storage. However, use of 31-bit storage is very strongly recommended. As previously mentioned, 24-bit storage is a scarce resource in many environments, and serious problems can arise if the entire pool of 24-bit storage is depleted by RPC applications.

An important consideration in this context is that Shadow RPC applications can run for a long time. This means that the storage used by these applications can be long lived. This is an important point for the overall design of RPCs running in the Shadow Server address space. The total virtual storage utilization of all host RPCs

executing concurrently must not exceed the available 24- and 31-bit virtual storage. In practice, there is often a large amount of unused 31-bit storage, whereas 24-bit storage may be sharply constrained.

One way of circumventing some of these constraints is to use Shadow RPC Direct RPCs which are in general much more short-lived than their Shadow RPC Direct counterparts.

RPC Libraries

Each NEON Client RPC is comprised of one or more load modules. The RPC name must be either a PDS member name or alias name in the RPC library. RPC load modules must be stored in the SDBRPCLD concatenation of the Shadow Server address space.

The Shadow Server JCL contains a SDBRPCLB DD statement. As many separate RPC libraries as necessary can be concatenated using this DD statement. If this DD statement is coded in the Shadow Server starter task JCL, all RPC load modules will be loaded from this library concatenation.

There are several reasons for storing RPC load modules in the RPCLIB concatenation:

- Data sets comprising the RPCLIB concatenation do ***not*** need to be APF authorized. By contrast, the libraries of the STEPLIB concatenation ***must*** be APF authorized. This approach allows host RPC applications developers to update the RPC library concatenation with a minimum of security. Stringent security procedures are often required before APF libraries can be updated. Additionally, this method provides isolation from the STEPLIB concatenation.
- Use of the RPC library concatenation improves performance.

Other DD Statements

Host RPCs can use any other DD statements in the Shadow Server address space JCL. Each installation can add additional DD statements for VSAM files, flat files, PDSs, etc. to meet application requirements. The DD statements used by the Shadow Server address space itself should not be changed. Please see the *Shadow Server User's Guide* for additional information about the Shadow Server address space JCL.

The *Shadow Server User's Guide* documents procedures for running the Shadow Server under TSO. This mechanism was specifically designed to facilitate the development of host RPCs. Each application developer can work with an individual copy of the Shadow Server running under TSO, allowing the developer to work independently. The Shadow Server runs unauthorized in the TSO environment and therefore cannot be used in any way to compromise system security. All accesses to data are associated with the TSO userid of the application developer. For additional information please see the *Shadow Server User's Guide*.

Data Transmission between the Client Application and the Host RPC

Data is transmitted between the client application and the host RPC using messages. Each message is a single buffer of data, and has a specific length (possible zero), specified by the sender. It will always be sent and received as a single entity. In other words the sender passes complete messages on the sending side, and the receiver receives complete messages on the receiving side. This approach is different from the TCP/IP stream approach where the data sent by one side can be received as multiple pieces on the other side, or multiple transmissions from the sender can be received together by the receiver.

In all cases, the messages sent between the client application and the host RPC are compressed to reduce network utilization. The compression/decompression process is completely transparent at both ends. The compression algorithm used is simple and fast, and consists of using compression factors of up to 10 and 20:1 for the sending and receiving of sparse data buffers (many blanks or binary zeros).

Using Host Data

Host RPCs can access and update many different types of host data including VSAM, DB2 and IMS. In general, host RPCs can use normal high-level language facilities to access and update these databases. However, special considerations do apply in some cases, and these considerations should be carefully reviewed before attempting to access and update the databases listed below.

DB2

Host RPCs can use either static or dynamic SQL to access DB2 databases, however, static SQL is used in most cases. Host RPC programs using either static or dynamic SQL can be prepared using standard DB2 program development procedures.

At run time, the plan name used by a host RPC with DB2 can be specified in the following two ways:

- **The client program that invokes the host RPC can specify the host DB2 subsystem name and the plan name.**

  In this case, both the DB2 subsystem name and the plan name will be padded with blanks and translated from ASCII to EBCDIC. Shadow Server will use the DB2 subsystem name and the plan name to establish a connection to DB2 immediately prior to the execution of the host RPC. This approach eliminates the need for the host RPC to establish its own connection to DB2.

  The connection to DB2 will automatically be broken as soon as the host RPC terminates. DSNALI close will be called to terminate the DB2 connection and release any DB2 resources. DSNALI close will be called with a close type of either SYNC or ABRT. By default, SYNC will be used to commit any

uncommitted changes. However, if the host RPC abended, ABRT will be used to roll back any uncommitted DB2 changes.

- **A host RPC can establish its own connection to DB2 using DSNALI.**

  This approach is not recommended. If a host application establishes its own connection with DB2 using DSNALI, then the authorization ID that DB2 uses to validate all host RPC requests to access/update data will be **undefined**. This restriction can be removed by installing the Shadow Server modification to the DSN3@ATH exit. The best approach is simply to let Shadow Server provide the DSNALI open and close calls rather than incorporating these calls into the host RPC application.

  Note:
  A slightly different link-edit procedure must be used for host RPCs versus DSNALI command applications. Host RPCs must be linked with DSNALI and DSNHLI2 rather than DSNELI. The correct approach is to include only DSNALI in the link-edit step.

IMS

Host RPCs can access and update IMS databases using the DBCTL interface. This interface allows any number of host RPCs to concurrently and independently access and update IMS databases, however, it does not provide access to IMS message queues. Host RPCs that use DBCTL to communicate with IMS can use all the IMS facilities that are available to CICS transactions. In other words, host RPCs have the same IMS programming facilities available as transactions running under CICS. Host RPCs use the same mechanism (DBCTL) to communicate with IMS as do CICS transactions running under CICS Version 3 and later releases.

DBCTL is a feature of IMS Version 3 and later releases. The DBCTL API is provided by either a separate DBCTL address space or by an IMS DB/DC system. See the IMS general information manual (GC26-4275) for additional information about installing and utilizing DBCTL. There are no special IMS generation or run-time parameters required to use the DBCTL API.

The DBCTL feature of the Shadow Server address space is not enabled by default. This feature must be enabled using the required product feature string letter, ‘I’. The product feature string is processed during product initialization.

Writing a Host RPC

Using DBCTL API

Host RPCs using the IMS interface can be written in any high-level language or assembler. In each case, the application programmer should call the language-specific interface routine. Host IMS RPCs are not passed a PCB list on entry. Instead they must schedule a PSB by calling the language-specific IMS interface function. This call (the function code is `'PCB'`) returns a PCB list to the caller.

The PCB list can then be used for subsequent IMS calls. Standard IMS function codes (`'GN'`, `'DELT'`, `'ISRT'`, `'REPL'`) can be used to get segments, delete segments, insert segments, and replace IMS segments.

The host RPC can either commit any changes it makes or terminate. If a host RPC neither commits changes nor terminates the PSB, Shadow Server will automatically perform these tasks after the host RPC terminates. Shadow Server will commit all changes made up to that point, if the RPC terminates normally. Shadow Server will roll back any uncommitted changes, if the RPC abends.

VSAM

Host RPCs running under Shadow Server can use VSAM data sets. Each host RPC must start by opening whatever data sets it needs. The DD statements for these data sets should be incorporated into the Shadow Server address space started task JCL. In the test environment, the VSAM data sets should be allocated by the TSO user running a test copy of the Shadow Server address space.

Host RPCs can use normal high-level language facilities to access and update VSAM records. Standard VSAM data sharing facilities can be used to coordinate updates to VSAM files.

RPC Debug Support

Version 4.5 of Shadow Direct provides access to a GUI, source level debugger for RPC. This feature is called the Visual Age ™ Remote Debugger (VAD) and is a product of IBM. It allows each programmer developing RPCs or stored procedures to debug them on his or her own personal computer.

A major benefit of this debugger is that a programmer no longer has to run a private copy of Shadow Server under TSO. In addition, this debugger requires no changes to the invoking ODBC application or to Shadow Direct on the host. Shadow Direct automatically invokes VAD on the same PC that is running the ODBC application.

For more information about the VAD debugger, check the IBM website at http://www-4.ibm.com/software/ad/c390/pt/.

To use the debugger:

1. Make sure that VAD and Shadow Direct, Version 4.5, have both been properly installed on your system. For more information about this step, check the IBM installation documentation for VAD, and the Shadow Installation Guide for Shadow Direct.

2. On the mainframe, compile and link the Stored Procedure written in C, C++, Cobol, or PL/I with the TEST option.

Note:
The output listing from the compile step must be stored in a file and not routed to SYSOUT.

3. Copy the load module to the standard RPC load module library, which will be a STEPLIB or RPCLIB in the started task environment. If the Shadow Server is invoked using TSO, the load module can also be stored in the ISPLLIB.

4. Use the ODBC Administrator to set the Host Debug option to the correct language type. To get to this screen, perform the following steps (for more detailed information about these steps, refer to the NEON Client User's Guide, Chapter 2, "Installing NEON Client"):

   a. Select the Neon_Client_Debug_Sample32 option from the User Data Sources screen.

   b. Click the <ADVANCED> button at the bottom of the Neon Client 32-bit screen.

   c. Click the <MORE> button at the bottom of the NEON Client Advanced Information Screen. This will take you to the NEON Optional Client Information Screen, as shown below:

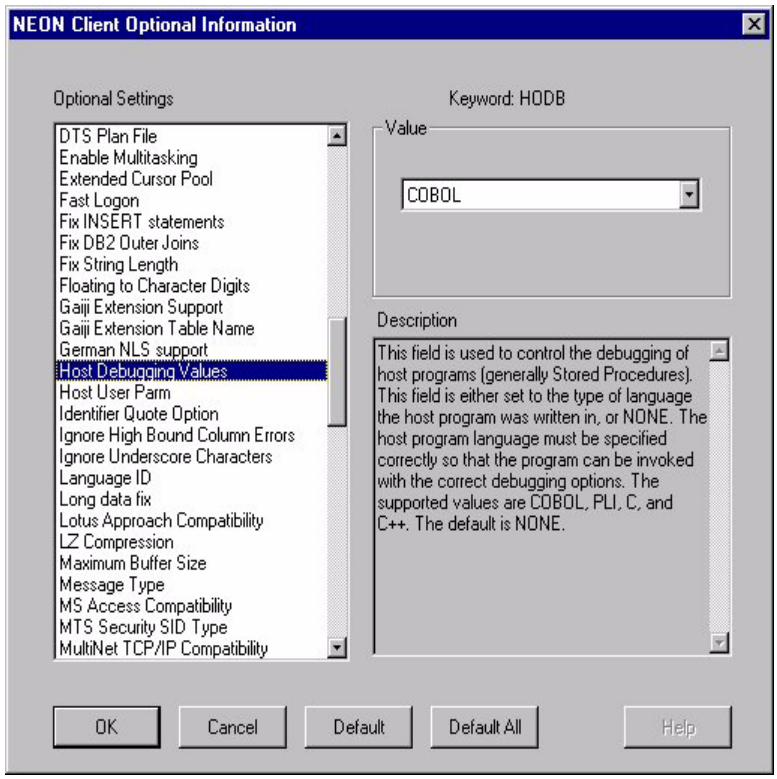


Figure 1–2. NEON Client Optional Information Screen

d. Select “Host Debugging Values” as the optional setting.

e. Select the appropriate keyword value. Options include:

- COBOL
- C
- C++
- PL/I

5. Start the VAD Remote Debugger Daemon from the Windows Start menu, or by clicking on an icon. You should see the following screen:

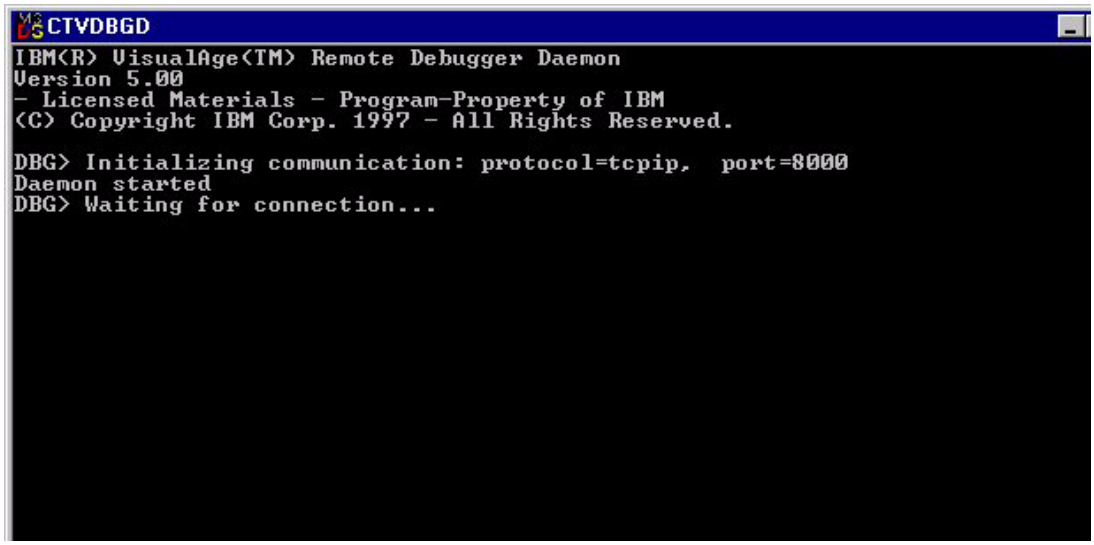


Figure 1–3. Daemon Startup Screen

6. Start the ODBC application that invokes the host RPC. The following figure shows an example of the GUI Debug screen. This screen will appear as soon as the ODBC appplication executes the SQL CALLS for the RPC:

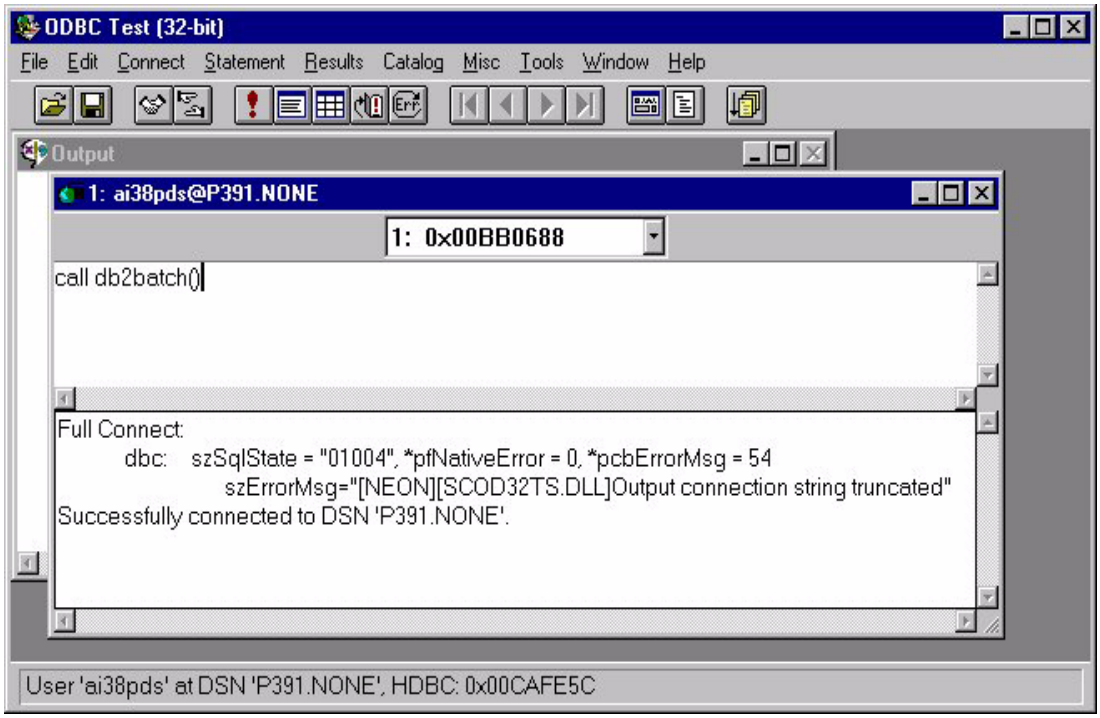


Figure 1–4. ODBC Test Debug Screen

7. Use the VAD as needed to debug the host program. Figure 1–5 and Figure 1–6 show examples of what the VAD screens look like when they are in the process of debugging.

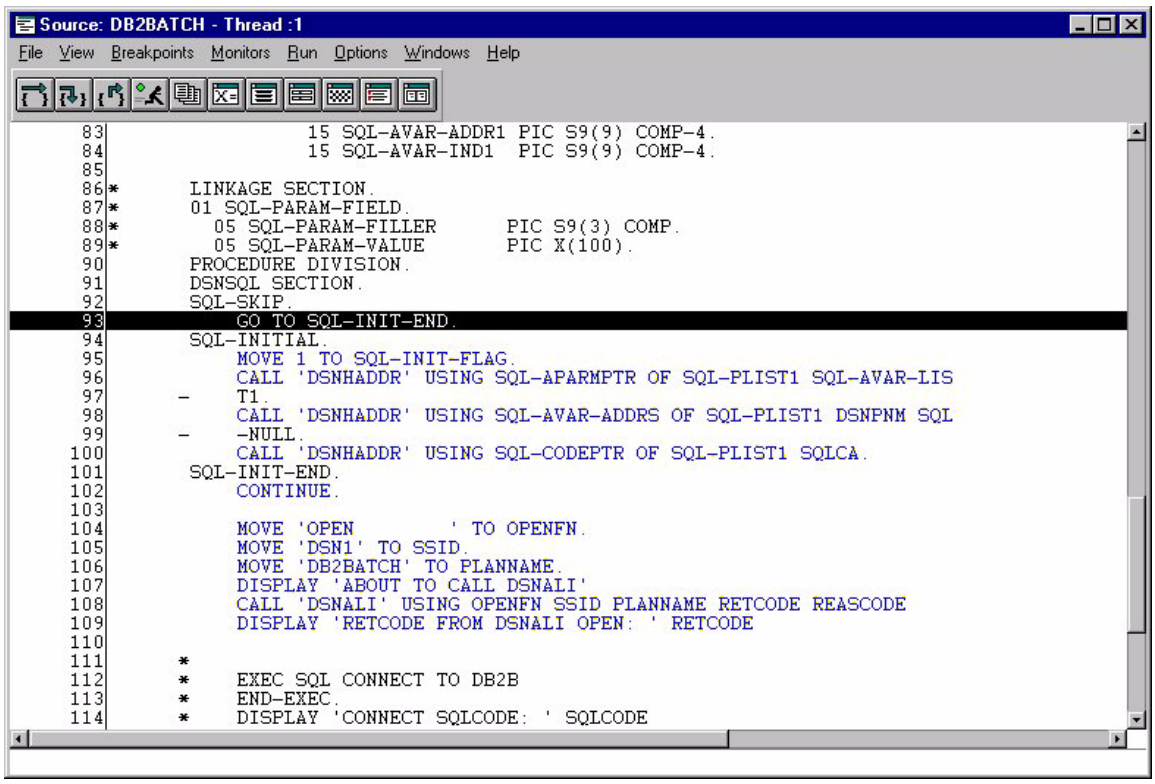


Figure 1–5.

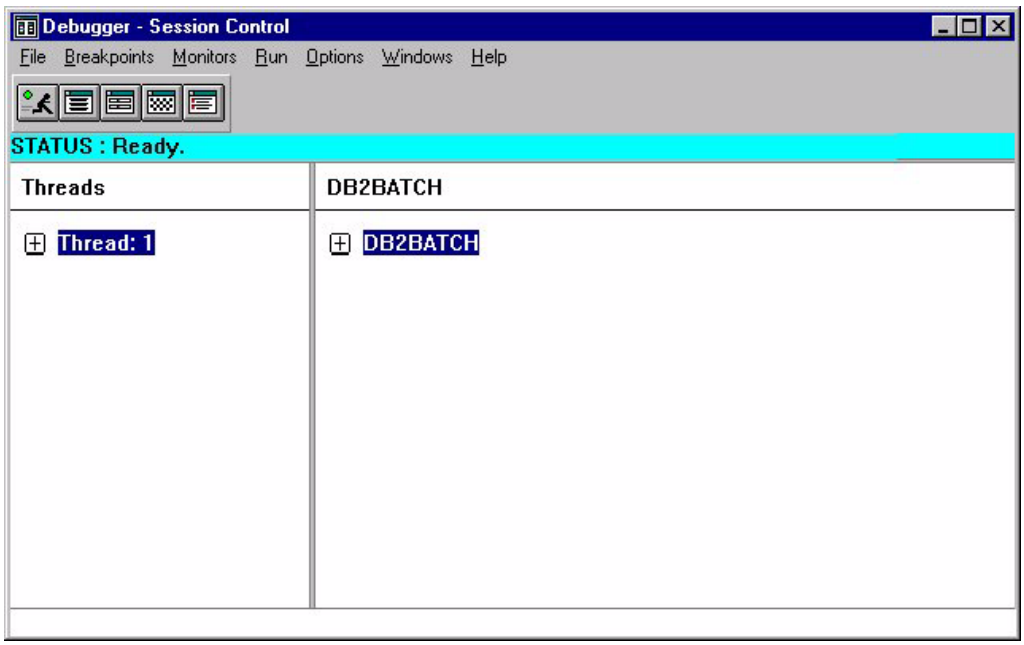


Figure 1–6.

The following information and restrictions apply to the VAD debugger:

- VAD can only be used to debug Shadow RPCs or Stored Procedure. It does not support DB2 Stored Procedures at this time, however, it will in the future.
- VAD can only be used with OE TCP/IP and Interlink TCP/IP as long as the interlink stack is being ussed via OE sockets. It does not, and never will, support LU6.2 and IUCV TCP/IP.
- VAD can only be used with LE/370, version 1.8 and later.
- VAD supports only OS/390 2.4 and later.
- VAD can be used with both started task and TSO versions of the Shadow Server. The started task version eliminates the need to configure a separate copy of Shadow Server (with its own port number) for each RPC developer, and the TSO version provides a separate copy of Shadow Server to each RPC developer.
- The current version of VAD gets a GPF in the termination phase. The GPF occurs after host RPC has completed execution, and has no harmful effect.

Client API Function Definitions

The Shadow RPC Direct API calls are used by Shadow RPC Direct applications to establish a connection to the host and to transmit data to and from the host. All these functions use the Pascal calling convention. None of these functions take a variable number of arguments.

The following functions are available:

- **SCAsciiToEbcdic:** Converts a string from ASCII to EBCDIC
- **SCEbcdicToAscii:** Converts a string from EBCDIC to ASCII
- **SCReadBuffer:** Receives a data buffer from the host
- **SCWriteBuffer:** Sends a data buffer to the host
- **SCWriteReadBuffer:** Writes a buffer to the host and receives data buffer

SCAsciiToEbcdic

IMS DirectSCAsciiToEbcdic converts character data from ASCII to RPC Direct EBCDIC. This function is normally used to convert data areas that are sent to the host.

Syntax

```
RETODBC SCAsciiToEbcdic(hdbc, rgbAscii, rgbEbcdic, cbValue)
```

Arguments

The **SCAsciiToEbcdic** function accepts the following arguments:

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
PTR	rgbAscii	Input	Pointer to storage area containing ASCII character data to be converted.
PTR	rgbEbcdic	Output	Pointer to output area where converted EBCDIC characters should be stored.
SDWORD	cbValue	Input	Number of bytes to convert from ASCII to EBCDIC.

Returns

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

When SCAsciiToEbcdic returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLError. The following table lists the SQLSTATE values commonly returned by SCAsciiToEbcdic and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message (function returns SQL_SUCCESS_WITH_INFO).
08003	Connection not open	Connection specified by hdbc argument was not open. Connection processes must be completed successfully (and the connection must be open) for the driver to perform this function.
S1009	Invalid argument value	Either rgbAscii or rgbEbcdic pointer was null.
S1090	Invalid string or buffer length	Value specified for cbValue was less than zero.

Comments

This function is used to convert a character string from ASCII to EBCDIC. The character string can be converted in place.

Note:
In the Visual Basic environment, fixed length character strings cannot be converted in place. Visual Basic imposes this restriction because it copies each of the character string arguments of this function into temporary data areas and then restores temporary data areas in an unpredictable order. However, variable length Visual Basic character strings may be converted in place using this function.

Code Example

None at this time.

Related Functions

For information about converting from EBCDIC to ASCII see the “SCEbcdicToAscii,” section on page 1-19.

SCEbcdicToAscii

IMS Direct/RPC Direct

SCEbcdicToAscii converts character data from EBCDIC to ASCII. This function is normally used to convert data areas that are received from the host.

Syntax

```
RETODBC SCEbcdicToAscii(hdbc, rgbEbcdic, rgbAscii, cbValue)
```

Arguments

The SCEbcdicToAscii function accepts the following arguments:

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
PTR	rgbEbcdic	Input	Pointer to storage area containing EBCDIC character data to be converted.
PTR	rgbAscii	Output	Pointer to output area where converted ASCII characters should be stored.
SDWORD	cbValue	Input	Number of bytes to convert from EBCDIC to ASCII.

Returns

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID HANDLE

Diagnostics

When SCEbcdicToAscii returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLError. The following table lists the SQLSTATE values commonly returned by SCEbcdicToAscii and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns `SQL_SUCCESS_WITH_INFO`.)
08003	Connection not open	Connection specified by hdbc argument was not open. Connection processes must be completed successfully (and the connection must be open) for driver to perform this function.
S1009	Invalid argument value	Either rgbAscii or rgbEbcdic pointer was null.
S1090	Invalid string or buffer length	Value specified for cbValue was less than zero.

Comments

This function is used to convert a character string from EBCDIC to ASCII. The character string can be converted in place.

Note:

In the Visual Basic environment, fixed length character strings cannot be converted in place. Visual Basic imposes this restriction because it copies each of the character string arguments of this function into temporary data areas and then restores temporary data areas in an unpredictable order. However, variable length Visual Basic character strings can be converted in place using this function

Code Example

None at this time.

Related Functions

For information about converting from ASCII to EBCDIC see the “SCAsciiToEbcdic,” section on page 1-17.

SCReadBuffer

RPC Direct

SCReadBuffer reads a complete data buffer from the host. Execution is suspended until the data buffer is available or an error is detected.

Syntax

```
RETODBC SCReadBuffer(hdbc, rgbValue, cbValueMax, pcbValue)
```

Arguments

The **SCReadBuffer** function accepts the following arguments:

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
PTR	rgbValue	Output	Buffer for input data. Part or all of this buffer may be filled with data obtained from host.
SDWORD	cbValueMax	Input	Maximum length of *rgbValue buffer.*
SDWORD FAR *	pcbValue	Output	Total number of bytes read into buffer pointed to by *rgbValue.*

Returns

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

When SCReadBuffer returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLError. The following table lists the SQLSTATE values commonly returned by SCReadBuffer and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	Connection specified by *hdbc argument was not open. Connection process must be completed successfully (and connection must be open) for driver to perform this function.*
08S01	Communication link failure	Communication link between driver and data source failed before or while the data buffer was being read from the host.

SQLSTATE	Error	Description
22003	Buffer size error	Buffer transmitted from host was larger than data area provided to receive buffer. Entire host buffer was discarded.
S1009	Invalid argument value	Value specified for argument *rgbValue was null*.
S1090	Invalid string or buffer length	Value specified for argument *cbvalueMax was less than zero.*

Comments

This function is used to read a buffer of data from the host. It will suspend execution until either a data buffer is received from the error or a communication error is detected. The size of the data buffer can range from zero to cbValueMax. Zero length buffers are supported and can be transmitted both to and from the host.

Note:

The buffer data area provided by the caller must be large enough to contain the entire buffer transmitted from the host. If the data area is not large enough, the entire buffer will be discarded and an error will be reported to the calling program.

Code Example

None at this time.

Related Functions

For information about	See
Converting data from ASCII to EBCDIC	SCAsciiToEbcdic
Converting data from EBCDIC to ASCII	SCEbcdicToAscii
Sending a buffer to the host	SCWriteBuffer
Writing a buffer to a host and receiving a reply	SCWriteReadBuffer

SCWriteBuffer

RPC Direct

SCWriteBuffer sends a buffer of data from the client application to the host RPC program. Execution of the client application program is suspended until the data is copied from the buffer provided by the caller.

Syntax

```
RETODBC SCWriteBuffer(hdbc, rgbValue, cbValue)
```

Arguments

The SCWriteBuffer function accepts the following arguments:

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
PTR	rgbValue	Input	rgbValue argument contains a pointer to data buffer that should be transmitted to host. This buffer is not null terminated.
SDWORD	cbValue	Input	cbValue argument contains number of bytes to be transmitted to host. This value must be greater than or equal to zero.

Returns

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE

Diagnostics

When SCWriteBuffer returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLError. The following table lists the SQLSTATE values commonly returned by SCWriteBuffer and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	Connection specified by hdbc argument was not open. Connection processes must be completed successfully (and connection must be open) for driver to perform this function.

SQLSTATE	Error	Description
08S01	Communication Link Failure	Communication link between driver and data source to which driver was connected failed before function completed processing.
S1009	Invalid argument value	rgbValue argument was a null pointer.
S1090	Invalid string or buffer length	Value specified for argument cbValue was less than zero.

Comments

This function is used to write a buffer of data from the client to the host. The length of the buffer can range from zero up to approximately 30,000 bytes. Control returns to the invoking application as soon as the data in the application buffer is copied into the communication buffers. There is no guarantee that when this function returns, the data has actually been transmitted to the host, nor is there any way of suspending execution until the data has been successfully transmitted.

This function will NOT turn the line around after the write operation is completed. This means that the host RPC will not be able to send a reply to the client. The SCWriteReadBuffer function should be used if the host RPC is expected to send a response buffer.

Note:
This is a consideration only for LU 6.2 client/server sessions.

Code Example

None at this time.

Related Functions

For information about	See
Writing a buffer to a host and receiving a reply	SCWriteReadBuffer
Reading a buffer of data from the host	SCReadBuffer
Converting data from ASCII to EBCDIC	SCAsciiToEbcdic
Converting data from EBCDIC to ASCII	SCEbcdicToAscii

SCWriteReadBuffer

RPC Direct

SCWriteReadBuffer writes a buffer of data to the host and receives a reply buffer from the host. Execution of the client application program is suspended until a buffer is received from the host, or a communication error occurs.

Syntax

```
RETODBC SCWriteReadBuffer (hdbc, rgbValue, cbValue, cbValueMax,
pcbValue)
```

Arguments

The SCWriteReadBuffer function accepts the following arguments:

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
PTR	rgbValue	I/O	rgbValue argument points to the data buffer used to both send and receive data. Buffer must initially contain data that will be transmitted to host. This buffer is not null-terminated. Upon successful completion of this function, buffer will contain data received from host.
SDWORD	cbValue	Input	cbValue argument contains number of bytes of data to send to host.
SDWORD	cbValueMax	Input	Maximum length of *rgbValue buffer.*
SDWORD FAR *	pcbValue	Output	Total number of bytes read into buffer pointed to by the *rgbValue argument*.

Returns

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO,
- SQL_ERROR
- SQL_INVALID_HANDLE.

Diagnostics

When SCWriteReadBuffer returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLError. The following table lists the SQLSTATE values commonly returned by SCWriteReadBuffer and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	Connection specified by *hdbc argument was not open. Connection processes must be completed successfully (and connection must be open) for driver to perform this function.*
08S01	Communication link failure	Communication link between driver and data source to which driver was connected failed before function completed processing.
22003	Numeric value out of range	Buffer transmitted from host was larger than data area provided to receive buffer. Entire host buffer was discarded.
S1009	Invalid argument value	rgbValue argument was a null pointer.
S1090	Invalid string or buffer length	Value specified for argument cbValue was less than zero.

Comments

This function is used to write a data buffer to the host and then receive a reply. Control is not returned to the invoking application until either the reply is received or a communication error is detected. This function is normally used to send "transactions" to the host and then receive a reply. If the data area is not large enough, the entire buffer will be discarded and an error will be reported to the calling program.

This function will turn the line around after the operation is completed. This means that the host PC will be able to send a reply to the client. The SCWriteBuffer function should be used if multiple buffers must be sent to the host without an intervening read.

Note:
This is a consideration only for LU 6.2 client/server sessions.

Code Example

None at this time.

Related Functions

For information about	See
Writing a buffer to a host	SCWriteBuffer
Reading a buffer of data from the host	SCReadBuffer
Converting data from ASCII to EBCDIC	SCAsciiToEbcdic
Converting data from EBCDIC to ASCII	SCEbcdicToAscii

CHAPTER 2:
ODBC CALL RPCs

This chapter covers programming information for ODBC CALL RPCs, which can be used to access most types of data residing on the mainframe.The information covered here includes ODBC CALL RPC examples, sample ODBC CALL RPCs for VSAM, and other sample RPCs.

This chapter applies to Shadow Direct and Shadow OS/390 Web Server.

Introduction

ODBC CALL RPCs can be used to access almost any type of data residing on the mainframe in the same manner RPC Direct RPCs can. This includes DB2, IMS, VSAM, PDSs, Flat Files, ADABAS, and M204. However, ODBC CALL RPCs have a major advantage over RPC Direct RPCs:

- ODBC CALL RPCs do not require any additional coding from the client application.
- Relational ODBC result sets are returned.
- Multiple RPCs can be executed from the same connection.
- ODBC CALL RPCs execute and end, unlike RPC Direct RPCs which remain loaded for the life of the connection or are terminated.

ODBC CALL RPCs can be executed with a CALL statement from any ODBC compliant client application, such as Visual Basic, Powerbuilder, MS-Access, etc. Since they are written using APIs provided on the host, they can return an ODBC result set to the client application. With ODBC CALL RPCs, the remote programs execute and end similar to the way a normal DB2 query executes and ends. The results of the RPC are accessed in the same manner as the results of a DB2 query.

ODBC CALL RPC Examples

All the source to the sample ODBC CALL RPCs is located in the `NEON.SV040100.SAMP` dataset on the MVS host. Compiled copies of these samples are also provided in the `NEON.SV040100.RPCLIB` dataset.

The Visual Basic application, VBDEMO, which is shipped with the client ODBC drivers, can be used to execute the sample ODBC CALL RPCs. In order to invoke the RPC, use the CALL statement preceding the RPC. Parameters to the RPCs are defined by placing the parameters with parenthesis. Each parameter is separated by a comma, literals are placed in quotes, and numeric data is left alone.

Example:

```
CALL RPCNAME('This is a literal','The next parameter is a
number',100)
```

Sample ODBC CALL RPC for VSAM

NEON.SV040100.SAMP(SDCOVSP) is a sample COBOL program that returns records from a VSAM dataset. This program can return all rows of the VSAM dataset or only the rows specified by an optional parameter.

Use the following steps to set up the sample application:

1. Allocate the sample VSAM cluster and populate it with sample data. Job DEFSTAFF in NEON.SV040100.CNTL will allocate the VSAM cluster and repro the sample data into the dataset.

2. Uncomment the SDBVS01 DDNAME in the main SDBB started task PROC. Then update the DSNAME with the fully qualified name of the VSAM dataset allocated in step 1, and restart Shadow Server. You can avoid allocating the VSAM dataset within the SDBB started task by using dynamic allocation within your RPC application.

3. The sample SDCOVSP program should already be compiled and linked and placed in the NEON.SV040100.RPCLIB dataset. If necessary, the sample program can be recompiled and re-linked using the source code provided in NEON.SV040100.SAMP.

To run the sample, you can use any ODBC-compliant application on your client workstation to issue a CALL *program_name*. The client application must have the ability to run a user-defined SQL script, such as MS-Query, Visual Basic, etc. The VBDEMO application that is shipped with Shadow Direct can be used for this.

The sample RPC to VSAM is a prime example of how to use Shadow Direct's unique ODBC CALL RPC approach for developing your client-server applications. The sample SDCOVSP RPC will:

- Read the VSAM dataset.
- Place the output in a relational ODBC result set.
- Return the results to the calling client application.

There are six ODBC calls used in this sample for this purpose. These calls and others have all been simplified for the application developer by a common header file that should be included in your RPC applications. These files provide all the definitions needed for support of the Host APIs, and also contain documentation for each API. The following header files are included in the SDB.SAMP dataset:

- SBCPHD For COBOL
- SCCPHD for C
- SPCPHD for PL/1

The following procedure division shows how to emulate ODBC calls in a host RPC, using the program SDCOVSP as an example (for detailed information about each parameter, please see the comments in the Header files):

1. `CALL 'SDCPGI' USING CONNECTION-HANDLE SQL-USER-NAME SQL-USERID SQL-USERID-LEN SQL-USERID-ACTUAL-LEN.`

   SDCPGI or SQLGETINFO can be used to return information from Shadow Server about the current environment. Examples of this information include Userid of the person calling the RPC, the DB2 subsystem being accessed, etc. For a complete list of possible values, refer to the Header file in the `SDB.SAMP` dataset.

   - **`SQL-USER-NAME`** is the actual information requested.
   - **`SQL-USERID`** is the pointer to storage for the information.
   - **`SQL-USERID-LEN`** is the maximum length of the requested data.
   - **`SQL-USERID-ACTUAL-LEN`** is the actual length of the returned information.

2. `CALL 'SDCPNP' USING STATEMENT-HANDLE SQL-PARAM-COUNT.`

   `SDCPNP` or `SQLNUMPARAMS` is used to return to the program the number of parameters sent to the RPC. This number will be returned in `SQL-PARAM-COUNT`.

3. `CALL 'SDCPDP' USING STATEMENT-HANDLE`

   `SQL-PARAM-NUMBER`

   `SQL-DATA-TYPE`

   `SQL-PRECISION`

   `SQL-SCALE`

   `SQL-NULLABLE-TYPE`

   `SQL-PARAM-TYPE`

   `SQL-PARAM-ADDRESS`

   `SQL-PARAM-LENGTH`

   `SDCPDP` or `SQLDESCRIBEPARAM` performs a describe parameter on behalf of an ODBC CALL RPC. This call is used to obtain information about the parameter passed from the client to the host. SDCPDP should be executed for each parameter being passed to the RPC.

   - **`SQL-PARAM-NUMBER`** should be set by the programmer to the parameter number being requested.

- **SQL-DATA-TYPE** returns the datatype of the parameter, SQL-VARCHAR, SQL-SMALLINIT, etc.
- **SQL-PRECISION** returns the precision of the parameter.
- **SQL-SCALE** returns the scale of the parameter, used mainly with decimal values.
- **SQL-NULLABLE-TYPE** specifies whether or not the parameter is nullable.
- **SQL-PARAM-TYPE** returns the type of parameter, which will mainly be SQL-PARAM-INPUT.
- **SQL-PARAM-ADDRESS** returns the pointer in storage for the parameter.
- **SQL-PARAM-LENGTH** returns the actual length of the column. The length will be the same as the precision except for variable length fields (character and binary). For variable length fields, the length will be the current length.

4. CALL 'SDCPBC' USING STATEMENT-HANDLE

   SQL-COLUMN-NUMBER

   SQL-C-DEFAULT

   SQL-SMALLINT

   SQL-PRECISION

   SQL-SCALE

   SQL-NO-NULLS

   ID-VALUE

   SQL-COLUMN-LEN

   SQL-COLUMN-NAME

   SQL-COLUMN-NAME-LEN

   SDCPBC or SQLBINDCOL performs a bind column on behalf of an ODBC call RPC. This call must be executed for each column being returned to the client. SQLBINDCOL assigns the storage and data type for a column in a result set. The following values will should be set by the RPC to properly bind the column:

   - **SQL-COLUMN-NUMBER** is the number of the column in the result set.
   - **SQL-C-DEFAULT** is the C data type column of the column data. This value must be set to SQL_C_DEFAULT at this time. This means that the C type must match the SQL type.

- **SQL-SMALLINIT** is the SQL data type of the column data.
- **SQL-PRECISION** is the precision of the data. This value is used primarily with character and decimal data. If the value is an integer, it should be set to "1".
- **SQL-SCALE** is the scale of the data type, used primarily with decimal data.
- **SQL-NO-NULLS** is the parameter specifying whether or not the data can be nullable. This value will either be SQL-NO-NULLS or SQL-NULLABLE.
- **ID-VALUE i**s the actual value to be bound. If this value is a variable field, the first two bytes must contain the length of the data.
- ***SQL-COLUMN-LEN** is used to determine if the data is NULL; should be set to "1" if the data is not NULL.
- **SQL-COLUMN-NAME** is the name of the column.
- **SQL-COLUMN-NAME-LEN** is the precision of the name of the column.

Note:
When using RPCs to access DB2, if you want to return NULL data to the client application, you will need to update the SQL-COLUMN-LEN field and set it to SQL-NULL-DATA or -1. For every column that may contain NULL data, a separate value should be used for SQL-COLUMN-LEN. When a row is fetched from the database, the RPC needs to inspect whether the data is NULL. If it is NULL, the corresponding SQL-COLUMN-LEN value defined in the SQLBINDCOL call must be set to SQL-NULL-DATA or -1.

```
5. CALL 'SDCPTH' USING STATEMENT-HANDLE SQL-THROW-ROW.
```

SDCPTH or SQLTHROWROW sends a row from the RPC into the output buffer created using the SQLBINDCOLUMN calls. This command is issued after a row of data has been retrieved from the requested database. Calling SDCPTH with a parameter of SQL-THROW-DONE signifies no more data is to be returned and flushes the buffer, sending the results down to the client application.

6. CALL 'SDCPSE' USING ENVIRONMENT-HANDLE

   CONNECTION-HANDLE

   STATEMENT-HANDLE

   SQLSTATE-DATA-AREA

   NATIVE-ERROR-CODE-AREA

   ERROR-MESSAGE-AREA

   FB256 ERROR-MSG-LENGTH-AREA

   SDCPSE or SQLERROR is used to obtain error information that may have been stored by a prior function call. Environment-Handle, Connection-Handle and Statement-Handle are ignored since only one host RPC can execute at a time.

   - **SQLSTATE-DATA-AREA** must be set to at least 6 bytes to allow for the return of the SQL-Code from this call.
   - **NATIVE-ERROR-CODE-AREA** returns a value describing the error.
   - **ERROR-MESSAGE-AREA** returns a pointer to storage for the error message text. This will always be null-terminated.
   - **FB256** is the maximum length of the error message buffer; the recommended setting is 256.
   - **ERROR-MSG-LENGTH-AREA** returns the actual length of the error message.

7. CALL 'SDCPRS' USING CONNECTION-HANDLE TRACE-MESSAGE-AREA SQL-NTS NATIVE-ERROR-CODE-AREA.

   SDCPRS or SQLRETURNSTATUS returns the status to the client from an ODBC call. The status data determines the return code from the SQLEXECDIRECT, SQLPREPARE, or SQLEXECUTE function that started the RPC. The client application can retrieve the status data (message and native code) by calling SQLERROR. The Connection-Handle is ignored since only one host RPC can execute at a time. The following parameters are required:

   - **ERROR-MSG-LENGTH-AREA** returns the actual length of the error message.
   - **TRACE-MESSAGE-AREA** is the address of the message text.
   - **SQL-NTS** is the length of the message text to be returned. This value can be an actual length or SQL-NTS can be specified if the message is null-terminated.
   - **NATIVE-ERROR-CODE-AREA** is the Native Error code. If this value is negative, the client return code will be SQL-ERROR. If this value is

positive, the client return code will be SQL-SUCCESS-WITH-INFO. This field cannot be zero.

Other Sample RPCs

Other sample RPC programs are available for use in the `NEON.SV040100.SAMP` dataset. These RPC samples access various other types of databases using COBOL, PL/1 and C. All use the same types of ODBC function calls as the VSAM sample above, in addition to other function calls necessary to access a desired database, e.g. IMS, M204. The following Cobol RPC samples are provided; () indicates that the sample supports an optional parameter.

SDCOIM

Reads sample data from the IMS parts database using IMS/DBCTL. The IMS PARTs database is provided by IBM during the IMS IVP install.

SDCOIMAP

Uses IMS/APPC to access the IMS Parts database. The IMS PARTs database is provided by IBM during the IMS IVP install.

SDCOM24P()

Uses IFAM calls to access the sample M204 database provided by M204.

SDCODB()

A sample DB2 RPC that reads the Q.staff table provided by QMF.

SDCOCIEC

A sample RPC to access CICS using EXCI.

Writing RPCs that Access DB2

With DB2 RPCs, such as the sample SDCODB, special considerations must be followed. Refer to the `NEON.SV040100.CNTL` dataset, member ***cob2db2,*** for the Sample JCL to compile and link a Cobol for MVS RPC.

1. Since SDCODB uses static SQL, the program must first be run through the DB2 precompiler to produce a DB2 DBRM.

2. The code is then compiled and linked. **On the link step, DSNALI must be linked into the RPC load module.**

3. Since Shadow Server will manage opening the thread to DB2 before connecting to the Shadow Server, either in the ODBC datasource definition or in the connection string, the PLAN parameter should be set to the name of the plan bound into DB2 for this RPC. Optionally multiple RPC DBRMs can be placed into a single plan.

> ***Note:***
> If the 4th character of the Plan Name is an R, the NEON Client ODBC driver assumes that your application is using a plan where the plan was bound using an Isolation value of Repeatable Read. If you are not using Repeatable Read please ensure that your plan name does NOT have an R in the 4th character of the plan name as does the Shadow default plan SDBR1010. If it is any other character than an R we assume the plan was bound with an Isolation Level of Cursor Stability.

4. In the datasource, Static SQL (CD) should be selected or set to YES in the connection string.

5. Under the Advanced/More option of the datasource definition, Always Convert Dynamic SQL (ALCD) should be set to NO. If it is not, the Shadow ODBC driver will attempt to convert ALL SQL sent through the driver to Static. Although all the SQL in the RPC will run statically, the initial CALL statement for the RPC is run dynamically. ALCD=YES can also be set in the connection string of the application.

   > ***Note:***
   > If you are running client code dated 11/19/97 or above, selecting Static SQL and setting ALCD=NO is no longer required.

6. Customer-written PL/I programs must **NOT** use the FETCH, CALL, or RELEASE statements if SSL is used for encryption of session data. This is due to Language Environment for VM & MVS restrictions.

Special Considerations for Cobol II

Cobol II was originally designed with CICS in mind. Under CICS, Cobol II programs are kept serialized and thus can be Reentrant. If you wish to use Cobol II with NEON Client, the following restrictions apply:

- All RPCs, including RPC Direct, must be compiled with NORENT, NORES and NODYNAM.
- All RPCs, including RPC Direct, must also be linked with NORENT**.**
- **All RPCs must be AMODE 31**, regardless of language, if you are passing parameters to the RPC.
- Programs must be linked with AMODE 31, since Shadow stores the parameter in a 31-bit address.

These restrictions, with the exception of being AMODE 31, are not necessary if:

- You are using the latest releases of the Cobol compiler, including Cobol for MVS or Cobol/370 as it was recently called.
- You are using PLI, C or Assembler.

The following information (*in italics*) is from the *IBM VS Cobol II Application Programming Guide for MVS and VSE*:

> *Multiple tasks within the same region are supported for RESIDENT run units only under CICS. Multiple OS tasks within the same region are not supported by COBOL in conjunction with the Library Management feature (RES option). COBOL does not preclude multitasking if the run units are compiled with NORES. However, any restrictions and conventions of multitasking imposed by the operating system, access methods, and so on, must be observed.*
>
> *With Cobol II you have the following valid combinations of DYNAM, RESIDENT, and RENT:*
>
> *There are five valid combinations of the DYNAM, RESIDENT, and RENT compiler options.*
>
> - ***NORENT and NORES and NODYNAM***
> *Nonreentrant code, no COBOL Library Management feature, CALL literal is static call.*
> - ***NORENT and RES and NODYNAM***
> *Nonreentrant code, COBOL library routines called dynamically, CALL literal is static call.*
> - ***NORENT and RES and DYNAM***
> *Nonreentrant code, COBOL library routines and user subprograms all called dynamically.*
> - ***RENT and RES and NODYNAM***
> *Reentrant code, COBOL library routines called dynamically, CALL literal is static call.*
> - ***RENT and RES and DYNAM***
> *Reentrant code, COBOL library routines and user subprograms all called dynamically.RENT or DYNAM causes the RESIDENT option to be forced on.*

As the IBM restrictions indicate, RES is only allowed running multiple tasks under CICS. As a result, the only valid combination for NORES is NORENT, NORES and NODYNAM. Attempting to run two simultaneous RPCs under Shadow with DYNAM or RENT will result in an abend followed by a IGZ015I Cobol error which states:

> *A recursive call was attempted to a program that was still active. COBOL does not allow reinvocation of a program which has begun execution, but has*

not yet terminated. For example, if program A calls program B, program B cannot call program A. The job was canceled.

Running Cobol II RPCs can also cause storage problems. Since all RPCs must be coded NORENT, NORES and NODYNAM, all external routines must be statically linked in with the RPC, causing a separate RPC to be loaded for each execution of the RPC. If the number of loaded Non-Reentrant RPCs exceeds the amount of private storage available to Shadow Server, an S806 abend will occur. In this case, one of the following actions must be chosen:

- Upgrade to Cobol for MVS.
- Set up multiple Shadow Servers and do work load balancing.
- Place the RPCs in the RPC Special Requirements parameter list.

The RPC Special Requirements parameter list is used to define the RPCs that should be serialized. By placing the RPC name on this list, Shadow Server will serialize all executions of these RPCs. This means that no more than one of these RPCs will be allowed to execute at a time. All other execution requests for the same RPC will be queued for execution. This function can cause significant performance degradation for a high-transaction environment.

The Special Requirements parameter list can be specified in the Shadow Initialization exec member, SDBxIN00, located in the dataset allocated to SYSEXEC in the Shadow Server started task. The parameter is `RPCxxSPECIALREQ` where `xx` is a number between 01 and 10. The syntax for placing this in the Shadow Initialization exec is:

`MODIFY PARM NAME (RPC`*`xx`*`SPECIALREQ)VALUE(`*`rpcname`*`)`

You can also add these RPCs dynamically using option 5.2 from the Shadow Server Primary Options menu (SDF ISPF panel), and selecting the the PRODRPC group of parameters. However, all changes made in this manner are only temporary until the next time the Shadow Server started task is recycled. To make the changes permanent, they should be placed in the SDBxIN00 initialization exec.

For more detailed information and restrictions about running Cobol II, please refer to the *IBM VS Cobol II Application Programming Guide for MVS and VSE*.

Special Considerations for Cobol for MVS and Other LE/370 Languages

With Cobol for MVS or other LE/370 languages, your RPCs will perform best if you minimize the amount of storage that your application programs use to below the 16 megabyte line. Whenever possible, design your RPCs to run above the 16 megabyte line. If you use the default LE/370 run-time options, you can only run a small number of RPCs concurrently within the Shadow Server address space. For programs that can run above the line, do the following:

- For COBOL programs, use the RES and DATA(31) compiler options.

- Link-edit the program with the AMODE(31) and RMODE(ANY) attributes.
- Use the following LE/370 run-time options:
  - HEAP(,,ANY) allocates program heap storage above the 16MB line.
  - STACK(,,ANY) allocates program stack storage above the line.
  - STORAGE(,,,4K) reduces the storage area below the 16MB line to 4K.
  - BELOWHEAP(4K,,) reduces the below the 16MB heap storage to 4K.
  - LIBSTACK(4K,,) reduces the library stack below the line to 4K.
- Compile and link-edit the RPC as reentrant.
- ALL31(ON) indicates that all programs contained in this stored procedure run with AMODE(31) and RMODE(ANY).

JCL for compiling and linking a user CEEUOPT module can be found in member CEEWUOPT of the `NEON.SV040100.CNTL` dataset. This JCL can be used to override system installation defaults for Cobol for MVS. CEEWUOPT has been modified to contain the current recommendations for running Cobol for MVS application under Shadow Server. This module should be linked with any Cobol for MVS programs in order for the options to be used. If not, the installation defaults for Cobol for MVS will be used. Optionally, the recommended changes can be made to CEEDOPT, thus making these options the default, and eliminating the required link for every RPC with CEEUOPT.

To improve performance, all eligible LE/370 runtime modules should be moved into the MLPA. The following information (*in italics*) was obtained from the *IBM Language Environment for MVS & VM Installation and Customization on MVS Release 5 Document Number SC26-4817-06.* Please reference this manual for up-to-date information.

Placing Language Environment Modules in Shared Storage

Placing routines in shared storage reduces overall system storage requirements. Also, initiate/terminate (init/term) time is reduced for each application, since load time decreases.

All of the re-entrant modules in CEE.V1R5M0.SCEERUN can be included in shared storage on MVS. To include them:

- *Authorize the data set CEE.V1R5M0.SCEERUN.*
- *Include CEE.V1R5M0.SCEERUN in the LNKLSTnn concatenation (optional for MVS/ESA Version 4).*
- *Create an IEALPAnn member in SYS1.PARMLIB that lists the modules to be made resident in the MLPA when the system is IPLed.*

Several members are installed in CEE.V1R5M0.SCEESAMP for you to use as examples in creating your IEALPAnn member. The table below lists the members, their content, and the level of MVS/ESA they are to be used in. Note that the

format of the IEALPAnn member changed between MVS/SP Version 3 and MVS/ESA SP Version 4. Use the format appropriate for the release level of MVS at your site.

Member Name	Description	MVS/ESA Level
CEEWMLPA	*All language Environment base modules eligible for the LPA except callable service stubs.*	*Version 3*
EDCWMLP1	*All C/C++ component modules eligible for LPA.*	*Version 3*
IGZWMLP1	*All language Environment COBOL component modules eligible for LPA assuming modified, full COBPACKs (must reside below the 16M line).*	*Version 3*
IGZWMLP2	*All language Environment COBOL component modules eligible for LPA assuming COBPACKs will reside above the 16M line (All routines with RMODE (ANY)).*	*Version 3*
IBMALLP1	*All language Environment PL/I component modules eligible for LPA.*	*Version 3*
AFHWMLP1	*All language Environment FORTRAN modules eligible for LPA.*	*Version 3*
CEEWMLP2	*See description for CEEWMLPA.*	*Version 4*
EDCWMLP2	*See description for EDCWMLP1.*	*Version 4*
IGZWMLP3	*See description for IGZWMLP1.*	*Version 4*
IGZWMLP4	*See description for IGZWMLP2.*	*Version 4*
IBMALLP2	*See description for IBMALLP1.*	*Version 4*
AFHWMLP2	*See description for AFHWMLP1.*	*Version 4*

If you want to load modules into the LPA under MVS/ESA Version 4, you do not need to place CEE.V1R5M0.SCEERUN in the LNKLSTnn concatenation. For earlier versions of MVS, you must do one of the following:

- *Add CEE.V1R5M0.SCEERUN to the LNKLSTnn concatenation.*
- *Make the non-LPA modules available to steps that run Language Environment applications by either:*
- *Copying the non-LPA modules to a data set that is in the LNKLSTnn concatenation, or*
- *Copying the non-LPA modules to a data set that can be used as a STEPLIB or a JOBLIB.*

Using the entire CEE.V1R5M0.SCEERUN dataset as a STEPLIB defeats the purpose of placing the modules in the LPA.

Shared Storage Considerations

Modules you copy into another (non-LPA) data set are not automatically updated by SMP/E when you apply a service update. You must rerun your copy job after

you apply service to Language Environment to make the updated modules available in the LNKLSTnn data set or in the STEPLIB.

Examine the lists carefully to make sure that you are installing the correct module for the national language support you have installed. Comments in CEEWMLPA, CEEWMLP2, EDCWMLP1, EDCWMLP2, IBMALLP1, and IBMALLP2 identify the mixed-case U.S. English modules and the Japanese modules. In IGZWMLP1, IGZWMLP2, IGZWMLP3, and IGZWMLP4 remove the module name IGZCMGEN if U.S. English mixed-case is not installed and add IGZCMGJA if Japanese is installed and you want it to be in the LPA.

Refer to the following books for more information on including modules in the LPA:

- *MVS/ESA System Programming Library: Initialization and Tuning for MVS/SP Version 3, GC28-1828*
- *MVS/ESA Initialization and Tuning Reference for MVS/ESA SP Version 4, GC28-1635*
- *MVS/ESA Initialization and Tuning Reference for MVS/ESA SP Version 5, SC28-1452*

When all recommended modules have been placed into MLPA, the runtime library from the Shadow Server started task SDBRPCLB concatenation can be removed to take advantage of the modules in MLPA.

It is also recommended that the PRELOAD option be used to preload the following LE runtime modules:

- CEEEV005
- IGZEINI
- IGZEPLF
- IGZEPCL
- CEEBINIT
- CEEPLPKA
- IGZCPAC

Using LE/370 languages for RPCs with Shadow Direct provides several important advantages:

- Shadow Direct can exploit the LE/370 Library Routine Retention feature. If the LIBKEEP parameter (in the PRODRPC parameter group) is set to YES, Library Routine Retention support is enabled for LE/370 programs. Shadow Direct will preload the required Library Retention support routines at initialization time, and create the Library Routine Retention environment for each transaction program TCB. Initial tests have shown a 50% decrease in total CPU time for RPCs executed with the new Library Retention support enabled. Benchmarks are easily performed by simply turning the LIBKEEP parameter on and off. The following `MODIFY PARM` statement placed in the Shadow Initialization exec, SDBxIN00 will turn on the Library Routine Retention support:

```
MODIFY PARM NAME(LE370LIBKEEP)VALUE(YES)
```

- Shadow Direct provides an RPC Preload feature. In high transaction volume environments, applications that utilize RPCs must repeatedly invoke operating system services to bring programs into memory for execution. This can cause severe performance degradation and overhead, to the point of eliminating the performance benefit of RPCs.

  Shadow Direct provides an extremely effective solution to this problem. The PRELOAD option when set to YES will preload programs from a special library designed to contain RPCs which have very sensitive response time requirements. Shadow Direct will also bypass operating systems services which are typically used to pass control to programs and give control to these programs directly. This library should be allocated to the SDBRPCPL ddname in the Shadow Server started task JCL.

  The performance improvement and CPU time reduction when using the Preload feature has shown to be substantial for high volume RPC transactions. The Preload feature can also be used with PLI or C. The only requirement for these RPCs is that they be compiled and linked with the REENTRANT option. With the following parameter set, all RPCs allocated to the SDBRPCPL ddname in the Shadow Server Started Task JCL will be preloaded at Shadow Server startup time.

  ```
  MODIFY PARM NAME(PRELOAD)VALUE(YES)
  ```

  Note:

  Any changes made to these RPCs will require the Shadow Server started task to be recycled in order to pick up the new changes. Also, since each preloaded RPC will be stored in the Shadow Server private area below the 16 meg line, the more RPCs loaded will decrease the amount of private storage available for Shadow Server's use. Only highly used RPCs that have sensitive response time requirements should be placed in this library.

- Support for compiling programs with DYNAM option.

Using ODBC CALL RPCs in Visual Basic

The following sample Visual Basic code is an example of using Visual Basic with DAO to return the results of an SQL query and returning the results from the sample ODBC CALL RPC, SDCOVSP. This sample can be found on the NEON Systems CD, in the `sample/vb4/daotest` directory.

```
Dim myDB As Database
Dim MyRs As Recordset
Dim MySQL As String
Dim MyRPC As String
Dim rc As String
```

```
Set myDB = Workspaces(0).OpenDatabase("", False, False, "ODBC")
MyRPC = “call sdcovsp”
MySQL = “Select * from Q.STAFF”

You can base the record set after a DB2 table, a query, or the
RPC.
‘SDCOVSP also returns the same column names as the Q.Staff table.

’Set MyRs = myDB.OpenRecordset(MyRPC, dbOpenDynaset,
dbSQLPassThrough)
’Set MyRs = myDB.OpenRecordset(MySQL, dbOpenDynaset,
dbSQLPassThrough)

‘Q.STAFFI is a DB2 table with a unique index on the id column.

Set MyRs = myDB.OpenRecordset(“Q.STAFFI”, dbOpenDynaset)

The below Text fields are text boxes on the main form
‘The record set is updatable if the record set is based on a table
with a unique index

rc = MyRs.Updatable
MyRs.MoveFirst
txtname.Text = MyRs(“name”)
Txtid.Text = MyRs(“id”)
txtjob.Text = MyRs(“job”)
txtdept.Text = MyRs(“dept”)
```

Using ODBC CALL RPCs in Powerbuilder

The following sample Powerbuilder script is an example of using Powerbuilder’s support for stored procedures to execute and return the results of the sample ODBC CALL RPC, SDCOVSP. This code is supplied on the NEON System’s CD-ROM, and can be found in directory `samples/pb/test.pbl`.

```
string ls_EmpName, ls_EmpJob
long ll_EmpID, ll_EmpDept, ll_EmpYears, ll_NewRow, ll_Row,
ll_InputNum
//
//Assign values to the ShadowDirect transaction
//
ShadowDirect = Create Transaction
ShadowDirect.DBMS       = “ODBC”
ShadowDirect.AutoCommit = TRUE
ShadowDirect.database   = “”
ShadowDirect.userid     = “”
ShadowDirect.dbpass     = “”
ShadowDirect.logid      = “”
ShadowDirect.logpass    = “”
ShadowDirect.servername = “DB2A”
ShadowDirect.dbparm     = “ConnectString='DSN=Sample_Direct'”
// Connect to the ShadowDirect ODBC transaction object
CONNECT USING ShadowDirect;
//
//Reset the DataWindow
```

```
//
DW_1.Reset()
//
//Get the employee ID entered from the input box
//
ll_InputNum = Long( em_1.Text )
//
//When the input is zero, call the RPC without a parameter
//
IF ll_InputNum = 0 THEN
//
//Declare the RPC
//
Declare GetStaffInfoWOParam Procedure for SDCOVSP
using ShadowDirect;
//
//Call the RPC
//
Execute GetStaffInfoWOParam;
//
//Process the result set
//
CHOOSE CASE ShadowDirect.SQLCode
 CASE 0
DO WHILE ShadowDirect.SQLCode = 0
FETCH GetStaffInfoWOParam INTO :ll_EmpID
,:ls_EmpName
,:ll_EmpDept
,:ls_EmpJob
,:ll_EmpYears;
IF ShadowDirect.SQLCode = 0 THEN
ll_NewRow = Dw_1.InsertRow(0)
ll_Row = DW_1.ScrollToRow( ll_NewRow )
DW_1.SetItem( ll_NewRow,"empid" , ll_EmpID )
DW_1.SetItem( ll_NewRow,"empdept" , ll_EmpDept)
DW_1.SetItem( ll_NewRow,"empjob" , ls_EmpJob )
DW_1.SetItem( ll_NewRow,"empname" , ls_EmpName )
DW_1.SetItem( ll_NewRow,"empyears" , ll_EmpYears ) END IF
LOOP
 CASE ELSE
  END CHOOSE
//
//Close the procedure
//
Close GetStaffInfoWOParam;
ELSE
//
//Declare the RPC with a parameter
//
Declare GetStaffInfo Procedure for SDCOVSP :ll_InputNum
using ShadowDirect;
//
//Call the RPC
//
```

```
Execute GetStaffInfo;
//
//Process the result set
//
  CHOOSE CASE ShadowDirect.SQLCode
CASE 0
DO WHILE ShadowDirect.SQLCode = 0
FETCH GetStaffInfo INTO :ll_EmpID
, :ls_EmpName
, :ll_EmpDep
, :ls_EmpJob
, :ll_EmpYears;
IF ShadowDirect.SQLCode = 0 THEN
ll_NewRow = Dw_1.InsertRow(0)
ll_Row = DW_1.ScrollToRow( ll_NewRow )
DW_1.SetItem( ll_NewRow,"empid" , ll_EmpID )
DW_1.SetItem( ll_NewRow,"empdept" , ll_EmpDept )
DW_1.SetItem( ll_NewRow,"empjob" , ls_EmpJob )
DW_1.SetItem( ll_NewRow,"empname" , ls_EmpName )
DW_1.SetItem( ll_NewRow,"empyears" , ll_EmpYears ) END IF
LOOP
 CASE ELSE
  END CHOOSE
//
//Close the procedure
//
Commit using ShadowDirect;
Close GetStaffInfo;
END IF
DISCONNECT USING ShadowDirect ;
```

Using ODBC CALL RPCs in /*EXECSQL

The following sample /*EXECSQL script is an example of using /*EXECSQL support for stored procedures to execute and return the results of the sample ODBC CALL RPC, SDCOVSP, which reads a VSAM file:

```
 /*WWW /NEON/IMSEXEC1
****************************************************************

*  SAMPLE APPLICATION THAT ILLUSTRATES THE USE OF AN EXECSQL    *

*  PROCESS SECTION.  THE AUTOFORMAT KEYWORD CALLS FOR THE ROW   *

*  DATA TO BE FORMATTED INTO AN HTML TABLE.                     *

*                                                              *

****************************************************************


/*EXECSQL MAXROWS(100) -
```

```
        SUBSYS(NONE) PLAN(NONE) -
            AUTOFORMAT( TITLE('SAMPLE RPC CALL USING /*EXECSQL') -
                        BODY('BGCOLOR="#FFCC33"') -
                      )
CALL SDCOVSP(100)
```

Note:
This applies to Shadow OS/390 Web Server **only**.

CHAPTER 3:
Running DB2 Stored Procedures

This chapter covers programming information for running DB2 stored procedures. It includes information about the execution, preparation, and troubleshooting of the DB2 stored procedures.

This chapter applies to Shadow Direct and Shadow OS/390 Web Server.

Introduction

IBM stored procedures are fully supported by Shadow Direct and Shadow Web Sever version 3.1 and above at maintenance level SVFX3270 and above. In order to run DB2 stored procedures, you must be running DB2 version 4.1 or above. For specific information on writing and using DB2 stored procedures, please refer to the *IBM DB2 Application Programming and SQL Guide*.

The Syntax

The syntax for invoking an IBM stored procedure using the Neon Client ODBC driver is

```
Call SYSPROC.procedure-name(parameter,parameter,….)
```

Where:

`SYSPROC`
: is the prefix that identifies the stored procedures as IBM rather than a NEON stored procedures (RPCs).

`procedure-name`
: is the procedure name in the DB2 catalog.

`parameter, parameter, ...`
: are the parameters for the DB2 stored procedures.

This syntax matches the IBM SQL stored procedure naming conventions.

DB2 Stored Procedures

Result Sets

Shadow Direct supports returning zero or one result set from an IBM stored procedure. If the DB2 stored procedure returns more than one result set, the remaining result sets are ignored (without any error messages or warnings). This is the same restriction for NEON RPCs.

Retrieving Column Names

If you want to retrieve column names from any DB2 stored procedures that return result sets, the **DESCSTAT** installation parameter **must** be set to YES. (DESCSTAT is set in the DB2 DSNZPARM member at DB2 install time. DSNZPARM is created via the DB2 installation job, DSNTIJUZ.).

Note:
If the DESCSTAT DB2 installation parameter is **not** set to YES, then the result set column names will **always** be zero length strings. Some IBM documentation incorrectly implies that this is a BIND parameter. It is not a BIND parameter and must be set as part of the installation of DB2.

You must bind the DBRM used by Shadow Direct (OPRXSQ) or Shadow OS/390 Web Server (SWRXSQ) with the package used by the DB2 stored procedure in order for this procedure to be called successfully.

Samples

- **Result Set.** A sample job, DB2PROC1, has been provided in the NEON.SV040100.CNTL dataset which shows how to properly compile and bind a DB2 stored procedure that returns a result set. This sample processes the example stored procedure, DB2PROC1, in the NEON.SV040100.SAMP dataset.

- **VBOutput Parameter.** A sample job, DB2PROC2, has been provided in the NEON.SV040100.CNTL dataset which shows how to properly compile and bind a sample DB2 stored procedure that returns an output parameter. This sample processes the example stored procedure, DB2PROC2, in the NEON.SV030100.SAMP dataset.

- **VB4.0 Program.** A sample VB 4.0 program is provided in the Shadow/samples/vb4/DB2Proc2 directory with the install of the Neon Client 32-bit driver version 3.02 or above. This sample shows how to properly use the ODBC API to invoke DB2PROC2.

- **VB5.0 Program.** A sample VB 5.0 program is provided in the Shadow/Samples/vb5/ADOSamp directory with the install of the NEON Client 32-bit driver version 3.04 or above. This sample shows how to do the following using ADO:

- Call a DB2 stored procedure that returns a result set.
- Call a DB2 stored procedure that returns an output parameter.
- Call a Shadow RPC program.
- Create an updateable recordset using an SQL query.

Preparing a DB2 Stored Procedure

The following steps need to be completed to properly prepare a DB2 stored procedure:

1. Update the DB2 `SYSIBM.SYSPROCEDURES` table with information regarding the stored procedure. If any of this information changes, you **must** stop the DB2 stored procedure and restart it using the **DB2 –START** and **–STOP PROCEDURE** command. If you don't, the changes will not take effect.

2. Run the DB2 Pre-Compiler against the DB2 stored procedure to create the DBRM.

3. Compile the DB2 stored procedure.

4. Link-edit the DB2 stored procedure. The DB2 stored procedure must be placed into a library in the `STEPLIB` of the DB2 stored procedure address space.

5. Bind the DB2 stored procedure into a DB2 package.

6. Bind the DB2 plan with the DB2 packlist and the Neon DBRM.

Note:
If the 4th character of the Plan Name is an R, the NEON Client ODBC driver assumes that your application is using a plan where the plan was bound using an Isolation value of Repeatable Read. If you are not using Repeatable Read please ensure that your plan name does NOT have an R in the 4th character of the plan name as does the Shadow default plan SDBR1010. If it is any other character than an R we assume the plan was bound with an Isolation Level of Cursor Stability.

The plan used for the connection to DB2 **must** have available the packages used by all of the stored procedures. Since only packages are supported for stored procedures, the plan must include a PKLIST that has all of the required packages.

Coding Cursors in Return Result Sets

When coding DB2 stored procedures, the cursors used in stored procedures should be declared WITH HOLD in addition to WITH RETURN. Otherwise, any COMMIT (either issued by DB2 or by Shadow Server) will destroy the result set(s). If WITH HOLD is not specified and if COMMIT_ON_RETURN is set to

"Y" in SYSIBM.SYSPROCEDURES, each stored procedure will not appear to return any result sets, **even if it actually does**.

Here's an example of an EXEC SQL statement in a DB2 stored procedure:

```
EXEC SQL
 DECLARE C1 CURSOR WITH HOLD WITH RETURN
   FOR SELECT ID, NAME, DEPT, JOB,
              YEARS, SALARY, COMM
              FROM Q.STAFF
END-EXEC.
```

Troubleshooting DB2 Stored Procedures

Be aware of the following situations that can arise when executing DB2 stored procedures:

- It is common to get a zero (0) SQLCODE from a stored procedure that actually failed because it could not access the associated package (timestamp errors, etc.). The reason is that all of the DB2 operations attempted by the stored procedure failed and there is no indication why.
- All stored procedures that actually create result sets exit with a +466 SQLCODE unless the result set is destroyed by a commit (as described above). This is neither an error nor a warning. It just means the stored procedure successfully created at least one result set.
- A -204 SQLCODE indicates that the DB2 stored procedure name is unknown to DB2. Unlike NEON stored procedures, all IBM stored procedures must be defined in the DB2 catalog.
- A -440 SQLCODE indicates the parameters passed by the application do not match the parameter definitions in the DB2 catalog.
- A -480 SQLCODE indicates that the stored procedure created one or more results without using WITH HOLD cursors and a subsequent COMMIT issued by Shadow Server destroyed those result sets before the DESCRIBE PROCEDURE was completed.
- A zero (0) SQLCODE can result from a DB2 stored procedure that is designed to return one or more results sets for at least two reasons.
- All of the EXEC SQL statements in the stored procedure are failing because the required package is not available.
- The cursors used to return the result sets were not declared with WITH HOLD, and COMMIT_ON_RETURN is set to 'Y'.

CHAPTER 4:
Shadow IMS Direct

This chapter provides programming information for Shadow IMS Direct, a component of *Shadow Direct*. Information includes the product architecture and installation, the progrmming of IMS applications, client API function definitions, and Sample IMS batch message program code.

This chapter applies specifically to Shadow Direct.

Introduction

Shadow IMS Direct allows you to write client DL/I applications that:

- Access and update IMS databases.
- Access and update the IMS message queue.

These client applications can be written in almost any programming language, including C, C++, Visual Basic (VB), and PowerScript.

This chapter assumes that the reader is generally familiar with IMS and specifically familiar with DL/I programming.

Product Architecture

Shadow IMS Direct allows two types of DL/I programs to be written:

- A Batch Message Program (BMP), providing single-threaded access to IMS databases and the message queue.
- A DBCTL program, providing multi-threaded access to IMS databases.

The type of program that is chosen depends on the type of access needed.

Single-Threaded Access to IMS Databases and the Message Queue

For accessing and updating the IMS message queue, a Batch Message Program (BMP) should be run in Shadow Server's address space. However, because only one copy of the BMP can run inside Shadow Server at a time, this IMS interface is

limited to one client application at any given time. In other words, only single-threaded access is possible.

Note:
If multiple requests are received to run a BMP, all subsequent requests will be enqueued. As each BMP request is completed, a request will be removed from the queue and processed. The queuing process is automatic and invisible (save for possible delays) to client applications.

This type of access is shown in Figure 4–1.

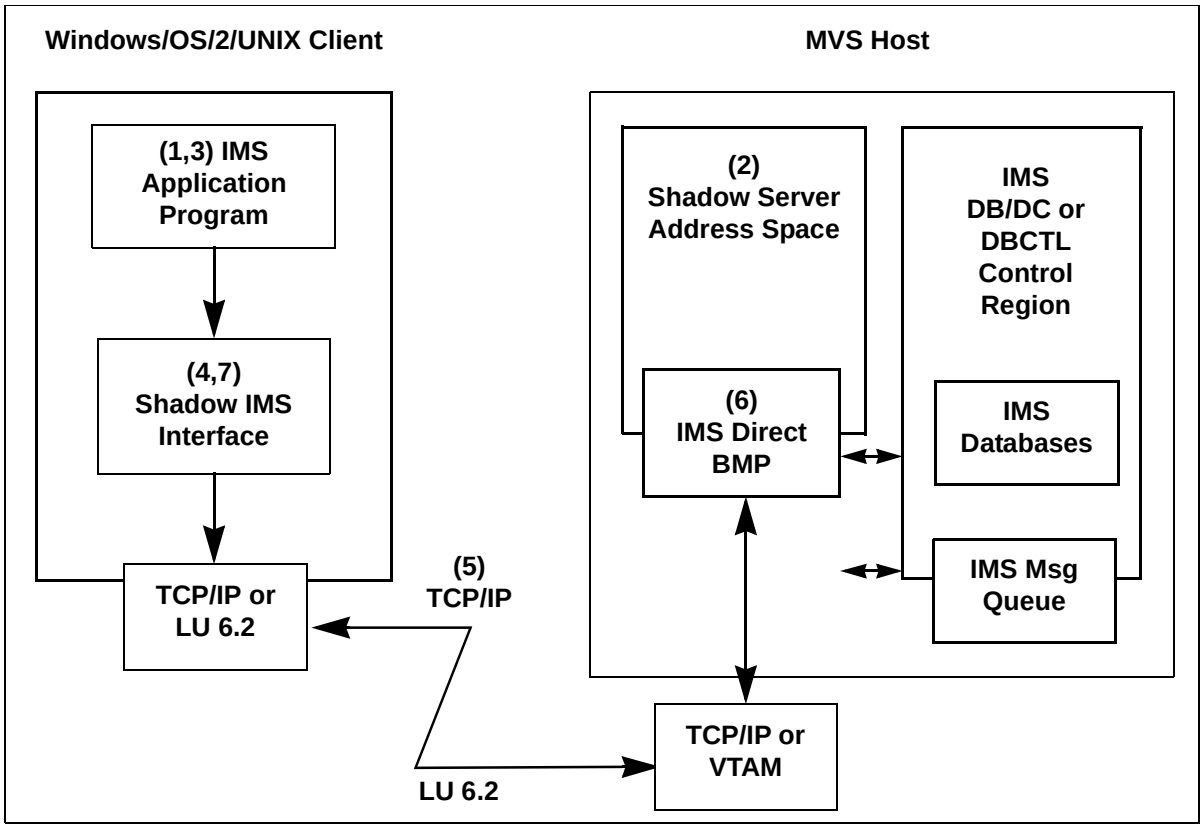


Figure 4–1. Shadow IMS Direct Product Architecture (IMS Message Queue and Database Access)

The steps in this process are:

1. A session begins when IMS Direct is invoked by a Windows/OS/2/UNIX client application and establishes a TCP/IP or LU 6.2 connection to Shadow Server.

2. The Shadow Server address space receives the request from the Windows/OS/2/UNIX client and attaches the IMS Batch Message Program (BMP). The IMS BMP then waits for work from the client system.

3. On the client system, the user's application program calls the IMS Direct DL/I interface function (SCCToDLI*) to access and update IMS databases or the IMS message queue.

4. The Shadow IMS interface routine analyzes and checks each DL/I call.

5. Shadow IMS Direct compresses each requests and sends it via TCP/IP or LU 6.2 to the BMP running inside the Shadow Server address space.

6. The BMP decompresses each request and invokes the actual IMS DL/I interface. The BMP compresses the data returned by IMS and sends it back to the client system.

7. The Shadow Direct SCCToDLI interface routine checks and analyzes each result and returns control to the application program. When the application program terminates, Shadow IMS Direct terminates the BMP running in the Shadow Server address space and closes the communication link.

Multi-Threaded Access to IMS Databases

If you do not need to access IMS message queues, you can write client applications that take advantage of Shadow IMS Direct's multi-threaded database access. These client applications architecturally resembles a CICS DL/I transaction program.† They are therefore subject to the same restrictions as CICS transactions that issue DL/I calls.

This type of access is illustrated in Figure 4–2.

* The SCCToDLI function is used with C and C++ language applications; the SCCToDLIPascal function is used with Visual Basic and PowerBuilder.

† Shadow IMS Direct does not use any CICS facilities.

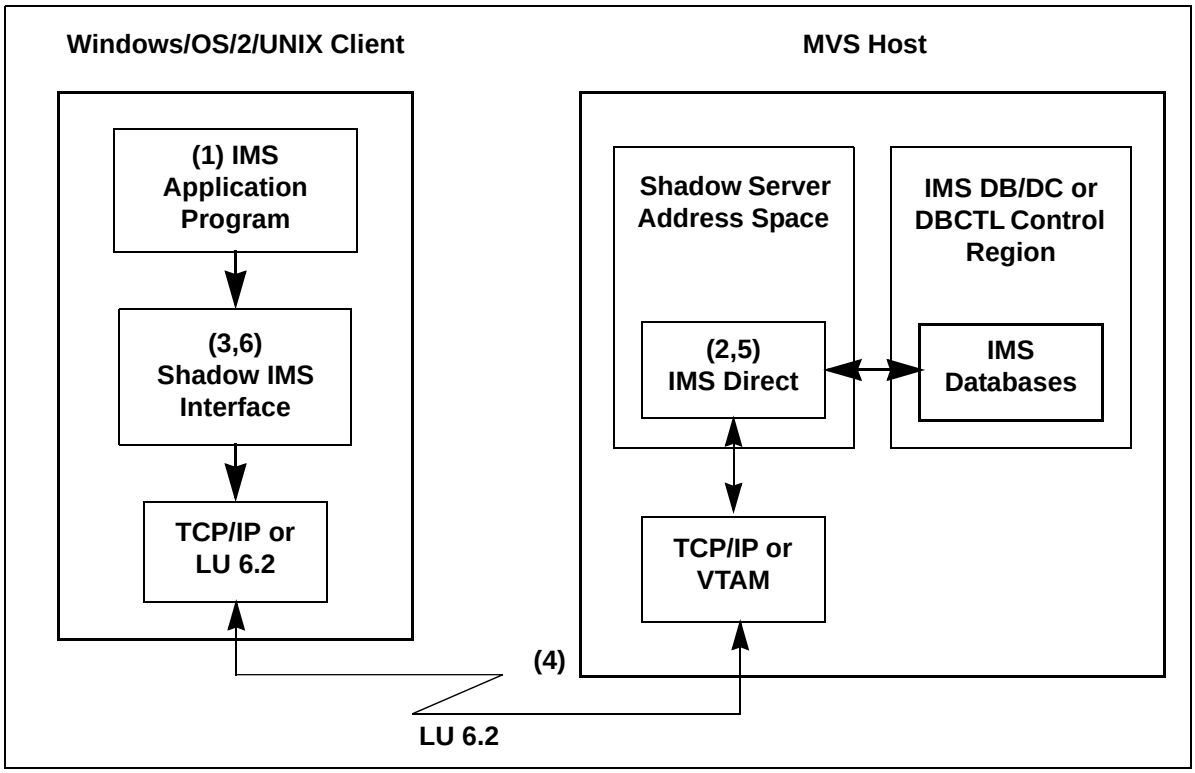


Figure 4–2. Shadow IMS Direct Product Architecture (Multi-Threaded Database Access)

The steps in the process are:

1. A session begins when Shadow IMS Direct is invoked by a Windows/OS/2/ UNIX client and establishes a TCP/IP or LU 6.2 connection to Shadow Server.

2. The Shadow Server address space recognizes the request from the Windows/ OS/2/UNIX client and establishes a link to IMS.

3. An IMS application program requests IMS services using the DL/I interface routine (SCCToDLI) to access and update IMS databases.

4. The Shadow DL/I interface routine checks and analyzes each request and sends it via TCP/IP or LU 6.2 to the Shadow Server address space.

5. Shadow Server decompresses requests and invokes the actual IMS DL/I interface. When IMS returns information, Shadow Server compresses it and sends it back to the client system.

6. The Shadow IMS Direct SCCToDLI routine decompresses the returned information and returns control to the application program.

Installing Shadow IMS Direct

Configuring Shadow Server

Before any programs created using Shadow IMS Direct can be used to communicate with the mainframe, several of Shadow Server's parameters must be properly set. This section will assume that Shadow Server is already installed and running on the mainframe. If not, refer to the Shadow IMS section in the *Shadow Server User's Guide*. This guide is useful for maneuvering through the various screens of the ISPF/SDB application.

By default, the Shadow Server address space does not allow either single-threaded or multi-threaded access to IMS. Each of these facilities must be enabled using a letter in the feature code string. IMS support is enabled using the feature letter code 'I'.

Note:
Only one single-threaded or multi-threaded IMS feature per Shaodw Server can be enabled at a time.

To specify which feature is to be used by Shadow Server, one of the following access parameters should be set to YES before the product is started:

- Single-threaded access.
- Multi-threaded access.

These parameters, which are explained in the following sections, can be set by entering them into the SDBxIN00 initialization REXX EXEC:DBCTL for multi-threaded or BMP for single-threaded. After start time, they can be modified using the SDB Parameters screen (SDB option 5.2) of the ISPF/SDB application.*

Setting Parameters for Single-Threaded Access

The following parameters are used for single-threaded access to IMS. Note that, in almost all cases, the default values are appropriate. The parameters are:

BMPPARM
Specifies the IMS BMP Parameter string. This parameter is not used at this time, but may be used at some point in the future.

BMPNAME
Enters the name of the IMS BMP Region Controller. The default is DFSRRC00.

* For information on the initialization EXEC, see Appendix A of the ***Shadow Server User's Guide***. For information about the SDB Parameters window, see Chapter 3 of the ***Shadow Server User's Guide***. Some parameters must be set before the Shadow Server address space is started; see the ***Shadow Server User's Guide*** for details.

IMSBMPTIMEOUT

Specifies the IMS BMP read time out value in units of seconds. This value is used to control how long the IMS BMP will wait for additional DL/I calls from the client application. If the time out limit is reached, the communication session with the client application will automatically be severed, and all IMS updates will automatically be rolled back. This value is used to prevent a failing client application or network problem from hanging the BMP in the main Shadow Server address space indefinitely. There is no default value for this field.

Setting Parameters for Multi-Threaded Access

The following parameters are used for multi-threaded access to IMS databases:

IMSID

Specifies the IMSID of the DBCTL region. This should be the four-character name of the DBCTL region, which is the same as the IMSID parameter in the DBCTL procedure. There is no default value for this parameter. A DBCTL region is not required to use multi-threaded access to IMS databases. All IMS DB/DC systems automatically provide DBCTL functionality as of IMS/ESA Release 3.0 or later. **Required.**

IMSUSERID

Specifies the userid of the product region. This should be the eight character name of the CCTL region. There is no default value. **Optional.**

IMSFUNCLEVEL

Specifies the function level of the product region. This parameter should indicate what DRA level the CCTL supports, so setting this parameter to 1 means that the CCTL uses the DRA at the IMS 3.1 level. The default value is 1. **Optional.**

IMSSUFFIX

Specifies the suffix of the DFSPZP module. The default value is “00”.

IMSMAXTHREADS

Sets the maximum number of DRA thread TCBs to be available at one time. The maximum number is 255. The default is “1”.

IMSMINTHREADS

Specifies the minimum number of DRA thread TCBs to be available at one time. The maximum number is 255. The default is “1”.

IMSWAITTIME

Sets the identity retry wait time. This parameter should indicate the amount of time (in seconds) that the DRA should wait between attempts to identify itself to CCTL during an INIT request. The default is 60 seconds.

IMSDDNAME

Specifies the DDname used to allocate RESLIB. This should be the one to eight character DDname that will be used to dynamically allocate the DBCTL RESLIB data set. The default DDname is CCTLDD. This library must contain the DRA modules.

IMSDSNAME

Specifies the DSName of the DRA RESLIB. This should be the one to forty-four character name of the DBCTL RESLIB data set. This library must contain the DRA modules and must be MVS APF authorized. The default data set name is 'IMS.RESLIB'.

IMSFPBUFFERS

Indicates the number of Fast Path DEDB buffers to be allocated and fixed per thread. The default is "0".

IMSFPOVERFLOW

Indicates the number of the Fast Path DEDB overflow buffers to be allocated per thread. The default is "0".

IMSCLASS

Specifies the snap dump sysout output class.

IMSGROUPNAME

Specifies the Application Group Name. This should be a one to eight character application group name to be used as part of the DBCTL security function.

IMSNBABUFFERS

Specifies the total number of Fast Path NBA buffers that the CCTLs can use.

IMSTIMEOUT

Enters the DRA term time out value. This should be the amount of time (in seconds) that a CCTL should wait for the successful completion of a DRA TERM request. This value should be specified only if the CCTL is coded to use it. This value is returned to the CCTL upon completion of an INIT request.

Note:

In almost all cases the default values are appropriate. The only parameter that must be set is the IMSID.

In order to set up the connection between Shadow Server and IMS using DBCTL, the IMS DBCTL interface will need to be active. DBCTL requires DBRC to be at the SHARECTL level; if it is not, DBCTL will not start. To initialize the RECON, specify (or let it default to) INIT.RECON SHARECTL. The example below shows some sample JCL you can copy to initialize the RECON. If you have CICS already connected to IMS and you are using DBCTL, this has probably already been done.

```
//INITREC JOB 1,PGMERID,CLASS=Q,MSGCLASS=A
//*
//RECON    EXEC PGM=DSPURX00,REGION=1000K
//STEPLIB  DD DSN=IMS.RESLIB,DISP=SHR
//DFSRESLB DD DSN=IMS.RESLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//RECON1   DD DSN=IMS.RECON1,DISP=SHR
//RECON2   DD DSN=IMS.RECON2,DISP=SHR
//SYSIN    DD *
INIT.RECON SSID(IMSA)
/*
```

For further information on how to setup DBCTL, refer to the *CICS-IMS Database Control Guide*.

Programming IMS Applications

Using Shadow IMS Direct always involves two separate programs:

- A client application.
- An MVS transaction program.

The client initiates the MVS transaction program. After the transaction program has been started, the client application sends requests to the host TP for execution. All requests are executed synchronously. By default, all data areas sent from the IMS application program to the host are converted, as needed. Most data areas are simply translated from ASCII to EBCDIC before being sent to the host, and from EBCDIC back to ASCII upon receipt from the host. PCBs are converted to host format on a field by field basis. All conversions can be turned off by passing negative length values to the Shadow IMS Direct interface function. The SCAsciiToEbcdic and SCEbcdicToAscii functions are provided to Shadow IMS Direct applications for any data conversions needed.

Client Applications

Shadow IMS Direct client applications are programs written in any one of several languages that use the Shadow IMS Direct API to execute IMS DL/I requests on the host. These applications are normally written in C or C++. However, these applications can be written in any language that can call DLL entry points, including Visual Basic (VB), PowerScript, Pascal, COBOL, etc. In practice, almost any client application programming language can be used to invoke the Shadow IMS Direct API.

The Shadow IMS Direct API is implemented as a DLL in the Windows, Windows NT, and OS/2 environments. The Shadow IMS Direct API is implemented as a shared library object in those UNIX environments (SunOS, etc.) that support shared libraries. In other UNIX environments, the API is implemented as an archive file.

Shadow IMS Direct applications must be linked using one of the two import libraries supplied with Shadow IMS Direct:

```
SCODBC.LIB, which is used with SCODBC.DLL
SCODBCTS.LIB, used with SCODBCTS.DLL
```

The SCODBCTS.DLL contains numerous diagnostic, debugging, and support tools. As a result, this DLL should be used for all application development purposes. However, since the SCODBCTS.DLL is substantially larger and slower than its production counterpart (SCODBC.DLL), production applications that have been fully debugged should be switched to SCODBC.DLL for improved performance.

Note:

ODBC.LIB must not be used with Shadow IMS Direct applications. The architecture of Shadow IMS Direct does not support passing calls from a Shadow IMS Direct application to the Shadow IMS Direct DLL, via the Microsoft driver manager (ODBC.DLL).

Shadow IMS Direct applications written in C or C++ must include the scpghd.h header file. This header file declares all the Shadow IMS Direct structures and API entry points, and must be included in all Shadow IMS Direct client application functions. The header file can be used with both ANSI and non-ANSI C compilers, however, ANSI C is the recommended choice for compiling and building Shadow RPC client application programs. This header file will also work in all client environments including Windows, OS/2, and UNIX.

Client API Function Definitions

The following functions are available with Shadow IMS Direct API calls:

- **SCCToDLI:** Execute DL/I calls from a C or C++ program.
- **SCCToDLIPascal:** Execute DL/I calls from a VB or PowerBuilder program.
- **SCPackedToAscii:** Convert packed decimal data to ASCII.
- **SCAsciiToPacked:** Convert an ASCII string to packed decimal data.

These API calls, which are detailed in the following sections, are used by Shadow IMS Direct applications to establish a connection to the host and to execute DL/I calls on the host.

SCCToDLI

IMS Direct

SCCToDLI executes DL/I requests on behalf of a client application. Most DL/I requests are passed to the host for processing, and the results are returned to the client application program. However, in a few cases, DL/I requests are executed locally. This function takes a variable number of arguments, and is intended to be called by C or C++ application programs. It cannot be called by Visual Basic or PowerScript applications.

Syntax

```
RETODBC SCCToDLI(hdbc, cpar, rgbIMSFunction, rgbValue1,
rgbValue2, rgbValue3, rgbValue4, rgbValue4, rgbValue5, rgbValue6,
rgbValue7, rgbValue8, rgbValue9, rgbValue10, rgbValue11,
rgbValue12, rgbValue13, rgbValue14, rgbValue15, rgbValue16,
rgbValue17, rgbValue18, rgbValue19, rgbValue20, rgbValue21,
rgbValue22, rgbValue23, rgbValue24, rgbValue25, rgbValue26,
rgbValue27, rgbValue28, rgbValue29, rgbValue30, rgbValue31,
rgbValue32, rgbValue33, rgbValue34)
```

Arguments

The SCCToDLI function accepts the following arguments:

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
SDWORD	cpar	Input	Number of parameters. This value includes the IMS function string, but does not include the connection handle, the number of parameters itself, or any length values provided for the other arguments.
PTR	rgbIMSFunction	Input	IMS function code. This field must point to a four-byte string containing the IMS function code. The function code does not need to be null-terminated, but must be in uppercase and padded with trailing blanks, if needed
PTR	rgbValue1-17	I/O	Use of this argument depends on the IMS function code string and the number of parameters. See the comments below for additional information.
PTR	rgbValue18-34	Input	Use of this argument depends on the IMS function code string and the number of parameters. See the comments below for additional information.

Returns

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- A positive IMS status code stored in the least significant two bytes of the return code.

Diagnostics

When SCCToDLI returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLError. The following table lists the SQLSTATE values commonly returned by SCCToDLI and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	Connection specified by *hdbc* argument was not open. Connection processes must be completed successfully (and the connection must be open) for driver to perform this function.
08S01	Communication link failure	Communication link between driver and data source to which driver was connected, failed before function completed processing.
S1000	General error	Error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. Error message returned by SQLError in the argument *szErrorMsg* describes error and its cause.
S1009	Invalid argument value	The parameter count *cpar* was less than one. The parameter count *cpar* exceeded the maximum value. The maximum value is 18. The *rgbIMSFunction* pointer was null. One of the *rgbValue* arguments was null. One of the argument length values was not set. For the “PCB “ IMS Function code, the parameter count *cpar* was not 3. For either the “GPCB” or “PPCB” IMS function code values, the *cpar* parameter count was not 4. The IMS function code *rgbIMSFunction* did not contain a valid IMS function code string. The PCB length value passed was invalid. An invalid PCB address was detected.

Comments

The SCCToDLI function provides all IMS services to an IMS Direct client application. This function can be used three ways

- To issue standard DL/I calls that get, replace, insert, and delete IMS segments.
- To obtain the PCB list and the number of PCBs on behalf of the caller.
- To access and update individual fields in a PCB. Different arguments are passed for each use of this function.

The correct arguments for each use of this function are described below.

- All of the arguments passed to this function must be four bytes long.

  Note:
  In many cases, large data areas are passed to this function, however, these data areas are passed using pointers that are actually four bytes long. The practical significance of this requirement is that all length and count values must be passed as four-byte integers. Length and count values can be cast to four-byte integers, if needed. The SDWORD cast can be used to ensure integers are passed to this function correctly.

- The SDWORD cast can be used to ensure integers are passed to this function correctly.

- Some forms of this function (see below) take a PCB as the fourth argument (*rgbValue1*). PCBs can be passed as either an address or a four-byte PCB number. Visual Basic and PowerScript applications must pass a PCB number value.

- The number of arguments passed to this function is variable. The actual number will range from three to thirty-seven.

  Note:
  The actual number will always be equal to the cpar argument value, multiplied by two, plus one. This relationship can be explained as follows: a length parameter must be supplied for all data arguments other than the IMS function code string pointed to by rgbIMSFunction; in addition, the cpar argument does not include itself and the connection handle.

- Length values are required for all arguments except for the first three. The length values always follow the data arguments. If, for example, *cpar* is two, the *cpar* argument will be followed by two data areas and one length. If *cpar* is three, the *cpar* argument will be followed by three data areas and two lengths. The number of lengths will always be one less than the number of data areas, because a length value is not provided for the *rgbIMSFunction* argument.

- The SCCToDLI function can only be called by languages that support passing a variable number of arguments. This means that this function cannot be called by Visual Basic and PowerBuilder applications. Visual Basic and PowerBuilder applications can, however, call the SCCToDLIPascal function described below. A C or C++ program can use this function without restriction.

- The return code from this function will either be an ODBC return code or an IMS status code. The ODBC return codes are described above. IMS status codes are two byte character strings stored in the return code variable. The sample IMS applications show how these status codes can be tested.

Note:
If an IMS operation succeeds, the IMS status code will be two blank characters. However, the two blank characters will NOT be stored in the return code variable. Instead, a return code of SQL_SUCCESS (which is actually zero) will be returned to the caller. This is the same convention that the mainframe CTDLI function uses. For more information about IMS status codes, see IMS Messages and Codes (SC26-4290).

Obtaining the PCB List Structure

This function can be used to obtain the PCB list from the Shadow IMS Direct interface. The PCB list is a structure that contains the number of PCBs, a vector of pointers to PCBs, and some additional information. The PCB list structure *imst* is typedef'ed in the `scpghd.h` header file. The PCBs pointed to by the PCB list in this structure are copies of the actual PCBs on the host. Shadow IMS Direct automatically synchronizes the client application PCBs and the actual PCBs on the host. The first PCB pointed to by the PCB list vector is the I/O PCB. The second, and all subsequent PCBs in the PCB list vector are database PCBs. This structure, and the PCBs pointed to by this structure, cannot be used in Visual Basic and PowerBuilder applications. Visual Basic and PowerBuilder do not have adequate capabilities for pointer manipulation.

SCCToDLI is called as follows, in order to obtain the PCB list structure:

```
rc = SCCToDLI(hdbc, 3, "PCB ", "DUMMYPSB", &lcimst, 8,
sizeof(imst))
```

The parameter count for retrieving the PCB list structure is always 3. The "PCB " function code is used on the host to schedule a PSB and returns a vector of PCB pointers. However, the **SCCToDLI** function returns the PCB vector list but does not actually schedule the PSB. The PSB is scheduled as part of the host session initiation process. The PSB name is specified using the PSB keyword in the connection string or in a section of the ODBC.INI file.

The "DUMMYPSB" argument is the PSB name; as mentioned above, this function cannot be used to schedule a PSB. However, PSB scheduling capability

may be added at some point in the future. The “DUMMYPSB” string should be passed to maintain upward compatibility with future versions of this function.

> ***Note:***
> The PSB string must be null-terminated.

The *lcimst* argument is the local data area structure into which the PCB list structure will be copied. This area should be allocated using the *imst* typedef.

The length values are the sizes of the “DUMMYPSB” string and the size of the local *imst* area, respectively. The *sizeof* function is used to determine the size of the *imst* as a convenience and to ensure future upward compatibility.

Accessing and Updating PCB Fields

The function can also be used to access and update fields in the PCB. This capability is provided for languages such as Visual Basic, or PowerScript, that cannot access PCB fields using pointers. Of course, this function can also be called from C or C++ applications. The IMS function code is “GPCB” for accessing PCB fields and “PPCB” for updating PCB fields. The **SCCToDLI** function is called as follows to access or update a PCB field:

```
rc = SCCToDLI(hdbc, 4, rgbIMSFunction, rgbValue1, data type, data
area, sizeof(PCB_STRUCT_8_TYPE), sizeof(data type), sizeof(data
area))
```

The parameter count for accessing and updating PCB fields is always four. The IMS function code string *rgbIMSFunction* must either be “GPCB” or “PPCB” as mentioned above. The PCB (argument 3) itself can be passed one of two ways

- As a pointer to a PCB.
- As the PCB number (cast to a SDWORD).

PCBs in the PCB vector list are numbered starting with “1” for the I/O PCB. In other words, the first database PCB is always PCB number 2.

The data type is either a pointer to a null-terminated character string or an integer cast to a SDWORD with the data type number. Make **absolutely sure** that, if the data type is passed as an **integer**, a **four-byte integer** is used. **Do not pass a two-byte integer in any case.** You will get a program fault.

> ***Note:***
> Do not pass the global const values defined in the Visual Basic sample programs without first assigning them to long integer values.

The possible data type values are:

Data Type Integer	Data Type String	Description
SC_DB_PCB	"SC_DB_PCB"	Entire DB PCB area
SC_DATABASE_NAME	"SC_DATABASE_NAME"	DB PCB database name
SC_SEGMENT_LEVEL_NUMBER	"SC_SEGMENT_LEVEL_NUMBER"	DB PCB segment level number
SC_STATUS_CODE	"SC_STATUS_CODE"	DB and DC PCB status code
SC_PROCESSING OPTIONS	"SC_PROCESSING OPTIONS"	DB PCB processing options
SC_SEGMENT_NAME	"SC_SEGMENT_NAME"	DB PCB segment name
SC_KEY_AREA_LENGTH	"SC_KEY_AREA_LENGTH"	DB PCB key feedback area length
SC_SENSITIVE_SEGMENTS	"SC_SENSITIVE_SEGMENTS"	DB PCB number of sensitive segments
SC_KEY_AREA	"SC_KEY_AREA"	DB PCB key feedback area
SC_DC_PCB	"SC_DC_PCB"	Entire DC PCB area
SC_TERMINAL_NAME	"SC_TERMINAL_NAME"	DC PCB terminal name
SC_CURRENT_DATE	"SC_CURRENT_DATE"	DC PCB current date
SC_CURRENT_TIME	"SC_CURRENT_TIME"	DC PCB current time
SC_SEQUENCE_NUMBER	"SC_SEQUENCE_NUMBER"	DC PCB input message sequence number
SC_DESCRIPTOR_NAME	"SC_DESCRIPTOR_NAME"	DC PCB output descriptor name
SC_USERID	"SC_USERID"	DC PCB user identification

The data area must be large enough to contain any data returned by the "GPCB" function code. If the "PPCB" function code is used, this area must contain all the data that will be copied into the PCB. The first length value must be the size of the PCB passed using *rgbValue1*. The actual size will depend on whether the client application program is passing an I/O or a DB PCB. The size of I/O PCBs is always fixed; the size of DB PCBs depends on the size of the key feedback area.

The size of the data type will depend on whether a character string or a numeric data type is passed. In either case, the correct length should be passed to this function. If a character string data type is passed, the size is the length of the string. If a numeric data type is passed, the size will always be four. The size of the data area will depend on field in the PCB that is being accessed or updated. The correct size of the data area should always be passed to this function.

Sending DL/I Requests to the Host

This most common use of this function is to pass DL/I calls to the host. The IMS function codes that can be used for this purpose are documented in the IMS

Application Programming: DL/I Calls manual (SC26-4274). See the CICS with DBCTL section of the table documenting which IMS calls are available in which environments. Of course, all the standard get, delete, insert, and replace calls can be used with the SCCToDLI function. SCCToDLI is called as follows for passing DL/I calls to the host.

```
 rc = SCCToDLI(hdbc, cpar, rgbIMSFunction, rgbValue1, rgbValue2,
...)
```

The *rgbIMSFunction* will contain the IMS function code that will be sent to the host and executed. This argument must point to a four-byte uppercase string containing IMS function code padded with blanks, if needed. This string is not null-delimited. *rgbValue1* can either point to a PCB or contain a four-byte PCB number. *rgbvalue2* through *rgbValue17* are used to pass a segment and segment search arguments to this function. The segment and the segment search arguments may or may not be passed, depending on the IMS function code. A length value must be supplied for each of the rgbValues that are actually passed. The number of arguments will be one plus the number of optional data areas actually passed. The length values must immediately follow the last data area passed to this function.

Code Example

None at this time.

Related Functions

For information about	See
Executing DL/I calls from a VB or PowerBuilder program	SCCToDLIPascal
Converting data from ASCII to EBCDIC	SCAsciiToEbcdic
Converting data from EBCDIC to ASCII	SCEbcdicToAscii

SCCToDLIPascal

IMS Direct

SCCToDLIPascal executes DL/I requests on behalf of a client application. Most DL/I requests are passed to the host for processing and results are returned to the client application program. However, in a few cases, DL/I requests are executed locally. This function takes a fixed number of arguments and is intended to be called by Visual Basic or PowerScript applications, however, it can also be called by C or C++ application programs.

Syntax

```
RETODBC SCCToDLIPascal(hdbc, cpar, rgbIMSFunction, rgbValue1,
rgbValue2, rgbValue3, rgbValue4, rgbValue5, rgbValue6, rgbValue7,
rgbValue8, rgbValue9, rgbValue10, rgbValue11, rgbValue12,
rgbValue13, rgbValue14, rgbValue15, rgbValue16, rgbValue17,
cbValue1, cbValue2, cbValue3, cbValue4, cbValue5, cbValue6,
cbValiue7, cbValue8, cbValue9, cbValue10, cbValue11, cbValue12,
cbValue13, cbValue14, cbValue15, cbValue16, cbValue17)
```

Arguments

The SCCToDLIPascal function accepts the following arguments:

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection handle.
SDWORD	cpar	Input	Number of parameters. This value includes the IMS function string, but does not include connection handle, number of parameters itself, or any length values provided for the other arguments.
PTR	rgbIMSFunction	Input	IMS function code. This field must point to a four-byte string containing the IMS function code. The function code does not need to be null-terminated, but must be in uppercase and padded with trailing blanks if necessary.
PTR	rgbValue1-17	I/O	Use of this argument depends on IMS function code string and number of parameters. See comments below for additional information.
SDWORD	cbValue1-17	Input	These arguments are the lengths of the *rgbValue1-17* arguments. Number of lengths actually used must be equal to number of *rgbValue1-17* values that are actually used.

Returns

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR, SQL_INVALID_HANDLE
- A positive IMS status code stored in the least significant two bytes of the return code.

Diagnostics

When SCCToDLIPascal returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLError. The following table lists the SQLSTATE values commonly returned by SCCToDLIPascal and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	Connection specified by *hdbc* argument was not open. Connection processes must be completed successfully (and connection must be open) for driver to perform this function.
08S01	Communication link failure	Communication link between driver and data source to which driver was connected, failed before function completed processing.
S1000	General error	Error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. Error message returned by SQLError in argument *szErrorMsg* describes error and its cause.
S1009	Invalid argument value	Parameter count *cpar* was less than one. Parameter count *cpar* exceeded maximum value. Maximum value is 18. IMS Function was null. One of the *rgbValue* arguments was null. One of the argument length values was not set. For “PCB “ IMS Function code, parameter count *cpar* was not 3. For either “GPCB” or “PPCB” IMS function code values, *cpar* argument was not 4. The IMS function code *rgbIMSFunction* did not contain a valid IMS function code string. PCB length value passed was invalid. An invalid PCB address was detected.

Comments

The SCCToDLIPascal function provides all IMS services to a direct client application. This function can be used three ways:

- To issue standard DL/I calls that get, replace, insert, and delete IMS segments.
- To obtain the PCB list and the number of PCBs on behalf of the caller.
- To access and update individual fields in a PCB.

The arguments passed to this function for each of these uses are different. The correct arguments for each use of this function are described below.

- All of the arguments passed to this function must be four bytes long.

  Note:
  In many cases, large data areas are passed to this function. However, these data areas are passed using pointers that are actually four bytes long. The practical significance of this requirement is that all length and count values must be passed as four-byte integers. Length and count values can be cast to four-byte integers, if needed. The SDWORD (no embedded blanks) cast can be used to ensure integers are passed to this function correctly.

- The number of arguments passed to this function is fixed. The actual number will always be thirty-seven. The *cpar* argument must be set to the number of rgbValues that are actually used plus one for the IMS function code string, pointed to by *rgbIMSFunction*. A null pointer must be passed for all of the unused rgbValues. Actual lengths must be passed using the cbValues for all the rgbValues that are actually used. Zero must be passed for all the unused length values. The number of used length values will always be equal to *cpar* minus one.

- Length values are required for all arguments except the first three. The length values always follow the data arguments. If, for example, *cpar* is two, then the *cpar* argument will be followed by two used data areas, 16 unused data areas, one actual length, and 16 unused lengths. If *cpar* is three, then the *cpar* argument will be followed by three data areas, 15 unused data areas, two actual lengths, and 15 unused lengths. The number of lengths will always be "1" less than the number of data areas because a length value is not provided for the *rgbIMSFunction* argument.

- Some forms of this function (see below) take a PCB as the fourth argument (rgbValue1). PCBs can be passed as either an address or a four-byte PCB number. Visual Basic and PowerScript applications must pass a PCB number value.

- The SCCToDLI function can be called by all languages. This function is intended for use by Visual Basic and PowerBuilder applications. C and C++ programs can call the SCCToDLI function without dummy arguments. The SCCToDLI is described above.

- The return code from this function will either be an ODBC return code or an IMS status code. The ODBC return codes are described above. IMS status codes are two-byte character strings stored in the return code variable. The sample IMS applications show how these status codes can be tested.

> ***Note:***
> If an IMS operation succeeds, the IMS status code will be two blank characters. However, the two blank characters will NOT be stored in the return code variable. Instead, a return code of SQL_SUCCESS (which is actually zero) will be returned to the caller. This is the same convention that the mainframe CTDLI function uses. For more information about IMS status codes, see IMS Messages and Codes (SC26-4290).

Obtaining the PCB List Structure

This function can be used to obtain the PCB list from the Shadow IMS Direct interface. The PCB list is a structure that contains the number of PCBs, a vector of pointers to PCBs, and some additional information. The PCB list structure *imst* is typedef'ed in the `scpghd.h` header file. The PCBs pointed to by the PCB list in this structure are copies of the actual PCBs on the host. Shadow IMS Direct automatically synchronizes the client application PCBs and the actual PCBs on the host. The first PCB pointed to by the PCB list vector is the I/O PCB. The second, and all subsequent PCBs in the PCB list vector, are database vectors. This structure and the PCBs pointed to by this structure cannot be used in Visual Basic and PowerBuilder applications. Visual Basic and PowerBuilder do not have adequate capabilities for pointer manipulation. SCCToDLIPascal is called as follows, in order to obtain the PCB list structure:

```
rc = SCCToDLIPascal(hdbc, 3, "PCB ", "DUMMYPSB", &lcimst, (PTR)
NULL, (PTR) NULL,
(PTR) NULL , (PTR) NULL,
(PTR) NULL , (PTR) NULL,
(PTR) NULL , (PTR) NULL ,
(PTR) NULL , (PTR) NULL ,
(PTR) NULL , (PTR) NULL ,
(PTR) NULL, 8, sizeof(imst),
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0)
```

The parameter count for retrieving the PCB list structure is always "3". "PCB " function code is used on the host to schedule a PSB and returns a vector of PCB pointers. However, the SCCToDLIPascal function returns the PCB vector list but does not actually schedule the PSB. The PSB is scheduled as part of the host session initiation process. The PSB name is specified using the PSB keyword in the connection string or in a section of the `ODBC.INI` file.

The “DUMMYPSB” argument is the PSB name; as mentioned above, this function cannot be used to schedule a PSB. However, PSB scheduling capability may be added at some point in the future. The “DUMMYPSB” string should be passed to maintain upward compatibility with future versions of this function.

Note:
The PSB string must be null-terminated.

The *lcimst* argument is the local data area structure into which the PCB list structure will be copied. This area should be allocated using the *imst* typedef.

The length values are the sizes of the “DUMMYPSB” string and the local *imst* area, respectively. The *sizeof* function is used to determine the size of the *imst* as a convenience and to ensure future upward compatibility.

Accessing and Updating PCB Fields

This function can also be used to access and update fields in the PCB. This capability is provided for languages such as Visual Basic or PowerScript, which cannot access PCB fields using pointers. Of course, this function can also be called from C or C++ applications. IMS function code is “GPCB” for accessing PCB fields and “PPCB” for updating PCB fields. The **SCCToDLIPascal** function is called as follows to access or update a PCB field:

```
rc = SCCToDLIPascal(hdbc, 4, rgbIMSFunction, rgbValue1, data type,
data area, (PTR) NULL,(PTR) NULL, (PTR) NULL, (PTR) NULL,
(PTR) NULL, (PTR) NULL, (PTR) NULL,
(PTR) NULL, (PTR) NULL, (PTR) NULL,
(PTR) NULL, (PTR) NULL, (PTR) NULL,
(PTR) NULL,
sizeof(PCB_STRUCT_8_TYPE),
sizeof(data type), sizeof(data area),
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0,
(SDWORD) 0, (SDWORD) 0)
```

The parameter count for accessing and updating PCB fields is always four. The IMS function code string *rgbIMSFunction* must either be “GPCB” or “PPCB” as mentioned above. The PCB itself can be passed one of two ways:

- As a pointer to a PCB.
- As the PCB number (cast to SDWORD).

PCBs in the PCB vector list are numbered starting from “1” for the I/O PCB. In other words, the first database PCB is always PCB number 2.

The data type is either a pointer to a null-terminated character string data type description or an integer cast to a SDWORD with the data type number. Make

absolutely sure that if the data type is passed as an **integer**, a **four-byte integer** is used. **Do not pass a two-byte integer in any case.** You will get a program fault. Do not pass the global const values defined in the Visual Basic sample programs without first assigning them to long integer values. See the **SCCToDLI** function located in this chapter for the data type list.

The data area must be large enough to contain any data returned by the “GPCB” function code. If the “PPCB” function code is used, this area must contain all the data that will be copied into the PCB. The first length value must be the size of the PCB passed using *rgbValue1*. The actual size will depend on whether the client application program is passing an I/O or a DB PCB. The size of I/O PCBs is always fixed; the size of DB PCBs depends on the size of the key feedback area.

The size of the data type will depend on whether a character string or a numeric data type is passed. In either case, the correct length should be passed to this function. The size of the data area will depend on what field in the PCB is being accessed or updated. The correct size of the data area should always be passed to this function.

Sending DL/I Requests to the Host

The most common use of this function is to pass DL/I calls to the host. The IMS function codes that can be used for this purpose are documented in the IMS Application Programming: DL/I Calls manual (SC26-4274). See the CICS with DBCTL section of the table. This section documents which IMS calls are available, and in which environments. Of course, all the standard delete, get, insert, and replace calls can be used with the SCCToDLIPascal function. SCCToDLIPascal is called as follows for passing DL/I calls to the host.

```
rc = SCCToDLIPascal(hdbc, cpar, rgbIMSFunction, rgbValue1,
rgbValue2, ..., cbValue1, cbValue2,...)
```

The *rgbIMSFunction* will contain the IMS function code that will be sent to the host and executed. This argument must point to a four-byte uppercase string containing IMS function code padded with blanks if needed. This string is not null-delimited. *rgbValue1* can either point to a PCB or contain a four-byte PCB number. *rgbvalue2* through *rgbValue17* are used to pass segments and segment search arguments to this function. The data area and the segment search arguments may or may not be passed, depending on the IMS function code. A length value must be supplied for each of the rgbValues that are actually passed. The number of arguments will be one plus the number of optional data areas actually passed. The length values must immediately follow the last data area passed to this function.

Code Example

None at this time.

Related Functions

For information about	See
Executing DL/I calls from a C or C++ program	SCCToDLI
Converting data from ASCII to EBCDIC	SCAsciiToEbcdic
Converting data from EBCDIC to ASCII	SCEbcdicToAscii

SCPackedToAscii

IMS Direct

SCPackedToAscii converts packed decimal data to ASCII. To properly display IMS Data that contains packed decimal fields, the data must be converted manually to ASCII. Since the SCCToDLIPascal API translates data from EBCDIC to ASCII automatically, conversion should be turned off to properly process the packed decimal data. To do this, specify a negative value for the length of the data being returned. If the value is negative, no EBCDIC to ASCII translation will occur using SCCToDLIPascal. Next, call the SCEbcdicToAscii API to convert the EBCDIC data, and call the SCPackedToAscii API to convert the packed decimal data.

Syntax

```
RETODBC SCPackedToAscii(hdbc, packed, precision, scale, rc, ascii)
```

Arguments

The SCPackedToASCII function accepts the following arguments::

Type	Argument	Use	Description
HDBC	hdbc	Input	ODBC connection handle.
PTR	cbpacked	Input	Pointer to the packed decimal data.
LONG	cbprecision	Input	Packed decimal data's `precision` value.
LONG	cbscale	Input	Packed decimal data's `scale` value.
INT	pcbrc	Output	Returned code.
CHAR	pcbascii	Output	Converted ASCII data buffer.

Returns

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_ERROR
- SQL_INVALID_HANDLE
- A positive IMS status code stored in the least significant two bytes of the return code

Diagnostics

When SCPackedToAscii returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLError. The following table lists the SQLSTATE values commonly returned by SCPackedToAscii and explains each one in the context of this

function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	Connection specified by the *hdbc* argument was not open. Connection processes must be completed successfully (and connection must be open) for driver to perform this function.
08S01	Communication link failure	Communication link between driver and data source to which driver was connected, failed before function completed processing.
S1000	General error	Error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. Error message returned by SQLError in argument *szErrorMsg* describes error and its cause.
S1009	Invalid argument value	Parameter count *cpar* was less than one. Parameter count *cpar* exceeded maximum value. Maximum value is 18. IMS Function was null. One of the *rgbValue* arguments was null. One of the argument length values was not set. For “PCB “ IMS Function code, parameter count *cpar* was not 3. For either “GPCB” or “PPCB” IMS function code values, *cpar* argument was not 4. IMS function code *rgbIMSFunction* did not contain a valid IMS function code string. PCB length value passed was invalid. An invalid PCB address was detected.

Example VB Code

```
 Dim rc As Integer
 Dim len1 As Long
 Dim sgar1 As String * 1015
 Dim sgar2 As String * 4
 Dim packed_data As String * 12
 Dim buffer_size, packed_precision, packed_scale As Long
len1 = 64
packed_precision = 8
packed_scale = 0
buffer_size = 12
rc = SCCToDLIPascal(hdbc, 4, "GN  ", 3, sgar, "STOKSTAT ", "", "",
"", "", "", "", "", "", "", "", "", "", "", "", 60, -1015, 10, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
'assign the packed decimal data at offset 65 to sgar2
sgar2 = Mid(sgar, 65, 4)
'convert data stored in sgar from Ebcdic to Ascii and store in
sgar1
rc = SCEbcdicToAscii(hdbc, sgar, sgar1, len1)
'convert packed data in sgar2 to Ascii and store in packed_data
rc = SCPackedToAscii(hdbc, packed_data, buffer_size, sgar2,
packed_precision, packed_scale)
```

SCAsciiToPacked

IMS Direct

SCAsciiToPacked converts an ASCII string to packed decimal data. This function can be called by any other routine. It returns standard ODBC retcode.

Notes:

The ASCII string passed by the caller is assumed to be properly null-terminated.

The size for the packed decimal buffer should be large enough to store all the numeric nibbles, specified by the precision argument, and the sign nibble at the end.

The converted packed decimal might lose significant digits if the precision of the ASCII string is higher.

Syntax

```
RETODBC SQL_API SCAsciiToPacked(hdbc, rgbAscii, rgbPacked,
cbValue, cbPrecision, cbScale)
```

Arguments

The SCAsciiToPacked function accepts the following arguments:

Type	Argument	Use	Description
HDBC	hdbc	Input	Connection block pointer
PTR	rgbAscii	Input	ASCII string
PTR	rgbPacked	Output	Packed decimal data
SDWORD	cbValue	Input	Packed decimal buffer length
SDWORD	cbPrecision	Input	Packed decimal's precision
SDWORD	cbScale	Input	Packed decimal's scale

C Example

Given an ASCII string of "100.02", and assuming the precision and the scale for the defined packed decimal field is also 5,2:

```
char * ascii_string = "100.02";
char   packed_buffer[3];    /* packed decimal data will takes up 3
bytes */

 rc = SCAsciiToPacked(hdbc, ascii_string, packed_buffer,
sizeof(packed_buffer), 5, 2);
```

Sample IMS Batch Message Program Code

The following sample program uses an IMS BMP to access segments from an IMS database. This program can be used interoperably in the Windows/OS/2/ UNIX environments.

Note:
In some cases, cross-platform portability is obtained using preprocessor statements.

In the Windows environment, this program is designed to run as a QuickWin application. In the OS/2 and UNIX environments, this program will write to standard output and standard error. Explanatory notes corresponding to the numbers in boxes are located immediately following the program.

```
/*****************************************************************/
/*---------------------------------------------------------------*/
/*  Include all of the standard header files                     */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
#include <stdio.h>              /* standard I/O library          */
#include <stdlib.h>             /* standard functions            */
#include <assert.h>             /* assert macro support          */
#include <string.h>             /* string functions              */
#include <time.h>               /* time functions and definitions*/
#include <sys/types.h>          /* more time functions and defs  */
#include <sys/timeb.h>          /* more time functions and defs  */
/*---------------------------------------------------------------*/
/*  A few defines so that the ODBC header files will work with UNIX */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
#ifndef MSDOS                   /* not MS-DOS environment?       */
#define  FAR                    /* FAR is not needed for UNIX    */
#define  far                    /* far is not needed for UNIX    */
#define  __stdcall              /* __stdcall is not needed for UNIX */
#define  __cdecl                /* __cdecl is not needed for UNIX */
#define  EXPORT                 /* EXPORT is not needed for UNIX */
#define  CALLBACK               /* CALLBACK is not needed for UNIX */
#define  HWND      int          /* HWND must be treated as integer */
#define  _timeb timeb           /* fix struct _timeb declaration */
#define  _ftime ftime           /* fix _ftime function name      */
#endif                          /* end MS-DOS environment check  */
/*---------------------------------------------------------------*/
/*  Include the ODBC header files                                */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
#ifdef MSDOS                    /* MS-DOS environment?           */
#define  __STDC__ 1             /* define the ANSI C constant    */
#include <windows.h>            /* standard Windows header file  */
#include <sql.h>                /* Core ODBC header file         */
#include <sqlext.h>             /* Extended ODBC header file     */
#define  RETODBC RETCODE        /* fix ODBC return code typedef  */
#else                           /* else, not MS-DOS environment  */
#include <x_sql.h>              /* Core ODBC header file         */
#include <x_sqlext.h>           /* Extended ODBC header file     */
```

1
2
3
4

```
#endif                          /* end MS-DOS environment check     */
#include <scpghd.h>             /* standard product header files    */
/*------------------------------------------------------------------*/
/*  Start the sample program                                        */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
#ifdef __STDC__                 /* check for ANSI C                 */
int __cdecl main(int argc , char *argv[])  /* ANSI function prototype */
#else                           /* else, K&R C                      */
int __cdecl main(argc, argv)    /* non-ANSI function prototype      */
int    argc;                    /* number of arguments              */
char  *argv[];                  /* argument pointer vector          */
#endif                          /* end of ANSI versus K&R c         */
  {                             /* start routine processing         */
/*------------------------------------------------------------------*/
/*  Declare a few variables needed below                            */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    HENV          henv;         /* environment block                */
    HDBC          hdbc;         /* connection block                 */
    imst          imar;         /* imst data area                   */
    int           alev;         /* environment allocated flag       */
    int           alcn;         /* connection allocated flag        */
    int           cnex;         /* connection executed flag         */
    int           firc;         /* final return code value          */
    int           acrc;         /* 'AC' return code value           */
    int           gbrc;         /* 'GB' return code value           */
    int           rc;           /* general purpose return code      */
    int           i;            /* count of DL/I calls              */
    typedef unsigned char u_char;  /* define an unsigned char type    */
/*------------------------------------------------------------------*/
/*  Define a few local values                                       */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
#define  STATUS_OK  0x0000      /* IMS status code of zeros         */
#define  STATUS_AC  0x4143      /* ASCII IMS status code AC         */
#define  STATUS_GB  0x4742      /* ASCII IMS status code GB         */
/*------------------------------------------------------------------*/
/*  Data areas for SCCToDLI                                         */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    static char dumy[8]      = "DUMMYPSB";  /* dummy PSB name string  */
    static char ssaunsol[10] = "SM40UMSR "; /* set segment search arg */
    typedef struct{PCB_STRUCT(24)} PCB_24_TYPE;  /* PCB type        */
/*------------------------------------------------------------------*/
/*  Segment structure                                               */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    struct                      /* start the structure              */
    {                           /* begin segment definition         */
      char type[2];             /* type                             */
      char terminal[5];         /* terminal                         */
      char message[25];         /* message                          */
    } unsol_root;               /* end segment definition           */
/*------------------------------------------------------------------*/
/*  I/O output structure                                            */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
#if (1 == 2)                    /* bypass the following struct      */
    struct                      /* start the structure              */
```

5

6

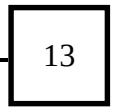
7

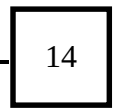
8

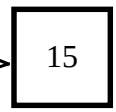
9

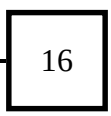
```
    {                           /* begin I/O area definition       */
      short ll;                 /* length field                    */
      short zz;                 /* other data area                 */
      char  msg[80];            /* message                         */
    } io_area;                  /* end I/O area definition         */
#endif                          /* struct not needed for now       */
/*-----------------------------------------------------------------*/
/*  Other local defines                                            */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    IO_PCB_TYPE far *io_pcb;    /* define I/O PCB                  */
    PCB_24_TYPE far *tbl_pcb;   /* define DB PCB                   */
/*-----------------------------------------------------------------*/
/*  A few length fields                                            */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    long szpc24 = sizeof(PCB_24_TYPE);  /* size of PCB             */
    long szioar = sizeof(unsol_root);  /* size of a segment        */
    long szssar = sizeof(ssaunsol);  /* size of the SSA             */
/*-----------------------------------------------------------------*/
/*  Initialize a few local variables                               */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    alev = alcn = cnex = 0;     /* clear the flags                 */
    gbrc = STATUS_GB;           /* set the status variable         */
    acrc = STATUS_AC;           /* set the status variable         */
/*-----------------------------------------------------------------*/
/*  Initialize a few local fields                                  */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    firc = 0;                   /* assume a zero return code       */
/*-----------------------------------------------------------------*/
/*  Issue program start message                                    */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    printf( "\nProgram Starts\n\n" );  /* start message            */
    i = 0;                      /* clear count of DL/I calls       */
/*-----------------------------------------------------------------*/
/*  Allocate the environment block                                 */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    rc = SQLAllocEnv(&henv);    /* allocate the environment block  */
    if (rc != SQL_SUCCESS &&    /* operation did not succeed?      */
        rc != SQL_SUCCESS_WITH_INFO)  /* nor success with info?    */
      goto exlb;                /* yes - exit to the caller        */
    alev = 1;                   /* show environment allocated      */
/*-----------------------------------------------------------------*/
/*  Allocate the connection block                                  */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    rc = SQLAllocConnect(henv, &hdbc);  /* allocate connection block */
    if (rc != SQL_SUCCESS &&    /* operation did not succeed?      */
        rc != SQL_SUCCESS_WITH_INFO)  /* nor success with info?    */
      goto exlb;                /* yes - exit to the caller        */
    alcn = 1;                   /* show connection block allocated */
/*-----------------------------------------------------------------*/
/*  Connect to the data source                                     */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    rc = SQLDriverConnect(hdbc,   /* pass the connection block     */
                          (HWND) NULL,  /* no window handle        */
                          (u_char far *)  /* cast the connection str */
```

```
   "UID=ai38pds;PWD=zrc003;"  /* usid          */
                         "PORT=1250;HOST=202.0.12.6;"  /* TCP/IP     */
                         "DSN=Tuld;"  /* Data source name            */
                         "APPL=BMP;"  /* application type string     */
                         "BMPA=BMP,SM400004,SM400004,SM563011;",
                         SQL_NTS,  /* string is null-terminated      */
                         NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
    if (rc != SQL_SUCCESS &&      /* operation did not succeed?       */
        rc != SQL_SUCCESS_WITH_INFO)  /* nor success with info?       */
      goto exlb;                  /* yes - exit to the caller         */
    cnex = 1;                     /* show connection complete         */
/*-------------------------------------------------------------------*/
/*  Get the IMS PCB list                                             */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    rc = SCCToDLI(hdbc,           /* pass the connection block        */
                  (SDWORD) 3,     /* number of parameters       */
                  "PCB ",         /* IMS function code                */
                  dumy,           /* dumy PSB string area             */
                  &imar,          /* IMS status area address          */
                  (SDWORD) sizeof(dumy),  /* dummy PSB area size*/
                  (SDWORD) sizeof(imar)); /* IMS status area size*/
    firc = (rc) ? rc : firc;     /* reset the return code           */
    if (firc < 0)                 /* some type of serious error?      */
      goto exlb;                  /* yes - exit with an error code    */
/*-------------------------------------------------------------------*/
/*  I/O PCB                                                          */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    io_pcb  = (IO_PCB_TYPE far *) imar.impcar[0];  /* define I/O PCB  */
/*-------------------------------------------------------------------*/
/*  Database PCB                                                     */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    tbl_pcb = (PCB_24_TYPE far *) imar.impcar[1];  /* define DB PCB   */
/*-------------------------------------------------------------------*/
/*  Attempt to retrieve the first segment. Note that the        */
/*  PCB can be passed either using the PCB address or using          */
/*  the PCB number. The call below passes an actual PCB address.     */
/*  The following call uses the __cdecl entry point.  The call       */
/*  below uses the pascal entry point. Both entry points may be      */
/*  used from a C program.  The pascal entry point is intended       */
/*  for Visual Basic programs and other desktop productivity         */
/*  tools.                                                           */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    rc = SCCToDLI(hdbc,           /* pass the connection block        */
                  (SDWORD) 4,     /* number of parameters       */
                  "GN  ",         /* IMS function code                */
                  tbl_pcb,        /* DB PCB                           */
                  &unsol_root,    /* segment work area                */
                  ssaunsol,       /* segment search argument    */
                  (SDWORD) szpc24,  /* size of DB PCB                 */
                  (SDWORD) szioar,  /* size of I/O area               */
                  (SDWORD) szssar);  /* size of SSA                   */
    firc = (rc) ? rc : firc;      /* reset the return code            */
    if (firc < 0)                 /* some type of serious error?      */
      goto exlb;                  /* yes - exit with an error code    */
```









```
/*--------------------------------------------------------------------*/
/*  Check for a special return code value                             */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    if (acrc == rc)              /* 'AC' return code?                 */
      goto exlb;                 /* yes - exit with an error code     */
/*--------------------------------------------------------------------*/
/*  Print the data returned                                           */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    if (rc == STATUS_OK)         /* first operation worked?           */
    {                            /* yes - display the data            */
      printf("unsol_root.type = %.2s\n",unsol_root.type);
      printf("unsol_root.terminal = %.5s\n",unsol_root.terminal);
      printf("unsol_root.message = %.25s\n",unsol_root.message);
    }                            /* end of first operation check      */
/*--------------------------------------------------------------------*/
/*  Attempt to retrieve the remaining segments                        */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    while ((rc == STATUS_OK) && (i < 4))  /* while return code zero   */
    {                            /* fetch each remaining segment      */
/*--------------------------------------------------------------------*/
/*    The pascal function can be used instead of the standard         */
/*    function. However, dummy values must be supplied for all        */
/*    of the unused addresses and lengths. Note that the PCB    */
/*    is passed as a PCB number rather than a PCB address.            */
/*    This approach is supported for both the SCCToDLI and            */
/*    the SCCToDLIPascal entry points.                                */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
      rc = SCCToDLIPascal(hdbc,  /* pass the connection block         */
                    (SDWORD) 4,  /* number of parameters       */
                    "GN  ",      /* IMS function code                 */
                    (SDWORD) 2,  /* DB PCB                            */
                    &unsol_root, /* segment work area                 */
                    ssaunsol,    /* segment search argument    */
                    (char far *) NULL, (char far *) NULL,  /* dummy   */
                    (char far *) NULL, (char far *) NULL,  /* dummy   */
                    (char far *) NULL, (char far *) NULL,  /* dummy   */
                    (char far *) NULL, (char far *) NULL,  /* dummy   */
                    (char far *) NULL, (char far *) NULL,  /* dummy   */
                    (char far *) NULL, (char far *) NULL,  /* dummy   */
                    (char far *) NULL, (char far *) NULL,  /* dummy   */
                    (SDWORD) szpc24,  /* size of the DB PCB           */
                    (SDWORD) szioar,  /* size of the I/O area         */
                    (SDWORD) szssar,  /* size of the SSA              */
                    (SDWORD) NULL, (SDWORD) NULL,  /* dummy values    */
                    (SDWORD) NULL, (SDWORD) NULL,  /* dummy values    */
                    (SDWORD) NULL, (SDWORD) NULL,  /* dummy values    */
                    (SDWORD) NULL, (SDWORD) NULL,  /* dummy values    */
                    (SDWORD) NULL, (SDWORD) NULL,  /* dummy values    */
                    (SDWORD) NULL, (SDWORD) NULL,  /* dummy values    */
                    (SDWORD) NULL, (SDWORD) NULL);  /* dummy values   */
      firc = (rc) ? rc : firc;   /* reset the return code             */
      if (firc < 0)              /* some type of serious error?       */
        goto exlb;               /* yes - exit with an error code     */
      printf("unsol_root.type = %.2s\n",unsol_root.type);  /* printf  */
```

17

18

```
      printf("unsol_root.terminal = %.5s\n",unsol_root.terminal);
      printf("unsol_root.message = %.25s\n",unsol_root.message);
      i++;                          /* increment DL/I call counter      */
    }                               /* end of fetch segment loop        */
/*-------------------------------------------------------------------*/
/*  If the status is not blank or "GB" then something is wrong       */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    if (rc != STATUS_OK && rc != STATUS_GB)  /* any type of error?    */
    {                               /* yes - report the error           */
      printf("Error Detected\n\n");  /* error heading                   */
      printf("ctdli rc     = %d/0x%04x\n",rc,rc);  /* display the rc  */
      printf("ctdli status = '%c%c'\n",  /* display IMS status code     */
             tbl_pcb -> dbpc_stcd[0],  /* first byte of the status    */
             tbl_pcb -> dbpc_stcd[1]);  /* second byte of the status  */
      printf("ctdli seg    = %s\n",  /* display the segment name      */
             tbl_pcb -> dbpc_sgna);  /* segment name area             */
      printf("ctdli kfb    = %s\n",  /* display the key               */
             tbl_pcb -> dbpc_kyar);  /* display the key area          */
      firc = 8;                     /* exit with an error code          */
    }                               /* end of error reporting           */
/*-------------------------------------------------------------------*/
/*  Standard exit label                                              */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
  exlb:                             /* standard exit label              */
    printf( "\nProgram Ends\n\n" );  /* terminate message               */
/*-------------------------------------------------------------------*/
/*  Release all resources and return to the caller                   */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    if (cnex)                       /* connection executed?       */
      SQLDisconnect(hdbc);          /* yes - disconnect from host       */
    if (alcn)                       /* connection allocated?            */
      SQLFreeConnect(hdbc);         /* yes - free the connection block  */
    if (alev)                       /* environment allocated?           */
      SQLFreeEnv(henv);             /* yes - free the environment block */
/*-------------------------------------------------------------------*/
/*  Return a value to the caller                                     */
/*--+----1----+----2----+----3----+----4----+----5----+----6----+----7*/
    return(firc);                   /* exit with a zero return code     */
  }                                 /* end routine processing           */
```

19

1. This program uses several standard C library functions. These header files declare the entry points to the standard C library. The standard C library can be used interoperably between the Windows, OS/2, and UNIX environments.

2. The #defines in this section allow the ODBC header files to be used in the OS/2 and UNIX environments. These #defines are not needed (nor are they used) in the Windows environment.

3. The standard ODBC header files used in the Windows environment are included here. These header files can only be used in the Windows environment.

4. The modified header files for OS/2 and UNIX are included here. These ODBC header files are slightly modified versions of the standard Windows ODBC header files.

5. The scpgd.h header file is the only Neon Systems-specific header file. This header file contains all the declarations and definitions needed for the Shadow IMS Direct and Shadow RPC Direct extensions to the ODBC specification. This header file must be included by all Shadow IMS/RPC Direct applications.

6. These statements define the environment block and the connection block handles used in this application. All calls to the ODBC API, including the Neon Systems extensions to the ODBC API, require that either an environment block or connection block handle be passed.

7. These defines are provided to help a programmer check the results from a DL/I call. DL/I calls will return either an ODBC return code or an IMS two byte status code stored in the return code itself. Each DL/I return code should be checked to see if it contains either SQL_SUCCESS, an ODBC error code (such as SQL_ERROR), or an IMS two byte status code.* Note that STATUSOK, and SQL_SUCCESS are both defined to the same value: zero.

8. The PSB name string should always be set to the value “DUMMYPSB”. The “DUMMYPSB” name string is not used in IMS BMP applications at this time; however, it may be used in IMS BMP applications in the future, and this value is strongly recommended for upward compatibility.

   The segment search argument is used to identify specific records in the IMS database. This value is unique to the current sample IMS BMP application and would, of course, be different in any user application. Note that the SSA string ends with a blank. This is the same SSA string termination convention used on the host.

   The PCB type defined here is a database PCB. Note that 24 bytes have been provided for the key feedback area. The size of the key feedback area is entirely application-specific, and must be set to the correct value for each user application.

9. The unsolicited root structure maps a segment in the IMS database used by this application. This structure is specific to this sample program and would, of course, need to be different in any Shadow IMS Direct user application program.

10. The pointers to the I/O PCB and DB PCB are defined here. These pointers are set so that the I/O PCB and DB PCB can be manipulated using standard C language programming constructs. The lengths of each of the data areas passed to and from the Shadow IMS Direct interface functions must be known. The C language statements in this section determine and store the length of each of these data areas. Note that the requirements for data area

* For more information, about IMS status codes, see the IBM Manual, IMS Messages and Codes (SC26-4290).

lengths is a significant difference between the Shadow IMS Direct DL/I interface function and the standard C language DL/I interface function CTDLI on the host. Note that if these lengths are converted to negative values, then all automatic data conversions will be turned off. In other words, data areas passed to and from the Shadow IMS Direct interface functions will not automatically be converted from ASCII to EBCDIC before they are sent to the host or from EBCDIC to ASCII when they are returned from the host. Automatic ASCII/EBCDIC conversion must be turned off if any data areas being passed to and from the IMS interface function contain binary fields such as integers.

Automatic data type conversion can even be turned off for PCBs passed to the IMS function. However, it's seldom necessary to disable conversion for PCBs. The conversion routines are aware of the internal structure of both DB and I/O PCBs and will automatically convert each field (including binary fields) correctly.

11. The environment block is allocated here. It is used as a handle for maintaining the list of connection blocks below. A Shadow IMS Direct application can actually establish any number of connections to the host. All of the connections can actually be connections to different hosts running completely different types of code for each different connection.

12. The connection block includes all of the connection-specific information. The connection block must be successfully allocated before a connection to the host can be attempted.

13. The **SQLDriverConnect** function is used to establish a connection to the host and to initiate the BMP running inside the Shadow Server address space. The connection string includes all the information needed to establish the host connection and to initiate the BMP in the Shadow Server address space. The application type is specified as BMP. This value is required for all BMP type connections to the host. The BMP parameter string is also specified as part of the overall connection string. The BMP parameter string includes the host PSB name and the LTERM name. When this function call returns to the application program, either a successful connection to the host will have been established, or the connection attempt will have failed. If a successful connection has been established, a BMP will now be waiting for requests from the client.

14. **SCCToDLI** obtains the IMS PCB list from the DL/I interface routine. An IMS PCB list contains the actual number of PCBs available and a vector of pointers to the individual PCBs. This call is executed entirely by the Shadow IMS Direct driver and does not require any communication with the host. The first operand to this call is the connection handle. Of course, only a connection handle that has been successfully used to establish a connection to the host can be used to obtain the PCB list. The second operand to this call is a parameter count. This value (3) is passed as a four-byte integer to the DL/I interface routine. The DL/I interface routine uses the number of parameters to analyze and check the rest of the parameter list. The parameter count includes

the IMS function code string, the PSB string and the IMS status area address, but does not include the two length values at the end of the parameter list.

The next argument to this function is the IMS function code, “PCB.” The IMS function code string is used to specify what type of processing the DL/I interface routine should perform. In this case, the “PCB“ function code is used to request the PCB list. The “PCB“ function code is used on the mainframe to schedule a PSB in the CICS environment and to obtain the PCB list. This is a significant difference between the mainframe and client implementation of the DL/I interface. However, in the future, the ”PCB“ function code may be supported as a means of scheduling a PSB in the client environment as well (but only for multi-threaded IMS operations).

The next argument to the **SCCToDLI** function code is the “DUMMYPSB” string. The dummy PSB string is not actually used by this call. However, the literal “PSB” should be passed for compatibility with future releases of this product. The fifth operand is the IMS status area address. This control block is filled in by this call to the IMS interface routine. This control block is defined in the scpghd.h header file.

The last two operands are the lengths of the dummy PSB string and the IMS status area. These lengths must be correctly specified. These length values are used to determine how much data is transmitted to the host and returned from the host. The requirement that these lengths be specified is another difference between the Shadow IMS Direct implementation of the **SCCToDLI** function and the mainframe implementation of the CTDLI function. Also, the length arguments are not included in the overall parameter count. In addition, no length value is specified for the IMS function code string. The IMS function code string is always four bytes long and, as a consequence, no length value is needed. The length values can be switched to negative values. If negative values are passed to this function then all argument conversion is turned off.

15. These statements extract the addresses of the I/O PCB and the DB PCB from the PCB list and store the PCB addresses in local variables. The PCB address values are used below in the **SCCToDLI** calls.

16. This call actually fetches a segment from an IMS database. The number of parameters is four, including the IMS function code, the address of the DB PCB, the address of the segment work area, and the address of the segment search argument (SSA). As mentioned above, only three length values are needed because the length of the IMS function code string is always known.

17. This statement checks to see if the DL/I call set a status code of AC. If the AC status code was returned, this program automatically terminates.

18. This call to the **SCCToDLIPascal** function is actually identical to the **SCCToDLI** function call above. The main difference is that the Pascal entry point has been used in this case. A C program can, of course, call either entry point, as needed. In general, it is easier for a C program to call the **SCCToDLI** entry point because dummy values are not needed for all of the missing arguments. Another minor difference between this call to **SCCToDLI**

and the earlier call is that the PCB has been selected by number rather than by address. It is always possible to specify which PCB should be used for a given call by specifying the PCB number as a four-byte integer. The PCB number is two in this case because PCBs are numbered from one, not zero. The PCB can be specified by number rather than by address for both the **SCCToDLIPascal** and **SCCToDLI** calls. The code in this section checks for any kind of unexpected error and uses the pointer to the PCB to extract fields from the PCB that are used to describe the error.

19. The calls in this section are used to break the connection with the host and release all of the resources that were obtained for connection to the host. The disconnect call will terminate the BMP running inside the Shadow Server address space. The two free calls release the connection block and the environment block, respectively.

CHAPTER 5:
Transaction Server for IMS

This chapter covers programming information for SHADOW_IMS, a generic RPC that allows you to invoke an existing transaction. Information includes examples showing use of SHADOW_IMS.

This chapter applies to Shadow Direct and Shadow OS/390 Web Server.

Introduction

SHADOW_IMS can be invoked from any ODBC-compliant application on the client workstation as a pass-through query.

The SHADOW_IMS RPC is invoked using the following ODBC CALL statement:

```
CALL SHADOW_IMS('IME','TRANSACTION PROGRAM NAME','IMS-PARTNER-LU
NAME ','SECURITY-TYPE','TP PARAMETERS','COLUMN NAME','LOCAL LU
NAME','MODE NAME','SYMDEST','USERID','PASSWORD','PROFILE','SEND-
TYPE','MESSAGE-LENGTH')
```

Parameter	Definition
'IME'	A keyword describing the interface.
'TRANSACTION PROGRAM NAME'	Name of IMS transaction to be executed.
'IMS-PARTNER-LUNAME'	The name of the IMS Partner LU as defined in SYS1.PARMLIB(APPCPMxx).
'SECURITY-TYPE'	Defines the type of security in use: • **NONE** specifies to omit access security information on this allocation request. • **SAME** specifies to use the userid and security profile (if present) from the allocation request that initiated the local program. The password (if present) is not used; instead, the userid is indicated as being already verified. If the allocation request that initiated execution of the local program contained no access security information, then access security information is omitted on this allocation request. • **PROGRAM** specifies to use the access security information that the local program provides on the call. The local program provides the information by means of the USERID, PASSWORD, and PROFILE parameters. These values are passed exactly as specified, without folding to uppercase.
'TP PARAMETERS'	Parameters for Transaction Program.

Parameter	Definition
'COLUMN NAME' **or** **'MAP NAME'**	Column name or map name used for returned data. For the **MAP** keyword, the syntax is: **'MAP(NAME(PARTREXX) FIELDS(*))'** **NAME:** This entry should correspond to the name assigned to the map during extraction. **FIELDS:** There are two ways to return data from all columns that are enabled in the map definition: • Use an asterisk after FIELDS. • Omit FIELDS altogether. To exclude some columns, enter the names of the enabled columns you do want returned in parentheses after FIELDS. For more information about the Data Mapping facility, refer to the *Shadow Server User's Guide*.
'LOCAL LU NAME'	**Optional.** Name of local LU where caller's allocate request originates. The ability to specify local LU name allows caller to associate its outbound conversations with particular LUs. The caller's address space must have access to the named LU. Otherwise, a parameter_error return code is returned. This is the new local LU Name specified in SYS1.PARMLIB(APPCPMxx) This parameter is optional. The default is to use the APPC Base LU, as defined in SYS1.PARMLIB(APPCPMxx). ***Note*:** It is recommended that a separate Local LU be defined for each Shadow Server you have running using IMS/APPC. Application developers should be informed of which LU should be used with which copy of the *Shadow Server*. The APPC base LU will work in most cases, however using a separate Local LU tends to be more reliable.
'MODE NAME'	**Optional.** Specifies the mode name designating the network properties for the session to be allocated for the conversation. The network properties include, for example, the class of service to be used. The mode name value of 'SNASVCMG' is reserved for use by APPC/MVS. If a mode name of 'SNASVCMG' is specified on the Allocate service, the request is rejected with a return code of parameter_error. If you specify a symbolic destination name in the SYMDEST name parameter, set MODE NAME to blanks to obtain the MODE NAME from the side information. If the partner LU is the same or on the same system as the LOCAL LU NAME, MODE NAME is ignored. If the partner LU is on a different system, and you do not specify a SYMDEST name, a blank MODE NAME defaults to any mode in effect for the local and partner LUs, or causes a return code of parameter_error.
'SYMDEST'	**Optional.** Specifies a symbolic name representing the partner LU, the PARTNER TP NAME, and the MODE NAME for the session on which the conversation is to be carried. The symbolic destination name must match that of an entry in the side information data set. The appropriate entry in the side information is retrieved and used to initialize the characteristics for the conversation. If you specify a SYMDEST name, the PARTNER LU NAME, MODE NAME, and TP NAME are obtained from the side information. If you also specify values for the PARTNER LU NAME, MODE NAME, or TP NAME parameters on the Allocate service, these values override any obtained from the side information. The SYMDEST name in this field can be from 1 to 8 characters long, with characters from character set 01134. If the SYMDEST name is shorter than eight characters, it must be left-justified in the variable field, and padded on the right with blanks. To not specify a SYMDEST name, set the SYMDEST name parameter value to 8 blanks and provide values for the PARTNER LU NAME, MODE NAME, and TP NAME parameters.

Parameter	Definition
'USERID'	**Optional.** Specifies the userid. The partner LU uses this value and the password to verify the identity of the end user that initiated the allocation request. The partner LU may use this value for auditing and accounting purposes, and, together with the security profile (if present), to determine which partner programs the local program can access. When the partner LU is on MVS with RACF protection, the userid must be 1-8 alphanumeric characters. This parameter is significant only when the Security_type parameter contains a value of Pgm. Otherwise, this parameter has no meaning and is ignored.
'PASSWORD'	**Optional.** Specifies the password. The partner LU uses this value and the userid to verify the identity of the end user that made the allocation request. When the partner LU is on MVS with RACF protection, the password must be 1-8 alphanumeric characters padded with blanks. This parameter is significant only when the Security_type parameter contains a value of Pgm. Otherwise, this parameter has no meaning and is ignored.
'PROFILE'	**Optional.** Specifies additional security information that may be used to determine what partner programs the local program may access, and which resources the local program may access. When the partner LU is on MVS with RACF protection, APPC/MVS treats the profile name as a RACF group name for verifying access to partner programs. The profile name must be 1-8 alphanumeric characters. This parameter is significant only when the Security_type parameter contains a value of Pgm. Otherwise, this parameter has no meaning and is ignored.
'SEND-TYPE'	**Optional.** SEND-TYPE specifies what, if any, information is to be sent to the partner program in addition to the data supplied. SEND-TYPE also lets you combine operations (for example, Send_and_confirm) and save extra calls to APPC. Default value is 1. Valid values for this parameter are: • **0 Buffer_data** Specifies that no additional information is to be sent to the partner program, and the data may be buffered until a sufficient quantity is accumulated. • **1 Send_and_flush** Specifies that no additional information is to be sent to the partner program. However, the supplied data is sent immediately rather than buffered. This is functionally equivalent to a Send_data call with the Send_type parameter set to Buffer_data followed by a Flush call. • **2 Send_and_confirm** Specifies that the supplied data is to be sent to the partner program immediately, along with a request for confirmation. This is functionally equivalent to a Send_data call with the Send_type parameter set to Buffer_data followed by a Confirm call. • **3 Send_and_prepare_to_receive** Specifies that the supplied data is to be sent to the partner program immediately, along with send control of the conversation. This is functionally equivalent to a Send_data call with the Send_type parameter set to Buffer_data followed by a Prepare_to_receive call with the prepare_to_receive_type set to sync_level and the locks parameter set to short. • **4 Send_and_deallocate** Specifies that the supplied data is to be sent to the partner program immediately, along with a deallocation notification. This is functionally equivalent to a Send_data call with the Send_type parameter set to Buffer_data followed by a Deallocate call with the deallocate_type set to sync_level.
'MESSAGE LENGTH'	**Optional.** Specifies the length of the messages that are written to or read from the message queue. Default value is 32k.

Examples of Using Shadow_IMS

COBOL

See sample program in hlq.SAMP(SDCOIM for *Shadow Direct* or SWCOIM for *Shadow OS/390 Web Server*) for DBCTL.

See sample program in hlq.SAMP(SDCOIMAP for *Shadow Direct* or SWCOIMAP for *Shadow OS/390 Web Server*) for Transaction Server for IMS.

Visual Basic 3.0

This sample code uses the IMS IVP sample database DI2IPART and the IMS sample transactions:

```
Set myDB = OpenDatabase(“”, False, False,
“ODBC;DSN=SHADOW_DIRECT”)
```

Part numbers:

```
AN960C10      7438995P002
3003806*      7618032P101*
3007228       922399-001
3013412       82125-869
652799
```

The transaction PART inquires into the part number database for information from the part master and standard information segments of a specific part number. The input format is *transaction code, part number* entered as follows:

```
MySQL = “CALL
SHADOW_IMS(””IMS””,””PART””,””IMSLU62””,””NONE””,””AN960C10””,””A
PPC-IMS-DATA””)”
```

The transaction DSPALLI displays all inventory, cycle count, and back-order information for a specific part. The input format is *transaction code, part number* entered as follows:

```
MySQL = “CALL
SHADOW_IMS(””IMS””,””DSPALLI””,””IMSLU62””,””NONE””,””AN960C10””,
””APPC-IMS-DATA””)”
```

To display inventory information for key 28009126 and part number an960c10, the input format is *transaction code, part number, inventory-location-key* entered as follows:

```
MySQL = “CALL
SHADOW_IMS(””IMS””,””DSPINV””,””IMSLU62””,””NONE””,””AN960C10,280
09126””,””APPC-IMS-DATA””)”

Set myset = myDB.CreateDynaset(MySQL, 64)
myset.MoveFirst
txtoutput.Text = myset(“APPC-IMS-DATA”)
I=1
myset.MoveLast
myset.MovFirst
Do Until myset.EOF
   N= myset(“APPC-IMS-DATA”) & I
   IstData.AddItem N
   myset.MoveNext
   I = I + 1
Loop
myset.MoveFirst
End Sub
```

PowerBuilder 4.0

The following PowerBuilder script invokes the sample IMS IVP PART transaction:

```
//
//Call SHADOW_IMS
//
LONG ll_Row, ll_NewRow
String ls_PartID,PARM1,PARM2,PARM3,PARM4,PARM5,PARM6
int li_Count
transaction ShadowDirect
//
//Create the ShadowDirect transaction object
//
ShadowDirect = Create Transaction
//
//Assign values to the ShadowDirect transaction
//

ShadowDirect.DBMS       = "ODBC"
ShadowDirect.database   = ""
ShadowDirect.userid     = ""
ShadowDirect.dbpass     = ""
ShadowDirect.logid      = ""
ShadowDirect.logpass    = ""
ShadowDirect.servername = "NONE"
ShadowDirect.dbparm  =
"ConnectString='DSN=SHADOW_DIRECT;PWD=xxxxxxx'"
```

```
// Connect to the ShadowDirect ODBC transaction object
CONNECT USING ShadowDirect ;
/* Sheet opening - reflect sheet count in title */
li_Count = w_genapp_frame.wf_getsheetcount ()
this.Title = "Sheet:" + string (li_Count)
/* Modify menu text for platform */
w_genapp_frame.wf_setmenutext (menuid)

//
//Declare the RPC with a parameter
//
   PARM1="IMS"
   PARM2="PART"
   PARM3="IMSLU62"
   PARM4="NONE"
   PARM5="AN960C10"
   PARM6="APPC-IMS-DATA"
Declare GetIMSData Procedure for SHADOW_IMS
:parm1,:parm2,:parm3,:parm4,:parm5,:parm6
using ShadowDirect;

//
//Call the RPC
//
Execute GetIMSData;
//
//Process the result set
//

CHOOSE CASE ShadowDirect.SQLCode
CASE 0
   DO WHILE ShadowDirect.SQLCode = 0
         FETCH GetIMSData INTO :ls_PartID;
         IF
END CHOOSE

//
//Close the procedure
//
//Commit using ShadowDirect;


Close GetIMSData;
```

/*EXECSQL

The following /*EXECSQL script invokes the sample IMS PARTS transaction for part number 3007228 and returns the results:

```
/*WWW /NEON/IMSEXEC1
*************************************************************
*    SAMPLE APPLICATION WHICH ILLUSTRATEST THE USE OF AN EXECSQL *
*    PROCESS SECTION.  THE AUTOFORMAT KEYWORD CALLS FOR THE      *
*    ROW DATA TO BE FORMATTED INTO AN HTML TABLE.                *
*************************************************************
*

/*EXECSQL MAXROWS(100) -
     SUBSYS(NONE) PLAN(NONE) -
       AUTOFORMAT( TITLE('SAMPLE IMS QUERY USING /*EXECSQL') -
                   BODY('BGCOLOR="#FFCC33"') -
                  )
CALL SHADOW_IMS('IMS','PART','IMSLU62','SAME','3007228','APPC-
DATA')
```

Note:
This script applies to Shadow OS/390 Web Server only.

CHAPTER 6:
Transaction Server for CICS

This chapter covers programming information for SHADOW_CICS, a generic RPC that allows you to invoke an existing transaction as long as the transaction does not send back a prompt requiring a response.

This chapter applies to Shadow Direct and Shadow OS/390 Web Server.

Introduction

SHADOW_CICS can be invoked from any ODBC-compliant application on the client workstation as a pass-through query (please see the Visual Basic and Powerbuilder samples).

You can invoke the RPC with the following ODBC CALL statement.

```
CALL
SHADOW_CICS('NNNN','CCCC','TTTT','PPPPPPPP',1,2,3,4,5,6,'DATA')
```

Parameter	Description
'NNNN'	Connection-type as defined in the SD exec, "EXCI".
'CCCC'	Connection-name as defined in the SD exec.
'TTTT'	Tran-ID as defined in CICS. For EXCI, this is the Tran-ID that is associated with the DFHMIRS program.
'PPPPPPPP'	Program name as defined in CICS.
'1'	First parameter expected by program.
'2'	Second parameter expected by program.
'3'	Third parameter expected by program.
'4'	Fourth parameter expected by program.
'5'	Optional. Length of commarea. If not present, default is 32k.
'6'	Indicates if recursive execution of transaction is required. Possible values are • **Y** for yes • **N** for no. Default value is **N**.

Parameter	Description
'DATA' **or** **'MAP'**	Column name or map name to be used for returned data. For MAP keyword, syntax is: 'MAP(NAME(EXCI) FIELDS(*))' **NAME:** This entry should correspond to name assigned to map during extraction. **FIELDS:** There are two ways to return data from all columns that are enabled in the map definition: • Use an asterisk after FIELDS • Omit FIELDS altogether To exclude some columns, enter the names of the enabled columns you do want returned in parentheses after FIELDS. For more information about the Data Mapping facility, refer to the *Shadow Server User's Guide*.
'NNNN'	Connection-type as defined in the SD exec, "EXCI".

Table 6–1. Shadow_CICS ODBC Call Parameters

Example of EXCI demo transaction:

```
CALLSHADOW_CICS('EXCI','EXCS','EXCI','DFH$AXCS',2,'FILEA   ',
'     1','',100,'','DATA')
```

Note:
In the above statement FILEA is followed by three spaces (for a total of eight characters) and "1" is preceded by five spaces (for a total of six characters).

Examples of Using Shadow for CICS

COBOL

See sample program in hlq.SAMP (SDCOCIEC for Shadow Direct or SWCOCIEC for Shadow OS/390 Web Server).

Visual Basic 3.0

This sample code uses the CICS IVP sample VSAM file FILEA and the sample CICS transaction EXCI:

```
Set myDB = OpenDatabase("", False, False,
"ODBC;DSN=SHADOW_DIRECT")
```

The transaction EXCI executes an external CICS interface program named DFH$AXCS that browses the sample VSAM file FILEA. The input format is:

```
mySQL = "CALL +
SHADOW_CICS(""EXCI"",""EXCS"",""EXCI"",""DFH$AXCS"",2,""FILEA "",
+
"                        "     1"",""""","""","""AI38GW"","""CICS-
DATA"")"
```

To call an RPC program that executes the same sample EXCI transaction, use the following format:

```
'mySQL = "CALL SDCOCIEC"

Set myset = myDB.CreateDynaset(mySQL, 64)
myset.MoveFirst
```

To display the first row of output from the SHADOW_CICS call in a text box, use the following example (the data is returned in a single column with multiple rows from call shadow_cics):

```
txtOutput.Text = myset("CICS-DATA")
```

To display the first of row output from the call rpc SDCOCIEC in a text box, use the following example (the rpc returns the data in seven columns with multiple rows):

```
'txtOutput.Text = (myset("Number") & " " & myset("Name") & " "  +
& myset("Address") & " " & myset("Phone"))

'txtOutput.Text = txtOutput.Text & (" " & myset("date") & " "   +
& myset("amount") & " " & myset("comment"))

I = 1
myset.MoveLast
myset.MoveFirst
```

To display the remaining data from the call:

```
Do Until myset.EOF
    ' Uncomment next line for use with CALL RPC SDCOCIEC
    'N = myset("Number") & " " & myset("name") & " "   +
    & myset("address") & " " & myset("phone")
   'Uncomment next line for use with SHADOW_CICS
    N = Left$(myset("CICS-DATA"), 80) & I
    lstData.AddItem N
    myset.MoveNext
    I = I + 1
Loop
myset.MoveFirst
End Sub

+  Indicates a continuation where code should all be on the same line
```

PowerBuilder 4.0

The following PowerBuilder script is used to invoke the sample SHADOW_CICS transaction:

```
//
//Call SHADOW_CICS
//
LONG ll_Row, ll_NewRow
String PARM1,PARM2,PARM3,PARM4,PARM6,PARM7,PARM8,PARM10,PARM11
string ls_PartID
int li_Count,PARM5,PARM9
transaction ShadowDirect
//
//Create the ShadowDirect transaction object
//
ShadowDirect = Create Transaction
//
//Assign values to the ShadowDirect transaction
//
ShadowDirect.DBMS       = "ODBC"
ShadowDirect.database   = ""
ShadowDirect.userid     = ""
ShadowDirect.dbpass     = ""
ShadowDirect.logid      = ""
ShadowDirect.logpass    = ""
ShadowDirect.servername = "NONE"
ShadowDirect.dbparm     = "ConnectString='DSN=P39016"

// Connect to the ShadowDirect ODBC transaction object
CONNECT USING ShadowDirect ;
/* Sheet opening - reflect sheet count in title */
li_Count = w_genapp_frame.wf_getsheetcount ()
this.Title = "Sheet:" + string (li_Count)
/* Modify menu text for platform */
w_genapp_frame.wf_setmenutext (menuid)
//
//Declare the RPC with a parameter
//
//call shadow_cics('EXCI','EXCS','EXCI','DFH$AXCS',2,'FILEA
','     1','',120,'','EXCI-DATA')
   PARM1="EXCI"
   PARM2="EXCS"
   PARM3="EXCI"
   PARM4="DFH$AXCS"
   PARM5=2
   PARM6="FILEA     "
   PARM7="     1"
   PARM8=""
   PARM9=120
   PARM10=""
   PARM11="EXCI-DATA"
```

```
Declare GetCICSData Procedure for SHADOW_CICS
:parm1,:parm2,:parm3,:parm4,:parm5,:parm6,:parm7,:parm8,:parm9,:p
arm10,:parm11
using ShadowDirect;
//
//Call the RPC
//
Execute GetCICSData;
//
//Process the result set
//
CHOOSE CASE ShadowDirect.SQLCode
CASE 0
DO WHILE ShadowDirect.SQLCode = 0
FETCH GetCICSData INTO :ls_PartID;
IF ShadowDirect.SQLCode = 0 THEN
ll_NewRow = Dw_1.InsertRow(0)
ll_Row = DW_1.ScrollToRow( ll_NewRow )
DW_1.SetItem( ll_NewRow,"part_id" , ls_PartID )
END IF
LOOP
CASE ELSE
END CHOOSE
//
//Close the procedure
//
//Commit using ShadowDirect;
Close GetCICSData;
```

/*EXECSQL

The following /*EXECSQL script is used to invoke the sample SHADOW_CICS transaction:

```
/*WWW /NEON/CICEXEC1
*****************************************************************
*     Sample application which illustratest the use of      *
*     an EXECSQL* process section.                          *
*                                                               *
*****************************************************************

/*EXECSQL MAXROWS(100) -
         outputformat( ddname(SAMPDATA) MEMBER(SQLEXEC3) -
                       CONTENTTYPE(text/html) -
                     )
call shadow_cics('EXCI','EWSS','EXCI','DFH$AXCS',2,'FILEA  ','
1',-
'',120,'','MAP(NAME(EXCIMAP) FIELDS(*))')
```


Note:

This script applies to Shadow OS/390 Web Server only.

CHAPTER 7:
Host Application API Function Calls

This chapter describes all Host Application Program Interface (API) functions, and ***applies to Shadow Direct and Shadow Web Server.***

API Description	DIRECT	WEB	SEF	WEB/RX
ODBC CALL RPC APIs				
To bind columns for result set:	SQLBINDCOL or SDCPBC	⛔	⛔	⛔
To describe passed parameter:	SQLDESCRIBEPARAM or SDCPDP	⛔	⛔	⛔
To access number of parameters:	SQLNUMPARAMS or SDCPNP	⛔	⛔	⛔
To reset parameters:	SQLRESETPARAM or SDCPRP	⛔	⛔	⛔
To return status to client:	SQLRETURNSTATUS or SDCPRS	⛔	⛔	⛔
To return row to result set:	SQLTHROW or SDCPTH	⛔	⛔	⛔
IMS/APPC APIs				
To connect to APPC for IMS:	SQLAPPCCONNECT or SDCPAC	SWSAPPCCONNECT or SWCPAC	SDBAPCON	SWSAPCON
To disconnect from APPC for IMS:	SQLAPPCDISCONNECT or SDCPAD	SWSAPPCDISCONNECT or SWCPAD	SDBAPDIS	SWSAPDIS
To receive and wait from APPC for IMS:	SQLAPPCRECEIVE or SDCPAR	SWSAPPCRECEIVE or SWCPAR	SDBAPRCV	SWSAPRCV
To perform a send to APPC for IMS:	SQLAPPCSEND or SDCPAS	SWSAPPCSEND or SWCPAS	SDBAPSND	SWSAPSND
CICS APIs				
To establish EXCI connect:	SQLEXCICONNECT or SDCPEC	SWSEXCICONNECT or SWCPEC	SDBEXCON	SWSEXCON
To perform DPL request using EXCI:	SQLEXCIDPLREQ or SDCPED	SWSEXCIDPLREQ or SWCPED	SDBEXDPL	SWSEXDPL
To perform EXCI initusr:	SQLEXCIINITUSR or SDCPEI	SWSEXCIINITUSR or SWCPEI	SDBEXINI	SWSEXINI
To perform EXCI disconnect:	SQLEXCIDISCONN or SDCPEL	SWSEXCIDISCONN or SWCPEL	SDBEXDIS	SWSEXDIS

API Description	DIRECT	WEB	SEF	WEB/RX
Web Server Specific APIs				
To transmit data to Web Server clients:	⊘	SWCPSN	⊘	SWSSEND
To buffer outbound HTTP response headers:	⊘	SWCPRE	⊘	SWSRESP
To transmit data directly to web client:	⊘	SWCPFI	⊘	SWSFILE
To provide new URL value:	⊘	SWCPSO	⊘	SWSSET
To provide a means to issue an MVS write to operator:	⊘	SWCPWT	⊘	SWSWTO
RPC Direct APIs				
To access current execution environment information:	sdcpif	⊘	⊘	⊘
To add text message to trace browser log:	sdcpmg	⊘	⊘	⊘
To read buffer of data from client:	sdcprd	⊘	⊘	⊘
To send buffer of data to client:	sdcpwr	⊘	⊘	⊘
General APIs				
To get error information:	SQLERROR or SDCPSE	SWSERROR or SWCPSE	SDBERROR	SWSERROR
To return information to ODBC CALL RPC:	SQLGETINFO or SDCPGI	SWSINFO or SWCPGI	SDBINFO	SWSINFO
To write message to trace browser:	SQLTRACEMSG or SDCPTM	SWSTRACEMSG or SWCPTM	SDBTRACE	SWSTRACE
To dynamically allocate a file:	SDBALLOC or SDCPAL	SWSALLOC or SWCPAL	SDBALLOC	SWSALLOC
To de-allocate datasets:	SDBFREE or SDCPFR	SWSFREE or SWCPFR	SDBFREE	SWSFREE
To fetch or set transaction run-time variable values:	SDBVALUE or SDCPVL	SWSVALUE or SWCPVL	SDBVALUE	SWSVALUE
To save and restore transaction-oriented data :	SQLTOKEN or SDCPTK	SWSTOKEN or SWCPTK	SDBTOKEN	SWSTOKEN
To concatenate multiple DDNames under a single DDName.	SDBCONCT or SDCPCC	SWSCONCT or SWCPCC.	SDBCONCT	SWSCONCT

API Description	DIRECT	WEB	SEF	WEB/RX
Web Server REXX and SEF only APIs				
To clear REXX external data queue:		SWSClearQueue or SWCPQL		SWSCLEDQ
To perform security authorization processing:			SDBECURE	SWSECURE
To serialize usage of resources:				SWSENQ
"PARSE PULL" operation in Shadow/ REXX:		SWSGetQueue or SWCPQG		
To set or display SWS product parameter values:			SDBPARM	SWSPARM
Equivalent to Shadow/REXX "Queue" (not "QUEUED()"):		SWSPutQueue or SWCPQP		
Partly equivalent to Shadow/REXX built-in function "QUEUED()":		SWSQueryQueue or SWCPQQ		
To create and write customized SMF records:			SDBSMF	SWSSMF
To transmit out-bound data to web server clients:				SWSXMIT

The High-Level Language (HLL) Interface

The Shadow Web Server's High-Level Language (HLL) interface is implemented using a number of small glue routines. There is a separate glue routine for each of the Web Server's Application Program Interfaces (APIs). Each glue routine is actually an alias pointing to a small routine within the SWCPBC, SWCPAC or SWCPIC load modules.

Compiling and Linking HLL Application Programs

When high-level language programs are linkage edited, the product's load library must be available to the linkage editor. Calls from a high-level language program's object module are resolved by including the NEON-supplied glue routines in the final load module built by the linkage editor.

The Shadow Web Server's load library must be included in the SYSLIB dataset concatenation whenever user-written web transaction programs are linkage edited.

Each glue routine receives control via standard MVS linkage conventions, and carries out the work of locating web server internals and then executing the requested function. When the glue routines are entered, the calling HLL programs may have been operating in any of the modes:

- AMODE(31) or AMODE(24)
- RMODE(24) or RMODE(ANY)

The glue routines handle the details of switching to the Server's normal AMODE(31), RMODE(ANY) operational mode. (We strongly recommend that, if possible, all routines be compiled and linked as AMODE(31), RMODE(ANY).)

NEON-Supplied Source Copy Members

In order to successfully invoke a Web Server API routine, each caller must provide one or more parameters in the form expected by the server. Three NEON-supplied sample library members, one for each of the supported languages, contain manifest constants and other data structures which you will need to invoke the server's API routines.

Samples for C, COBOL, and PL/I are available in the **SAMP** dataset.

Layout of the HLL Reference Pages

Each of the HLL API interface is documented separately. Each reference page is laid out in substantially the same format.

- Each HLL reference page contains a leading section which briefly explains the use of the API interface.
- This is followed by the Arguments Section which defines each of the parameters which can be passed to the function. For some Web Server functions, this section can be further broken down into sub-sections (one for each of the major sub-functions of the interface).

- The Arguments Section describes parameter values using the terminology given below. The argument type, for each supported language is also shown. (Note that for C/370, the NEON-supplied header file contains typedef statements which define the values shown.)
- Additional details showing how to create and use arguments of the proper type, are illustrated in the examples for each supported language.
- After the arguments' section, the return code values are explained.
- Finally, there is an example for each of the supported languages; C, COBOL, and PL/I.

Note:
All manifest constants shown outside of the COBOL language examples are given with underbar characters (e.g. `SWS_SUCCESS`). The corresponding COBOL constant definitions use the hyphen character wherever underbars are shown (e.g. `SWS-SUCCESS`).

Call by Reference

Some compilers employ a call by value mechanism. When subroutines are invoked using a call by value mechanism, some parameter values are passed directly to the called program using the 1st-level parameter list area, or a general purpose registers. Values are frequently returned directly within a general purpose register.

All Shadow Web Server API interface routines expect to be invoked using a call by reference mechanism, implemented with the standard MVS parameter list format.

Upon entry to any Web Server API routine, general purpose register 1 must point to a parameter list composed of 1 to n fullwords. Bit zero of the last fullword within the parameter list must be set to one.

Each fullword of the parameter list contains an address value referring to the main storage location containing the actual function argument.

The method for ensuring that call by reference conventions are used for subroutine calls varies by language.

- For C/370 each sub-routine prototype contains the specification:
  #pragma linkage(name,os)
- For COBOL this is automatic.
- For PL/I, each API routine is declared as:
  DCL NAME ENTRY EXTERNAL OPTIONS(RETCODE, INTER, ASM);

Terminology Used in the Reference Pages

The following terminology is used within the HLL API Reference page:

Address

A main-storage address, specified by the low-order 31-bits of a four-byte (32-bit) fullword. In the various high-level languages, the term 'address' is generally equivalent to 'pointer'.

Buffer Area

An arbitrary number of contiguously located bytes in main storage. The size of the buffer area can be a pre-defined value, as specified for the individual API interface. Most buffer areas, however, are user-created and variable in length. Either the length of the entire buffer, or the length of the data value contained within the buffer, is specified as an argument to the API function.

Flag-Word

As pertains to the Web Server APIs, a four-byte signed or un-signed integer, stored within a fullword. The storage format, in most cases, is identical to un-signed integers, except that the fullword is not processed as an integer value. Instead, each individual bit position specifies a parameter option value.

Usually manifest constants, defined within the header files, are used to specify the value of flag-word arguments. These manifest constants can often be added or logically OR-ed together to form the four-byte flag word.

Fullword

A four-byte storage area, generally aligned on a four-byte boundary. A fullword, can contain either a storage address or a signed or un-signed binary integer.

Manifest Constant

A value defined within a NEON-supplied header file and used (possibly added or OR-ed with other constants) as a function call argument.

Depending on the high-level language, manifest constants can be compiler pre-processor symbols (C and PL/I) or actual program constant data areas (COBOL and PL/I). Manifest constants can be formed as Flag-word values or as signed or un-signed integers.

Null-terminated String

Within a buffer area, null terminated strings can often be used to indicate the actual length of the data value. The end of the value (and by derivation the value length) is delimited by a one-byte binary zero (X'00'). The value begins in the first position of the buffer area (if non-zero), and continues up-to, but not including, the delimiting binary zero at the end. A zero-length null-terminated string is recognized when the first byte of the buffer is a binary zero.

Signed Integer

As pertains to the Web Server APIs, a four-byte, signed binary integer, stored within a fullword.

Unsigned Integer

As pertains to the Web Server APIs, a four-byte, un-signed binary integer, stored within a fullword.

ODBC CALL Host APIs

This section covers the following ODBC CALL Host APIs:

API Description	DIRECT	WEB	SEF	WEB/RX
ODBC CALL RPC APIs				
To bind columns for result set:	SQLBINDCOL or SDCPBC	⛔	⛔	⛔
To describe passed parameter:	SQLDESCRIBEPARAM or SDCPDP	⛔	⛔	⛔
To access number of parameters:	SQLNUMPARAMS or SDCPNP	⛔	⛔	⛔
To reset parameters:	SQLRESETPARAM or SDCPRP	⛔	⛔	⛔
To return status to client:	SQLRETURNSTATUS or SDCPRS	⛔	⛔	⛔
To return row to result set:	SQLTHROW or SDCPTH	⛔	⛔	⛔

The standard ODBC Host APIs (SQLDescribeParam and SQLResetParam) support long parameters without any new or additional programming.

- The two-byte length prefix for variable strings will never contain a value greater than 32767, nor can it ever be set to a value greater than 32767. This does not mean that the variable length strings are restricted to a length of 32767 bytes. They can be used to set lengths greater than 32767 (the two-byte length prefix will contain 32767, if the actual length is greater than 32767) and it will report the actual length even if it is greater than 32767 (the two-byte length prefix will contain 32767, if the actual length is greater than 32767).
- The actual length can never be greater than the precision for any parameter, long or otherwise.
- Long parameters must be used with parameter markers. Long literals are not supported.
- Long parameters can only be used with NEON stored procedures. They are not supported using IBM stored procedures.
- Long parameters can not be used with MDI stored procedures.

SQLBINDCOL (SDCPBC) Function

SQLBINDCOL performs a bind column on behalf of an ODBC CALL RPC. This call is used to bind a column to return sets back to the client. The caller must provide information, which is used to build an SQLDA describing the result set.

Syntax

The general form for invocation of SQLBINDCOL is:

```
CALL 'SDCPBC' USING STATEMENT-HANDLE
                    SQL-COLUMN-NUMBER
                    SQL-C-DEFAULT
                    SQL-SMALLINT
                    SQL-PRECISION
                    SQL-SCALE
                    SQL-NO-NULLS
                    ID-VALUE
                    SQL-COLUMN-LEN
                    SQL-COLUMN-NAME
                    SQL-COLUMN-NAME-LEN
```

CALL Arguments

The SQLBINDCOL function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Column number of the result data. Columns are numbered from the left, starting with 1.
3	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	C data type of column data. Value must be SQL-C-default at this time. This means that C type must match SQL type.
4	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	SQL data type of column data. All DB2 SQL data types are supported except for graphic (DBCS) data.
5	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Precision of column. This value is primarily used for decimal and character string data.
6	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Scale of column. This value is primarily used for decimal data.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
7	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Indicates if column can have null values. Possible values are SQL-NO-NULLS and SQL-NULLABLE.
8	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Pointer to storage for data. Actual data must be at this location when SQLTHROW function is called to send a row.
9	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Pointer to a fullword which serves as a null indicator for subsequent SQLTHROW call. Before SQLTHROW function is called to send a row, this fullword should be set to “-1” if the data for the column is NULL. Otherwise, it should be set to “0”. Notice that NULL data is only applicable for DB2. If you write rpc that accesses non-DB2 data, simply initialize this fullword to “0”.
10	CHAR*	PIC (X)X	CHAR(X)	INPUT	Pointer to storage containing column name. Column name must be a valid DB2 column name.
11	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Length of column name string. This must be a valid DB2 column name length.

Return Values

SQLBINDCOL always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation succeeded, but a warning was issued. Call SQLERROR to get warning message.
SQL_NO_DATA_FOUND	Error condition. No data returned.
SQL_ERROR	A parameter validation error was found. The error will be logged to the wrap-around trace, and is available using the SQLERROR function.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR.
SQL_STILL_EXECUTING	Error condition. Another function is still executing.
SQL_NEED_DATA	Error Condition. The application needs to send parameter data values.

Diagnostics

When SQLBINDCOL returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLERROR. The following table lists the SQLSTATE values commonly returned by SQLBINDCOL and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
SS1000	General error	Invalid parameter list detected.
S1002	Invalid column number	Column number is zero.
S1002	Invalid column number	Column number exceeds maximum value.
S1010	Function sequence error	Result set has already been started.

PL/I Example

```
%INCLUDE SPCPHD
   DCL ST   PTR;                       /* STATEMENT HANDLE    */
   DCL ID   FIXED BIN(15);             /* ID VALUE            */
   DCL CBID FIXED BIN(31);             /* LENGTH OF THE ID    */
   DCL FB00 FIXED BIN(31) INIT(0);     /* LITERAL VALUE       */
   DCL FB01 FIXED BIN(31) INIT(1);     /* LITERAL VALUE       */

CALL SQLBINDCOL(ST,      /* PASS THE STATEMENT HANDLE         */
   FB01,                 /* COLUMN NUMBER                     */
   SQL_C_DEFAULT,        /* REQUIRED C DATA TYPE              */
   SQL_SMALLINT,         /* USE A TWO-BYTE INTEGER            */
   FB00,                 /* PRECISION DOESN'T MATTER          */
   FB00,                 /* SCALE DOESN'T MATTER              */
   SQL_NO_NULLS,         /* DATA IS NEVER NULL                */
   ID,                   /* ID FIELD ADDRESS                  */
   CBID,                 /* ID LENGTH ADDRESS                 */
   'ID',                 /* COLUMN NAME STRING                */
   FB02);                /* COLUMN NAME LENGTH                */
RC = PLIRETV();          /* GET THE RETURN CODE               */
```

C Example

```
SDWORD        cbpanm;       /* length of the parameter number*/

RC=SQLBINDCOL(&st,          /* pass the statement handle     */
              1,            /* column number                 */
              SQL_C_DEFAULT,/* required C data type          */
              SQL_SMALLINT, /* use a two-byte integer        */
              0,            /* precision doesn't matter      */
              0,            /* scale doesn't matter          */
              SQL_NO_NULLS, /* data is never NULL            */
              &panm,        /* parameter number address      */
              &cbpanm,      /* parameter number length address*/
              "Number",     /* column name string            */
              SQL_NTS);     /* string is null-terminated     */
 if (rc != SQL_SUCCESS &&   /* not successful execution?     */
     rc != SQL_SUCCESS_WITH_INFO)/* not success with info?   */
```

COBOL Example

```
77 SQL-COLUMN-NUMBER            PIC S9(5) COMP VALUE 1.
77 SQL-C-DEFAULT                PIC S9(5) COMP VALUE IS 99.
77 SQL-SMALLINT                 PIC S9(5) COMP VALUE IS 5.
77 SQL-PRECISION                PIC S9(5) COMP VALUE 0.
77 SQL-SCALE                    PIC S9(5) COMP VALUE 0.
77 SQL-NO- NULLS                PIC S9(5) COMP VALUE IS 0.
77 SQL-COLUMN- LEN              PIC S9(5) COMP VALUE 1.
77 COLUMN-NULL-INDICATOR         PIC S9(5) COMP VALUE 0.
77 SQL-COLUMN-NAME              PIC X(10) VALUE 'ID'.
77 SQL-COLUMN-NAME-LEN          PIC S9(5) COMP VALUE 2.

CALL 'SDCPBC' USING STATEMENT-HANDLE
SQL-COLUMN-NUMBER
SQL-C-DEFAULT
SQL-SMALLINT
SQL-PRECISION
SQL-SCALE
SQL-NO-NULLS
ID-VALUE
COLUMN-NULL-INDICATOR
SQL-COLUMN-NAME
SQL-COLUMN-NAME-LEN.
```

SQLDESCRIBEPARAM (SDCPDP) Function

SQLDESCRIBEPARAM is used to obtain information about a parameter passed from the client to the host.

Note:
The client can pass parameters to the host using both parameter markers (?) and parameter literals. Both types of client parameters are treated the same way on the host.

Syntax

The general form for invocation of SQLDESCRIBEPARAM is:

```
CALL 'SDCPDP' USING STATEMENT-HANDLE
                    SQL-PARAM-NUMBER
                    SQL-DATA-TYPE
                    SQL-PRECISION
                    SQL-SCALE
                    SQL-NULLABLE-TYPE
                    SQL-PARAM-TYPE
                    SQL-PARAM-ADDRESS
                     SQL-PARAM-LENGTH
```

CALL Arguments

The SQLDESCRIBEPARAM function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Parameter number. All parameters including literals are numbered from the left starting at 1.00
3	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	SQL data type of parameter data. All DB2 SQL data types are supported except for graphic (DBCS) data.
4	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Precision of parameter. This value is primarily used for decimal and character string data.
5	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Scale of parameter. This value is primarily used for decimal data.
6	LONG	PICS9(5) COMP	FIXED BIN(31)	OUTPUT	Indicates whether or not parameter allows null values.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
7	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Indicates input/output type of parameter. Parameters can be used to send data to host (input), receive data from host (output), or both (input/output).
8	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Pointer to storage for parameter. Parameter can be accessed and updated at this storage location.
9	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Actual length of column. Length will be same as precision except for variable length fields (character and binary). For variable length fields, length will be current length. For all types, this field can contain SQL-NULL-DATA.

Return Values

SQLDESCRIBEPARAM always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_NO_DATA_FOUND	Indicates that the DDNAME, DSNAME or PDS member name is not valid because the dataset or member does not exist, or because the dataset is being held exclusively by some other address space.
SQL_ERROR	Indicates that the DDNAME, DSNAME or PDS member name is not valid because the dataset or member does not exist, or because the dataset is being held exclusively by some other address space.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR.
SQL_STILL_EXECUTING	Error condition. Another function is still executing.
SQL_NEED_DATA	Error Condition. The application needs to send parameter data values.

Diagnostics

When SQLDESCRIBEPARAM returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLERROR. The following table lists the SQLSTATE values commonly returned by SQLDESCRibeparam and explains each one in the context of

this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
SS1000	General error	Invalid parameter list detected.
S1000	General error	No room in buffer for column description.
S1004S1002	SQL data type out of range	SQL data type is invalid.
S1009	Invalid argument value	Column name address not set.
S1009	Invalid argument value	Column data length address not set.
S1009	Invalid argument value	Column data address is not set.
S1090	Invalid string or buffer length	Column name length is invalid.
S1090	Invalid string or buffer length	Column name length is not valid.
S1094	Invalid scale value	Decimal scale value is invalid.
S1099	Nullable type out of range	Nullable status value is invalid.
S1104S1002	Invalid precision value	Decimal precision value is invalid.
S1104S1002	Invalid precision value	String or binary precision value is invalid.
S1C00S	Driver not capable	Data type is not SQL_C_DEFAULT.

PL/I Example

```
DCL FB01   FIXED BIN(31) INIT(1);/* LITERAL VALUE              */
DCL SQDATY FIXED BIN(31);        /* SQL_DATA_TYPE             */
DCL SQPRSN FIXED BIN(31);        /* SQL_PRECISION             */
DCL SQNUTY FIXED BIN(31);        /* SQL_NULLABLE_TYPE         */
DCL SQSCAL FIXED BIN(31);        /* SQL_SCALE                 */
DCL SQPATY FIXED BIN(31);        /* SQL_PARAM_TYPE            */
DCL SQPAAD PTR;                  /* SQL_PARAM_ADDRESS         */
DCL SQPALN FIXED BIN(31);        /* SQL_PARAM_LENGTH          */


CALL SQLDESCRIBEPARAM(ST,        /* PASS THE STATEMENT HANDLE */
                FB01,            /* PARAMETER NUMBER          */
                SQDATY,          /* REQUIRED C DATA TYPE      */
                SQPRSN,          /* USE A TWO-BYTE INTEGER    */
                SQSCAL,
                SQNUTY,
                SQPATY,
                SQPAAD,
                SQPALN);
RC = PLIRETV();                  /* GET THE RETURN CODE       */
```

C Example

```
SDWORD         pasq;          /* parameter SQL type           */
UDWORD         papr;          /* parameter precision          */
SDWORD         pasc;          /* parameter scale              */
SDWORD         panl;          /* parameter nullable status    */
SDWORD         paty;          /* parameter type value         */
PTR            paad;          /* parameter address            */
SDWORD         paln;          /* parameter length             */

rc = SQLDescribeParam(&st,    /* pass the statement handle     */
                      i+1,    /* parameter number              */
                      &pasq,  /* address of the SQL type field */
                      &papr,  /* address of the precision field*/
                      &pasc,  /* address of the scale field    */
                      &panl,  /* address of the nullable status*/
                      &paty,  /* address of the parameter type */
                      &paad,  /* address of the parameter      */
                      &paln); /* address of parameter length   */
 if (rc != SQL_SUCCESS &&     /* not successful execution?     */
     rc != SQL_SUCCESS_WITH_INFO)  /* not success with info?  */
```

COBOL Example

```
77 STATEMENT-HANDLE          USAGE IS POINTER.
77 SQL-PARAM-NUMBER          PIC S9(5) COMP VALUE 1.
77 SQL-DATA-TYPE             PIC S9(5) COMP VALUE 0.
77 SQL-PRECISION             PIC S9(5) COMP VALUE 0.
77 SQL-SCALE                 PIC S9(5) COMP VALUE 0.
77 SQL-NULLABLE-TYPE         PIC S9(5) COMP VALUE 0.
77 SQL-PARAM-TYPE            PIC S9(5) COMP VALUE 0.
77 SQL-PARAM-ADDRESS         USAGE IS POINTER.
77 SQL-PARAM-LENGTH          PIC S9(5) COMP VALUE 0.

CALL 'SDCPDP' USING STATEMENT-HANDLE
SQL-PARAM-NUMBER
SQL-DATA-TYPE
SQL-PRECISION
SQL-SCALE
SQL-NULLABLE-TYPE
SQL-PARAM-TYPE
SQL-PARAM-ADDRESS
SQL-PARAM-LENGTH
```

SQLNUMPARAMS (SDCPNP) Function

SQLNUMPARAMS is used to obtain the number of parameters passed from the client to the host. This value will be zero or greater.

Note:
The client can pass parameters to the host using both parameter markers (?) and parameter literals. Both types of client parameters are treated the same way on the host.

Syntax

The general form for invocation of SQLNUMPARAMS is:

```
CALL 'SDCPNP' USING STATEMENT-HANDLE SQL-PARAM-COUNT
```

CALL Arguments

The SQLNUMPARAMS function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Number of RPC parameters passed to host from client. This argument is a pointer to a signed four-byte integer.

Return Values

SQLNUMPARAMS always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_NO_DATA_FOUND	Indicates that the DDNAME, DSNAME or PDS member name is not valid because the dataset or member does not exist, or because the dataset is being held exclusively by some other address space.

Return Value	Description
SQL_ERROR	A parameter validation error was found. The error will be logged to the wrap-around trace, and is available using the SQLERROR function.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR.
SQL_STILL_EXECUTING	Error condition. Another function is still executing.
SQL_NEED_DATA	Error Condition. The application needs to send parameter data values.

Diagnostics

When SQLNUMPARAMS returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLERROR. The following table lists the SQLSTATE values commonly returned by SQLNUMPARAMS and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
SS1000	General error	Invalid parameter list detected.
S1009S	Invalid argument value	Parameter count address not set.

PLI/I Example

```
DCL ST     PTR;                 /* STATEMENT HANDLE             */
DCL SQPACN FIXED BIN(31);       /* SQL_PARAMATER_COUNT          */

CALL SQLNUMPARAMS(ST,           /* PASS THE STATEMENT HANDLE    */
                  SQPACN);      /* PARAMETER COUNT              */
 RC = PLIRETV();                /* GET THE RETURN CODE          */
```

C Example

```
rc = SQLNumParams(&st, &pacn); /* get the number of parameters*/
 if (rc != SQL_SUCCESS &&      /* not successful execution?   */
     rc != SQL_SUCCESS_WITH_INFO) /* not success with info?   */
```

COBOL Example

```
77 STATEMENT-HANDLE          USAGE IS POINTER.
77 SQL-PARAM-COUNT           PIC S9(5) COMP VALUE 0.

CALL 'SDCPNP' USING STATEMENT-HANDLE SQL-PARAM-COUNT.
```

SQLRESETPARAM (SDCPRP) Function

SQLRESETPARAM is used to reset the length of a parameter passed from the client to the host.

> ***Note:***
> The client can pass parameters to the host using both parameter markers (?) and parameter literals. This routine can only be used with parameter markers.

In practice, this routine is really only used to change null parameters to non-null parameters and vice versa.

Syntax

The general form for invocation of SQLRESETPARAM is:

```
RETCODE SQLRESETPARAM (hstmt, rgbMsgText, cbMsgText, fOption)
```

CALL Arguments

The SQLRESETPARAM function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Parameter number. All parameters including literals are numbered from the left starting at 1.
3	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	New parameter length value. Parameter becomes null if new value is SQL-NULL-DATA. Parameter becomes NON-NULL if new value is not SQL-NULL-DATA.

Return Values

SQLRESETPARAM always sets a signed numeric return code value. Possible values are:

Return Value	Description
*SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_NO_DATA_FOUND	Indicates that the DDNAME, DSNAME or PDS member name is not valid because the dataset or member does not exist, or because the dataset is being held exclusively by some other address space.
SQL_ERROR	A parameter validation error was found. The error will be logged to the wrap-around trace, and is available using the SQLERROR function.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR.
SQL_STILL_EXECUTING	Error condition. Another function is still executing.
SQL_NEED_DATA	Error Condition. The application needs to send parameter data values.

*

Diagnostics

When SQLRESETPARAM returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLERROR. The following table lists the SQLSTATE values commonly returned by SQLRESETPARAM and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
SS1000	General error	Invalid parameter list detected.
S1000	General error	Trace message construction failed.
S1000	General error	Invalid option value detected.
S1009S	Invalid argument value	Message area address not set.
S1009	Invalid argument value	Trace message insertion failed.
S1090S1002	Invalid string or buffer length	Message area length is invalid.
S1090	Invalid string or buffer length	Message text length is not valid.

SQLRETURNSTATUS (SDCPRS) Function

SQLRETURNSTATUS is used to return status information to the client from an ODBC CALL RPC. The status data determines the return code from the SQLEXECDIRECT, SQLPREPARE, or SQLEXECUTE function that started the RPC. The client application can retrieve the status data (message and native code) by calling SQLERROR.

The actual return code returned to the ODBC application will be SQL-SUCCESS-WITH-INFO if this routine provides a positive return code and SQL-ERROR if this routine provides a negative return code. The return code provided by this routine is returned to the client application as the native error code (see the SQLERROR function description in the ODBC programmer's reference manual, not the SQLERROR function description here).

Syntax

The general form for invocation of SQLRETURNSTATUS is:

```
CALL 'SDCPAS' CONNECTION-HANDLE MESSAGE LENGTH ERROR-CODE
```

CALL Arguments

The SQLRETURNSTATUS function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	CHAR*	PICX(X)	CHAR(X)	INPUT	Address of message text to be returned. Text must be set before function is called.
3	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Length of message text to be returned. Value can be an actual length or can be specified as SQL-NTS if the message text is NULL-TERMINATED.
4	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Native error code. If value is negative, client return code will be SQL-ERROR. If value is positive, client return code will be SQL-SUCCESS-WITH-INFO. This field must not be zero.

Return Values

SQLRETURNSTATUS always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_NO_DATA_FOUND	Indicates that the DDNAME, DSNAME or PDS member name is not valid because the dataset or member does not exist, or because the dataset is being held exclusively by some other address space.
SQL_ERROR	A parameter validation error was found. The error will be logged to the wrap-around trace, and is available using the SQLERROR function.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR.
SQL_STILL_EXECUTING	Error condition. Another function is still executing.
SQL_NEED_DATA	Error Condition. The application needs to send parameter data values.

Diagnostics

When SQLRETURNSTATUS returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLERROR. The following table lists the SQLSTATE values commonly returned by SQLReturnStatus and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
SS1000	General error	Invalid parameter list detected.
S1000	General error	Invalid native error code detected.
S1009S	Invalid argument value	Message area address not set.
S1090S1002	Invalid string or buffer length	Message area length is invalid.
S1090	Invalid argument value	Message text length is not valid.

PL/I Example

```
DCL CN     PTR;                  /* CONNECTION HANDLE          */
DCL TRMG   CHAR(256);            /* TRACE MESSAGE AREA         */
DCL NAER   FIXED BIN(31);        /* NATIVE ERROR CODE AREA     */

CALL SQLRETURNSTATUS(CN,         /* GET SOME INFORMATION       */
                     TRMG,       /* TRACE MESSAGE AREA         */
                      SQL_NTS,   /* STRING IS NULL-TERMINATED  */
                      NAER);     /* PASS THE NATIVE ERROR CODE */
RC = PLIRETV();                  /* GET THE RETURN CODE        */
```

C Example

```
SDWORD         naer;             /* native error code area     */

rc = SQLReturnStatus(&cn,        /* get some information       */
                     trmg,       /* trace message area         */
                     SQL_NTS,    /* string is null-terminated  */
                     naer);      /* pass the native error code */
```

COBOL Example

```
77 CONNECTION-HANDLE          USAGE IS POINTER.
77 TRACE-MESSAGE-AREA         PIC X(256) VALUE IS SPACES.
77 SQL-NTS                    PIC S9(5) COMP VALUE IS -3.
77 NATIVE-ERROR-CODE-AREA     PIC S9(5) COMP VALUE 0.
```

SQLTHROW (SDCPTH) Function

SQLTHROW is used to send a row from the host ODBC call RPC back to the client.

Note:
One or more columns must be bound before this routine is called.

SQLTHROW is called for each row in the result set. When populating the result set, SQLTHROW is called with a parameter of SQL-THROW-ROW. Once the result set is populated and you wish to send the result to the client, SQLTHROW is called once again with a parameter of SQL-THROW-DONE.

Syntax

The general form for invocation of SQLTRHOW is:

```
CALL 'SDCPTH' STATEMENT-HANDLE MESSAGE LENGTH THROW-OPTION
```

CALL Arguments

The SQLTHROW function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Top of operation needed. This value is used to indicate that row is being provided or that result set is complete.

Return Values

SQLTHROW always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_NO_DATA_FOUND	Indicates that the DDNAME, DSNAME or PDS member name is not valid because the dataset or member does not exist, or because the dataset is being held exclusively by some other address space.

Return Value	Description
SQL_ERROR	A parameter validation error was found. The error will be logged to the wrap-around trace, and is available using the SQLERROR function.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR.
SQL_STILL_EXECUTING	Error condition. Another function is still executing.
SQL_NEED_DATA	Error Condition. The application needs to send parameter data values.

Diagnostics

When SQLTHROW returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLERROR**.** The following table lists the SQLSTATE values commonly returned by SQLThrow and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
SS1000	General error	Invalid parameter list detected.
S1000S	General error	Invalid option value detected.
S1000	General error	Null data specified for a non-null column.
S1000	General error	Maximum row count limit has been reached.
S1001	Memory allocation failure	Buffer space allocation failed.
S1010S1002	Function sequence error	Result set has already been completed.
24000	Invalid cursor state	No columns have been bound so far.

PL/I Example

```
DCL ST     PTR;                    /* STATEMENT HANDLE         */

CALL SQLTHROW(ST,                  /* PASS THE STATEMENT HANDLE*/
              SQL_THROW_ROW);      /* OPTION VALUE             */
RC = PLIRETV();                    /* GET THE RETURN CODE      */
```

C Example

```
rc = SQLThrow(&st,                 /* pass the statement handle*/
              SQL_THROW_ROW);      /* option value             */
 if (rc != SQL_SUCCESS &&          /* not successful execution?*/
     rc != SQL_SUCCESS_WITH_INFO) /* not success with info?   */
```

COBOL Sample

```
77 STATEMENT-HANDLE          USAGE IS POINTER.
77 SQL-THROW-DONE            PIC S9(5) COMP VALUE IS 2.

CALL 'SDCPTH' USING STATEMENT-HANDLE SQL-THROW-DONE.
```

IMS/APPC APIs

> ***Note:***
> In order to use these APIs, you need the IMS Transaction Server.

If you want to write an RPC to invoke a transaction, a sample is provided in the `NEON.SV040100.SAMP` dataset, member SDCOIMAP. This sample is a ODBC CALL RPC and is executed in the same fashion as the SHADOW_IMS RPC. (Please see the *Shadow Direct* User's Guide regarding ODBC CALL RPCs.)

A sample Visual Basic 4.0 program that demonstrates the use of the IMS Transaction Server for ODBC is shipped on the CD-ROM in the Shadowcd/samples/vb4/imsappc directory. This sample demonstrates the use of the Parts application that is shipped with IMS. It also allows for other transactions to be executed. Depending on your site, the calls to SHADOW_IMS may need to be modified.

The following API call functions, implemented by Shadow OS/390 Web Server and ShadowDirect Server, include those for APPC connections between Transaction Server for IMS and IMS, V4.1 and above, as well as APPC connections via REXX-language interfaces.

This section covers the following IMS/APPC APIs:

API Description	DIRECT	WEB	SEF	WEB/RX
IMS/APPC APIs				
To connect to APPC for IMS:	SQLAPPCCONNECT or SDCPAC	SWSAPPCCONNECT or SWCPAC	SDBAPCON	SWSAPCON
To disconnect from APPC for IMS:	SQLAPPCDISCONNECT or SDCPAD	SWSAPPCDISCONNECT or SWCPAD	SDBAPDIS	SWSAPDIS
To receive and wait from APPC for IMS:	SQLAPPCRECEIVE or SDCPAR	SWSAPPCRECEIVE or SWCPAR	SDBAPRCV	SWSAPRCV
To perform a send to APPC for IMS:	SQLAPPCSEND or SDCPAS	SWSAPPCSEND or SWCPAS	SDBAPSND	SWSAPSND

High-Level Language Interface SQLAPPCCONNECT (SDCPAC) or SWSAPPCCONNECT (SWCPAC) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPAC/SWCPAC**.

This API function call:

- Connects to IMS.
- Sends a transaction for IMS to execute and receive data from the transaction.

Syntax

The general form for invocation of SDCPAC/SWCPAC is:

```
set sql-appc-type-ims to true.
set atb-security-none to true.
CALL 'SDCPAC/SWCPAC' USING STATEMENT-HANDLE
                     SQL-APPC-TYPE
                     TP-NAME
                     TP-NAME-LENGTH
                     PARTNER-LU-NAME
                     ATB-SECURITY
                     CONVERSATION-ID
                     SEND-LENGTH
                     SEND-BUFFER
                     REQUESTED-LENGTH
                     RECEIVE-BUFFER
                     ATB-RETCODE
                      LOCAL LUNAME
                      MODE NAME
                      SYMBOLIC DESTINATION NAME
                      USERID
                      PASSWORD
                      PROFILE
                       ATB-DATA-RECVD
                       ATB-SEND-TYPE
                       ATB-SYNC-LEVEL
                      CONVERSATION TYPE
```

CALL Arguments

The SQLAPPCCONNECT/SWSAPPCONNECT (SWCPAC/SDCPAC) function arguments are described in the following table. All parameters are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Statement Handle. Currently ignored however, it must contain zeros.
2	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	IMS Call type: • **SWS-APPC-TYPE-IMS** for non-conversational IMS transactions. • **SWS-APPC-TYPE-IMSCONV** for conversational IMS transactions.
3	CHAR*	PIC X(64)	CHAR(64)	INPUT	IMS Transaction Name. A field containing the name of an IMS Transaction Code.
4	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	IMS Transaction Name length.
5	CHAR*	PIC X(17)	CHAR(17)	INPUT	Partner LU Name. The APPC LU Name of the IMS System.
6	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Security Type: • **CM_SECURITY_NONE** to omit access security information • **CM_SECURITY_SAME** to use the UserID and security profile of the user that initiated the request. • **CM_SECURITY_PROGRAM** to use the UserID and security profile supplied by the program.
7	CHAR*	PIC X(8)	CHAR(8)	OUTPUT	Conversation ID of the APPC Call.
8	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Length of data to be sent.
9	CHAR*	PIC X(2-32704)	CHAR (2-32704)	INPUT	Input buffer. The input buffer is comprised of a 2-byte prefix containing the binary length of the buffer data followed by the IMS Message Input Descriptor (MID) data.
10	LONG	PIC S9(9) COMP	FIXED BIN(31)	OUTPUT	Output buffer length.
11	CHAR*	PIC X(2-32704)	CHAR (2-32704)	OUTPUT	Output buffer. The output buffer is comprised of a 2-byte prefix containing the binary length of the buffer data followed by the IMS Message Output Descriptor (MOD) data.
12	LONG	PIC S9(9) COMP	FIXED BIN(31)	OUTPUT	Return Code.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
13	CHAR*	PIC X(8)	CHAR(8)	INPUT	Local LU Name. Specifies the name of a local LU from which the caller's allocate request is to originate. This is provides the ability to associate a transaction request with a particular LU name. ***Note:*** Optional parameter. If used, it requires the use of Mode Name, Symbolic Partner LU Name, User ID, Password and Security Profile be coded.
14	CHAR*	PIC X(8)	CHAR(8)	INPUT	Mode Name. Specifies the Mode name designating the network properties for the local LU Name. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Symbolic Partner LU Name, User ID, Password and Security Profile be coded.
15	CHAR*	PIC X(8)	CHAR(8)	INPUT	Symbolic Name. Specifies the symbolic name representing the IMS APPC LU Name, Mode Name, Transaction Name. The symbolic destination name must match that of an entry in the side information dataset. If you specify any pass any of the parameters (Local LU Name, Mode Name or Transaction Name), these will override the information retrieved and used to initialize the characteristics of the conversation. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Mode Name, User ID, Password and Security Profile.
16	CHAR*	PIC X(10)	CHAR(10)	INPUT	User ID. The Partner LU uses this value and the Password to validate the identity of the end-user that initiated the request. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Mode Name, Symbolic Partner LU Name, Password and Security Profile.
17	CHAR*	PIC X(10)	CHAR(10)	INPUT	Password. The Partner LU uses this value and the User ID to validate the identity of the end-user that initiated the request. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Mode Name, Symbolic Partner LU Name, User ID and Security Profile.
18	CHAR*	PIC X(10)	CHAR(10)	INPUT	Security Profile. Specifies additional security information that can be used to determine what partner programs the local program can access. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Mode Name, Symbolic Partner LU Name, User ID and Password.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
19	LONG	PIC S9(9) COMP	FIXED BIN(31)	OUTPUT	Data Type Received. Specifies the action taken to receive the output data. • **CM_NO_DATA_RECEIVED** specifies that no additional data is to be sent to the Partner LU, and the data can be buffered until a sufficient quantity is accumulated. • **CM_DATA_RECEIVED** specifies that no additional data is to be sent to the Partner LU, and the data is to be sent immediately. • **CM_COMPLETE_DATA_RECIEVED** specifies that the data is to be sent immediately along with a request for confirmation. • **CM_INCOMPLETE_DATA_RECEIVED** specifies that the data is to be sent immediately along with send control of the conversation. ***Note:*** Optional parameter. If used, it requires that placeholders be specified for Local LU Name, Mode Name, Symbolic Partner LU Name, User ID, Password and Security Profile.
20	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Data Type Sent. Specifies the action to take once the APPC Conversation is established. • **CM_BUFFER_DATA** specifies that no additional data is to be sent to the Partner LU, and the data can be buffered until a sufficient quantity is accumulated. • **CM_SEND_AND_FLUSH** specifies that no additional data is to be sent to the Partner LU, and the data is to be sent immediately. • **CM_SEND_AND_CONFIRM** specifies that the data is to be sent immediately along with a request for confirmation. • **CM_SEND_PREP_TO_RECEIVE** specifies that the data is to be sent immediately along with send control of the conversation. • **CM_SEND_AND_DEALLOCATE** specifies that the data is to be sent immediately along with a deallocation notification. ***Note:*** Optional parameter. If used, it requires that placeholders be specified for Local LU Name, Mode Name, Symbolic Partner LU Name, User ID, Password and Security Profile.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
21	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Synchronization Level. Specifies whether or not confirmation processing will be performed on this conversation. • **CM_NONE**— no confirmation processing is required for this conversation. • **CM_CONFIRM** — confirmation processing is required for this conversation. ***Note:*** Optional parameter. If used, it requires that placeholders be specified for Local LU Name, Mode Name, Symbolic Partner LU Name, User ID, Password, Security Profile and Data Type Sent.
22	LONG	PIC S9(9) COMP	FIXED BIN (31)	INPUT	Conversation Type.

Return Values

SQLAPPCCONNECT/SWSAPPCCONNECT always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. By using the SQLERROR/SWSERROR interface, you can obtain the error message pertinent to the error.

PL/I Example

```
%INCLUDE SPCPHD
%INCLUDE ATBCMPLI
%INCLUDE ATBPBPLI
      .
      .
      .
DCL  STMTHDL FIXED BIN(31);              /* Statement Handle */
DCL  RC FIXED BIN(31);                   /* RETURN CODE      */
DCL  SENDLEN FIXED BIN(31);              /* SEND LENGTH      */
DCL  RECVLEN FIXED BIN(31);              /* RECEIVE LENGTH   */
DCL  SENDBUF CHAR(82);                   /* SEND BUFFER      */
DCL  RECVBUF CHAR(82);                   /* RECEIVED BUFFER  */
DCL  LOCAL_LUNAME CHAR(8) INIT(' ');     /* LOCAL LUNAME     */
DCL  MODE_NAME CHAR(8) INIT(' ');        /* MODE NAME        */
DCL  SYMBOLIC_
     DESTINATION_NAME CHAR(8) INIT (' '); /* SYMBOLIC NAME    */
DCL  USERID CHAR(10) INIT(' ');          /* USERID           */
DCL  PASSWORD CHAR(10) INIT(' ');        /* PASSWORD         */
DCL  PROFILE CHAR(10) INIT(' ');         /* SECURITY PROFILE */
DCL  CONVERSATION_TYPE CHAR(10) INIT(' ');/* CONVERSATION TYPE*/
DCL  ATB_DATA_RECVD FIXED BIN(31);       /* RECEIVE DATA TYPE*/
DCL  ATB_SEND_TYPE FIXED BIN(31);        /* SEND DATA TYPE   */
DCL  ATB_SYNC_LEVEL FIXED BIN(31);       /* SYNC LEVEL       */


CALL SWSAPPCCONNECT(STMTHDL,       /* STATEMENT HANDLE       */
                    SWS_APPC_TYPE_IMS,
                    TP_NAME,
                    TP_NAME_LENGTH,
                    PARTNER_LU_NAME,
                    ATB_SECURITY_NONE,
                    CONVERSATION_ID,
                    SENDLEN,
                    SENDBUF,
                    RECVLEN,
                    RECVBUF,
                    RC,
                    LOCAL_LUNAME,
                    MODE_NAME,
                    SYMBOLIC_DESTINATION_NAME,
                    USERID,
                    PASSWORD,
                    PROFILE,
                    RECVTYPE,
                    SENDTYPE,
                    SYNCTYPE
                    CONVERSATION_TYPE,
                    CM_SEND_AND_FLUSH,
                    CM_NONE.
 RC = PLIRETV();                   /* GET RETURN CODE        */
 IF RC ^= SWS_SUCCESS THEN         /* EXIT PROGRAM IF BAD RC */
   EXIT;
```

C Example

```
#include "sccphd.h"                /* Neon headers           */
#include "atbcmc.h"                /* CPI Communications     */
#include "atbpbc.h"                /* LU6.2                  */
     .
long RC;                           /* return code            */
long stmtHDL;                      /* statement handle       */
long recvtype;                     /* data type received     */
long sendtype;                     /* data type sent         */
long synctype;                     /* syncronization level   */
long sendlen;                      /* input buffer length    */
long recvlen;                      /* output buffer length   */
long tp_name_length;               /* tp name length         */
char sendbuf[82]                   /* input buffer area      */
char recvbuf[82]                   /* output buffer          */
char tp_name[64]                   /* tp name                */
char local_luname[8]= '     '      /* local luname           */
char mode_name[8]   = '     '      /* mode name              */
char symbolic_
destination_ name[8]= '     '      /* symbolic name          */
char userid[10]     = '     '      /* user id                */
char password[10]   = '     '      /* password               */
char profile[10]    = '     '      /* security profile       */
char conversation_type[10]         /* conversation type      */
char conversation_id[8]            /* conversation id        */
char partner_lu_name[17]           /* partner lu name        */

CALL SWSAPPCCONNECT(stmtHDL,  /* statement handle            */
     SWS_APPC_TYPE_IMS
         tp_namE
         tp_name_length
         partner_lu_name
         ATB_SECURITY_NONE
         conversation_id
         sendlen
         sendbuf
         recvlen
         recvbuf
         RC
         local_luname
         mode_name
         symbolic_destination_name
         userid
         password
         profile
         recvtype
         sendtype
         synctype
         connection_type
         CM_SEND_AND_FLUSH
         CM_NONE.
    if (rc ^= SWS_SUCCESS)
        return rc;
```

COBOL Example

```
COPY SBCPHD.                          Neon Copybook
COPY ATBCMCOB.                        CPI COMMUNICATIONS COPYBOOK
COPY ATBPBCOB.                        LU6.2 COPYBOOK
       .
77 STATEMENT-HANDLE                 USAGE IS POINTER.
77 LOCAL-LUNAME                     PIC X(8) VALUE IS SPACES.
77 MODE-NAME                        PIC X(8) VALUE IS SPACES.
77 SYMBOLIC-DESTINATION-NAME        PIC X(8) VALUE IS SPACES.
77 USERID                           PIC X(10) VALUE IS SPACES.
77 PASSWORD                         PIC X(10) VALUE IS SPACES.
77 PROFILE                          PIC X(10) VALUE IS SPACES.
77 CONVERSATION-TYPE                PIC X(10) VALUE IS SPACES.
01 SEND-BUFFER.
  05 SEND-BUFFER-LENGTH             PIC 9(4) COMP-4.
  05 SEND-BUFFER-CONTENTS           PIC X(100) VALUE IS SPACES.
  05 SEND-EXTRA-AREA                PIC X(10) COMP-4.
  05 SEND-DATA-TYPE                 PIC 9(9) COMP-4.
  05 SEND-LEVEL-SYNC                PIC 9(9) COMP-4.
01 RECEIVE-BUFFER.
  05 RECEIVE-BUFFER-LENGTH          PIC 9(4) COMP-4.
  05 RECEOVE-BUFFER-CONTENTS        PIC X(100) VALUE IS SPACES.
  05 RECEIVE-EXTRA-AREA             PIC X(10).
  05 RECEIVE-DATA-TYPE              PIC 9(9) COMP-4.



       .
SET CM-NONE           TO TRUE.
SET ATB-SECURITY-NONE TO TRUE.
SET SWS-APPC-TYPE-IMS TO TRUE.
MOVE 'PART'           TO TP-NAME.
MOVE 4                TO TP-NAME-LENGTH.
MOVE 'P390.P392AIMS'  TO PARTNER-LU-NAME.
CALL 'SDCPAC' USING STATEMENT-HANDLE
                    SWS-APPC-TYPE
                    TP-NAME
                    TP-NAME-LENGTH
                    PARTNER-LU-NAME
                    ATB-SECURITY
                    CONVERSATION-ID
                    SEND-LENGTH
                    SEND-BUFFER
                    REQUESTED-LENGTH
                    RECEIVE-BUFFER
                    ATB-RETCODE
                    LOCAL-LUNAME
                    MODE-NAME
                    SYMBOLIC-DESTINATION-NAME
                    USERID
                    PASSWORD
                    PROFILE
                    ATB-DATA-RECVD
```

```
                        ATB-SEND-TYPE
                        ATB-SYNC-LEVEL.
                        CONVERSATION-TYPE
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS
    GOBACK.
```

SDBAPCON/SWSAPCON Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreter.
✔	High-level language interface.

The REXX-language SDBAPCON/ SWSAPCON built-in function can be used for APPC connections between Shadow OS/390 Web Server and Shadow Direct for IMS and IMS v4.1 and above. A call to this function

- Connects to IMS.
- Sends a transaction for IMS to execute and receive the data from the executed transaction.

The data received is in the format of the Message Output Descriptor (MOD) used by the application program that processed the transaction.

Upon return from the call, the following REXX variables are populated with data:

REXX Variable	Description
APPC.STMTHDL	The statement handle for the interface call.
APPC.CONVID	The Conversation ID for the IMS APPC conversation.
APPC.OUTBUFF.0	The length of the data contained within the APPC.OUTBUFF.1 REXX variable.
APPC.OUTBUFF.1	The returned Message Output Descriptor data. The output message contains the MOD data as well as a two byte length prefix. This variable contains the data, including any supplied trailing blanks.
APPC.RETCODE	The APPC Interface return code.
APPC.DATATYPE	The returned datatype.

Syntax

The general form for a REXX-language invocation of SDBAPCON/SWSAPCON is:

```
rc = SDBAPCON/SWSAPCON("Connection Type", ,
                       "Transaction Name", ,
                       "Transaction Name Length", ,
                       "Partner LU Name", ,
                       "Security Type", ,
                       "Transaction Data", ,
                       "Local LU Name", ,
                       "Mode Name", ,
                       "Symbolic Partner LU Name", ,
                       "Userid", ,
                       "Password", ,
                       "Security Profile", ,
                       "Data Type Sent", ,
                       "Message Length Sent", ,
                       "Synchronization Level"
```

Valid Arguments

Connection Type	Specifies the type of IMS transaction to execute: • **IMS** for IMS Non-conversational transaction. • **IMS CONV** for IMS Conversational transaction. ***Note:*** Required parameter.
Transaction Name	Specifies the IMS Transaction Code ***Note:*** Required parameter.
Transaction Name Length	Specifies the length of the IMS Transaction Code ***Note:*** Required parameter.
Partner LU Name	Specifies the APPC LU Name for the IMS system ***Note:*** Required parameter.
Security Type	Specifies the type of security in use. • **NONE** specifies that no access security information is to be passed on the APPC Allocation request. • **SAME** specifies to use the userid and security profile (if present) from the allocation request that initiated the local program. The password (if present) is not used; instead, the userid is indicated as "already been validated". If the allocation request that initiated the local program contained no access security information, then access security information is omitted on this allocation request. • **PROGRAM** specifies to use the security information, provided by the local program, on the API call. The local program provides the information by means of the Userid, Password and Security Profile parameters. These values are passed exactly as specified (without folding the characters to upper case). ***Note:*** Required parameter.
Transaction Data	Specifies any data required by the application program in order to process the transaction. This would be data in the format of the Message Input Descriptor (without the LLZZ prefix). ***Note:*** Required parameter.

Local LU Name	Specifies the name of a local LU from which the caller's allocate request is to originate. This is provides the ability to associate a transaction request with a particular LU name. ***Note:*** Optional parameter. If used, it requires the use of Mode Name, Symbolic Partner LU Name, User ID, Password and Security Profile be coded.
Mode Name	Specifies the Mode name designating the network properties for the local LU Name. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Symbolic Partner LU Name, User ID, Password and Security Profile be coded.
Symbolic Partner LU Name	Specifies the symbolic name representing the IMS APPC LU Name, Mode Name, Transaction Name. The symbolic destination name must match that of an entry in the side information dataset. If you specify any pass any of the parameters (Local LU Name, Mode Name or Transaction Name), these will override the information retrieved and used to initialize the characteristics of the conversation. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Mode Name, User ID, Password and Security Profile.
User ID	The Partner LU uses this value and the Password to validate the identity of the end-user that initiated the request. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Mode Name, Symbolic Partner LU Name, Password and Security Profile.
Password	The Partner LU uses this value and the User ID to validate the identity of the end-user that initiated the request. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Mode Name, Symbolic Partner LU Name, User ID and Security Profile.
Security Profile	Specifies additional security information that can be used to determine what partner programs the local program can access. ***Note:*** Optional parameter. If used, it requires the use of Local LU Name, Mode Name, Symbolic Partner LU Name, User ID and Password.
Data Type Sent	Specifies the action to take once the APPC Conversation is established. • **BUFFER** specifies that no additional data is to be sent to the Partner LU, and the data can be buffered until a sufficient quantity is accumulated. • **SENDFLSH** specifies that no additional data is to be sent to the Partner LU, and the data is to be sent immediately. • **SENDCONF** specifies that the data is to be sent immediately along with a request for confirmation. • **SENDPREP** specifies that the data is to be sent immediately along with send control of the conversation. • **SENDDEAL** specifies that the data is to be sent immediately along with a deallocation notification. ***Note:*** Optional parameter. If used, it requires that placeholders be specified for Local LU Name, Mode Name, Symbolic Partner LU Name, User ID, Password and Security Profile.
Synchronization Level	Specifies whether or not confirmation processing will be performed on this conversation. • **NONE** — no confirmation processing is required for this conversation. • **CONFIRM** — confirmation processing is required for this conversation. ***Note:*** Optional parameter. If used, it requires that placeholders be specified for Local LU Name, Mode Name, Symbolic Partner LU Name, User ID, Password, Security Profile and Data Type Sent.

SDBAPCON/SWSAPCON Examples

```
/*-------------------------------------------------------*/
/* initialize some system values                         */
/*-------------------------------------------------------*/

 address SWSSEND
 imsappc = 'P390.P392AIMS'
 imstran = 'NEONDISP'
 parms   = ''

 /*-------------------------------------------------------*/
 /* execute the ims transaction                           */
 /*-------------------------------------------------------*/

 rc = sdbapcon/swsapcon
('ims',imstran,length(imstran),imsappc,'NONE'
,parms)

 /*-------------------------------------------------------*/
 /* parse the output into usable variables                */
 /*-------------------------------------------------------*/

 pars.msg      = substr(APPC.OUTBUFF.1,1,79)
 pars.page     = substr(APPC.OUTBUFF.1,80,2)
 pars.index    = substr(APPC.OUTBUFF.1,82,2)
 pars.scroll   = substr(APPC.OUTBUFF.1,84,150)

 pars.area     = substr(APPC.OUTBUFF.1,234,380)
 pars.len      = 380
 pars.data     = ''
 do i = 1 to 10
   pars.part.i = substr(pars.area,4,15)
   pars.desc.i = substr(pars.area,19,20)
   pars.len    = pars.len - 38
   pars.area   = substr(pars.area,39,pars.len)
   pars.data   = pars.data||' '||pars.part.i||pars.desc.i
 end
```

High-Level Language Interface SQLAPPCDISCONNECT (SDCPAD) or SWSAPPCDISCONNECT (SWCPAD) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPAD/SWCPAD**.

This call disconnects from IMS.

Syntax

The general form for invocation of SDCPAD/SWCPAD is

```
CALL 'SDCPAD/SWCPAD' USING STATEMENT-HANDLE
                           SQL-APPC-TYPE
                           CONVERSATION-ID
                           ATB-RETCODE
```

CALL Arguments

The SQLAPPCDISCONNECT/SWSAPPCDISCONNECT (SDCPAC/SWS-PAC) function arguments are described in the table which follows. All parameters are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Statement Handle. Currently ignored however, it must contain zeros.
2	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	IMS Call type: • **SWS-APPC-TYPE-IMS** for non-conversational IMS transactions. • **SWS-APPC-TYPE-IMSCONV** for conversational IMS transactions.
3	CHAR*	PIC X(8)	CHAR(8)	OUTPUT	Conversation ID of the APPC Call.
4	LONG	PIC S9(9) COMP	FIXED BIN(31)	OUTPUT	Return Code.

Return Values

SWSAPPCDISCONNECT always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. By using the SQLERROR/SWSERROR interface, you can obtain the error message pertinent to the error.

PL/I Example

```
%INCLUDE SPCPHD
%INCLUDE ATBCMPLI
%INCLUDE ATBPBPLI
       .
       .
       .
 DCL  STMTHDL      FIXED BIN(31);   /* Statement Handle       */
 DCL  RC           FIXED BIN(31);   /* RETURN CODE            */
 DCL  SENDLEN      FIXED BIN(31);   /* SEND LENGTH            */
 DCL  RECVLEN      FIXED BIN(31);   /* RECEIVE LENGTH         */
 DCL  SENDBUF      CHAR(82)         /* SEND BUFFER            */
 DCL  RECVBUF      CHAR(82)         /* RECEIVED BUFFER        */
 DCL (FILL_1       initial('        ')
      FILL_2       initial('        ')
      FILL_3       initial('        ')) char(8);
 DCL (FILL_4       initial('        ')
      FILL_5       initial('        ')
      FILL_6       initial('        ')) char(10);

CALL SWSAPPCDISCONNECT(STMTHDL,     /* STATEMENT HANDLE       */
                       SWS_APPC_TYPE_IMS,
                       CONVERSATION_ID,
                       RC.
RC = PLIRETV();                     /* GET RETURN CODE        */
  IF RC ^= SWS_SUCCESS THEN         /* EXIT PROGRAM IF BAD RC */
    EXIT;
```

C Example

```
# include "sccphd.h"            /* Neon headers              */
# include "atbcmc.h"            /* CPI Communications        */
# include "atbpbc.h"            /* LU6.2                     */
     .
     .
     .
 long RC;                       /* return code               */
 long stmtHDL;                  /* statement handle          */
 long recvtype;                 /* data received type        */
 long sendlen;                  /* input buffer length       */
 long recvlen;                  /* output buffer length      */
 long tp_name_length;           /* tp name length            */
 char sendbuf[82]               /* input buffer area         */
 char recvbuf[82]               /* output buffer             */
 char tp_name[64]               /* tp name                   */
 char fill_1[] = '      '       /* eight byte filler field   */
 char fill_2[] = '      '       /* eight byte filler field   */
 char fill_3[] = '      '       /* eight byte filler field   */
 char fill_4[] = '      '       /* ten byte filler field     */
 char fill_5[] = '      '       /* ten byte filler field     */
 char fill_6[] = '      '       /* ten byte filler field     */
 char conversation_id[8]        /* conversation id           */
 char partner_lu_name[17]       /* partner lu name           */

 CALL SWSAPPCDISCONNECT(stmtHDL, /* statement handle         */
        SWS_APPC_TYPE_IMS
        conversation_id
        RC.
 if (rc ^= SWS_SUCCESS)
        return rc;
```

COBOL Example

```
COPY SBCPHD.                          Neon Copybook
COPY ATBCMCOB.                        CPI COMMUNICATIONS COPYBOOK
COPY ATBPBCOB.                        LU6.2 COPYBOOK
       .
       .
       .
       .
 77 STATEMENT-HANDLE        USAGE IS POINTER.
 77 FILLER-PARMLIST-01      PIC X(8) VALUE IS SPACES.
 77 FILLER-PARMLIST-02      PIC X(8) VALUE IS SPACES.
 77 FILLER-PARMLIST-03      PIC X(8) VALUE IS SPACES.
 77 FILLER-PARMLIST-04      PIC X(10) VALUE IS SPACES.
 77 FILLER-PARMLIST-05      PIC X(10) VALUE IS SPACES.
 77 FILLER-PARMLIST-06      PIC X(10) VALUE IS SPACES.
 01 SEND-BUFFER.
 05 SEND-BUFFER-LENGTH      PIC 9(4) COMP-4.
 05 SEND-BUFFER-CONTENTS    PIC X(100) VALUE IS SPACES.
 05 SEND-EXTRA-AREA         PIC X(10).
 01 RECEIVE-BUFFER.
 05 RECEIVE-BUFFER-LENGTH   PIC 9(4) COMP-4.
 05 RECEIVE-BUFFER-CONTENTS PIC X(80).
 05 RECEIVE-EXTRA-AREA      PIC X(10).
       .
       .
       .
 SET CM-NONE           TO TRUE.
 SET ATB-SECURITY-NONE TO TRUE.
 SET SWS-APPC-TYPE-IMS TO TRUE.
 MOVE 'PART'           TO TP-NAME.
 MOVE 4                TO TP-NAME-LENGTH.
 MOVE 'P390.P392AIMS'  TO PARTNER-LU-NAME.
 CALL 'SDCPAD' USING STATEMENT-HANDLE
                     SWS-APPC-TYPE
                     CONVERSATION-ID
                     ATB-RETCODE.

 MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
 IF NOT SWS-SUCCESS
     GOBACK.
```

SDBAPDIS/SWSAPDIS Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level language interface available.

The REXX-language SDBAPDIS/SWSAPDIS built-in function can be used to drop APPC connections between the Shadow OS/390 Web Server for IMS and IMS v4.1 and above. A call to this function disconnects IMS.

Upon return from the call, the following REXX variables are used:

REXX Variable	Description
APPC.STMTHDL	The statement handle for the interface call.
APPC.RETCODE	The APPC Interface return code.

Syntax

The general form for a REXX-language invocation of SDBAPDIS/SWSAPDIS is:

```
rc = SDBAPDIS/SWSAPDIS(“Connection Type”)
```

SDBAPDIS/SWSAPDIS Example

```
/*-------------------------------------------------------------*/
/* disconnect the ims transaction                              */
/*-------------------------------------------------------------*/

rc = sdbapdis/swsapdis('ims')
     end
```

or

```
/*-------------------------------------------------------------*/
/* disconnect the ims transaction                              */
/*-------------------------------------------------------------*/

rc = sdbapdis/swsapdis('imsconv')
     end
```

High-Level Language Interface SQLAPPCRECEIVE (SDCPAR) or SWSAPPCRECEIVE (SWCPAR) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPAR/SWCPAR.**

Use this call to do a receive and wait from APPC for IMS information.

Syntax

The general form for invocation of SDCPAR/SWCPAR is

```
CALL 'SDCPAR/SWCPAR' USING STATEMENT-HANDLE
                           SQL-APPC-TYPE
                           CONVERSATION-ID
                           REQUESTED-LENGTH
                           RECEIVE-BUFFER
                           ATB-RETCODE
                           PARTNER-LU-NAME
                           ATB-DATA-RECVD
                           TPNAME-FOR-TESTIMS
                           TPNAME-LENGTH-FOR-TESTIMS
                           CONVERSATION-TYPE
```

CALL Arguments

The SQLAPPCRECEIVE/SWSAPPCRECEIVE (SWCPAR/SDCPAR) function arguments are described in the table which follows. All parameters are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Statement Handle. Currently ignored however, it must contain zeros.
2	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	IMS Call type: • **SWS-APPC-TYPE-IMS** for non-conversational IMS transactions. • **SWS-APPC-TYPE-IMSCONV** for conversational IMS transactions.
3	CHAR*	PIC X(8)	CHAR(8)	OUTPUT	Conversation ID of the APPC Call.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
4	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT/ OUTPUT	Output buffer length.
5	CHAR*	PIC X(2-32704)	CHAR (2-32704)	OUTPUT	Output buffer. The output buffer is comprised of a 2 byte prefix containing the binary length of the buffer data followed by the IMS Message Output Descriptor (MOD) data.
6	LONG	PIC S9(9) COMP	FIXED BIN(31)	OUTPUT	Return Code.
7	CHAR*	PIC X(17)	CHAR(17)	INPUT	Partner LU Name. The APPC LU Name of the IMS System.
8	LONG	PIC S9(9) COMP	FIXED BIN(31)	OUTPUT	Data Type Received. Specifies the action taken to receive the output data: • **CM_NO_DATA_RECEIVED** specifies that no additional data is to be sent to the Partner LU, and the data can be buffered until a sufficient quantity is accumulated. • **CM_DATA_RECEIVED** specifies that no additional data is to be sent to the Partner LU, and the data is to be sent immediately. • **CM_COMPLETE_DATA_RECIEVED** specifies that the data is to be sent immediately along with a request for confirmation. • **CM_INCOMPLETE_DATA_RECEIVED** specifies that the data is to be sent immediately along with send control of the conversation. ***Note:*** Optional parameter. If used, it requires that placeholders be specified for Local LU Name, Mode Name, Symbolic Partner LU Name, User ID, Password and Security Profile.
9	CHAR*	PIC X(8)	CHAR(8)	INPUT	TPNAME for TESTIMS.
10	LONG	PIC S9(9) COMP	FIXED BIN (31)	OUTPUT	TPNAME length for TESTIMS.
11	LONG	S9 (4) COMP	FIXED BIN (31)	INPUT	Conversation type.

Return Values

SQLAPPCRECEIVE/SWSAPPCRECEIVE always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.

Return Value	Description
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. By using the SQLERROR/SWSERROR interface, you can obtain the error message pertinent to the error.

PL/I Example

```
%INCLUDE SPCPHD
%INCLUDE ATBCMPLI
%INCLUDE ATBPBPLI
       .
       .
       .
       .
 DCL  STMTHDL FIXED BIN(31);              /* Statement Handle */
 DCL  RC FIXED BIN(31);                   /* RETURN CODE      */
 DCL  SENDLEN FIXED BIN(31);              /* SEND LENGTH      */
 DCL  RECVLEN FIXED BIN(31);              /* RECEIVE LENGTH   */
 DCL  SENDBUF CHAR(82)                    /* SEND BUFFER      */
 DCL  RECVBUF CHAR(82)                    /* RECEIVED BUFFER  */
 DCL  LOCAL_LUNAME CHAR(8) INIT(' ');     /* LOCAL LUNAME     */
 DCL  MODE_NAME CHAR(8) INIT(' ');        /* MODE NAME        */
 DCL  SYMBOLIC_
      DESTINATION_NAME CHAR(8) INIT (' '); /* SYMBOLIC NAME    */
 DCL  USERID CHAR(10) INIT(' ');          /* USERID           */
 DCL  PASSWORD CHAR(10) INIT(' ');        /* PASSWORD         */
 DCL  PROFILE CHAR(10) INIT(' ');         /* SECURITY PROFILE */
 DCL  TPNAME_FOR_TESTIMS
 DCL  TPNAME_LENGTH_FOR_TESTIMS
 DCL  CONVERSATION_TYPE CHAR(10) INIT(' ');/* CONVERSATION TYPE*/

 CALL SWSAPPCRECEIVE(STMTHDL,       /* STATEMENT HANDLE       */
                     SWS_APPC_TYPE_IMS,
                     PARTNER_LU_NAME,
                     CONVERSATION_ID,
                     RECVLEN,
                     RECVBUF,
                     RC,
                     RECVTYPE,
                     TPNAME_FOR_TESTIMS,
                     TPNAME_LENGTH_FOR_TESTIMS,
                     CONVERSATION_TYPE.


  RC = PLIRETV();                   /* GET RETURN CODE        */
  IF RC ^= SWS_SUCCESS THEN         /* EXIT PROGRAM IF BAD RC */
    EXIT;
```

C Example

```
#include "sccphd.h"                /* Neon headers           */
#include "atbcmc.h"                /* CPI Communications     */
#include "atbpbc.h"                /* LU6.2                  */

    .
    .
long RC;                           /* return code            */
long stmtHDL;                      /* statement handle       */
long recvtype;                     /* data received type     */
long sendlen;                      /* input buffer length    */
long recvlen;                      /* output buffer length   */
long tp_name_length;               /* tp name length         */
char sendbuf[82]                   /* input buffer area      */
char recvbuf[82]                   /* output buffer          */
char tp_name[64]                   /* tp name                */
char local_luname[8]= '     '      /* local luname           */
char mode_name[8]   = '     '      /* mode name              */
char symbolic_
destination_ name[8]= '     '      /* symbolic name          */
char userid[10]     = '     '      /* user id                */
char password[10]   = '     '      /* password               */
char profile[10]    = '     '      /* security profile       */
char tpname_for_testims            /* ten byte filler field  */
char tpname_length_for_testims     /* ten byte filler field  */
char conversation_type[10]         /* ten byte filler field  */
char conversation_id[8]            /* conversation id*/
char partner_lu_name[17]           /* partner lu name        */

CALL SWSAPPCRECEIVE(stmtHDL, /* statement handle             */
     SWS_APPC_TYPE_IMS
         partner_lu_name
         conversation_id
         recvlen
         recvbuf
         tpname_for_testims
         tpname_length_for_testims
         conversation_type
         RC.

   if (rc ^= SWS_SUCCESS)
        return rc;
```

COBOL Example

```
COPY SBCPHD.                              Neon Copybook
 COPY ATBCMCOB.                            CPI COMMUNICATIONS COPYBOOK
 COPY ATBPBCOB.                            LU6.2 COPYBOOK
        .
        .
        .
        .
 77 STATEMENT-HANDLE                    USAGE IS POINTER.
 77 LOCAL-LUNAME                        PIC X(8) VALUE IS SPACES.
 77 MODE-NAME                           PIC X(8) VALUE IS SPACES.
 77 SYMBOLIC-DESTINATION-NAME           PIC X(8) VALUE IS SPACES.
 77 USERID                              PIC X(10) VALUE IS SPACES.
 77 PASSWORD                            PIC X(10) VALUE IS SPACES.
 77 PROFILE                             PIC X(10) VALUE IS SPACES.
 77 TPNAME-FOR-TESTIMS                  PIC X(10) VALUE IS SPACES.

 77 TPNAME-LENGTH-FOR-TESTIMS           PIC X(10) VALUE IS SPACES.
 77 CONVERSATION-TYPE                   PIC X(10) VALUE IS SPACES.
 01 SEND-BUFFER.
 05 SEND-BUFFER-LENGTH                  PIC 9(4) COMP-4.
 05 SEND-BUFFER-CONTENTS                PIC X(100) VALUE IS SPACES.
 05 SEND-EXTRA-AREA                     PIC X(10).
 01 RECEIVE-BUFFER.
 05 RECEIVE-BUFFER-LENGTH               PIC 9(4) COMP-4.
 05 RECEIVE-BUFFER-CONTENTS             PIC X(80).
 05 RECEIVE-EXTRA-AREA                  PIC X(10).
        .
        .
        .
 SET CM-NONE           TO TRUE.
 SET ATB-SECURITY-NONE TO TRUE.
 SET SWS-APPC-TYPE-IMS TO TRUE.
 MOVE 'PART'           TO TP-NAME.
 MOVE 4                TO TP-NAME-LENGTH.
 MOVE 'P390.P392AIMS'  TO PARTNER-LU-NAME.
 CALL 'SDCPAR' USING STATEMENT-HANDLE
                     SWS-APPC-TYPE
                     PARTNER-LU-NAME
                     CONVERSATION-ID
                     REQUESTED-LENGTH
                     RECEIVE-BUFFER
                     ATB-RETCODE
                     TPNAME-FOR-TESTIMS
                     TPNAME-LENGTH-FOR-TESTIMS
                     CONNECTION-TYPE.

 MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
 IF NOT SWS-SUCCESS
        GOBACK.
```

SDBAPRCV/ SWSAPRCV Function

✔	Can be used in Shadow/REXX.
✔	Can be used from Other REXX interpreters.
✔	HLL entry point name is **SDCPAR/SWCPAR**.

The REXX-language SDBAPRCV/SWSAPRCV built-in function can be used to continue receiving message output from an IMS transaction (i.e. multisegment output messages). Once you receive a non-zero return code, the IMS conversation is deallocated and the transaction is complete.

In the case of conversational IMS transactions, the non-zero return code will indicate that their is no more data to receive.

Upon return from the call, the following REXX variables are populated with data:

REXX Variable	Description
APPC.STMTHDL	The statement handle for the interface call.
APPC.CONVID	The Conversation ID for the IMS APPC conversation.
APPC.OUTBUFF.0	The length of the data contained within the APPC.OUTBUFF.1 REXX variable.
APPC.OUTBUFF.1	The returned Message Output Descriptor data. The output message contains the MOD data as well as a two byte length prefix. This variable contains the data, including any supplied trailing blanks.
APPC.RETCODE	The APPC Interface return code.
APPC.DATATYPE	The returned datatype.

Syntax

The general form for a REXX-language invocation of SDBAPRCV/SWSAPRCV is:

```
rc = SDBAPRCV/SWSAPRCV("Connection Type", ,
                       "Conversation ID", ,
                       "Partner LU Name")
```

Valid Arguments

Connection Type	Specifies the type of IMS transaction to execute: • **IMS** for IMS Non-conversational transaction. • **IMSCONV** for IMS Conversational transaction. ***Note:*** Required parameter.
Conversation ID	Specifies the IMS Conversation ID. ***Note:*** Required parameter.
Partner LU Name	Specifies the APPC LU Name for the IMS system ***Note:*** Optional parameter.

SDBAPRCV/SWSAPRCV Example

```
/*-----------------------------------------------------*/
/* initialize some system values                       */
/*-----------------------------------------------------*/

address SWSSEND
imsappc = 'P390.P392AIMS'
imstran = 'NEONDISP'
parms   = ''

/*-----------------------------------------------------*/
/* Retrieve more data from the IMS transaction         */
/*-----------------------------------------------------*/

rc = SDBAPRCV/SWSAPRCV('ims',APPC.CONVID,imsappc)

/*-----------------------------------------------------*/
/* parse the output into usable variables              */
/*-----------------------------------------------------*/

pars.msg      = substr(APPC.OUTBUFF.1,1,79)
pars.page     = substr(APPC.OUTBUFF.1,80,2)
pars.index    = substr(APPC.OUTBUFF.1,82,2)
pars.scroll   = substr(APPC.OUTBUFF.1,84,150)
pars.area     = substr(APPC.OUTBUFF.1,234,380)
pars.len      = 380
pars.data     = ''
do i          = 1 to 10
   pars.part.i= substr(pars.area,4,15)
   pars.desc.i= substr(pars.area,19,20)
   pars.len   = pars.len - 38
   pars.area  = substr(pars.area,39,pars.len)
   pars.data  = pars.data||' '||pars.part.i||pars.desc.i

end
```

High-Level Language Interface SQLAPPCSEND (SDCPAS) or SWSAPPCSEND (SWCPAS) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from Other REXX interpreters.
✔	HLL entry point name is **SDCPAS/SWCPAS.**

This call performs a send to APPC for IMS.

Syntax

The general form for invocation of SDCPAS/SWCPAS is

```
CALL 'SDCPAS' USING STATEMENT-HANDLE
                    SQL-APPC-TYPE
                    CONVERSATION-ID
                    SEND-LENGTH
                    SEND-BUFFER
                    PARTNER-LU-NAME
                    ATB-SEND-TYPE
                    ATB-RETCODE.
```

CALL Arguments

The SQLAPPCSEND/SWSAPPCSEND (SWCPAS/SDCPAS) function arguments are described in the table which follows. All parameters are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Statement Handle. Currently ignored however, it must contain zeros.
2	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	IMS Call type: • **SWS-APPC-TYPE-IMS** for non-conversational IMS transactions. • **SWS-APPC-TYPE-IMSCONV** for conversational IMS transactions.
3	CHAR*	PIC X(8)	CHAR(8)	OUTPUT	Conversation ID of the APPC Call.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
4	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Length of the data to be sent.
5	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Input buffer. The input buffer is comprised of a 2 byte prefix containing the binary length of the buffer data followed by the IMS Message Input Descriptor (MID) data.
6	CHAR*	PIC X(17)	CHAR(17)	INPUT	Partner LU Name. The APPC LU Name of the IMS System.
7	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT	Data Type Sent. Specifies the action to take once the APPC Conversation is established. • **CM_BUFFER_DATA** specifies that no additional data is to be sent to the Partner LU, and the data can be buffered until a sufficient quantity is accumulated. • **CM_SEND_AND_FLUSH** specifies that no additional data is to be sent to the Partner LU, and the data is to be sent immediately. • **CM_SEND_AND_CONFIRM** specifies that the data is to be sent immediately along with a request for confirmation. • **CM_SEND_PREP_TO_RECEIVE** specifies that the data is to be sent immediately along with send control of the conversation. • **CM_SEND_AND_DEALLOCATE** specifies that the data is to be sent immediately along with a deallocation notification. ***Note:*** Optional parameter. If used, it requires that placeholders be specified for Local LU Name, Mode Name, Symbolic Partner LU Name, User ID, Password and Security Profile.
8	LONG	PIC S9(9) COMP	FIXED BIN(31)	INPUT/ OUTPUT	Return Code.

Return Values

SQLAPPCSEND/SWSAPPCSEND always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.

Return Value	Description
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. By using the SQLERROR/SWSERROR interface, you can obtain the error message pertinent to the error.

PL/I Example

```
%INCLUDE SPCPHD
 %INCLUDE ATBCMPLI
 %INCLUDE ATBPBPLI
       .
       .
       .
       .
 DCL  STMTHDL      FIXED BIN(31);   /* Statement Handle       */
 DCL  RC           FIXED BIN(31);   /* RETURN CODE            */
 DCL  SENDLEN      FIXED BIN(31);   /* SEND LENGTH            */
 DCL  RECVLEN      FIXED BIN(31);   /* RECEIVE LENGTH         */
 DCL  SENDBUF      CHAR(82)         /* SEND BUFFER            */
 DCL  RECVBUF      CHAR(82)         /* RECEIVED BUFFER        */
 DCL (FILL_1       initial('        ')
      FILL_2       initial('        ')
      FILL_3       initial('        ')) char(8);
 DCL (FILL_4       initial('        ')
      FILL_5       initial('        ')
      FILL_6       initial('        ')) char(10);


 CALL SWSAPPCSEND(STMTHDL,          /* STATEMENT HANDLE       */
                  SWS_APPC_TYPE_IMS,
                  PARTNER_LU_NAME,
                  CONVERSATION_ID,
                  SENDLEN,
                  SENDBUF,
                  RC,
                  RECVTYPE.

  RC = PLIRETV();                   /* GET RETURN CODE        */
  IF RC ^= SWS_SUCCESS THEN         /* EXIT PROGRAM IF BAD RC */
    EXIT;
```

C Example

```
#include "sccphd.h"              /* Neon headers                */
#include "atbcmc.h"              /* CPI Communications          */
#include "atbpbc.h"              /* LU6.2                       */

     .
     .
 long RC;                        /* return code                 */
 long stmtHDL;                   /* statement handle            */
 long recvtype;                  /* data received type          */
 long sendlen;                   /* input buffer length         */
 long recvlen;                   /* output buffer length        */
 long tp_name_length;            /* tp name length              */
 char sendbuf[82]                /* input buffer area           */
 char recvbuf[82]                /* output buffer               */
 char tp_name[64]                /* tp name                     */
 char fill_1[] = '      '        /* eight byte filler field     */
 char fill_2[] = '      '        /* eight byte filler field     */
 char fill_3[] = '      '        /* eight byte filler field     */
 char fill_4[] = '      '        /* ten byte filler field       */
 char fill_5[] = '      '        /* ten byte filler field       */
 char fill_6[] = '      '        /* ten byte filler field       */
 char conversation_id[8]         /* conversation id             */
 char partner_lu_name[17]        /* partner lu name             */

 CALL SWSAPPCSEND(stmtHDL,       /* statement handle            */
      SWS_APPC_TYPE_IMS
          partner_lu_name
          conversation_id
          sendlen
          sendbuf
          RC
          sendtype.

     if (rc ^= SWS_SUCCESS)
         return rc;
```

COBOL Example

```
COPY SBCPHD.                          Neon Copybook
 COPY ATBCMCOB.                        CPI COMMUNICATIONS COPYBOOK
 COPY ATBPBCOB.                        LU6.2 COPYBOOK
        .
        .
        .
        .
 77 STATEMENT-HANDLE         USAGE IS POINTER.
 77 FILLER-PARMLIST-01       PIC X(8) VALUE IS SPACES.
 77 FILLER-PARMLIST-02       PIC X(8) VALUE IS SPACES.
 77 FILLER-PARMLIST-03       PIC X(8) VALUE IS SPACES.
 77 FILLER-PARMLIST-04       PIC X(10) VALUE IS SPACES.
 77 FILLER-PARMLIST-05       PIC X(10) VALUE IS SPACES.
 77 FILLER-PARMLIST-06       PIC X(10) VALUE IS SPACES.
 01 SEND-BUFFER.
 05 SEND-BUFFER-LENGTH       PIC 9(4) COMP-4.
 05 SEND-BUFFER-CONTENTS     PIC X(100) VALUE IS SPACES.
 05 SEND-EXTRA-AREA          PIC X(10).
 01 RECEIVE-BUFFER.
 05 RECEIVE-BUFFER-LENGTH    PIC 9(4) COMP-4.
 05 RECEIVE-BUFFER-CONTENTS PIC X(80).
 05 RECEIVE-EXTRA-AREA       PIC X(10).
        .
        .
        .
 SET CM-NONE            TO TRUE.
 SET ATB-SECURITY-NONE TO TRUE.
 SET SWS-APPC-TYPE-IMS TO TRUE.
 MOVE 'PART'            TO TP-NAME.
 MOVE 4                 TO TP-NAME-LENGTH.
 MOVE 'P390.P392AIMS'   TO PARTNER-LU-NAME.
 CALL 'SDCPAS' USING STATEMENT-HANDLE
                     SWS-APPC-TYPE
                     CONVERSATION-ID
                     SEND-LENGTH
                     SEND-BUFFER
                     PARTNER-LU-NAME
                     ATB-SEND-TYPE
                     ATB-RETCODE.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
 IF NOT SWS-SUCCESS
     GOBACK.
```

SDBAPSND/SWSAPSND Function

✔	Can be used in Shadow/REXX.
✔	Can be used from Other REXX interpreters.
✔	HLL entry point name is **SDCPAS/SWCPAS**.

The REXX-language SDBAPSND/SWSAPSND built-in function can be used to send data to an IMS transaction. This function requires that an active connection already be established to IMS.

Upon return from the call, the following REXX variables are populated with data:

REXX Variable	Description
APPC.STMTHDL	The statement handle for the interface call.
APPC.CONVID	The Conversation ID for the IMS APPC conversation.
APPC.OUTBUFF.0	The length of the data contained within the APPC.OUTBUFF.1 REXX variable.
APPC.OUTBUFF.1	The returned Message Output Descriptor data. The output message contains the MOD data as well as a two byte length prefix. This variable contains the data, including any supplied trailing blanks.
APPC.RETCODE	The APPC Interface return code.
APPC.DATATYPE	The returned datatype.

Syntax

The general form for a REXX-language invocation of SDBAPSND/SWSAPSND is:

```
rc = SDBAPSND/SWSAPSND(“Connection Type”, ,
                       “Conversation ID”, ,
                       “Transaction Data”, ,
                       “Partner LU Name”)
```

Valid Arguments

Connection Type	Specifies the type of IMS transaction to execute: • **IMS** for IMS Non-conversational transaction. • **IMSCONV** for IMS Conversational transaction. ***Note:*** Required parameter.
Conversation ID	Specifies the IMS Conversation ID. ***Note:*** Required parameter.
Transaction Data	Specifies any data required by the application program in order to process the transaction. This would be data in the format of the Message Input Descriptor (without the LLZZ prefix). ***Note:*** Required parameter.
Partner LU Name	Specifies the APPC LU Name for the IMS system ***Note:*** Optional parameter.
Synchronization Level	Specifies whether or not confirmation processing will be performed on this conversation. • **NONE** — no confirmation processing is required for this conversation. • **CONFIRM** — confirmation processing is required for this conversation. ***Note:*** Optional parameter. If used, it requires that placeholders be specified for the Partner LU Name.

SDBAPSND/SWSAPSND Example

```
/*------------------------------------------------------------ */
/* initialize some system values                               */
/*------------------------------------------------------------ */

address
SWSSEND imsappc = 'P390.P392AIMS'
imstran = 'NEONDISP'
parms = ''

/*-------------------------------------------------------------*/
/* Send data into IMS                                          */
/*-------------------------------------------------------------*/

rc = swsapsnd('ims',APPC.CONVID,parms,imsappc)
```

CICS APIs

Note:
In order to use these APIs, you need the CICS Transaction Server.

Two copy members supplied by IBM **must be included** before using the EXCI interface for CICS. The following tables represent the language, member names, and libraries where these members can be found.

Copybook Name	Language	Library
DFHXCPLD	Assembler	CICS410 SDFHMAC
DFHXCPLH	C	CICS410.SDFHC370
DFHXCPLO	COBOL	CICS410.SDFHCOB
DFHXCPLL	PL/I	CICS410.SDFHPL1

Copybook Name	Language	Library
DFHXCRCD	Assembler	CICS410 SDFHMAC
DFHXCRCH	C	CICS410.SDFHC370
DFHXCRCO	COBOL	CICS410.SDFHCOB
DFHXCRCL	PL/I	CICS410.SDFHPL1

The following API call functions, implemented by Shadow Web Server and Shadow Direct, include those for APPC connections between Transaction Server for CICS and CICS, V4.1 and above, as well as APPC connections via REXX-language interfaces.

API Description	DIRECT	WEB	SEF	WEB/RX
CICS APIs				
To establish EXCI connect:	SQLEXCICONNECT or SDCPEC	SWSEXCICONNECT or SWCPEC	SDBEXCON	SWSEXCON
To perform DPL request using EXCI:	SQLEXCIDPLREQ or SDCPED	SWSEXCIDPLREQ or SWCPED	SDBEXDPL	SWSEXDPL
To perform EXCI initusr:	SQLEXCIINITUSR or SDCPEI	SWSEXCIINITUSR or SWCPEI	SDBEXINI	SWSEXINI
To perform EXCI disconnect:	SQLEXCIDISCONN or SDCPEL	SWSEXCIDISCONN or SWCPEL	SDBEXDIS	SWSEXDIS

High-Level Language Interface SQLEXCICONNECT (SDCPEC) or SWSEXCICONNECT (SWCPEC) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPEC/SWCPEC.**

This call establishes an EXCI connect on behalf of an ODBC *CALL RPC* to a CICS region. It can be called by any host ODBC call RPC, and returns standard ODBC return codes.

In order to execute a CICS Transaction using this interface, you will need to execute the following API calls in the sequence listed below.

Service Name	Service Description
SWSEXCIINITUSR	To initialize the CICS EXCI connection.
SWSEXCICONNECT	To connect to CICS through the EXCI interface.
SWSEXCIDPLREQ	To issue a DPL request to CICS. This API can be called repetitively in order to complete the processing required for the transaction.
SWSEXCIDISCONN	To disconnect a CICS EXCI connection.

Syntax

The general form for invocation of SDCPEC/SWCPEC is:

```
CALL SQLEXCICONNECT USING STATEMENT-HANDLE
     SQL-CICS-TYPE
     CONNECTION-NAME
     EXCI-RETURN-CODE
     USER-TOKEN
     PIPE-TOKEN.
```

CALL Arguments

The SQLEXCICONNECT/SWSEXCICONNECT (SDCPEC/SWCPEC) function arguments are described in the table which follows. All parameters are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement Handle. Currently ignored however, it must contain zeros.
2	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	CICS Connection Type. Must be set to SWS-CICS-TYPE-EXCI or SWS-CICS-TYPE-NEON. The "NEON" type is currently not supported.
3	CHAR*	PIC X(4)	CHAR(4)	INPUT	CICS Connection Name. A field containing the name of a Defined Connection. The Connection Name must be defined (using the "DEFINE CONNECTION" command) in the Shadow Web Server startup exec (SWS_IN00).
4	exci_return_code*	EXCI-RETURN-CODE	EXCI_ RETURN_ CODE	OUTPUT	The CICS EXCI Return Code Copybook layout for the output return code area.
5	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	User Token.
6	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Pipe Token.

Return Values

SQLEXCICONNECT/SWSEXCICONNECT always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. By using the SQLERROR/SWSERROR interface, you can obtain the error message pertinent to the error.

PL/I Example

```
%INCLUDE DFHXCPLL
%INCLUDE DFHXCRCL
%INCLUDE SPCPHD
       .
       .
DCL  STMTHDL      FIXED BIN(31);      /* Statement Handle      */
DCL  USER_TOKEN   FIXED BIN(31);      /* User Token            */
DCL  PIPE_TOKEN   FIXED BIN(31);      /* User Token            */
DCL  CONNECTION_NAME    CHAR(4);      /* Connection Name       */
DCL  RC           FIXED BIN(31);      /* RETURN CODE           */

CONNECTION_NAME = 'EWST'              /* SET name              */

CALL SWSEXCICONNECT(STMTHDL           /* STATEMENT HANDLE      */
     SWS_CICS_TYPE_EXCI,              /* connection type       */
     CONNECTION_NAME,                 /* connection name       */
     EXCI_RETURN_CODE,                /* cics exci return area */
     USER_TOKEN,                      /* user token            */
     PIPE_TOKEN                       /* pipe token            */

RC = PLIRETV();                       /* GET RETURN CODE       */
IF RC ^= SWS_SUCCESS THEN             /* EXIT PROGRAM IF BAD RC*/
       EXIT;
```

C Example

Note:
The Neon Header file must be included after the CICS EXCI headers.

```
#include "dfhxcplh.h"            /* CICS Return Area header    */
#include "dfhxcrch.h"            /* CICS Response Codes        */
#include "sccphd.h"              /* Neon headers               */
exci_return_code exciRET;        /* cics exci return area      */
.
long RC;                         /* return code                */
long stmtHDL;                    /* statement handle           */
long userTOKEN;                  /* user token                 */
long pipeTOKEN;                  /* user token                 */
char connection_name[] = "EWST"; /* Connection Name            */

CALL SWSEXCICONNECT(stmtHDL,     /* statement handle           */
     SWS_CICS_TYPE_EXCI,         /* connection type            */
     connection_name,            /* connection name            */
     exciRET,                    /* cics exci return area      */
     userTOKEN,                  /* user token                 */
     pipeTOKEN                   /* pipe token                 */

if (rc ^= SWS_SUCCESS)
     return rc;
```

COBOL Example

```
COPY DFHXCPLO.                          CICS EXCI Return Areas
COPY DFHXCRCO.                          CICS EXCI Response Codes
COPY SBCPHD.                            Neon Copybook
        .
        .
77 CONNECTION-NAME          PIC X(4) VALUE IS 'EWST'.
77 STATEMENT-HANDLE         USAGE IS POINTER.
77 USER-TOKEN               PIC S9(5) COMP VALUE IS ZERO.
77 PIPE-TOKEN               PIC S9(5) COMP VALUE IS ZERO.
        .
        .
        .
SET SWS-CICS-TYPE-EXCI TO TRUE.
CALL 'SDCPEC' USING STATEMENT-HANDLE
                    SWS-CICS-TYPE
                    CONNECTION-NAME
                    EXCI-RETURN-CODE
                    USER-TOKEN
                    PIPE-TOKEN.

MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS
    GOBACK.
```

SDBEXCON/SWSEXCON Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level Language Interface available.

The REXX-language SDBEXCON/SWSEXCON built-in function is used to establish a CICS EXCI connect on behalf of the user.

Upon return from the call, the following REXX variables are populated with data:

REXX Variable	Description
EXCI.STMTHDL	The statement handle for the interface call.
EXCI.RETCODE	The EXCI Return Code Area as mapped by the CICS Copybooks.
EXCI.PIPETOKN	The Pipe Token required as input to subsequent API calls.

Syntax

The general form for a REXX-language invocation of SDBEXCON/SWSEXCON is:

```
rc = swsexcon(Connection Type, ,
              Connection Name, ,
              User Token)
```

Valid Arguments

Connection Type	Specifies the type of connection: • **EXCI** — use CICS EXCI interface. • **NEON** — currently not supported. ***Note:*** Required parameter.
Connection Name	Logical name of the connection as specified on the “DEFINE CONNECTION” statement in the Shadow Initialization exec member (SWS_IN00) ***Note:*** Required parameter.
User Token	Token created as a result of the SWSEXINI API Call. Use the EXCI.USERTOKN REXX variable name. ***Note:*** Required parameter.

SDBEXCON/SWSEXCON Examples

```
     contype  = 'EXCI'       /*  Connection Type             */
     conname  = 'EWST'       /*  Connection Name from DEFINE */
     cicstran = 'EXCI'       /*  CICS Transaction Code       */
     cicspgm  = 'DFH$AXCS'   /*  CICS Program Name           */
     address swssend

     /*-------------------------------------------------------*/
     /* Initialize the user                                   */
     /*-------------------------------------------------------*/
         rc = swsexini(contype,conname)

     /*-------------------------------------------------------*/
     /* Allocate a pipe                                       */
     /*-------------------------------------------------------*/
         rc = swsexcon(contype,conname,EXCI.USERTOKN)

     /*-------------------------------------------------------*/
     /* Issue DPL Request                                     */
     /*-------------------------------------------------------*/
         parm  = '00000001'x||'FILEA   000001'
         rc = swsexdpl(contype,conname,cicstran,cicspgm, ,
                       parm,EXCI.USERTOKN,EXCI.PIPETOKN)

         pgmrc = substr(EXCI.COMMAREA.1,1,4)
         pgmrc = c2x(pgmrc)

         parm  = substr(EXCI.COMMAREA.1,5,14)
         parm  = '00000002'x||parm

         data  = substr(EXCI.COMMAREA.1,19,80)

     /*-------------------------------------------------------*/
     /* Disconnect the pipe                                   */
     /*-------------------------------------------------------*/
         rc =
swsexdis(contype,conname,EXCI.USERTOKN,EXCI.PIPETOKN)
```

High-Level Language Interface SQLEXCIDPLREQ (SDCPED) or SWSEXCIDPLREQ (SWCPED) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPED/SWCPED.**

This function is used to disconnect a CICS EXCI Connection. In order to execute a CICS Transaction using this interface, you will need to execute the following API calls in the sequence listed below:

Service Name	Service Description
SWSEXCIINITUSR	To initialize the CICS EXCI connection.
SWSEXCICONNECT	To connect to CICS through the EXCI interface.
SWSEXCIDPLREQ	To issue a DPL request to CICS. This API can be called repetitively in order to complete the processing required for the transaction.
SWSEXCIDISCONN	To disconnect a CICS EXCI connection.

Syntax

The general form for invocation of SDCPED/SWCPED is:

```
CALL SQLEXCIDPLREQ  USING STATEMENT-HANDLE
          SQL-CICS-TYPE
          CONNECTION-NAME
          TRANS-ID
          EXCI-RETURN-CODE
          PROGRAM-NAME
          CICS-BUFFER-COMMAREA
          CICS-BUFFER-COMMAREA-LENGTH
          CICS-BUFFER-INPUT-LENGTH
          UOWID
          USER-ID
          USER-TOKEN
          PIPE-TOKEN
          EXCI-DPL-RETAREA
```

CALL Arguments

The SQLEXCIDPLREQ/SWSEXCIDPLREQ (SDCPED/SWCPED) function arguments are described in the table which follows. All parameters are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement Handle. Currently ignored however, it must contain zeros.
2	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	CICS Connection Type. Must be set to SWS-CICS-TYPE-EXCI or SWS-CICS-TYPE-NEON. The "NEON" type is currently not supported.
3	CHAR*	PIC X(4)	CHAR(4)	INPUT	CICS Connection Name. A field containing the name of a Defined Connection. The Connection Name must be defined (using the "DEFINE CONNECTION" command) in the Shadow Web Server startup exec (SWS_IN00).
4	CHAR*	PIC X(4)	CHAR(4)	INPUT	CICS Transaction Name. The name of the mirror transaction with which the target program is to run.
5	exci_return_ code*	EXCI-RETURN-CODE	EXCI_ RETURN_ CODE	OUTPUT	The CICS EXCI Return Code Copybook layout for the output return code area.
6	CHAR*	PIC S9(5) COMP	FIXED BIN(31)	INPUT	CICS Program Name. The program to run and interact with the DPL request.
7	CHAR*	PIC X(1 - 32704)	CHAR (1 - 32704)	INPUT/ OUTPUT	CICS COMMAREA.
8	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	CICS COMMAREA Length. The total length of the COMMAREA.
9	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Input data length. The total length of the data within the COMMAREA.
10	CHAR*	PIC X(8)	CHAR(8)	INPUT	User ID. This is used as a work field and should be spaces upon entry to the API call.
11	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	User Token.
12	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Pipe Token.
13	exci-dpl-retarea *	EXCI-DPL-RETAREA	EXCI_DPL_ RETAREA	OUTPUT	The CICS EXCI DPL Return area Copybook layout for the output return area.

Return Values

SQLEXCIDPLREQ/SWSEXCIDPLREQ always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. By using the SQLERROR/SWSERROR interface, you can obtain the error message pertinent to the error.

PL/I Example

```
%INCLUDE DFHXCPLL
%INCLUDE DFHXCRCL
%INCLUDE SPCPHD
       .
       .
       .
DCL  STMTHDL     FIXED BIN(31);     /* Statement Handle       */
DCL  USER_TOKEN  FIXED BIN(31);     /* User Token             */
DCL  PIPE_TOKEN  FIXED BIN(31);     /* User Token             */
DCL  CONNECTION_NAME   CHAR(4);     /* Connection Name        */
DCL  RC          FIXED BIN(31);     /* RETURN CODE            */

CONNECTION_NAME = 'EWST'            /* SET name               */

CALL SWSEXCIDPLREQ(STMTHDL          /* STATEMENT HANDLE       */
     SWS_CICS_TYPE_EXCI,            /* CONNECTION TYPE        */
     CONNECTION_NAME,               /* CONNECTION NAME        */
     TRANS_ID,                      /* TRANSACTION ID         */
     EXCI_RETURN_CODE,              /* CICS EXCI RETURN AREA  */
     PROGRAM_NAME,                  /* PROGRAM NAME           */
     COMMAREA,                      /* COMMAREA               */
     SIZEOF(COMMAREA),              /* COMMAREA LENGTH        */
     STRLEN(COMMAREA),              /* COMMAREA LENGTH        */
     NULL,                          /* UNIT OF WORK ID AREA   */
     USER_ID,                       /* USER ID                */
     USERTOKEN,                     /* USER TOKEN             */
     PIPETOKEN,                     /* PIPE TOKEN             */
     EXCI_DPL_RETAREA)              /* DPL RETURN CODE AREA   */

RC = PLIRETV();                     /* GET RETURN CODE        */
IF RC ^= SWS_SUCCESS THEN           /* EXIT PROGRAM IF BAD RC */
   EXIT;
```

C Example

Note:
The Neon Header file must be included after the CICS EXCI headers.

```
#include "dfhxcplh.h"             /* CICS Return Area header  */
#include "dfhxcrch.h"             /* CICS Response Codes      */
#include "sccphd.h"               /* Neon headers             */

exci_return_code exciRET;         /* cics exci return area    */
exci_dpl_retarea exciDPL;         /* cics exci DPL return area*/
       .
       .
       .
       .
long RC;                          /* return code              */
long stmtHDL;                     /* statement handle         */
long userTOKEN;                   /* user token               */
long pipeTOKEN;                   /* user token               */
char commarea[32704];             /* Connection Name          */
char connection_name[] = "EWST";  /* Connection Name          */
char trans_id[] = "EXCI";         /* Transaction id           */
char user_id[] = "        ";      /* userid                   */

CALL SWSEXCIDPLREQ(stmtHDL,       /* statement handle         */
     SWS_CICS_TYPE_EXCI,          /* connection type          */
     connection_name,             /* connection name          */
     trans_id,                    /* transaction id           */
     exciRET,                     /* cics exci return area    */
     program_name,                /* program name             */
     commarea,                    /* commarea                 */
     sizeof(commarea),            /* commarea length          */
     strlen(commarea),            /* commarea length          */
     NULL,                        /* unit of work id area     */
     user_id,                     /* user id                  */
     userTOKEN,                   /* user token               */
     pipeTOKEN,                   /* pipe token               */
     exciDPL)                     /* DPL Return Code Area     */

if (rc ^= SWS_SUCCESS)
     return rc;
```

COBOL Example

```
COPY DFHXCPLO.                          CICS EXCI Return Areas
COPY DFHXCRCO.                          CICS EXCI Response Codes
COPY SBCPHD.                            Neon Copybook
         .
         .
77 CONNECTION-NAME                PIC X(4) VALUE IS 'EWST'.
77 TRANS-ID                       PIC X(4) VALUE IS 'EXCI'.
77 PROGRAM-NAME                   PIC X(8) VALUE IS 'DFH$AXCS'.
77 FILL-8                         PIC X(8) VALUE IS SPACES.
77 STATEMENT-HANDLE               USAGE IS POINTER.
77 USER-TOKEN                     PIC S9(5) COMP VALUE IS ZERO.
77 PIPE-TOKEN                     PIC S9(5) COMP VALUE IS ZERO.
77 CICS-BUFFER-INPUT-LENGTH       PIC S9(5) COMP VALUE IS ZERO.
77 CICS-BUFFER-COMMAREA-LENGTH    PIC S9(5) COMP VALUE IS ZERO.
01 UOWID                          USAGE IS POINTER.
01 CICSBUFFER.
05 CICS-BUFFER-COMMAREA.
10 CICS-BUFFER-COMMAREA-RETURNCD  PIC S9(5) COMP.
10 CICS-BUFFER-COMMAREA-FILENAME  PIC X(8).
10 CICS-BUFFER-COMMAREA-RIDFIELD  PIC X(6).
10 CICS-BUFFER-COMMAREA-RECORD    PIC X(512).
         .
         .
SET SWS-CICS-TYPE-EXCI TO TRUE.
CALL 'SDCPED' USING STATEMENT-HANDLE
                    SWS-CICS-TYPE
                    CONNECTION-NAME
                    TRANS-ID
                    EXCI-RETURN-CODE
                    PROGRAM-NAME
                    CICS-BUFFER-COMMAREA
                    CICS-BUFFER-COMMAREA-LENGTH
                    CICS-BUFFER-INPUT-LENGTH
                    UOWID
                    FILL-8
                    USER-TOKEN
                    PIPE-TOKEN
                    EXCI-DPL-RETAREA.

MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS
   GOBACK.
```

SDBEXDPL/SWSEXDPL Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level Language Interface available.

The REXX-language SDBEXDPL/SWSEXDPL built-in function is used to establish a CICS EXCI connect on behalf of the user.

Upon return from the call, the following REXX variables are populated with data:

REXX Variable	Description
EXCI.STMTHDL	The statement handle for the interface call.
EXCI.RETCODE	The EXCI Return Code Area as mapped by the CICS Copybooks.
EXCI.COMMAREA.1	The output COMMAREA from the CICS Transaction.
EXCI.DPLCODE	The DPL Return Code Area as mapped by the CICS Copybooks.

Syntax

The general form for a REXX-language invocation of SDBEXDPL/SWSEXDPL is:

```
rc = swsexdpl(Connection Type, ,
              Connection Name, ,
              CICS Transaction Name, ,
              CICS Program Name, ,
              CICS Transaction Input, ,
              User Token, ,
              Pipe Token)
```

Valid Arguments

Connection Type	Specifies the type of connection: • **EXCI** — use CICS EXCI interface. • **NEON** — currently not supported. ***Note:*** Required parameter.
Connection Name	Logical name of the connection as specified on the “DEFINE CONNECTION” statement in the Shadow Initialization exec member (SWS_IN00). ***Note:*** Required parameter.
CICS Transaction Name	The Trans-ID of the mirror transaction with which the target program is to run under. ***Note:*** Required parameter.

CICS Program Name	The program to run and interact with via DPL requests. ***Note:*** Required parameter.
CICS Transaction Input	The input to the executing transaction passed through the COMMAREA. A COMMAREA will of 32,704 bytes will be allocated, the transaction input will be copied into this area and passed to the transaction. ***Note:*** Required parameter.
User Token	The token created as a result of the SWSEXINI API Call. Use the EXCI.USERTOKN REXX variable name. ***Note:*** Required parameter.
Pipe Token	The token created as a result of the SWSEXCON API Call. Use the EXCI.PIPETOKN REXX variable name. ***Note:*** Required parameter.

SDBEXDPL/SWSEXDPL Examples

```
     contype  = 'EXCI'       /*  Connection Type             */
     conname  = 'EWST'       /*  Connection Name from DEFINE */
     cicstran = 'EXCI'       /*  CICS Transaction Code       */
     cicspgm  = 'DFH$AXCS'   /*  CICS Program Name           */
     address swssend

     /*-------------------------------------------------------*/
     /* Initialize the user                                   */
     /*-------------------------------------------------------*/
         rc = swsexini(contype,conname)

     /*-------------------------------------------------------*/
     /* Allocate a pipe                                       */
     /*-------------------------------------------------------*/
         rc = swsexcon(contype,conname,EXCI.USERTOKN)

     /*-------------------------------------------------------*/
     /* Issue DPL Request                                     */
     /*-------------------------------------------------------*/
         parm  = '00000001'x||'FILEA   000001'
         rc = swsexdpl(contype,conname,cicstran,cicspgm, ,
                       parm,EXCI.USERTOKN,EXCI.PIPETOKN)

         pgmrc = substr(EXCI.COMMAREA.1,1,4)
         pgmrc = c2x(pgmrc)

         parm  = substr(EXCI.COMMAREA.1,5,14)
         parm  = '00000002'x||parm

         data  = substr(EXCI.COMMAREA.1,19,80)

     /*-------------------------------------------------------*/
     /* Disconnect the pipe                                   */
     /*-------------------------------------------------------*/
         rc =
swsexdis(contype,conname,EXCI.USERTOKN,EXCI.PIPETOKN)
```

High-Level Language Interface SQLEXCIINITUSR (SDCPEI) or SWSEXCIINITUSR (SWCPEI) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPEI/SWCPEI.**

This call is used to initialize a user for access to CICS using EXCI. Standard ODBC return codes are returned. In order to execute a CICS Transaction using this interface, you will need to execute the following API calls in the sequence listed below.

Service Name	Service Description
SWSEXCIINITUSR	To initialize the CICS EXCI connection .
SWSEXCICONNECT	To connect to CICS through the EXCI interface.
SWSEXCIDPLREQ	To issue a DPL request to CICS. This API can be called repetitively in order to complete the processing required for the transaction.
SWSEXCIDISCONN	To disconnect a CICS EXCI connection.

Syntax

The general form for invocation of SDCPEI/SWCPEI is:

```
CALL SQLEXCIINITUSR USING STATEMENT-HANDLE
     SQL-CICS-TYPE
     CONNECTION-NAME
     EXCI-RETURN-CODE
     USER-TOKEN.
```

CALL Arguments

The SQLEXCIINITUSR/SWSEXCIINITUSR (SDCPEI/SWCPEI) function arguments are described in the table which follows. All parameters are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement Handle. Currently ignored however, it must contain zeros.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
2	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	CICS Connection Type. Must be set to SWS-CICS-TYPE-EXCI or SWS-CICS-TYPE-NEON. The “NEON” type is currently not supported.
3	CHAR*	PIC X(4)	CHAR(4)	INPUT	CICS Connection Name. A field containing the name of a Defined Connection. The Connection Name must be defined (using the “DEFINE CONNECTION” command) in the Shadow Web Server startup exec (SWS_IN00).
4	exci_return_ code*	EXCI-RETURN-CODE	EXCI_ RETURN_ CODE	OUTPUT	The CICS EXCI Return Code Copybook layout for the output return code area.
5	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	User Token.

Return Values

SQLEXCIINITUSR/SWSEXCIINITUSR always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. By using the SQLERROR/SWSERROR interface, you can obtain the error message pertinent to the error.

PL/I Example

```
%INCLUDE DFHXCPLL
%INCLUDE DFHXCRCL
%INCLUDE SPCPHD
       .
       .
DCL  STMTHDL      FIXED BIN(31);  /* Statement Handle         */
DCL  USER_TOKEN   FIXED BIN(31);  /* User Token               */
DCL  CONNECTION_NAME    CHAR(4);  /* Connection Name          */
DCL  RC           FIXED BIN(31);  /* RETURN CODE              */

CONNECTION_NAME = 'EWST'          /* SET name                 */

CALL SWSEXCIINITUSR(STMTHDL,      /* STATEMENT HANDLE         */
     SWS_CICS_TYPE_EXCI,          /* connection type          */
     CONNECTION_NAME,             /* connection name          */
     EXCI_RETURN_CODE,            /* cics exci return area    */
     USER_TOKEN)                  /* user token               */
RC = PLIRETV();                   /* GET RETURN CODE          */
IF RC ^= SWS_SUCCESS THEN         /* EXIT PROGRAM IF BAD RC   */
   EXIT;
```

C Example

Note:
The Neon Header file must be included after the CICS EXCI headers.

```
#include "dfhxcplh.h"             /* CICS Return Area header   */
#include "dfhxcrch.h"             /* CICS Response Codes       */
#include "sccphd.h"               /* Neon headers              */

exci_return_code exciRET;         /* cics exci return area     */
       .
       .
       .
long RC;                          /* return code               */
long stmtHDL;                     /* statement handle          */
long userTOKEN;                   /* user token                */
char connection_name[] = "EWST";  /* Connection Name           */

CALL SWSEXCIINITUSR(stmtHDL,      /* statement handle          */
     SWS_CICS_TYPE_EXCI,          /* connection type           */
     connection_name,             /* connection name           */
     exciRET,                     /* cics exci return area     */
     userTOKEN)                   /* user token                */

if (rc ^= SWS_SUCCESS)
    return rc;
```

COBOL Example

```
COPY DFHXCPLO.                          CICS EXCI Return Areas
COPY DFHXCRCO.                          CICS EXCI Response Codes
COPY SBCPHD.                            Neon Copybook
        .
        .
77 CONNECTION-NAME          PIC X(4) VALUE IS 'EWST'.
77 STATEMENT-HANDLE         USAGE IS POINTER.
77 USER-TOKEN               PIC S9(5) COMP VALUE IS ZERO.
        .
        .
        .
SET SWS-CICS-TYPE-EXCI TO TRUE.
CALL 'SDCPEI' USING STATEMENT-HANDLE
                    SWS-CICS-TYPE
                    CONNECTION-NAME
                    EXCI-RETURN-CODE
                    USER-TOKEN.

MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS
    GOBACK.
```

SDBEXINI/SWSEXINI Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level Language Interface available.

The REXX-language SDBEXINI/SWSEXINI built-in function is used to initialize a CICS EXCI interface between Shadow for CICS and CICS v4.1 and above.

Upon return from the call, the following REXX variables are populated with data:

REXX Variable	Description
EXCI.STMTHDL	The statement handle for the interface call.
EXCI.RETCODE	The EXCI Return Code Area as mapped by the CICS Copybooks.
EXCI.USERTOKN	The User Token required as input to subsequent API calls.

Syntax

The general form for a REXX-language invocation of SDBEXINI/SWSEXINI is:

```
rc = swsexini(Connection Type, ,
              Connection Name)
```

Valid Arguments

Connection Type	Specifies the type of connection: • **EXCI** — use CICS EXCI interface. • **NEON** — currently not supported. ***Note:*** Required parameter.
Connection Name	Logical name of the connection as specified on the “DEFINE CONNECTION” statement in the Shadow Initialization exec member (SWS_IN00) ***Note:*** Required parameter.

SDBEXINI/SWSEXINI Examples

```
contype  = 'EXCI'           /*  Connection Type              */
conname  = 'EWST'           /*  Connection Name from DEFINE  */
cicstran = 'EXCI'           /*  CICS Transaction Code        */
cicspgm  = 'DFH$AXCS'       /*  CICS Program Name            */
address swssend

/*-------------------------------------------------------*/
/* Initialize the user                                   */
/*-------------------------------------------------------*/
    rc = swsexini(contype,conname)

/*-------------------------------------------------------*/
/* Allocate a pipe                                       */
/*-------------------------------------------------------*/
    rc = swsexcon(contype,conname,EXCI.USERTOKN)

/*-------------------------------------------------------*/
/* Issue DPL Request                                     */
/*-------------------------------------------------------*/
    parm  = '00000001'x||'FILEA   000001'
    rc = swsexdpl(contype,conname,cicstran,cicspgm, ,
                  parm,EXCI.USERTOKN,EXCI.PIPETOKN)

    pgmrc = substr(EXCI.COMMAREA.1,1,4)
    pgmrc = c2x(pgmrc)

    parm  = substr(EXCI.COMMAREA.1,5,14)
    parm  = '00000002'x||parm

    data  = substr(EXCI.COMMAREA.1,19,80)

/*-------------------------------------------------------*/
/* Disconnect the pipe                                   */
/*-------------------------------------------------------*/
    rc = swsexdis(contype,conname,EXCI.USERTOKN,EXCI.PIPETOKN)
```

High-Level Language Interface SQLEXCIDISCONN (SDCPEL) or SWSEXCIDISCONN (SWCPEL) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPEL/SWCPEL.**

SQLEXCIDISCONN/SWSEXCIDISCONN is the Web Server API function used to disconnect a CICS EXCI Connection. In order to execute a CICS Transaction using this interface, you will need to execute the following API calls in the sequence listed below.

Service Name	Service Description
SWSEXCIINITUSR	To initialize the CICS EXCI connection.
SWSEXCICONNECT	To connect to CICS through the EXCI interface.
SWSEXCIDPLREQ	To issue a DPL request to CICS. This API can be called repetitively in order to complete the processing required for the transaction.
SWSEXCIDISCONN	To disconnect a CICS EXCI connection.

Call Arguments

The SQLEXCIDISCONN/SWSEXCIDISCONN (SDCPEL/SWCPEL) function arguments are described in the table which follows. All parameters are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Statement Handle. Currently ignored however, it must contain zeros.
2	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	CICS Connection Type. Must be set to SWS-CICS-TYPE-EXCI or SWS-CICS-TYPE-NEON. The “NEON” type is currently not supported.
3	CHAR*	PIC X(4)	CHAR(4)	INPUT	CICS Connection Name. A field containing the name of a Defined Connection. The Connection Name must be defined (using the “DEFINE CONNECTION” command) in the Shadow Web Server startup exec (SWS_IN00).

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
4	exci_return_ code*	EXCI-RETURN-CODE	EXCI_ RETURN_ CODE	OUTPUT	The CICS EXCI Return. Code Copybook layout for the output return code area.
5	LONG	PIC S9(5) COMP	FIXED BIN(31)	INPUT	User Token.
6	LONG	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	Pipe Token.

Return Values

SQLEXCIDISCONN/SWSEXCIDISCONN always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. By using the SQLERROR/SWSERROR interface, you can obtain the error message pertinent to the error.

PL/I Example

```
%INCLUDE DFHXCPLL
%INCLUDE DFHXCRCL
%INCLUDE SPCPHD
      .
       .
DCL  STMTHDL    FIXED BIN(31);      /* Statement Handle       */
DCL  USER_TOKEN  FIXED BIN(31);     /* User Token             */
DCL  PIPE_TOKEN  FIXED BIN(31);     /* User Token             */
DCL  CONNECTION_NAME  CHAR(4);      /* Connection Name        */
DCL  RC           FIXED BIN(31);    /* RETURN CODE            */

CONNECTION_NAME = 'EWST'            /* SET name               */

CALL SWSEXCIDISCONN(STMTHDL         /* STATEMENT HANDLE       */
     SWS_CICS_TYPE_EXCI             /* connection type        */
     CONNECTION_NAME                /* connection name        */
     EXCI_RETURN_CODE,              /* cics exci return area  */
     USER_TOKEN,                    /* user token             */
     PIPE_TOKEN)                    /* pipe token             */

RC = PLIRETV();                     /* GET RETURN CODE        */
IF RC ^= SWS_SUCCESS THEN           /* EXIT PROGRAM IF BAD RC */
   EXIT;
```

C Example

Note:
The Neon Header file must be included after the CICS EXCI headers.

```
#include "dfhxcplh.h"           /* CICS Return Area header    */
#include "dfhxcrch.h"           /* CICS Response Codes        */
#include "sccphd.h"             /* Neon headers               */

exci_return_code exciRET;       /* cics exci return area      */
  .
    .
long RC;                        /* return code                */
long stmtHDL;                   /* statement handle           */
long userTOKEN;                 /* user token                 */
long pipeTOKEN;                 /* user token                 */
char connection_name[] = "EWST"; /* Connection Name           */

CALL SWSExciDisconn(stmtHDL,    /* statement handle           */
     SWS_CICS_TYPE_EXCI,        /* connection type            */
     connection_name,           /* connection name            */
     exciRET,                   /* cics exci return area      */
     userTOKEN,                 /* user token                 */
     pipeTOKEN)                 /* pipe token                 */
if (rc ^= SWS_SUCCESS)
   return rc;
```

COBOL Example

```
 COPY DFHXCPLO.                      CICS EXCI Return Areas
 COPY DFHXCRCO.                      CICS EXCI Response Codes
 COPY SBCPHD.                        Neon Copybook
        .
 77 CONNECTION-NAME         PIC X(4) VALUE IS 'EWST'.
 77 STATEMENT-HANDLE        USAGE IS POINTER.
 77 USER-TOKEN              PIC S9(5) COMP VALUE IS ZERO.
 77 PIPE-TOKEN              PIC S9(5) COMP VALUE IS ZERO.
        .
SET SWS-CICS-TYPE-EXCI TO TRUE.
 CALL 'SDCPEL' USING STATEMENT-HANDLE
                     SWS-CICS-TYPE
                     CONNECTION-NAME
                     EXCI-RETURN-CODE
                     USER-TOKEN
                     PIPE-TOKEN.

 MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
 IF NOT SWS-SUCCESS
    GOBACK.
```

SDBEXDIS/SWSEXDIS Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreter.s
✔	High-level Language Interface available.

The REXX-language SDBEXDIS/SWSEXDIS built-in function is used to establish a CICS EXCI connect on behalf of the user.

Upon return from the call, the following REXX variables are populated with data:

REXX Variable	Description
EXCI.STMTHDL	The statement handle for the interface call.
EXCI.RETCODE	The EXCI Return Code Area as mapped by the CICS Copybooks.

Syntax

The general form for a REXX-language invocation of SDBEXDIS/SWSEXDIS is:

```
rc = swsexdis(Connection Type, ,
              Connection Name, ,
              User Token, ,
              Pipe Token)
```

Valid Arguments

Connection Type	Specifies the type of connection. • **EXCI** — use CICS EXCI interface. • **NEON** — currently not supported. ***Note:*** Required parameter.
Connection Name	Logical name of the connection as specified on the “DEFINE CONNECTION” statement in the Shadow Initialization exec member (SWS_IN00) ***Note:*** Required parameter.
User Token	The token created as a result of the SWSEXINI API Call. Use the EXCI.USERTOKN REXX variable name. ***Note:*** Required parameter.
Pipe Token	The token created as a result of the SWSEXCON API Call. Use the EXCI.PIPETOKN REXX variable name. ***Note:*** Required parameter.

SDBEXDIS/SWSEXDIS Examples

```
contype  = 'EXCI'            /*  Connection Type              */
conname  = 'EWST'            /*  Connection Name from DEFINE  */
cicstran = 'EXCI'            /*  CICS Transaction Code        */
cicspgm  = 'DFH$AXCS'        /*  CICS Program Name            */
address swssend

/*-------------------------------------------------------*/
/* Initialize the user                                   */
/*-------------------------------------------------------*/
    rc = swsexini(contype,conname)
/*-------------------------------------------------------*/
/* Allocate a pipe                                       */
/*-------------------------------------------------------*/
    rc = swsexcon(contype,conname,EXCI.USERTOKN)

/*-------------------------------------------------------*/
/* Issue DPL Request                                     */
/*-------------------------------------------------------*/
    parm  = '00000001'x||'FILEA   000001'
    rc = swsexdpl(contype,conname,cicstran,cicspgm, ,
                  parm,EXCI.USERTOKN,EXCI.PIPETOKN)

    pgmrc = substr(EXCI.COMMAREA.1,1,4)
    pgmrc = c2x(pgmrc)

    parm  = substr(EXCI.COMMAREA.1,5,14)
    parm  = '00000002'x||parm

    data  = substr(EXCI.COMMAREA.1,19,80)

/*-------------------------------------------------------*/
/* Disconnect the pipe                                   */
/*-------------------------------------------------------*/
    rc = swsexdis(contype,conname,EXCI.USERTOKN,EXCI.PIPETOKN)
```

Web Server Specific APIs

API Description	DIRECT	WEB	SEF	WEB/RX
Web Server Specific APIs				
To transmit data to Web Server clients:		SWCPSN		SWSSEND
To buffer outbound HTTP response headers:		SWCPRE		SWSRESP
To transmit data directly to web client:		SWCPFI		SWSFILE
To provide new URL value:		SWCPSO		SWSSET
To provide a means to issue an MVS write to operator:		SWCPWT		SWSWTO

:

High-Level Language Interface SWSSEND (SWCPSN) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SWCPSN**.

SWSSEND is the Web Server API function used to transmit out-bound data to web server clients, flush buffered data to the client, or to purge buffers which have not been transmitted.

CALL Arguments

The SWSSEND function takes four arguments. All four arguments must be specified on the call.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A four-byte flag-word indicating the type of operation to be performed. One of the following manifest constants should be used to indicate the desired operation. The values are mutually exclusive; only one can be used. • **SWS_SEND_BINARY** indicates that the third argument contains binary-format data which should be transmitted to the web client, as is. • **SWS_SEND_TEXT** indicates that the third argument contains text-format data which is processed by the web server to remove trailing blanks, is translated from EBCDIC to ASCII, and a CR character appended to the end. • **SWS_SEND_FLUSH** indicates that any unsent data within Web Server buffers should be immediately transmitted to the client program. • **SWS_SEND_PURGE** indicates that any unsent data within the Web Servers transmission buffers should be discarded without being transmitted to the client.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
3	PTR	PIC X(nnn)	CHAR (nnn)	Input	The buffer area containing the data to be transmitted to the web client. The data value can be no longer than 8K in length. This argument must be specified, even if the second argument specifies a flush or purge operation. The length of the data value within the buffer area is specified by the forth argument.
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the data value given by the third argument. This can be an integer fullword value in the range 0 to 8K. You can also use the manifest constant, SWS_NTS, to specify that data is a null-terminated string. This value should be zero for flush or purge operations.

Return Values

SWSSEND always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The output data has been buffered, or for flush and purge operations, the buffer operation has been completed.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_INVALID_HANDLE, SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR/SWSERROR.
4	The operation failed due to loss of the communications session. Further out-bound transmissions will not be possible.
8	The requested operation was invalid within the overall context of the transaction process. When this return code is set, the cause for the request rejection can normally be found in the wrap-around trace. Possible reasons for return code **8** being set are: • An additional transmission buffer was needed, but could not be obtained. Transmission of data cannot be performed in cross-memory mode. • An output request (TEXT or BINARY) is being made, but is invalid at the current time. This occurs if you attempt to transmit data after having issued an SWSFILE(SEND) request.

The SWSSEND operation is not logged to the Server's wrap-around trace file unless an error occurs. If you need to trace information sent using SWSSEND, use the SENDTRACE keyword of the `/*WWW` rule header.

PL/I Example

```
DCL  SCONN   PTR;                   /* Connection Handle      */
DCL  SDATA   CHAR(256);             /* Text output area       */
DCL  SSIZE   FIXED BIN(31);         /* Text length area       */
DCL  RC      FIXED BIN(31);         /* return code            */
DCL  DMHX    FIXED BIN(31) BASED;   /* Dummy Handle field     */
ADDR(SCONN)->DMHX = 0;              /* Clear Connection Handle*/
SDATA    = 'Hello World!';          /* Set output area       */
SSIZE    = 12;                      /* set lengt              */
CALL SWSSEND( SCONN                 /* send the text data     */
      SWS_SEND_TEXT,
      SDATA,
      SSIZE );
RC = PLIRETV();                     /* get return code        */
IF RC ^= SWS_SUCCESS THEN           /* exit program if bad RC */
 EXIT;
```

C Example

```
HDBC sConn   = NULL;                /* Connection Handle      */
char sData[] = "Null-terminated!";  /* Text string definition */
long RC;                            /* return code            */
rc = SWSSend( &sConn,               /* send the text data     */
     SWS_SEND_TEXT,
     sData,
     SWS_NTS );
if (rc ^= SWS_SUCCESS) return;      /* exit program if bad RC */
```

COBOL Example

```
77  SCONN                   SAGE IS POINTER.
77  SDATA                   PIC X(80).
77  SSIZE                   PIC S9(5) COMP.
MOVE 'HELLO WORLD!'         TO SDATA.
MOVE 12                     TO SSIZE.
CALL 'SWCPSN' USING SCONN,
       SWS-SEND-TEXT,
       SDATA,
       SSIZE.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK
```

SWSSEND Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level language interface available.
✔	Also see **SWSSEND** Host Command Environment.

SWSSEND is a built-in function used to transmit out-bound data to web server clients from REXX-Language event procedures. SWSSEND can only be used from within WWW event procedures and will return an error if invoked from other event procedure types.

Syntax

The general form for invocation of SWSSEND is:

```
z = SWSSEND( arg1 {, arg2 } )
```

Valid Arguments

The SWSSEND function takes one or two arguments.

The first argument always specifies the data to be transmitted to the web server client. A NULL string can be passed as the first argument, or the argument can be omitted entirely by coding a single comma in its place.

The second argument can be one of the following string constants:

TEXT	Indicates that the data specified by the first argument is text data. The string is converted from EBCDIC to ASCII before transmission and a trailing CRLF is added. Trailing blanks, if any, are also removed from the string. Text format data is assumed, if the second argument is omitted.
BINARY	Indicates that the data specified by the first argument is binary data. The string is transmitted to the Web Client, as is, without additional processing.
FLUSH	Indicates any data already in the out-bound buffers should be written to the client immediately. The first argument is ignored.
PURGE	Indicates that all data currently un-transmitted within buffers should be discarded. The first argument is ignored.

Return Values

SWSSEND always returns a numeric value. If the value is zero the operation has completed successfully. A non-zero return value indicates that the communications session has been lost.

The SWSSEND operation is not logged to the Server's wrap-around trace file unless an error occurs. If you need to trace information sent using SWSSEND, use the SENDTRACE keyword of the /*WWW rule header.

Examples

The following call will buffer the HTML data for out-bound transmission. A Line-Feed character will be added following the data and the data will be translated to ASCII before transmission:

```
htmldata = "<h1>This is a Header</h1>"
z=SWSSEND( htmldata )
```

The following call will place the data into the out-bound buffer with no additional processing:

```
z=SWSSEND( gifdata , "BINARY" )
```

The following call will cause all buffered data to be sent to the client immediately.

```
z=SWSSEND( , "FLUSH" )
```

The following call will purge all previously buffered data. Data which was flushed prior to this call, will have already been sent to the web client.

```
z=SWSSEND( , "PURGE" )
```

High-Level Language Interface SWSRESP (SWCPRE) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SWCPRE.**

SWSRESP (entry point SWCPRE) is a high level function used to buffer customized out-bound HTTP response headers for subsequent transmission to web client browsers.

SWSRESP can be used to buffer an HTTP response header at any time during the life of a web transaction. Using SWSRESP to buffer HTTP response headers differs from merely writing these headers using SWSSEND:

- When using SWSSEND, HTTP headers must be written before any *message body* data (e.g. an HTML page or binary GIF image) has been output.
- SWSRESP can be used before, during or after output of the *message body.*

The Server *merges* the HTTP response headers which have been buffered using SWSRESP with any other data generated by the web transaction. This merge processing takes place when the web transaction ends and causes the complete output stream to be assembled and transmitted to the client.

SWSRESP is only valid when a web transaction procedure is operating in *server-parsed header mode* since the merging/assembly, described above, is disabled in *non-parsed-header* mode. A call to SWSRESP will return an error if the web transaction is not operating in server-parsed header mode.

CALL Arguments

The SWSRESP (entry point SWCPRE) function takes six arguments. All six arguments must be specified on the call.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The connection handle. The connection handle is an opaque, four byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A four-byte flag-word indicating the type of operation to be performed. The only value currently supported is: SWS_RESPONSE_ADD
3	PTR	PIC X(nnn)	CHAR (nnn)	Input	The buffer area containing the header to be transmitted to the web client. The data value can be no longer than 8K in length. The length of the header is specified by the 4th argument. *The colon after the response/general header is added by the SWSRESP function; therefore it is not necessary to include it in the third argument of the SWSRESP call.*
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the data value given by the third argument. This can be an integer fullword value in the range of 0 to 8K. You can also use the manifest constant SWS_NTS to specify that the data is a null terminated string.
5	PTR	PIC X(nnn)	CHAR (nnn)	Input	The buffer area containing the entity body text to be transmitted to the web client. The data value can be no longer than 8K in length. The length of the entity body text is specified by the 5th argument.
6	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the data value given by the fourth argument. This can be an integer fullword value in the range of 0 to 8K. You can also use the manifest constant SWS_NTS to specify that the data is a null terminated string.

Return Values

SWSRESP always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	The operation succeeded. The specified header has been placed in the output buffer.
SWS_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SWSERROR.
Any other value	The operation failed.

The SWSRESP operation is not logged to the Server's warp-around trace file unless an error occurs. If you need to trace information sent using SWSRESP, use the SENDTRACE keyword of the /*WWW rule header.

PL/I Example

```
DCL   SCONN     PTR;                 /* Connection Handle      */
DCL   HDATA     CHAR(256);           /* Header area            */
DCL   HSIZE     FIXED BIN(31);       /* Header length          */
DCL   BDATA     CHAR(256);           /* Body area              */
DCL   BSIZE     FIXED BIN(31);       /* Body length            */
DCL   DMHX      FIXED BIN(31) BASED;/* Dummy Handle field     */
ADDR(SCONN)->DMHX=0;                 /* Zero connection handle */
HDATA="Pragma";                      /* Set header data        */
HSIZE=6;                             /* Set header length      */
BDATA="no-cache";                    /* Set body data          */
BSIZE=8;                             /* Set body length        */
CALL SWSRESP(SCONN                   /* Send the response      */
             SWS_RESPONSE_ADD,
             HDATA,
             HSIZE,
             BDATA,
             BSIZE);
RC=PLIRETV();                        /* Get return code        */
IF RC ^=SWS_SUCCESS THEN             /* exit if bad RC         */
         EXIT;
```

C Example

```
HDBC   sConn     = NULL;             /* Connection Handle      */
char   hData[]   = "Pragma";         /* Header text            */
char   bData[]   = "no-cache";       /* body text              */
long   RC;                           /* return code            */
rc = SWSResp(&sConn,                 /* send the response      */
     SWS_RESPONSE_ADD,
     hData,
     SWS_NTS,
     bData,
     SWS_NTS);
If(rc ^=SWS_SUCCESS) return;         /* exit if bad rc         */
```

COBOL Example

```
77 SCONN         USAGE IS POINTER.
77 HDATA         PIC X(80).
77 HSIZE         PIC S9(5) COMP.
77 BDATA         PIC X(80).
77 BSIZE         PIC S9(5) COMP.
MOVE 'Pragma'    TO HDATA.
MOVE 6           TO HSIZE.
MOVE 'no-cache'  TO BDATA.
MOVE 8           TO BSIZE.
CALL 'SWCPRE' USING SCONN,
      SWS-RESPONSE-ADD,
      HDATA,
      HSIZE,
      BDATA,
      BSIZE.
MOVE RETURN CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.
```

SWSRESP Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High Level language available.

SWSRESP is a high level function used to buffer customized out-bound HTTP response headers for subsequent transmission to web client browsers.

SWSRESP can be used to buffer an HTTP response header at any time during the life of a web transaction. Using SWSRESP to buffer HTTP response headers differs from merely writing these headers using SWSSEND:

- When using SWSSEND, HTTP headers must be written before any *message body* data (e.g. an HTML page or binary GIF image) has been output.
- SWSRESP can be used before, during or after output of the *message body.*

The Server *merges* the HTTP response headers which have been buffered using SWSRESP with any other data generated by the web transaction. This merge processing takes place when the web transaction ends and causes the complete output stream to be assembled and transmitted to the client.

SWSRESP is only valid when a web transaction procedure is operating in *server-parsed header mode* since the merging/assembly, described above, is disabled in *non-parsed-header* mode. A call to SWSRESP will return an error if the web transaction is not operating in server-parsed header mode.

Syntax

The general form for invocation of SWSRESP is:

```
Z = SWSRESP( func, hname, hvalue )
```

Valid Arguments

The SWSRESP takes three arguments:

func	Specifies the function to be performed. At this time the only supported function value is ADD. Any other argument value will return an error.
hname	Specifies the name of the HTTP response/general header to be transmitted to the web server client. The HTTP response header name *should be one which is universally recognized as defined in the Hypertext Transfer Protocol - HTTP/1.1(RFC2068) specification. However, any string can be coded.* The colon which ends each HTTP response header name must be omitted from the string, since SWSRESP always inserts a trailing colon (e.g. Code "Content-type", not "Content-type:").

hvalue	The third argument, hvalue, specifies the value for the corresponding HTTP response header. The content of the third argument will, of course, vary depending on the response/general header specified in the second argument. The entity body text is converted to ASCII and then transmitted exactly as provided by this argument.

Return Values

SWSRESP always returns a numeric value. A zero return value indicates successful buffering (though not actual transmission) of the HTTP response header. A non-zero return value indicates that an error has occurred.

The SWSRESP operation is not logged to the Server's wrap-around trace file unless an error occurs. If you need to trace information actually sent as a result of an SWSRESP request, use the SENDTRACE keyword of the /*WWW rule header.

Example

The following call will cause two HTTP response headers to be merged with other headers generated by SWSFILE. A "Pragma: no-cache" and "Expires: Mon, 14 APR 1998 12:02:03 GMT" HTTP response header will be included in the final output transmission stream.

```
/*WWW  /testswsresp  sendtrace(yes)
/*REXX
Z = SWSRESP('ADD','Pragma','no-cache')
Z = FILE('SEND','DDN','SAMPDATA','ICINFO','image/gif','BINARY')
Z = SWSRESP ('ADD', 'Expires', 'Mon, 14 APR 1998 12:02:03 GHT')
```

High-Level Language Interface SWSFILE (SWCPFI) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from Other REXX interpreters.
✔	HLL entry point name is **SDCPAR/SWCPAR**.

SWSFILE is the Web Server API function used to retrieve data from an MVS dataset and transmit it to a Web Client program. The caller specifies the MVS dataset from which data is to be retrieved and the MIME content type to be used when the information is transmitted.

For text format data, the Server processes HTML Extension Statements, if any, within the data file before the information is transmitted to the Web Client. This facility allows the information to be specially tailored in response to run-time conditions.

The SWSFILE (SWCPFI) interface supports the following operation request types:

SWS_FILE_SEND	This request type is used to invoke out-bound transmission of a PDS dataset member (BPAM or PDSE), or a sequential dataset (QSAM). This request type contains all of the functionality formerly incorporated into the SWS_FILE_PDSSEND request type. We recommend that you use the SWS_FILE_SEND function for all PDS member or sequential dataset transmission requests. This request type supercedes and obsoletes the older SWS_FILE_PDSSEND request type.
SWS_FILE_QUEUE	This request type is used to cause *small* data members to be read into the transaction's external data queue. Because the size of the external data queue is limited, this function should not be used for members containing more than 2-300 logical records. The logical record length is limited to 752 bytes in length and any excess bytes are truncated without notification. Once read to the external data queue member data records can be retrieve using the SWSGetQueue HLL API interface.
SWS_FILE_PDSSEND	*This request type is being retired, and should not be used for future Web transaction HLL program development.* Use the newer SWS_FILE_SEND request type for future HLL program development, since it supercedes this PDS-only, DDNAME-only API interface. The Shadow Web Server will continue to support this request type so that HLL programs developed before SWS_FILE_SEND was available will continue to operate correctly without requiring a re-write. Future enhancements will only be made to the preferred SWS_FILE_SEND type. The SWS_FILE_PDSSEND request type is documented separately in the Deprecated High-Level Language SWSFILE (SWCPFI) Request Types page.

Call Arguments

The SWSFILE (SWCPFI) function arguments are described in the table which follows. For send requests, each of the six arguments described must be present on the API CALL. Omit the sixth argument for queue operation requests.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	USAGE POINTER	PTR	INPUT	The Web Server connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	INPUT	A four-byte flag-word indicating the request type of file-related operation to be performed. One of the following values must be specified for this argument: • **SWS_FILE_SEND** is used to request out-bound transmission of PDS members or an entire sequential datasets. This request type should be used for all future HLL development. • **SWS_FILE_QUEUE** is used to request that the file data records be read into the transactions external data queue. • **SWS_FILE_PDSSEND** has been superceded by the SWS_FILE_SEND request type, which should be used for all future HLL program development. This request type continues to be supported and is documented separately.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
3	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	INPUT	A four-byte flag-word specifying various options for the API CALL request. The options available for specification are defined as manifest constants within each HLL header file. You combine the options which define your request by adding (or logically ORing) the manifest constants together to form a single, 32-bit flag word. The options are broken into three groups. One option constant can be selected from each of the groups shown below. **Name Specification Options** One option from this group ***must*** be coded. This option indicates whether the call refers to the MVS file using a DD name or fully-qualified dataset name. • **SWS_FILE_DDNAME** specifies the DDNAME of the MVS dataset. • **SWS_FILE_DSNAME** specifies a fully-qualified MVS dataset name. **Data Contents Format Options** One option from this group ***must*** be coded. This option specifies the format of the data contained within the dataset; either text or binary. The Server transmits binary format data with no modifications, but applies various translation and editing options to text format data before transmission. • **SWS_SEND_TEXT** contains text format data. Before out-bound transmission, the Server translates the information from EBCDIC to ASCII, strips trailing blanks, and appends a carriage return character to each line. • If the file contains HTML Extension Statements the Server will process these to tailor the final output, unless extension processing is surpressed by the **SWS_FILE_NOHTX** option. • **SWS_SEND_BINARY** contains binary format data. The Server transmits the data, as is, without modification. **HTML Extension Processing Options** One option from this group can optionally be coded, but none are required. No option from this group should be coded when SWS_FILE_BINARY has been specified.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
					This option controls how, or if, HTML Extension Statements within a text format file are processed by the Server before transmission. • **SWS_FILE_HTX** indicates the Server will process HTML Extension Statements. This option is the default for all text format files, so it need not be explicitly coded except for documentation purposes. • **SWS_FILE_NOHTX** indicates HTML Extension Statement processing will be surpressed. • **SWS_FILE_HTXREXXRULES** indicates that the Server will replace uninitialized variables located during HTML Extension processing with the upper-case variable name. If this option is not specified, the Server replaces uninitialized variables with a NULL string.
4	UCHAR *	PIC X(8) or PIC X(44)	CHAR(8) or CHAR(44)	INPUT	This argument specifies either the DDNAME or the DSNAME of the file to be transmitted. You must indicate which type of value is specified by coding one of the Name Specification Options, given above. When the SWS_FILE_DDNAME option is used, this value must be the 8-byte, blank padded DDNAME for the file. When the SWS_FILE_DSNAME option is used, this value must be the 44-byte, blank padded dataset name for the file. For either DDNAME or DSNAME specifications, a null-terminated string can be specified, if desired.
5	UCHAR *	PIC X(8)	CHAR(8)	INPUT	When the file referenced by this call is a PDS or PDSE (BPAM) dataset, this argument **must** contain the 8-byte, blank padded member name within the library which is to be transmitted. A null-terminated string can be specified, if desired. For operations on other file types (non-BPAM), this is not examined by the API function. You should specify a NULL value of a type appropriate to the particular high-level language.
6	UCHAR *	PIC X(50)	CHAR(50)	INPUT	The 50-byte, blank-padded MIME content type value to be used when the data is transmitted. A shorter string can be specified if the string is null terminated. Omit this argument for SWS_FILE_QUEUE requests.

Return Values

SWSFILE always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_NO_DATA_FOUND	Indicates that the DDNAME, DSNAME or PDS member name is not valid because the dataset or member does not exist, or because the dataset is being held exclusively by some other address space.
SWS_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SWSERROR.
Any other value	The operation failed. Generally this indicates an unrecoverable loss of the communications session between the Shadow Web Server and the client's web browser program.

PL/I Example

```
DCL SCONN   PTR;                              /* Connectionhandle */
DCL SDDNA   CHAR(8) INIT('HTMFILE');          /* File DDNAME      */
DCL SDSNA   CHAR(44) INIT('MY.HTML.DATA');    /* File DSNAME      */
DCL SDSNA2  CHAR(44) INIT('SOME.QSAM.DATA')/* File DSNAME      */
DCL SMENA   CHAR(8) INIT('HLLFILE');          /* Member name      */
DCL SMENA2  CHAR(8) INIT('MYFILE');           /* Member name      */
DCL SDUMMY  FIXED BIN(31) INIT(0);            /* NULL Argument    */
DCL SCOTY   CHAR(50) INIT('text/html');       /* Content type     */
DCL SCOTY2  CHAR(50) INIT('image/gif');       /* Content type     */
DCL SCOTY3  CHAR(50) INIT('text/plain');      /* Content type     */
DCL RC      FIXED BIN(31);                    /* return code      */
DCL DMHX    FIXED BIN(31) BASED;              /* Dummy Handle     */

ADDR(SCONN)->DMHX = 0;             /*Clear Connection Handle   */

/* Send the member HLLFILE from the PDS allocated to the       */
/* HTMFILE DD name using MIME type text/html.                  */

CALL SWSFILE     (SCONN,              /* Connection handle      */
                  SWS_FILE_SEND,      /* Send a file            */
                  SWS_FILE_DDNAME +   /* Arg 4 is a DD name     */
                  SWS_SEND_TEXT       /* data is text format    */
                  SDDNA,              /* Send from this DDNAME  */
                  SMENA               /* Send this member       */
                  SCOTY ),            /* File content           */
RC = PLIRETV();                       /* get return code        */
IF RC ^= SWS_SUCCESS THEN             /* exit program if bad RC */
     EXIT;

/* Send binary data from the PDS 'MY.HTML.DATA(MYFILE) as      */
/* MIME type image/gif.                                        */

CALL SWSFILE    ( SCONN,               /* Connection handle    */
                  SWS_FILE_SEND,       /* Send a file          */
                  SWS_FILE_DSNAME +    /* Arg 4 is a DSNAME    */
                  SWS_FILE_BINARY,     /* data in binary format */
                  SDSNA,               /* Send from this DSNAME */
                  SMENA2,              /* Send this member     */
                  SCOTY2 );            /* File content         */
RC=PLIRETV();                          /* get return code      */
IF RC ^= SWS_SUCCESS THEN              /* exit program if bad RC*/
     EXIT;

/* Send the sequential file 'SOME.QSAM.DATA' as MIME type      */
/* text/plain.                                                 */

CALL SWSFILE    ( SCONN,               /* Connection handle    */
                  SWS_FILE_SEND,       /* Send a file          */
                  SWS_FILE_DSNAME +    /* Arg 4 is a DSNAME    */
                  SWS_FILE_TEXT        /* data in text format  */
                  SDSNA2,              /* Send from this DSNAME */
                  SDUMMY,              /* NULL argument        */
```

```
                  SCOTY3 );          /* File content          */
RC = PLIRETV();                      /* get return code       */
IF RC ^= SWS_SUCCESS THEN            /* exit program if bad RC*/
      EXIT;
```

C Example

```
HDBC sConn   = NULL;               */Connection Handle       */
long RC;                           /* return code            */

/* Send the HLLFILE member from the PDS allocated with the    */
/* DD name HTMFILE as MIME type text/html.                    */

rc = SWSFILE( &sConn,              /* connection handle      */
              SWS_FILE_SEND,       /* Send a file            */
              SWS_FILE_DDNAME |    /* Arg 4 is a DD name     */
              SWS_SEND_TEXT,       /* data is text format    */
              "HTMFILE",           /* Send from this DDNAME  */
              "HLLFILE",           /* Send this member       */
              "text/html" );       /* File content           */
if (rc ^= SWS_SUCCESS) return;     /* exit program if bad RC */


/* Send 'MY.HTML.DATA(MYFILE)' as MIME type image/gif         */
rc = SWSFILE( &sConn,              /* Connection handle      */
     SWS_FILE_SEND,                /* Send a file            */
     SWS_FILE_DSNAME |             /* Arg 4 is a DSNAME      */
     SWS_FILE_BINARY,              /* data is binary format  */
     "MY.HTML.DATA",               /* Send from this DSNAME  */
     "MYFILE",                     /* Send this member       */
     "image/gif" );                /* File content           */

if (rc ^= SWS_SUCCESS) return;     /* exit program if bad RC */

/* Send 'SOME.QSAM.DATA' as MIME type text/plain.             */

rc = SWSFILE( &sConn,              /* connection handle      */
     SWS_FILE_SEND,                /* Send a file            */
     SWS_FILE_DSNAME |             /* Arg 4 is a DSNAME      */
     SWS_SEND_TEXT,                /* data is binary format  */
     "SOME.QSAM.DATA",             /* Send from this DSNAME  */
     "",                           /* NULL argument          */
     "text/plain"                  /* File content           */
if (rc ^= SWS_SUCCESS) return;     /* exit program if bad RC */
```

COBOL Example

```
77  SCONN              USAGE IS POINTER.
77  SDDNA             PIC X(8) VALUE 'HTMFILE'.
77  SDSNA             PIC X(44) VALUE 'MY.HTML.DATA'.
77  SDSNA2            PIC X(44) VALUE 'SOME.QSAM.DATA'
77  SMENA             PIC X(8) VALUE 'HLLFILE'.
77  SMENA2            PIC X(8) VALUE 'MYFILE'.
77  SCOTY             PIC X(50) VALUE 'text/html'.
77  SCOTY2            PIC X(50) VALUE 'image/gif'.
77  SCOTY3            PIC X(50) VALUE 'text/plain'.
77  SDUMMY            PIC S9(5) COMP VALUE 0.
77  SOPTION           PIC S9(5) COMP.

*     SEND THE HLLFILE MEMBER FROM THE PDS ALLOCATED TO
*     THE HTMFILE DD NAME AS MIME TYPE text/html.

COMPUTE SOPTION = SWS-FILE-DDNAME + SWS-SEND-TEXT.
CALL 'SWCPFI' USING SCONN,
              SWS-FILE-SEND,
              SOPTION,
              SDDNA,
              SMENA,
              SCOTY.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.

*     SEND 'MY.HTML.DATA(MYFILE)' AS MIME TYPE image/gif.

COMPUTE SOPTION = SWS-FILE-DSNAME + SWS-SEND-BINARY.
CALL 'SWCPFI' USING SCONN,
              SWS-FILE-SEND,
              SOPTION,
              SDSNA,
              SMENA2,
              SCOTY2.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.

*     SEND 'SOME.QSAM.DATA AS MIME TYPE text/plain.

COMPUTE SOPTION = SWS-FILE-DSNAME + SWS-SEND-TEXT.
CALL 'SWCPFI' USING SCONN,
              SWS-FILE-SEND,
              SOPTION,
              SDSNA2,
              SDUMMY,
              SCOTY3.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.
```

Deprecated SWSFILE (SWCPFI) Sub-Function: SWS_FILE_PDSSEND

This sub-function is being retired in favor of the enhanced **SWS_FILE_SEND** sub-function. In future, new enhancements will only be made to the enhanced sub-function.

We recommend that all new HLL application be written using the enhanced **SWS_FILE_SEND** function.

Previously written HLL programs need not be re-written; the Server will continue to support the deprecated **SWS_FILE_PDSSEND** interface as documented on this page.

CALL Arguments

The SWSFILE function takes six arguments. All six arguments must be specified.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	USAGE POINTER	PTR	INPUT	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	INPUT	Specify **SWS_FILE_PDSSEND** to invoke the deprecated interface as documented here. Refer to the current SWSFILE API for **all** other cases.
3	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	INPUT	A four-byte flag-word indicating options to be used in performing the requested sub-function. The following option flags can be specified, either singly, or in combination: • **SWS_SEND_TEXT** indicates the data to be sent exists in text format. It is translated by the server, during output, to ASCII. • **SWS_SEND_BINARY** indicates the data to be sent exists in binary format. It is transmitted by the server, as is. • **SWS_FILE_NOHTX** indicates the value indicates that HTML extension processing should not be performed during the output operation. This flag can be combined with the SWS_SEND_TEXT flag. • **SWS_FILE_HTX_REXXRULES** indicates the vale indicates that uninitialized variables located during HTML extension processing are replaced with the upper case variable name. If SWS_FILE_HTX_REXXRULES is not specified, uninitialized variables are replaced with a NULL string. This option can be combined with the SWS_SEND_TEXT flag.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
4	UCHAR *	PIC X(8)	CHAR(8)	INPUT	The 8-byte, blank padded DD name for the file. A shorter string can be specified if the string is null terminated.
5	UCHAR *	PIC X(8)	CHAR(8)	INPUT	The 8-byte, blank padded member name to be transmitted. A shorter string can be specified if the string is null terminated.
6	UCHAR *	PIC X(50)	CHAR(50)	INPUT	The 50-byte, blank padded MIME content type value to be used when the data is transmitted. A shorter string can be specified if the string is null terminated.

Return Values

SWSFILE always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR	A parameter validation error was found. The error will be logged to the wrap-around trace, and is available using the SWSERROR function.
SWS_NO_DATA_FOUND	Indicates that the DD name or PDS member name was invalid.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SWSERROR.
Any other value	The operation failed. Generally this indicates an unrecoverable loss of the communications session between the Shadow Web Server and the client's web browser program.

PL/I Example

```
DCL  SCONN  PTR;                    /* Connection Handle      */
DCL  SDDNA  CHAR(8) INIT('HTMFILE');/* File DD name           */
DCL  SMENA  CHAR(8) INIT('HLLFILE');/* Member name            */
DCL  SCOTY  CHAR(50) INIT('text/html');/* Content type        */
DCL  RC     FIXED BIN(31);          /* return code            */
DCL  DMHX   FIXED BIN(31) BASED;    /* Dummy Handle field     */

ADDR(SCONN)->DMHX = 0;              /* Clear Connection Handle*/

CALL SWSFILE( SCONN                 /* send the text data     */
              SWS_FILE_PDSSEND,
              SWS_SEND_TEXT,
              SDDNA,
              SMENA,
              SCOTY );
RC = PLIRETV();                     /* get return code        */
IF RC ^= SWS_SUCCESS THEN           /* exit program if bad RC */
   EXIT;
```

C Example

```
HDBC sConn   = NULL;                /* Connection Handle      */
long RC;                            /* return code            */

rc = SWSFILE( &sConn,               /* send the text data     */
              SWS_FILE_PDSSEND,
              SWS_SEND_TEXT,
              "HTMFILE",
              "HLLFILE",
              "text/html" );
if (rc ^= SWS_SUCCESS) return;      /* exit program if bad RC */
```

COBOL Example

```
77  SCONN              USAGE IS POINTER.
77  SDDNA              PIC X(8) VALUE 'HTMFILE'.
77  SMENA              PIC X(8) VALUE 'HLLFILE'.
7   SCOTY              PIC X(50)VALUE 'text/html'.


CALL 'SWCPFI' USING SCONN,
              SWS-FILE-PDSSEND,
              SWS-SEND-TEXT,
              SDDNA,
              SMENA,
              SCOTY.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK
```

SWSFILE Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level Language Interface available.

When used with Other REXX-language Interpreters SWSFILE built-in function can be used to transmit data with an external dataset directly to a web client. It can also be used to retrieve information about external datasets for the REXX-language procedure.

Syntax

The general form for a REXX-language invocation of SWSFILE is:

```
rc = SWSFILE( opertype, arg1, ... , argn )
```

The SWSFILE function incorporates several operation types. The first argument to the function call must be the name of the specific operation to be performed. The remaining arguments needed for the function call depend on which operation is being invoked.

Valid Operation Types

One of the following operation types must be coded as the first argument to the call (shown as opertype in the example above).

SEND	Transmit file-resident data to out-bound to a web client. This is a generic function which can operate upon PDS, PDSE, or QSAM datasets.
STATS	Return statistical information about a PDS member or a sequential dataset.
MBRLIST	Queue a list of PDS members to the REXX external data queue.
QUEUE	Read a PDS member or sequential dataset into the REXX external data queue. A return code value of **28** is returned if the external data queue is not sufficiently large to contain the entire file, or PDS(E) member. ***Note:*** You can increase the size of the external data queue using the QUEUESIZE( ) keyword).
PDSSEND	Send a PDS member out-bound to the web client.
PDSSTATS	Return statistical information about a PDS member.
PDSQUEUE	Read a PDS member into the REXX external data queue.

The remaining arguments which must be coded for each operation type is given below. A table, at the end of this page, explains how to code each of the arguments.

Whenever text format data is transmitted to a web client (for the **SEND** or **PDSSEND** operations) or is placed into the REXX external data queue (for the **QUEUE** or **PDSQUEUE** operations), the Server processes HTML Extension Statements to tailor the output. This facility allows the file-resident data to be customized at run-time.

Note:

The PDS-based operations (**PDSSEND**, **PDSSTATS**, and **PDSQUEUE** are legacy operations from an earlier release of the Server. They support *only* operations upon PDS datasets using a DD name specification. These legacy operations continue to be supported, however future enhancements will not be made to these operation types.

New REXX-language applications should use the enhanced operation types (**SEND**, **STATS**, and **QUEUE**) since they support both PDS and sequential dataset organizations and allow dataset names to be specified using either a DD name or fully-qualified dataset name.

The SEND Operation

SEND is used to transmit a member of a PDS dataset or a sequential dataset to the web client. The function will create and transmit an HTML response header, followed by the file-resident data.

Coding SEND Requests

To code the SWSFILE function call for SEND, use the following format:

```
zrc = SWSFILE("SEND",nametype,filename,mbrname,
              mimetype,datafmt,htxopt)
```

The arguments for the call are explained in the SWSFILE Arguments Table below.

Run-time Operation of SEND

SEND can operate on RECFM F, FB, V, VB, or U datasets. RECFM VB is suggested for binary data, and RECFM FB is suggested for text data.

When the PDS member or sequential dataset is transmitted out-bound, any record-length fields (such as are present for VB format records) are removed and only the raw data is transmitted.

The input file data can be ASCII binary data or EBCDIC text data.

If TEXT format data is specified, trailing EBCDIC blanks are removed from each data record before an ending CR (carriage return) character is appended. Each line is translated from EBCDIC to ASCII before transmission.

Unless NOHTX was explicitly specified on the function request, HTML Extension values are processed before the member is transmitted. When HTML Extensions are *actually* present within the input member, transmission of the `Last-Modified:` HTTP response header is suppressed. Instead, an `Expires:` header is generating giving the current time.

If ISPF-type statistics exist for the PDS member the ISPF last-modified data is used to generate the `Last-Modified:` HTML response header; otherwise, the `Last-modified:` HTML response header is not generated. The `Last-modified:` response header is surpressed if HTML Extension Statements are present and might cause the data to be tailored different for a future request.

SEND checks the HTTP transaction headers before actually transmitting the full PDS member outbound. If the in-bound HTTP transaction *method* is HEAD, only the HTML response header information is transmitted. Similarly, if the in-bound URL contained an `If-modified-since:` specification, the Server may have transmitted a "Not Modified" (304) response instead of sending the contents of the file.

The out-bound transmission generated by a successful SEND operation constitutes an entire and complete HTTP response. The REXX process which issues the SWSFILE function should not transmit additional data to the web client unless some error is returned by the function.

SEND Return Values

Return Value	Description
0	The function was completed successfully. The out-bound data stream was flushed from Web Server buffers.
4	The communciations link failed during out-bound transmission.
8	The specified member name does not exist within the PDS.
12	Either the specified DDNAME (the filename operand) is not allocated to the Shadow Web Server address space or the specified DSNAME does not exist.
16 or 20	An internal processing error, or abend condition.
24	The SECURITY(USERID) is in effect for the dataset and the effective userid does not have authority to the dataset.
28	The file exceeds the FILESATAGINGSIZELIMIT value set for pre-staging of data files. Normally such requests are re-driven automatically without pre-staging, but DDNAME format requests made from REXX procedures cannot be re-driven automatically and are rejected with this return code.

The STATS Operation

STATS is used to check the status of a PDS member or sequential dataset and return to the REXX procedure information about that member or sequential dataset. This information can be useful in dynamically creating transaction responses from within the REXX procedure.

Coding STATS Requests

```
zrc = SWSFILE("STATS",nametype,filename,mbrname,datafmt)
```

The arguments for the STATS operation are described in the SWSFILE Arguments Table below.

Run-Time Operation of STATS

STATS can operate of RECFM F,FB, V, VB, or U datasets.

STATS returns a content length, record count and last modification date value which corresponds to information which would be required to transmit the member out-bound to a web client.

STATS Return Values

For binary format data the member size or sequential dataset size returned will reflect an exact count of the data bytes within each logical record of the PDS member or sequential file.

For text format data the member or sequential file is treated as EBCDIC text. In calculating the member or sequential file size, trailing EBCDIC blanks are eliminated, and 1 additional byte per logical record (for the trailing carriage return character) is assumed. If a zero-length record is encountered within a RECFM=V or RECFM=VB file, a single blank is substituted for the zero-length record (zero-length records cannot be created in text files using ISPF, however, then can be present if the data has been uploaded to MVS via FTP).

STATS returns a NULL string if the dataset does not exist, or member does not exist within the specified PDS, or if the dataset cannot be accessed.

The STATS function returns a character string suitable for processing by REXX using blank-delimited word type processing.

The returned string contains the following information:

Word 1

Set to the value YES or NO to indicate whether or not ISPF statistics were present within the PDS directory for the member.

Word 2

Contains the integer value representing the length, in bytes, of the member or sequential dataset.

Word 3

Contains the count of logical records within the member or sequential dataset.

Word 4&ff

The remainder of the string contains the HTTP formatted Last-Modified: data and time stamp and is derived from the ISPF last-update time. This information is omitted if ISPF statistics are not present for the member.

The MBRLIST Operation

MBRLIST is used to obtain a list of the PDS members within a library. The list of member names is returned in the REXX external data queue. MBRLIST only operates on PDS or PDSE dataset types.

Coding MBRLIST

To code the SWSFILE function call for MBRLIST, use the following format:

```
zrc = SWSFILE("MBRLIST",nametype,filename)
```

The SWSFILE arguments for the MBRLIST operation are described in the SWSFILE Argument Table below.

Run-Time Operation of MBRLIST

MBRLIST only operations upon PDS or PDSE datasets. The MBRLIST function queues the list of PDS(E) members to the REXX external data queue. The list can be retrieved using the PARSE PULL REXX operation.

MBRLIST Return Values

Return Value	Description
0	The function was completed successfully. The member list isqueued.
12	Either the specified DDNAME (the filename operand) is not allocated to the Shadow Web Server address space or the specified DSNAME does not exist.
16 or 20	An internal processing error, or abend condition.
24	The SECURITY(USERID) is in effect for the dataset and the effective userid does not have authority to the dataset.
28	The external data queue is not sufficiently large to contain the entire member list. (**Note:** You can increase the size of the external data queue using the QUEUESIZE( ) keyword)

If no members exist within the PDS, the function returns with an empty external data queue and sets return code **0**.

The QUEUE Operation

QUEUE is used to read a member of a PDS dataset or a sequential file into the REXX external data queue. The QUEUE function is far more efficient than using EXECIO for a similar purpose, and allows you to reference a single DD name for all members of a PDS dataset.

The input source can be ASCII binary data or EBCDIC text data. If the input source is in text format, HTML Extensions within the source are processed before the source data is placed into the external queue.

Coding QUEUE

To code the SWSFILE function call for QUEUE, use the following format:

```
zrc = SWSFILE("QUEUE",nametype,filename,mbrname,datafmt,htxopt)
```

The SWSFILE arguments for the QUEUE operation are described in the SWS-FILE Argument Table below.

Run-Time Operation of QUEUE

When the PDS member or sequential dataset is read and copied to the external data queue any record-length fields (such as are present for VB format records)

are removed and only the raw data is copied. Trailing blanks are eliminated for TEXT format records. Additionally, for TEXT format records, if a zero-length VB record is read, a single blank is used for the corresponding queue entry (zero-length VB records can, and frequently are, created when members are shipped to an MVS system using FTP).

For binary format data no editing of any kind is applied to the data before it is copied to the external data queue. The data is copied to the queue as a continuous string of bytes with each queue entry being exactly as long as the corresponding source logical record.

For text format data each input record is stripped of trailing blanks before being copied to the external data queue.

Note:

The maximum logical record length which can be stored in a REXX external data queue entry is 652 bytes. Logical records longer than 623 bytes will be truncated.

After checking the return code from the QUEUE operation, you can retrieve the records from the external data queue with REXX coding, such as:

```
DO WHILE QUEUED() > 0
PARSE PULL nextlogicalrecord
END
```

QUEUE Return Values

Return Value	Description
0	The function was completed successfully. The data was copied to the external data queue.
8	The specified member name does not exist within the PDS.
12	Either the specified DDNAME (the filename operand) is not allocated to the Shadow Web Server address space or the specified DSNAME does not exist.
16 or 20	There was an internal processing error, or abend condition. The *most likely* cause of such an error is that the REXX external data queue has become full. The size of the REXX external data queue is fixed, but can be altered for Web Transaction programs by changing the SEFMAXQUEUE start-up parameter value.
24	The SECURITY(USERID) is in effect for the dataset and the effective userid does not have authority to the dataset.
28	The external data queue is not sufficiently large to contain the entire file, or PDS(E) member. (**Note:** You can increase the size of the external data queue using the QUEUESIZE( ) keyword).

The PDSSEND Operation

PDSSEND is a legacy operation type from an earlier release of the Server. Equivalent operations can be requested using the SEND request, which is the preferred method for new applications.

Coding PDSSEND Requests

To code SWSFILE function calls using PDSSEND, use the following format:

```
zrc = SWSFILE("PDSSEND",filename,mbrname,mimetype,datafmt,htxopt)
```

Note:

This operation type must supply a DD name as the filename argument. Use of a fully-qualified dataset name is not supported for this interface.

Also, note that the MVS dataset referred to must be either a PDS or PDSE dataset. Access to sequential datasets is not supported.

Run-Time Operation of PDSSEND

The operation of PDSSEND is the same as for the SEND request, shown above.

PDSSEND Return Values

Return values set by PDSSEND are the same as for the SEND request, shown above.

The PDSSTAT Operation

PDSSTAT is a legacy operation type from an earlier release of the Server. Equivalent operations can be requested using the STATS request, which is the preferred method for new applications.

Coding PDSSTAT Requests

To code SWSFILE function calls using PDSSTAT, use the following format:

```
zrc = SWSFILE("PDSSTAT",filename,mbrname,datafmt)
```

Note:
This operation type must supply a DD name as the filename argument. Use of a fully-qualified dataset name is not supported for this interface.

Also, note that the MVS dataset referred to must be either a PDS or PDSE dataset. Access to sequential datasets is not supported.

Run-Time Operation of PDSSTAT

The operation of PDSSTAT is the same as for the STATS request, shown above.

PDSSTAT Return Values

Return values set by PDSSTAT are the same as for the STATS request, shown above.

The PDSQUEUE Operation

PDSQUEUE is a legacy operation type from an earlier release of the Server. Equivalent operations can be requested using the QUEUES request, which is the preferred method for new applications.

Coding PDSQUEUE Requests

To code SWSFILE function calls using PDSQUEUE, use the following format:

```
zrc = SWSFILE("PDSQUEUE",filename,mbrname,datafmt,htxopt)
```

Note:

This operation type must supply a DD name as the filename argument. Use of a fully-qualified dataset name is not supported for this interface.

Also, note that the MVS dataset referred to must be either a PDS or PDSE dataset. Access to sequential datasets is not supported.

Run-Time Operation of PDSQUEUE

The operation of PDSQUEUE is the same as for the QUEUE request, shown above.

Return Values

Return values set by PDSQUEUE are the same as for the QUEUE request, shown above.

ArgumentsTable

The table which follows describes each of the arguments which can be supplied for an SWSFILE function call.

Argument Name given in Syntax Examples	Description of Argument
nametype	Indicates whether the filename argument on the function call is an MVS DD name or a fully-qualified dataset name. The nametype argument must be one of the following: • DDNAME or DD indicates that the filename argument is a DD name value. The DD name must be allocated to the Shadow Web Server address space. • DSNAME or DSN indicates that the filename argument is a fully-qualified MVS dataset name.
Filename	Specifies the dataset to be operated upon by the function call. The dataset specification can be made using either an MVS DD name value or as a fully-qualified dataset name. must be a PDS, PDSE, or sequential dataset. • DD name values must be given as an 8-byte, blank-padded string. • Fully-qualified dataset names must be given as a 44-byte, blank-padded string. • When a DD name is used, it must be pre-allocated to the Server address space. • You specify whether a DD name or dataset name is being passed on the call using the nametype argument shown above. • For the legacy operations (PDSSEND, PDSSTATS, and PDSQUEUE) only an 8-byte DD name is permitted. The DD name must reference a PDS or PDSE dataset.
mbrname	The 8-byte, blank padded member name to be transmitted from a PDS or PDSE dataset. This argument must be specified when the SWSFILE operation refers to a PDS or PDSE dataset. It should be omitted when a sequential dataset is referred to (code a single comma in place of the argument).

Argument Name given in Syntax Examples	Description of Argument
mimetype	The 50-byte, blank padded MIME content type value to be used when the data is transmitted. The value coded here is used to generate the Content-type: out-bound HTML response header. It can be set to any value, but would normally be set to 'image/gif', 'Text/HTML' or some other widely known value. The server does not perform a validity test upon this argument. If this argument is omitted, the Server assumes a default value of 'text/plain'.
datafmt	The format of the data contained within the file to be transmitted. Valid values are: • **TEXT** indicates that the data to be sent exists in text format. It is translated by the server, during output, to ASCII, trailing blanks are stripped, and a carriage return character is appended to each line. HTML Extension Statements, if any, within the data are processed by the Server. • **BINARY** indicates that the data to be sent exists in binary format. It is transmitted by the server, as is. If this argument is omitted, the Server uses BINARY as the default value.
htxopt	Indicate how (or if) HTML Extension Statements within the file data are processed by the Server before output. This argument should be omitted for binary format files, and is optional when text format data is processed. Valid values are: • **HTX** indicates that HTML extension processing should be performed during the output operation. • **NOHTX** indicates that HTML extension processing should not be performed during the output operation. • **HTXREXXRULES** indicates that un-initialized variables located during HTML extension processing are replaced with the upper-case variable name. Normal extension processing replaces un-initialized variables with a NULL string. If this argument is omitted, the Server assumes a value of HTX for all text format files.

The SWSFILE Function with Other REXX-language Interpreters

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SWCPFI**.

When used with Other REXX-language Interpreters SWSFILE built-in function can be used to transmit data with an external dataset directly to a web client. It can also be used to retrieve information about external datasets for the REXX-language procedure.

SWSFILE Syntax

The general form for a REXX-language invocation of SWSFILE is:

```
rc = SWSFILE( opertype, arg1, ... , argn )
```

The SWSFILE function incorporates several operation types. The first argument to the function call must be the name of the specific operation to be performed. The remaining arguments needed for the function call depend on which operation is being invoked.

Valid Operation Types

One of the following operation types must be coded as the first argument to the call (shown as **`opertype`** in the example above).

SEND	Transmit file-resident data to out-bound to a web client. This is a generic function which can operate upon PDS, PDSE, or QSAM datasets.
STATS	Return statistical information about a PDS member or a sequential dataset.

The remaining arguments which must be coded for each operation type is given below. A table, at the end of this page, explains how to code each of the arguments.

Whenever text format data is transmitted to a web client (for the **SEND** or **PDSSEND** operations) or is placed into the REXX external data queue (for the **QUEUE** or **PDSQUEUE** operations), the Server processes HTML Extension

Statements to tailor the output. This facility allows the file-resident data to be customized at run-time.

Note:

The PDS-based operations (**PDSSEND**, **PDSSTATS**, and **PDSQUEUE** are legacy operations from an earlier release of the Server. They support *only* operations upon PDS datasets using a DD name specification. These legacy operations continue to be supported, however future enhancements will not be made to these operation types.

New REXX-language applications should use the enhanced operation types (**SEND**, **STATS**, and **QUEUE**) since they support both PDS and sequential dataset organizations and allow dataset names to be specified using either a DD name or fully-qualified dataset name.

The SEND Operation

SEND is used to transmit a member of a PDS dataset or a sequential dataset to the web client. The function will create and transmit an HTML response header, followed by the file-resident data.

Coding SEND Requests

To code the SWSFILE function call for SEND, use the following format:

```
zrc = SWSFILE("SEND",nametype,filename,mbrname,mimetype,
      datafmt,htxopt)
```

The arguments for the call are explained in the SWSFILE Arguments Table below.

Run-time Operation of SEND

SEND can operate on RECFM F, FB, V, VB, or U datasets. RECFM VB is suggested for binary data, and RECFM FB is suggested for text data.

When the PDS member or sequential dataset is transmitted out-bound, any record-length fields (such as are present for VB format records) are removed and only the raw data is transmitted.

The input file data can be ASCII binary data or EBCDIC text data.

If TEXT format data is specified, trailing EBCDIC blanks are removed from each data record before an ending CR (carriage return) character is appended. Each line is translated from EBCDIC to ASCII before transmission.

Unless NOHTX was explicitly specified on the function request, HTML Extension values are processed before the member is transmitted. When HTML Extensions are *actually* present within the input member, transmission of the `Last-`

`Modified:` HTTP response header is suppressed. Instead, an `Expires:` header is generating giving the current time.

If ISPF-type statistics exist for the PDS member, the ISPF last-modified data is used to generate the `Last-Modified:` HTML response header; otherwise, the `Last-modified:` HTML response header is not generated. The `Last-modified:` response header is surpressed if HTML Extension Statements are present and might cause the data to be tailored different for a future request.

SEND checks the HTTP transaction headers before actually transmitting the full PDS member outbound. If the in-bound HTTP transaction *method* is HEAD, only the HTML response header information is transmitted. Similarly, if the in-bound URL contained an `If-modified-since:` specification, the Server may have transmitted a "Not Modified" (304) response instead of sending the contents of the file.

The out-bound transmission generated by a successful SEND operation constitutes an entire and complete HTTP response. The REXX process which issues the SWSFILE function should not transmit additional data to the web client unless some error is returned by the function.

SEND Return Values

Return Value	Description
0	The function was completed successfully. The out-bound data stream was flushed from Web Server buffers.
4	The communciations link failed during out-bound transmission.
8	The specified member name does not exist within the PDS.
12	Either the specified DDNAME (the filename operand) is not allocated to the Shadow Web Server address space or the specified DSNAME does not exist.
16 or 20	An internal processing error, or abend condition.
24	The SECURITY(USERID) is in effect for the dataset and the effective userid does not have authority to the dataset.
28	The file exceeds the FILESATAGINGSIZELIMIT value set for pre-staging of data files. Normally such requests are re-driven automatically without pre-staging, but DDNAME format requests made from REXX procedures cannot be re-driven automatically and are rejected with this return code.

The STATS Operation

STATS is used to check the status of a PDS member or sequential dataset and return to the REXX procedure information about that member or sequential dataset. This information can be useful in dynamically creating transaction responses from within the REXX procedure.

Coding STATS Requests

```
zrc = SWSFILE("STATS",nametype,filename,mbrname,datafmt)
```

The arguments for the STATS operation are described in the SWSFILE Aruguments Table below.

Run-Time Operation of STATS

STATS can operate of RECFM F,FB, V, VB, or U datasets.

STATS returns a content length, record count and last modification date value which corresponds to information which would be required to transmit the member out-bound to a web client.

STATS Return Values

For binary format data, the member size or sequential dataset size returned will reflect an exact count of the data bytes within each logical record of the PDS member or sequential file.

For text format data, the member or sequential file is treated as EBCDIC text. In calculating the member or sequential file size, trailing EBCDIC blanks are eliminated, and one additional byte per logical record (for the trailing carriage return character) is assumed. If a zero-length record is encountered within a RECFM=V or RECFM=VB file, a single blank is substituted for the zero-length record (zero-length records cannot be created in text files using ISPF, however, then can be present if the data has been uploaded to MVS via FTP).

STATS returns a NULL string if the dataset does not exist, or member does not exist within the specified PDS, or if the dataset cannot be accessed.

The STATS function returns a character string suitable for processing by REXX using blank-delimited word type processing.

The returned string contains the following information:

Word 1

Set to the value YES or NO to indicate whether or not ISPF statistics were present within the PDS directory for the member.

Word 2

Contains the integer value representing the length, in bytes, of the member or sequential dataset.

Word 3

Contains the count of logical records within the member or sequential dataset.

Word 4&ff

The remainder of the string contains the HTTP formatted `Last-Modified:` data and time stamp and is derived from the ISPF last-

update time. This information is omitted if ISPF statistics are not present for the member.

Arguments Table

The table which follows describes each of the arguments which can be supplied for an SWSFILE function call.

Argument Name given in Syntax Examples	Description of Argument
nametype	Indicates whether the filename argument on the function call is an MVS DD name or a fully-qualified dataset name. The nametype argument must be one of the following: • DDNAME or DD indicates that the filename argument is a DD name value. The DD name must be allocated to the Shadow Web Server address space. • DSNAME or DSN indicates that the filename argument is a fully-qualified MVS dataset name.
filename	Specifies the dataset to be operated upon by the function call. The dataset specification can be made using either an MVS DD name value or as a fully-qualified dataset name. must be a PDS, PDSE, or sequential dataset. • DD name values must be given as an 8-byte, blank-padded string. • Fully-qualified dataset names must be given as a 44-byte, blank-padded string. • When a DD name is used, it must be pre-allocated to the Server address space. • You specify whether a DD name or dataset name is being passed on the call using the nametype argument shown above. • For the legacy operations (PDSSEND, PDSSTATS, and PDSQUEUE) only an 8-byte DD name is permitted. The DD name must reference a PDS or PDSE dataset.
mbrname	The 8-byte, blank padded member name to be transmitted from a PDS or PDSE dataset. This argument must be specified when the SWSFILE operation refers to a PDS or PDSE dataset. It should be omitted when a sequential dataset is referred to (code a single comma in place of the argument).
mimetype	The 50-byte, blank padded MIME content type value to be used when the data is transmitted. The value coded here is used to generate the Content-type: out-bound HTML response header. It can be set to any value, but would normally be set to 'image/gif', 'Text/HTML' or some other widely known value. The server does not perform a validity test upon this argument. If this argument is omitted, the Server assumes a default value of 'text/plain'.
datafmt	The format of the data contained within the file to be transmitted. Valid values are: • **TEXT** indicates that the data to be sent exists in text format. It is translated by the server, during output, to ASCII, trailing blanks are stripped, and a carriage return character is appended to each line. HTML Extension Statements, if any, within the data are processed by the Server. • **BINARY** indicates that the data to be sent exists in binary format. It is transmitted by the server, as is. If this argument is omitted, the Server uses BINARY as the default value.

Argument Name given in Syntax Examples	Description of Argument
htxopt	Indicate how (or if) HTML Extension Statements within the file data are processed by the Server before output. This argument should be omitted for binary format files, and is optional when text format data is processed. Valid values are: • **HTX** indicates that HTML extension processing should be performed during the output operation. • **NOHTX** indicates that HTML extension processing should not be performed during the output operation. • **HTXREXXRULES** indicates that un-initialized variables located during HTML extension processing are replaced with the upper-case variable name. Normal extension processing replaces un-initialized variables with a NULL string. If this argument is omitted, the Server assumes a value of HTX for all text format files.

High-Level Language Interface SWSSET (SWCPSO) Function

✔	Can be used in Shadow/REXX. See RETURN RESCAN or RETURN FLUSH.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SWCPSO.**

The SWSSET built-in function allows the caller to control various runtime environmental options. Following is a list of functions that can be specified:

Function Name	Description
SWS_OPTION_RESCAN	Provides a new URL value which the server uses during the rescan operation. The rescan operation does not actually begin until *after* the current application program terminates normally. The rescan function provides high-level language programs with an equivalent functionality which the Shadow/REXX Interpreter intrinsically provides. For Shadow/REXX, the equivalent function is performed by using the RESCAN option of the RETURN statement.
SWS_OPTION_FLUSH	Allows the user to send the current "buffered" output. This function normally occurs *after* the current application program terminates normally.
SWS_OPTION_NOFLUSH	Allows the user to inhibit the sending of the "buffered" output. This function occurs *after* the current application program terminates normally.
SWS_OPTION_SENDTRACE	Allows the user to specify the output tracing option.
SWS_OPTION_PARSETRACE	Allows the user to specify the URL tracing option.
SWS_OPTION_RESPMODE	Allows the user to specify the server parsing mode.
SWS_OPTION_DPRTY	Allows the user to adjust the current processing priority of their TCB within the Shadow Server address space. This can be a positive or negative whole number which is added to the TCB's current dispatching priority.
SWS_OPTION_AUTOFLUSH	Allows the user to specify the maximum number of buffers to allow before automatically sending them. Value must be specified as a whole number between 0 and 32767.
SWS_OPTION_MAXRESPBYTES	Allows the user to specify the maximum number of bytes which can be sent in response to a single URL. Value must be specified as a whole number between 0 and 2147483647.
SWS_OPTION_MASRESPBUFFERS	Allows the user to specify the maximum number of buffers which can be sent in response to a single URL. Value must be specified as a whole number between 0 and 32767.
SWS_OPTION_USERDATA1	Allows the user to specify information to be added to the Shadow Web Server SMF record.

Function Name	Description
SWS_OPTION_USERDATA2	Allows the user to specify information to be added to the Shadow Web Server SMF record. This option can only be executed from a WWW Master rule.
SWS OPTION ASCIIEBCDICMAP	Allows the user to specify a language conversion value to be used for translating any subsequent output generated by this rule.

CALL Arguments

The SWSSET function takes four arguments. All four arguments must be specified on the call.

Arg	HLL Argument Type			I/O	Description of Argument	
	C	COBOL	PL/I			
1	HDBC	Usage pointer	PTR	Input	The Web Server connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).	
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A flag word indicating the function which the SWSSET invocation should perform. You must specify one of the aforementioned functions for this parameter.	
3	CHAR*	PIC X(8)	CHAR(8)	OUTPUT	The data value to be set. The format of this argument varies, depending on the value supplied for the second argument.	
					Function	**Value**
					SWS_OPTION_ RESCAN	Specify a 1-to-128 byte URL rescan value. The rescan URL value can be a null terminated string.
					SWS_OPTION_ FLUSH	Parameter not required.
					SWS_OPTION_ NOFLUSH	Parameter not required.
					SWS_OPTION_ SENDTRACE	• **YES** to turn the option on. • **NO** to turn the option off.
					SWS_OPTION_ RESPMODE	• **NONE** for Non-parsed headers. • **SERVER** for Server-parsed headers.
					SWS_OPTION_ DPRTY	A value from -255 to 255.

Arg	HLL Argument Type			I/O	Description of Argument	
	C	COBOL	PL/I			
					SWS_OPTION_ AUTOFLUSH	A value from 0 to 32767.
					SWS_OPTION_ MAXRESPBYTES	A value from 0 to 2147483647.
					SWS_OPTION_ MAXRESPBUFFERS	A value from 0 to 32767.
					SWS_OPTION_ PARSETRACE	• **YES** to turn the option on. • **NO** to turn the option off.
					SWS_OPTION_ USERDATA1	Any text string from 1 to 256 bytes long
					SWS_OPTION_ USERDATA2	Any text string from 1 to 256 bytes long
					SWS_OPTION_ ASCIIEBCDICMAP	Any valid 3 or 4 character language code string
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument specifies the size of the function data given by the third argument. The required size varies depending on the function specified by the second argument. If the data specified as parameter 3 is a null-terminated string, you can specify the manifest constant SWS_NTS.	

The tape will contain a serial number of the form “NSnnnn” on its external label. Use this label in the JCL above and in the SSINSTAL job below.

Return Values

SWSSET always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	The operation succeeded.
SWS_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_INVALID_HANDLE	The connection handle is invalid. No error information is available.

Return Value	Description
Any other value	The operation failed. By using the SQLERROR/SWSERROR interface, you can obtain the error message pertinent to the error.

SWS_OPTION_ASCIIEBCDICMAP Language Codes

The following list of language codes are used to translate text.

Code	Lanuguage
BEL	BELGIAN
CBL	CANADIAN BILINGUAL
DAN	DANISH (MS)
DAN2	DANISH/NORWEGIAN
DEU	GERMAN (MS)
DEU2	AUSTRIAN/GERMAN
ENG	U.K. ENGLISH (MS)
ENG2	U.K. ENGLISH
ENU	U.S. ENGLISH
ENU2	U.S. ENGLISH (ORIG. SWS VERSION)
ESN	MODERN SPANISH (MS)
ESP	CASTILIAN SPANISH (MS)
ESP2	SPANISH
FIN	FINISH (MS)
FIN2	FINISH/SWEDISH
FRA	FRENCH (MS)
FRA2	FRENCH
FRC	CANADIAN FRENCH
ISL	ICELANDIC (MS)
ITA	ITALIAN (MS)
ITA2	ITALIAN
JPE	JAPANESE/ENGLISH
NLD	DUTCH (MS)
NLD2	DUTCH
NOR	NORWEGIAN (MS)
PTG	PORTUGUESE (MS)

Code	Lanuguage
PTG2	PORTUGUESE
SVE	SWEDISH (MS)
SWF	SWISS/FRENCH
SWG	SWISS/GERMAN

PL/I Example

```
DCL TCONN PTR;                    /* Connection Handle          */
DCL TDATA CHAR(256);              /* data buffer area           */
DCL TSIZE FIXED BIN(31);          /* data length                */
DCL RC FIXED BIN(31);             /* return code                */
DCL DMHX FIXED BIN(31) BASED;     /* Dummy Handle field         */

ADDR(TCONN)->DMHX = 0;            /* Clear Connection Handle    */

TDATA = 'SYSTEM/ERROR/500';       /* Set rescan URL value       */
TSIZE = 16;                       /* set length                 */
CALL SWSSET( TCONN                /* Set rescan URL value       */
             SWS_OPTION_RESCAN,
             TDATA,
             TSIZE );

RC = PLIRETV();                   /* get return code            */
IF RC ^= SWS_SUCCESS THEN         /* if bad RC, then go         */
 GOTO ERROR_LABEL;                /* report the error or        */
                                  /* abort the transaction      */

CALL PLIRETC(0);                  /* Rescan URL is set up, so   */
RETURN(0);                        /* let Server do re-match     */

Other function call formats:
CALL SWSSET( TCONN
             SWS_OPTION_FLUSH);
CALL SWSSET( TCONN
             SWS_OPTION_NOFLUSH);
TSIZE = 3;
CALL SWSSET( TCONN,
             SWS_OPTION_SENDTRACE,
             SWS_OPTION_YES,
             TSIZE );
TSIZE = 2;
CALL SWSSET( TCONN,
             SWS_OPTION_PARSETRACE,
             SWS_OPTION_NO,
             TSIZE );
TSIZE = 4;
CALL SWSSET( TCONN,
```

```
            SWS_OPTION_RESPMODE,
            SWS_OPTION_NONE,
            TSIZE );
TDATA = '5'
TSIZE = 1;
CALL SWSSET( TCONN,
            SWS_OPTION_DPRTY,
            TDATA, TSIZE );

TDATA = '100'
TSIZE = 3;
CALL SWSSET( TCONN,
            SWS_OPTION_AUTOFLUSH,
            TDATA,
            TSIZE );
TDATA = '32000'
TSIZE = 5; CALL SWSSET( TCONN,
                        SWS_OPTION_MAXRESPBYTES,
                        TDATA,
                        TSIZE );
TDATA = '55';
TSIZE = 2;
CALL SWSSET( TCONN,
            SWS_OPTION_MAXRESPBUFFERS,
            TDATA,
            TSIZE );
TDATA = 'USER DATA MESSAGE';
TSIZE = 17;
CALL SWSSET( TCONN,
            SWS_OPTION_USERDATA1,
            TDATA,
            TSIZE );
TDATA = 'SECURED USER DATA MESSAGE';
TSIZE = 25;
CALL SWSSET( TCONN,
            SWS_OPTION_USERDATA2,
            TDATA,
            TSIZE );
TDATA = 'FRA2'
TSIZE = 4; CALL SWSSET( TCONN,
                        SWS_OPTION_ASCIIEBCDICMAP,
                        TDATA,
                        TSIZE );
```

C Example

```
HDBC tConn = NULL;                    /* Connection Handle       */
char tData[] = "SYSTEM/ERROR/500";  /* rescan URL value        */
long RC;                              /* return code             */
rc = SWSSet( &tConn,                  /* set rescan URL value    */
             SWS_OPTION_RESCAN,
             tdata,
             strlen(tdata) );
if (rc ^= SWS_SUCCESS) return;        /* if bad RC from SWSSET   */
{                                     /* perform some appl-dep.  */
...                                   /* error recovery or abort */
};                                    /* processing.             */
else                                  /* Otherwise, URL is set so*/
return 0;                             /* let Server do re-match  */

Other function call formats:
rc = SWSSet( &tConn,
             SWS_OPTION_FLUSH);
rc = SWSSet( &tConn,
             SWS_OPTION_NOFLUSH);
rc = SWSSet( &tConn,
             SWS_OPTION_SENDTRACE,
             SWS_OPTION_NO,
             strlen(SWS_OPTION_NO));
rc = SWSSet( &tConn,
             SWS_OPTION_PARSETRACE,
             SWS_OPTION_NO,
             strlen(SWS_OPTION_NO));
rc = SWSSet( &tConn,
             SWS_OPTION_RESPMODE,
             SWS_OPTION_SERVER,
             strlen(SWS_OPTION_SERVER));
char tData[] = "-40";
rc = SWSSet( &tConn,
             SWS_OPTION_DPRTY,
             tdata,
             strlen(tdata) );
char tData[] = "50";
rc = SWSSet( &tConn,
             SWS_OPTION_AUTOFLUSH,
             tdata,
             strlen(tdata) );
char tData[] = "2000000";
rc = SWSSet( &tConn,
             SWS_OPTION_MAXRESPBYTES,
             tdata,
             strlen(tdata) );
char tData[] = "80";
rc = SWSSet( &tConn,
             SWS_OPTION_MAXRESPBUFFERS,
             tdata,
             strlen(tdata) );
char tData[] = "USER DATA MESSAGE";
```

```
rc = SWSSet( &tConn,
             SWS_OPTION_USERDATA1,
             tdata,
             strlen(tdata) );
char tData[] = "SECURED USER DATA MESSAGE";
rc = SWSSet( &tConn,
             SWS_OPTION_USERDATA2,
             tdata,
             strlen(tdata) );
char tData[] = "FRA2";
rc = SWSSet( &tConn,
             SWS_OPTION_ASCIIEBCDICMAP,
             tdata,
             strlen(tdata) );
```

COBOL Example

```
77 TCONN                                                    USAGE IS
POINTER.
77 TDATA                                                  PIC X(128).
77 TSIZE                                                    PIC S9(5)
COMP.

MOVE 'SYSTEM/ERROR/500'                                     TO TDATA.
MOVE 16                                                     TO TSIZE.
CALL 'SWCPSO' USING TCONN,
             SWS-OPTION-RESCAN,
             TDATA,
             TSIZE.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS PERFORM 9999-ABORT-PROCEDURE.
MOVE 0 to RETURN-CODE.
GOBACK.
Other function call formats:
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-FLUSH.
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-NOFLUSH.
MOVE 3                                                      TO TSIZE.
CALL 'SWCPSO' USING TCONN,
                    SWS-OPTION-SENDTRACE,
                    SWS-OPTION-YES,
                    TSIZE.
MOVE 2                                                      TO TSIZE.
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-PARSETRACE,
           SWS-OPTION-NO,
           TSIZE.
MOVE 4                                                      TO TSIZE.
CALL 'SWCPSO'
     USING TCONN,
            SWS-OPTION-RESPMODE,
            SWS-OPTION-NONE,
            TSIZE.
MOVE '-17'                                                  TO TDATA.
MOVE 3                                                      TO TSIZE.
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-DPRTY,
           TDATA,
           TSIZE.
MOVE '100'                                                  TO TDATA.
MOVE 3                                                      TO TSIZE.
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-AUTOFLUSH,
```

```
           TDATA,
           TSIZE.
MOVE '100000'                                          TO TDATA.
MOVE 6                                                 TO TSIZE.
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-MAXRESPBYTES,
           TDATA,
           TSIZE.
MOVE '255'                                             TO TDATA.
MOVE 3                                                 TO TSIZE.
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-MAXRESPBUFFERS,
           TDATA,
           TSIZE.
MOVE 'USER DATA MESSAGE'                                    TO
TDATA.
MOVE 17                                                TO TSIZE.
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-USERDATA1,
           TDATA,
           TSIZE.
MOVE 'SECURED USER DATA MESSAGE'                            TO
TDATA.
MOVE 25                                                TO TSIZE.
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-USERDATA2,
           TDATA,
           TSIZE.
MOVE 'FRA2'                                            TO TDATA.
MOVE 4                                                 TO TSIZE.
CALL 'SWCPSO'
     USING TCONN,
           SWS-OPTION-ASCIIEBCDICMAP,
           TDATA,
           TSIZE.
```

SWSSET Function

✔	Can be used in Shadow/REXX. See RETURN RESCAN or RETURN FLUSH.
✔	Can be used from other REXX interpreters.
✔	High-level language interface available.

The SWSSET built-in function allows the caller to control various runtime environmental options. The following is a list of functions that can be specified:

Function Name	Description	Subparameter
RESCAN	Provides a new URL value which the Server uses during the rescan operation. The rescan operation does not actually begin until *after* the current application program terminates normally.	Specify a 1-to-128 byte URL rescan value.
FLUSH	Provides a new URL value which the Server will use during the rescan operation. The rescan operation does not actually begin until *after* the current application program terminates normally.	This function has no sub-parameters.
NOFLUSH	Forces any scheduled output to be sent to the user and terminates any further processing.	This function has no sub-parameters.
SENDTRACE	Allows the user to specify the output tracing option.	• **YES** to turn the option on. • **NO** to turn the option off.
PARSETRACE	Allows the user to specify the URL tracing option.	• **YES** to turn the option on. • **NO** to turn the option off.
RESPMODE	Allows the user to specify the server parsing mode.	• **NONE** for Non-parsed headers. • **SERVER** for Server-parsed headers.
DPRTY	Allows the user to adjust the current processing priority of their TCB within the Shadow Server address space. This can be a positive or negative whole number which is added to the TCB's current dispatching priority.	A value from -255 to 255.
AUTOFLUSH	Allows the user to specify the maximum number of buffers to allow before automatically sending them.	A value from 0 to 32767.
MAXRESPBYTES	Allows the user to specify the maximum number of bytes that can be sent in response to a single URL.	A value from 0 to 2147483647.
MAXRESPBUFFERS	Allows the user to specify the maximum number of buffers that can be sent in response to a single URL.	A value from 0 to 32767.

Function Name	Description	Subparameter
USERDATA1	Allows the user to specify a text sting to be inserted in the Shadow Web Server SMF record.	Must be a text string from 1 to 256 bytes long.
USERDATA2	Allows the user to specify a text sting to be inserted in the Shadow Web Server SMF record. This string of text is secured in as much as it can only be specified in a WWW Master rule.	Must be a text string from 1 to 256 bytes long.

Return Values

The function always returns 0 (zero) to the caller.

SWSSET always sets a signed numeric return code value. Possible values are:

Return Value	Description
0	The function succeeded.
1	The API call cannot "connect back" into the subsystem code properly.

Coding Samples

```
rc = SWSSET( 'RESCAN', newurl )

rc = SWSSET( 'FLUSH' )

rc = SWSSET( 'NOFLUSH' )

rc = SWSSET( 'SENDTRACE', 'YES' )

rc = SWSSET( 'PARSETRACE', 'YES' )

rc = SWSSET( 'RESPMODE', 'SERVER' )

rc = SWSSET( 'DPRTY', '-2' )

rc = SWSSET( 'AUTOFLUSH', '25' )

rc = SWSSET( 'MAXRESPBYTES', '10000' )

rc = SWSSET( 'MAXRESPBUFFERS' '100' )

rc = SWSSET( 'USERDATA1' 'USER MESSAGE DATA' )

rc = SWSSET( 'USERDATA2' 'SECURED USER MESSAGE DATA' )

rc = SWSSET( 'ASCIIEBCDICMAPPING' 'FRA2' )
```

High-Level Language Interface SWSWTO (SWCPWT) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SWCPWT.**

The SWSWTO function allows a message to be written to the MVS operator console. Optionally, a route code can be supplied. If a zero route code is coded, the default will be used. The route code is one of four constants, which are described below. Each constant determines a set of route and descriptor codes, which is described in the IBM publication - - *Assembler Services Reference*.:

CALL Arguments

The SWSWTO function takes four arguments. All four arguments must be specified on the call.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The Web Server connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	PTR	PIC X(nnn)	CHAR(nnn)	Input	The message passed to the WTO service. You can specify a null terminated string, or explicitly provide the value length via the third argument. The maximum length is 70 bytes.
3	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the data value given by the second argument passed to WTO. You can optionally specify SWS_NTS to indicate the data is a null terminated string.
4	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument is one of the following constants: SWS-WTO-INFO rtcde(1)desc(4,9) SWS-WTO-WARN rtcde(1,11)desc(4,9) SWS-WTO-SEVERE rtcd(1,11)desc(1,11) SWS-WTO-HARDCOPY none

Return Values

SWSWTO always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	The operation succeeded. The specified data was written to the product's wrap-around trace.
SWS_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_INVALID_HANDLE	The connection handle is invalid. No error information is available.
Any other value	The operation failed.

PL/I Example

```
DCL  TCONN     PTR;                 /* Connection Handle       */
DCL  TDATA     CHAR(70);            /* Text output area        */
DCL  TSIZE     FIXED BIN(31);       /* Text length area        */
DCL  RC        FIXED BIN(31);       /* return code             */
DCL  DMHX      FIXED BIN(31) BASED; /* Dummy Handle field      */


ADDR(TCONN)->DMHX = 0;              /* Clear Connection Handle */

TDATA    = 'WTO Message Text';      /* Set output area         */
TSIZE    = 16;                      /* set length              */
CALL SWSWTO( TCONN                  /* output trace message    */
             TDATA,
             TSIZE,
             SWS-WTO-INFO );
RC = PLIRETV();                     /* get return code         */
IF RC ^= SWS_SUCCESS THEN           /* exit program if bad RC  */
       EXIT;
```

C Example

```
HDBC tConn   = NULL;                /* Connection Handle       */
char tData[] = "Null-terminated!";  /* Text string definition  */
long RC;                            /* return code             */

rc = SWSWto( &tConn,                /* output trace message    */
              tData,
              SWS_NTS,
              SWS-WTO-INFO );
if (rc ^= SWS_SUCCESS) return;      /* exit program if bad RC  */
```

COBOL Example

```
77  TCONN                USAGE IS POINTER.
77  TDATA                PIC X(70).
77  TSIZE                PIC S9(5) COMP.
77  DESC                 PIC S9(5) COMP VALUE 12.
77  ROUTE                PIC S9(5) COMP VALUE 1.

MOVE 'WTO MESSAGE'   TO TDATA.
MOVE 11              TO TSIZE.

CALL 'SWCPWT' USING TCONN,
                    TDATA,
                    TSIZE,
                    SWS-WTO-INFO.

MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.
```

SWSWTO Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level language interface available.

The SWSWTO built-in function provides a means to issue an MVS write to operator.

The SWSWTO function allows a message to be written to the MVS operator console. Optionally, a route code may be supplied. If a zero route code is supplied, the default will be used. The route code is one of four string constants, which aredescribed below. Each constant determines a set of route and descriptor codes, which is described in the IBM publication - - *Assembler Services Reference.*

Coding SWSWTO

To code the SWSWTO function, use the following format:

```
RC = SWSWTO( textstring ,<route code>)
```

Valid Route Codes

Route Code	Description
Informational	desc codes 4,9 route code 1
Warning	desc codes 4,9 route codes 1,11
Severe	desc codes 2,11 route codes 2,11
Hardcopy	None
Route Code	Description

Return Values

The function returns 0 (zero) if successful and non-zero if the WTO failed.

RPC Direct Host APIs

This section covers the following RPC Direct APIs:

API Description	DIRECT	WEB	SEF	WEB/RX
RPC Direct APIs				
To access current execution environment information:	sdcpif			
To add text message to trace browser log:	sdcpmg			
To read buffer of data from client:	sdcprd			
To send buffer of data to client:	sdcpwr			

Host RPCs can use a variety of APIs provided by Shadow Server to communicate with the Shadow Server address space. These APIs are used to:

- Transmit data to and from the client.
- Add messages to Trace Browse.
- Obtain additional information about the execution environment.

Host RPCs can be used in any high-level language. The same routines are provided for all languages. All of the host RPC API functions can be invoked by either AMODE 24 or AMODE 31 callers. All data areas passed to these functions can either be above or below the 16 MB line. All functions accept a fixed number of arguments passed using an OS parameter list with the VL bit set for the last parameter in the list. The VL bit must be correctly set to maintain compatibility with future releases of the host RPC API. The functions of the Shadow Server host RPC API are:

sdcpif

Access and update execution environment information.

sdcpmg

Add a user message to the Trace Browse log.

sdcprd

Read a buffer of data from the client application.

sdcpwr

Write a buffer of data to the client application.

sdcpif Function

RPC Direct

sdcpif is used to gain information about the current execution environment. Information requests and a buffer data area are passed to this function. This function either updates the buffer with the requested data or uses the data in the buffer to update the current execution environment.

Syntax

The general form for invoication of **sdcpif** is:

```
long sdcpif(rqsr, ouar)
```

CALL Arguments

The **sdcpif** function accepts the following arguments:

Type	Argument	Use	Description
scrqsr *	rqsr	Input	Request string. This request string is passed using a variable length string. Variable length string starts with a two byte prefix followed by name of current request. Request name must be passed in uppercase, and length prefix must be equal to number of characters in request name.
char *	ouar	I/O	Buffer area. This data area is used to either return information about the current execution environment or provide a new value for some element of the current execution environment.

Return Values

sdcpif returns:

Return Value	Description
CMCPCMOK	The information request was successfully handled.
CMCPIVRQ	The request name string was invalid.

Comments

sdcpif provides supports requests for several different types of information about the current execution environment. The supported request types are:

- **HOSTNAME.** This request is used to obtain the host name of the client system that initiated the current RPC. The host name is returned in the output buffer as a variable length string. The first two bytes of the output buffer will contain the length of the host name string. Up to 16 bytes of host name information will be returned after the two-byte length prefix. The size of the return buffer should be at least 102 bytes to allow for longer host names in the

future. Currently, host names are limited to 16 bytes, however, 100 bytes should be reserved in the result area to allow for future expansion of this field. The host name will be on of the following:

- A TCP/IP host name in character string format
- An IP address in dotted decimal notation (for example, 140.252.14.65)
- The SNA LU name of the system running the client application.

- **USERID.** The userid that the client application program provided to logon to the host system will be returned in the buffer area as a variable length string. The first two bytes will contain the userid length followed by the userid string. The userid string can be up to 8 bytes long and will not be padded with trailing blanks.

- **PROGRAM.** This is the name of the currently executing RPC. The program name will be either a load module name or a load module alias. The buffer will contain the program name as a variable length string. The first two bytes will contain the program name length followed by up to 8 bytes of the program name. The program name will not be padded with trailing blanks.

- **CPUTIME.** CPU time is the amount of TCB time used by the current task so far. The amount of TCB CPU time will be returned to the output buffer area as an 8-byte double-precision floating-point value in units of seconds.

- **UNIQUETOKEN.** This is an 8-byte unique token value returned in the output buffer area. The token value can be used as needed by the host RPC application. It will always monotonically increase and can be assumed to be unique across all of the CPU engines of a system image.

- **USERAREA.** This is a 4K scratchpad area shared by all RPCs running in the Shadow Server address space. Shadow Server does not provide any serialization for this user area. This area can be accessed and updated by all RPCs using the 4-byte USERAREA address returned in the output buffer area.

The **sdcpif** function can also be used to update certain elements of the current execution environment. The following update request type is supported:

PLANNAME. The plan name is updated using the contents of the buffer area. The buffer area must be a variable length string containing the plan name. The first 2 bytes of the buffer area indicate the length of the plan name, followed

by the actual plan name. This name will be padded with trailing blanks if needed when it copied into the plan name area.

Note:

This call does not actually alter the DB2 plan (if any) used by the current RPC. It will only update the plan names displayed by the Shadow Server diagnostic facility and the SDB ISPF application on the host. This request type is normally only used by RPCs that establish their own connections to DB2 with DSNALI, and then update the Shadow execution environment plan name for diagnostic purposes.

Example

None at this time.

Related Functions

For information about	See
Adding a user message to the Trace Browse log	**sdcpmg**
Reading a buffer of data from the client	**sdcprd**
Writing a buffer of data to the client	**sdcpwr**

sdcpmg Function

RPC Direct

sdcpmg adds a text message to the Trace Browse log.

Syntax

The general form for invoication of **sdcpmg** is:

```
long sdcpmg(mgsr, mgln)
```

CALL Arguments

The **sdcpmg** function accepts the following arguments:

Type	Argument	Use	Description
char *	mgsr	Input	Text message. The *mgsr argument points to text message that should be added to Trace Browse log. This text message should not start with a length prefix of any kind and should not be null terminated. The text message can contain any combination of characters, however, printable characters are preferred for usability purposes.*
long	mgln	Input	Text message length. The *mgsr argument must be greater than or equal to zero.*

Return Values

sdcpmg returns:

Return Value	Description
CMCPCMOK	The user message was successfully added to Trace Browse.
CMCPEXER	General execution errors were detected.
CMCPTBER	The message could not be added to Trace Browse for any other reason.

Comments

This function adds user messages to the Trace Browse area. User messages have an event type of message and can be added at any time and in any number, however, some caution should be used in adding user messages to Trace Browse. If there are too many messages added, there will be less room for system messages that can be needed for debugging purposes. A user message consists only of message text. The message text can contain any combination of characters passed by the caller and will be truncated if it exceeds the maximum message text length.

Example

None at this time.

Related Functions

For information about	See
Accessing and updating execution environment information	**sdcpif**
Reading a buffer of data from the client application	**sdcprd**
Writing a buffer of data to the client application	**sdcpwr**

sdcprd Function

RPC Direct

sdcprd reads a buffer of data from the client.

Syntax

The general form for invocation of **sdcprd** is:

`long sdcprd(`*`buar, buln`*`)`

CALL Arguments

The **sdcprd** function accepts the following arguments:

Type	Argument	Use	Description
char *	buar	Output	Buffer area for message received from client application. This area must be at least as large as the size value specified in next argument.
long	buln	Input	Buffer length. Size of the buar buffer area.

Return Values

sdcprd returns:

Return Value	Description
A Non-Negative Buffer Length Value	A message was successfully read from the client. ***Note:*** If the input buffer length is zero then the return code from this function will also be zero.
CMCPBUER	The message buffer reset failed.
CMCPEXER	A general execution error occurred.
CMCPSXSZ	The buffer sent by the client application was larger than the buffer area provided by the host RPC.
CMCPREER	The client application failed, the system running the client application failed, or the network failed.
CMCPTPEN	The client application terminated.

Comments

This function is used to read one buffer of data from the client. The return code from this function will be either an actual buffer size or a negative error code. If the client system network or application fails, this function will return a negative error code.

The buffer should be large enough for the largest possible message sent by the client application. In general, client applications are limited to sending messages of up to 30 kilobytes.

Note:
If the receiving buffer is smaller than the message transmitted by the client, the client message will be discarded, not truncated.

The buffer of data received from a client will not be translated or converted in any way, so if the buffer contains binary data values, these will be returned to the host application unchanged. In other words, all bit combinations can be transmitted from the client application to the host RPC without alteration.

Calling this function will suspend execution of a host RPC until a data buffer is available or until a communication I/O error is detected. In other words, the host RPC will be suspended until the client application transmits a data buffer or until a communication failure occurs. If the client application fails to supply a buffer of data, the host application will be suspended indefinitely. There is no time out associated with this function at this time.

Examples

None at this time.

Related Functions

For information about	See
Accessing and updating execution environment information	**sdcpif**
Adding a user message to the Trace Browse log	**sdcpmg**
Writing a buffer of data to the client application	**sdcpwr**

sdcpwr Function

RPC Direct

This function is used to send a buffer of data from the host application to the client.

Syntax

The general form for invocation of **sdcpwr** is:

`long sdcpwr(`*`buar, buln, cmfg`*`)`

CALL Arguments

The **sdcpwr** function accepts the following arguments:

Type	Argument	Use	Description
char *	buar	Input	Buffer area containing message to be sent to client application. This data area does not start with a length prefix and need not be null-delimited. This data area must be at least as large as size value specified by next argument.
long	buln	Input	Buffer length. This argument contains number of bytes of data to be transmitted from host to client. This value must be greater than or equal to zero. This value should not exceed maximum size buffer that can be transmitted from host to client, or approximately 30K.
long	cmfg	Input	Communication flags. This argument is used to pass flags to write routine to control how write operation is done. See comments section below for definition of flags.

Return Values

sdcpwr returns:

Return Value	Description
CMCPCMOK	The data buffer was successfully transmitted to the client. ***Note:*** There is no guarantee that the buffer was successfully received even if the return code is CMCPCMOK.
CMCPBUER	The transmission buffer could not be reset.
CMCPEXER	A general execution error occurred.
CMCPSEER	A communication error of some kind was detected. A communication error will be reported if the network failed, the client application program failed, or the system running the client application program failed.

Comments

This buffer of data can contain any combination of characters or binary values, and will not be translated or converted in any way as it is transmitted from the host system to the client. This function will return to the caller before the data has actually been transmitted from the host, at which time the host RPC can assume that the data has been copied from the buffer. However, no assumptions can be made about when the data will actually be delivered to the client application.

The host RPC buffer write function can be called consecutively any number of times. There is no requirement that any data buffer be received from the client before, after, or in between buffer write calls. However, if the host application attempts to write “too much” data to the client, then the host communication buffers can be filled causing the call to be suspended until a sufficient amount of data has actually been sent to the client.

The communication flags argument is optional. If this argument is omitted, all flags will be assumed to be off. The communication flag is:

SDCPWRNO. This flag is set to prevent the line from being turned around after each write. By default, the line is turned around after each write to allow the client to send a buffer to the host. This flag must be set if another write will follow the current write without an intervening read.

Note:
This flag will only have an effect on LU 6.2 client/server sessions. This flag should be set if there is any possibility that the host RPC will ever have to use LU 6.2 to communicate with the client.

Examples

None at this time.

Related Functions

For information about	See
Accessing and updating execution environment information	**sdcpif**
Adding a user message to the Trace Browse log	**sdcpmg**
Reading a buffer of data form the client application	**sdcprd**

General APIs

This section covers the following General APIs:

API Description	DIRECT	WEB	SEF	WEB/RX
General APIs				
To get error information:	SQLERROR or SDCPSE	SWSERROR or SWCPSE	SDBERROR	SWSERROR
To return information to ODBC CALL RPC:	SQLGETINFO or SDCPGI	SWSINFO or SWCPGI	SDBINFO	SWSINFO
To write message to trace browser:	SQLTRACEMSG or SDCPTM	SWSTRACEMSG or SWCPTM	SDBTRACE	SWSTRACE
To dynamically allocate a file:	SDBALLOC or SDCPAL	SWSALLOC or SWCPAL	SDBALLOC	SWSALLOC
To de-allocate datasets:	SDBFREE or SDCPFR	SWSFREE or SWCPFR	SDBFREE	SWSFREE
To fetch or set transaction run-time variable values:	SDBVALUE or SDCPVL	SWSVALUE or SWCPVL	SDBVALUE	SWSVALUE
To save and restore transaction-oriented data :	SQLTOKEN or SDCPTK	SWSTOKEN or SWCPTK	SDBTOKEN	SWSTOKEN
To concatenate multiple DDNames under a single DDName.	SDBCONCT or SDCPCC	SWSCONCT or SWCPCC.	SDBCONCT	SWSCONCT

High-Level Language Interface SQLERROR (SDCPSE) SWSERROR (SWCPSE) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPSE/SWCPSE.**

SQLERROR/SWSERROR is the Web Server API function used to fetch information pertaining to the last Application Program Interface error detected for this transaction.

CALL Arguments

The SQLERROR/SWSERROR function call requires eight arguments. None can be omitted from the function call.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HENV	USAGE POINTER	PTR	INPUT	The Web Server environment handle. The environment handle is an opaque, four-byte address pointer. The environment handle is currently not used, and must be set to zero (NULL).
2	HDBC	USAGE POINTER	PTR	INPUT	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
3	HSTMT	USAGE POINTER	PTR	INPUT	The Web Server statement handle. The statement handle is an opaque, four-byte address pointer. The statement handle is currently not used, and must be set to zero (NULL).
4	UCHAR*	PIC X(6)	CHAR(6)	OUTPUT	This argument should specify a character string buffer of at least 6 bytes in length. A state value, compatible in format with the ODBC specification is returned in this area, as a null terminated string.
5	SDWORD*	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	The 'native' error code is returned within this area. This is some value that describes the error condition.
6	UCHAR*	PIC X (nnn)	CHAR (nnn)	OUTPUT	The buffer area which receives the error message text. Note that the error message text will always be null-terminated. Room for the trailing null must be provided.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
7	SDWORD*	PIC S9(5) COMP	FIXED BIN(31)	INPUT	The total size of the error message buffer area supplied by the sixth argument. The error message will be truncated if it does not fit into this buffer, including room for the trailing null terminator.
8	SDWORD*	PIC S9(5) COMP	FIXED BIN(31)	OUTPUT	The API returns the total size of the error message (excluding the null terminator). The returned size value will be larger than the buffer size if the error message has been truncated.

Return Values

SQLERROR/SWSERROR always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The return values have been set.
SWS_SUCCESS_WITH_INFO, SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_NO_DATA_FOUND, SQL_NO_DATA_FOUND	There is no prior error condition upon which to report.
SWS_INVALID_HANDLE, SQL_INVALID_HANDLE	One of the handle arguments is invalid.

PL/I Example

```
DCL  SENVH   PTR;                      /* Environment Handle  */
DCL  SCONN   PTR;                      /* Connection Handle   */
DCL  SSTMT   PTR;                      /* Statment Handle     */
DCL  SSQST   CHAR(6);                  /* ODBC State          */
DCL  SNATV   FIXED BIN(31);            /* Native Error Code   */
DCL  SERMG   CHAR(256);                /* error message text  */
DCL  SMGSZ   FIXED BIN(31) INIT(256);  /* Buffer size         */
DCL  SRTSZ   FIXED BIN(31);            /* Fetched value size  */
DCL  RC      FIXED BIN(31);            /* return code         */
DCL  DMHX    FIXED BIN(31) BASED;      /* Dummy Handle field  */

ADDR(SENVH)->DMHX = 0;              /* Clear Environment Hndl.*/
ADDR(SCONN)->DMHX = 0;              /* Clear Connection Handle*/
ADDR(SSTMT)->DMHX = 0;              /* Clear Statement Handle */

CALL SWSERROR( SENVH,               /* get last error info    */
               SCONN,
               SSTMT,
               SSQST,
               SNATV,
               SERMG,
               SMGSZ,
               SRTSZ );
RC = PLIRETV();                     /* get return code        */
IF (RC ^= SWS_SUCCESS &             /* exit program if bad RC */
    RC ^= SWS_SUCCESS_WITH_INFO) THEN
       EXIT;
```

C Example

```
HDBC   sEnvh   = NULL;              /* Environment Handle     */
HDBC   sConn   = NULL;              /* Connection Handle      */
HSTMT  sStmt   = NULL;              /* Statement Handle       */
char   sSqst[6];                    /* ODBC-Compatible state  */
SDWORD sNatv;                       /* Native Error Code      */
char   sErmg[256];                  /* Error message text     */
SDWORD sRtsz;                       /* Error message size     */

rc = SWSError( &sEnvh,              /* get error information  */
               &sConn,
               &sStmt,
               sSqst,
               &sNatv,
               sErmg,
               sizeof(sErmg),
               &sRtsz );
if (rc ^= SWS_SUCCESS) return;      /* exit program if bad RC */
```

COBOL Example

```
77  SENVH              USAGE IS POINTER.
77  SCONN              USAGE IS POINTER.
77  SSTMT              USAGE IS POINTER.
77  SSQST              PIC X(6).
77  SNATV              PIC S9(5) COMP.
77  SERMG              PIC X(256).
77  SMGSZ              PIC S9(5) COMP VALUE 256.
77  SRTSZ              PIC S9(5) COMP.
CALL 'SWCPSE' USING SENVH,
                    SCONN,
                    SSTMT,
                    SSQST,
                    SNATV,
                    SERMG,
                    SMGSZ,
                    SRTSZ.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACKGOBACK.
```

SDBERROR/SWSERROR Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level language interface available.

SDBERROR/SWSERROR is a built-in function used to retrieve information about the last error condition encountered by a Web Server API function. If no error condition has been encountered, the function returns a NULL string. Otherwise a description of the last error is returned.

Syntax

The general form for invocation of SDBERROR/SWSERROR is:

```
string = SDBERROR( )
```

or

```
string = SWSERROR( )
```

The SDBERROR/SWSERROR function is coded without arguments.

Return Values

After the call shown above completes, the variable string will be set to a NULL value if no error condition has been encountered. Otherwise, it will contain a text-format description of the error.

High-Level Language Interface SQLGETINFO (SDCPGI) SWSINFO (SWCPGI) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPGI/SWCPGI**.

SQLGETINFO/SWSINFO is the Web Server API function used to fetch information about the current transaction execution environment and return it to the caller.

CALL Arguments

The SQLGETINFO/SWSINFO function takes five arguments. All five arguments must be specified on the call.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A four-byte binary integer indicating the information item to be returned by the function. Specify any one of the manifest constants, shown in the table below, to indicate the data item to be fetched.
3	UCHAR *	PIC X(nnn)	CHAR (nnn)	Output	The data buffer to receive the fetched information. Depending on the value of the second argument, the returned data can be a null-terminated string; a 16-bit integer value, a 32-bit flag-word value, or a 32-bit signed or unsigned integer.
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the data buffer area given by the third argument.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
5	SDWORD *	PIC S9(5) COMP	FIXED BIN(31)	Output	Return area receiving the total size, in bytes, of the requested information value, regardless of whether the fetched value could be completely stored within the buffer area. For character format data items, which are null terminated, this value does not include the null termination byte. For requests which return character data: If the total size of the requested information is greater than or equal to the size of the data buffer the returned character string is truncated, and a null terminationbyte is placed into the last available of the buffer area. For requests which return any other data type:The value given by the forth argument is ignored. The size of the return buffer area is assumed to be at least four bytes.

The following table shows the values which can be specified for the second argument. Note that for COBOL, the value names contain hyphens instead of underbar characters.

Manifest Constant	Value Returned
SWS_GET_ASID	The ASID as a 2-byte binary value.
SWS_GET_BYTES	The number of saved bytes
SWS_GET_CLOCK	The current TOD clock value as an 8-byte binary TOD value. Note that this is the un-adjusted STCK value.
SWS_GET_CONNECTID	The unique CONNECTION ID value as a 4-byte binary value.
SWS_GET_CPUDELTA	The 8-byte task CPU time delta value.
SWS_GET_CPUTIME	The 8-byte task CPU time value.
SWS_GET_DB2PLAN	The DB2 plan name.
SWS_GET_DB2SUBSYS	The DB2 subsystem name.
SWS_GET_EVENTTYPE	An indication of the event type associated with the invocation of the rule/ program.
SWS_GET_HOSTDOMAIN	The host (server) domain associated with the current request.
SWS_GET_HOSTNAME	The HOSTNAME (CLIENT) associated with the current request.
SWS_GET_IPADDRESS	The IP Address for the current connection. The returned value is 4-byte binary value.
SWS_GET_JOBNAME	The MVS job name related to the current primary address space.
SWS_GET_LASTCONNECTID	The last CONNECTION ID used on the current link.
SWS_GET_LASTUSERID	The last Userid used on the current link

Manifest Constant	Value Returned
SWS_GET_LINKTYPE	The link type for the current request
SWS_GET_LU	The LU NAME for the current request
SWS_GET_MAINPGM	The name of the main REXX program or rule.
SWS_GET_MODE	The mode name for the current request
SWS_GET_PRODUCT	The Product Identification string
SWS_GET_PRODUCTSTATUS	The current product status
SWS_GET_PROGRAM	The name of the REXX program or rule.
SWS_GET_ROWS	The number of source rows
SWS_GET_SEFFEATURE	A single blank if SEF is not enabled.
SWS_GET_SUBSYS	The accessed subsystem ID from the current OPMS image.
SWS_GET_SUBSYSASID	The ASID of the active subsystem from the real OPMS as a two-byte binary value.
SWS_GET_SMFID	The SMFID of the MVS system.
SWS_GET_TASKTYPE	The task type.
SWS_GET_TRANSTYPE	The transaction program type.
SWS_GET_USERID	The Userid value.
SWS_GET_VERSION	The version string of the product subsystem under which this rule/program is running.

Return Values

SQLGETINFO/SWSINFO always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, **SQL_SUCCESS**	The operation succeeded. The requested data has been fetched and placed into the buffer area. The actual size of the data is set into the sixth argument.
SWS_SUCCESS_WITH_INFO, **SQL_SUCCESS_WITH_INFO**	The return buffer area was not large enough to store the fetched item. The fetched item was truncated. The size of the fetched item, before truncation, is returned to the sixth argument. For character data, a null termination byte is always placed into the last buffer position.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.

Return Value	Description
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_INVALID_HANDLE, SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR/SWSERROR.

PL/I Examples

Note:
SQLGETINFO/SWSINFO can be used as an alias for the pre-processor symbol SWSINFO.

```
DCL  SCONN     PTR;                      /* Connection Handle   */
DCL  SBUFF     CHAR(256);                /* Return Buffer area  */
DCL  SBFSZ     FIXED BIN(31) INIT(256); /*  Size of buffer     */
DCL  SRTSZ     FIXED BIN(31);            /* Actual item size    */
DCL  RC FIXED  BIN(31);                  /* return code         */
DCL  DMHX      FIXED BIN(31) BASED;      /* Dummy Handle field  */
ADDR(SCONN)->  DMHX = 0;              /* Clear Connection Handle*/
CALL SWSINFO(SCONN                    /* fetch the IP address   */
ADDR(SCONN)->  DMHX = 0;              /* Clear Connection Handle*/
_IPADDRESS,
               SBUFF,
               SBFSZ,
               SRTSZ );
RC = PLIRETV();                       /* get return code        */
IF RC ^= SWS_SUCCESS THEN             /* exit program if bad RC */
 EXIT;
```

C Example

Note:
SQLGETINFO/SWSINFO can be used as an alias for the pre-processor symbol SWSINFO.

```
HDBC          sConn   = NULL;        /* Connection Handle      */
char          sBuff[256];            /* Return Buffer Area     */
SDWORD        sRTSZ;                 /* Return item size       */
long          RC;                    /* return code            */
 rc = SWSINFO( &sConn,               /* obtain the IP Address  */
               SWS_GET_IPADDRESS,
               &sBuff[0],
               (SDWORD) sizeof(sBuff),
               &sRTSZ );
if (rc ^= SWS_SUCCESS) return;      /* exit program if bad RC */
```

COBOL Example

```
77  SCONN             USAGE IS POINTER.
77  SBUFF             PIC X(80).
77  SBFSZ             PIC S9(5) COMP VALUE 80.
77  SRTSZ             PIC S9(5) COMP.
CALL 'SWCPGI' USING SCONN,
                    SWS-GET-IPADDRESS,
                    SBUFF,
                    SBFSZ,
                    SRTSZ.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.
```

SDBINFO/SWSINFO Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level language interface available.

SDBINFO/SWSINFO is a built-in function used to retrieve environmental information from the Shadow Web Server Subsystem.

Syntax

The general form for invocation of SDBINFO/SWSINFO is:

```
var = SDBINFO/SWSINFO( arg1 )
```

Valid Arguments

The SDBINFO/SWSINFO function takes one argument. The input argument can be one of the following string constants:

Manifest Constant	Value Returned
ASID	The ASID as a 2-byte binary value, when invoked via the program API. The ASID is returned as a 4-byte value when invoked from REXX.
BYTES	The number of saved bytes
CLOCK	The current TOD clock value as an 8-byte binary TOD value. Note that this is the un-adjusted STCK value.
CONNECTID	The unique CONNECTION ID value
CPUDELTA	The 8-byte task CPU time delta value
CPUTIME	The 8-byte task CPU time value
DB2PLAN	The DB2 plan name
DB2SUBSYS	The DB2 subsystem name
EVENTTYPE	An indication of the event type associated with the invocation of the rule/program.
HOSTDOMAIN	The host (server) domain associated with the current request.
HOSTNAME	The HOSTNAME (CLIENT) associated with the current request.
IPADDRESS	The IP Address for the current request. The function returns a 4-byte binary value, when invoked via the program API. A formatted character value in the form, 10.123.2.12 is returned when this function is invoked from REXX.
JOBNAME	TheMVS job name related to the current primary address space

Manifest Constant	Value Returned
LASTCONNECTID	The last CONNECTION ID used on the current link
LASTUSERID	The last Userid used on the current link
LINKTYPE	The link type for the current request
LU	The LU NAME for the current request
MAINPGM	The name of the main REXX program or rule
MODE	The mode name for the current request
ODBCDATE	The compile date of the ODBC driver
ODBCVERSION	The ODBC driver version
PRODUCT	The Product Identification string
PRODUCTSTATUS	The current product status
PROGRAM	The name of the REXX program or rule
ROWS	The number of source rows
SEFFEATURE	A single blank if SEF is not enabled.
SUBSYS	The accessed subsystem ID from the current OPMS image
SUBSYSASID	The ASID of the active subsystem from the real OPMS
SMFID	The SMFID
TASKTYPE	The task type
TRANSTYPE	The transaction program type
USERID	The Userid value
USERPARM	The user parameter string from the client
VERSION	The version string of the product subsystem under which this rule/program is running.

Return Values

The function always returns the indicated value. If the value requested is not valid for the environment, a NULL string is returned.

Examples

The following request will set the REXX variable, IPA, to the 4-byte binary TCP/IP address of the Web Client program:

```
IPA = SDBINFO/SWSINFO('IPADDRESS')
```

High-Level Language Interface SQLTRACEMSG (SDCPTM) SWSTRACEMSG (SWCPTM) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX Intepreters.
✔	HLL entry point name is **SDCPTM/SWCPTM.**

SQLTRACEMSG/SWSTRACEMSG is used to write a message into the Shadow Web Server or Shadow Direct's wrap-around trace browse dataset. The message can contain any text desired. If the message is too long to fit within a trace browse record, it is truncated. Truncation is not considered an error.

CALL Arguments

The SQLTRACEMSG/SWSTRACEMSG function takes four arguments. All four arguments must be specified on the call.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	PTR	PIC X(nnn)	CHAR (nnn)	Input	The data value which is to be written to the trace browse wrap-around dataset. You can specify a null terminated string, or explicitly provide the value length via the third argument. The maximum useable length for a trace browse record is approximately 730 bytes.
3	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the data value given by the second argument which is to be written to the trace record. You can optionally specify SWS_NTS, to indicate that the data is a null terminated string.
4	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument is currently not used, but can be in future releases. You must specify a zero value.

Return Values

SQLTRACEMSG/SWSTRACEMSG always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified data was written to the product's wrap-around trace.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_INVALID_HANDLE, SQL_INVALID_HANDLE	The connection handle is invalid. No error information is available
Any Other Value	The operation failed.

PL/I Example

```
DCL  TCONN   PTR;                    /* Connection Handle      */
DCL  TDATA   CHAR(256);              /* Text output area       */
DCL  TSIZE   FIXED BIN(31);          /* Text length area       */
DCL  RC      FIXED BIN(31);          /* return code            */
DCL  DMHX    FIXED BIN(31) BASED;    /* Dummy Handle field     */
DCL  FB00    FIXED BIN(31) INIT(0); /* Dummy argument         */
ADDR(TCONN)->DMHX = 0;               /* Clear Connection Handle*/
TDATA    = 'Trace Message Text';     /* Set output area        */
TSIZE    = 18;                       /* set length             */
CALL SWSTRACE( TCONN                 /* output trace message   */
       TDATA,
       TSIZE,
       FB00 );
RC = PLIRETV();                      /* get return code        */
IF RC ^= SWS_SUCCESS THEN            /* exit program if bad RC */
EXIT;
```

C Example

```
HDBC tConn   = NULL;                /* Connection Handle      */
char tData[] = "Null-terminated!";  /* Text string definition */
long RC;                            /* return code            */
rc = SWSTRACE( &tConn,              /* output trace message   */
       tData,
       SWS_NTS,
       0 );
if (rc ^= SWS_SUCCESS) return;      /* exit program if bad RC */
```

COBOL Example

```
77  TCONN              USAGE IS POINTER.
77  TDATA              PIC X(80).
77  TSIZE              PIC S9(5) COMP.
77  FB00               PIC S9(5) COMP VALUE 0.
MOVE 'TRACE MESSAGE' TO TDATA.
MOVE 13 TO TSIZE.
CALL 'SWCPTM' USING TCONN,
       TDATA,
       TSIZE,
       FB00.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.
```

SDBTRACE/SWSTRACE Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level language interface available.

The SWSTRACE built-in function provides a means of logging text information to the Shadow Web Server Wrap-around trace.

The SWSTRACE function is primarily intended to allow logging of trace messages from other REXX interpreters. You can generate log messages from Shadow/REXX using this function, if you desire. However, REXX SAY statements issued from Shadow/REXX event procedures are automatically logged to the wrap-around trace without the overhead of invoking this function.

Coding SDBTRACE/SWSTRACE

To code the SWSTRACE function, use the following format:

```
RC = SWSTRACE( textstring )
```

Return Values

The function always returns 0 (zero) to the caller.

High-Level Language Interface SDBALLOC (SDCPAL) SWSALLOC (SWCPAL) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPAL/SWCPAL**.

SDBALLOC/SWSALLOC is used to dynamically allocate an MVS dataset for use by Web Client program.

The format of this command is similar in features and functions to the TSO/E Allocate command. A text string is used as input in order to provide the parameters necessary to allocate the specified dataset. Use the SWSFREE command to de-allocate datasets allocated with the SDBALLOC/SWSALLOC command.

Note:
Because of comparable functionality of SWSALLOC to IBM's ALLOC function, this documentation is similar to IBM's TSO/E online help.

Call Arguments

The SDBALLOC/SWSALLOC (SDCPAL/SWCPAL) function arguments are described in the table which follows. Only two of the three arguments are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BINARY (31)	INPUT	The length of the allocation command string. If the length is longer than the actual command, trailing nulls or blanks will be ignored. If the length is less than the actual command string, the allocation command string will be truncated and possibly cause execution errors. The maximum string length is 32768 bytes.
2	CHAR*	PIC X(nnnnn)	CHAR (nnnnn)	INPUT	The allocation command string. See Supported Dynamic Allocation Keywords below.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
3	SWSASB*	Usage Pointer	PTR	OUTPUT	The Shadow Web Server Allocation Status Block. This is an optional argument that provides information concerning the status of the allocation request. If you do not specify this argument, you will not have access to the reason code nor the DAIR code.

Return Values

SDBALLOC/SWSALLOC always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR, SQLERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. Generally this indicates that the file was not allocated. There will be an error message in the Allocation Status Block describing the error.

Supported Dynamic Allocation Keywords

The SWSALLOC (SWCPAL) interface supports the following dataset allocation request parameters:

Allocation Keyword	Description
DSN(DSNAME)	Specifies the name of the dataset to be allocated. You can only specify a single dataset name. Dataset names must be fully qualified as there will be no prefix appended to the supplied name. ***Note:*** *This is a required parameter unless you specify a PATH parameter.*
DDN(DDNAME)	Specifies the DDNAME to associate with the allocated file. If you do not specify one, one will be dynamically generated for you. The generated DDNAME can be obtained from the Allocation Status Block which is input to the allocation request.

Allocation Keyword	Description
DEST(DESTINATION/NODE USERID)	Remote destination or a User at a specified node to which SYSOUT data sets are to be routed.
DISP(STATUS NORMAL ABNORMAL)	Specifies the disposition of file upon normal and abnormal (conditional) session termination. Status disposition: Indicates the disposition of the file upon normal session termination. • **SHR** = Dataset exists and exclusive control is not required. • **OLD** = Dataset exists and exclusive control is required. • **MOD** = Additions are to be made to the dataset. • **NEW** = Dataset is to be created . Normal termination disposition: Indicates the disposition of the file upon normal session termination. • **UNCATALOG** = Specifies that the file should be uncatalogued. • **CATALOG** = Specifies that the file should be catalog. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted. Abnormal (conditional) termination disposition: indicates the disposition of the file upon abnormal (conditional) session termination. • **UNCATALOG** = Specifies that the file should be uncatalogued. • **CATALOG** = Specifies that the file should be catalog. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted.
SYSOUT(CLASS)	Dataset is to be a system output dataset.
VOLUME(SERIAL(s))	Volume(s) on which the dataset resides or is to reside.
BLKSIZE(VALUE)	Blocksize; must be 0 to 32760.
BLOCKS(PRIMARY SECONDARY)	Space is to be allocated by BLOCKS. ***Note:*** Requires the BLKSIZE parameter.
TRACKS(PRIMARY SECONDARY)	Space is to be allocated by tracks.
CYLINDERS(PRIMARY SECONDARY)	Space is to be allocated by cylinders.
DIR(INTEGER)	Number of Directory Blocks required.

Allocation Keyword	Description
LIKE(MODEL_DATASET_ NAME)	The model dataset is a dataset whose attributes are to be used to allocate a new dataset. The following attributes are copied from the model dataset: • Primary and Secondary space quantities (SPACE). • Directory space quantity (DIR). • Dataset Organization (DSORG). • Record Format (RECFM). • Optional Services Codes (OPTCD). • Logical Record Length (LRECL). • Key Length (KEYLEN). • Blocksize (BLKSIZE). • Volume Sequence Number (VSEQ). • Expiration Date (EXPDT). If SMS is active the following attributes are not copied: • Optional Services Codes (OPTCD). • Blocksize (BLKSIZE). • Volume Sequence Number (VSEQ). • Expiration Date (EXPDT). Any attribute(s) of the model data set can be overridden by explicitly specifying the appropriate keyword(s) on the allocate command.
HOLD	Dataset is to be placed on a hold queue upon de-allocation.
UNIT(UNIT_TYPE)	Device type to which a file or data set is to be allocated.
UCOUNT(COUNT)	Maximum number of devices to which a file or data set can be allocated.
PARALLEL	One device mounted for each volume specified on the volume parameter.
MAXVOL(VOL_COUNT)	Maximum number of volumes a data set can use.
PRIVATE	A volume which is not permanently resident or reserved is to be assigned the private volume use attribute.
RELEASE	Unused space is to be deleted when the data set is closed.
VSEQ(VOL_SEQ_NUM)	Which volume of a multi-volume data set to begin processing with.
ROUND	Allocated space should be equal to one or more cylinders.
BFALN(VALUE)	Buffer boundary alignment. Legitimate values are: • **D** = Double word boundary. • **F** = Full word boundary.
BFTEK(VALUE)	Type of buffering. Legitimate values are: • **A** = Automatic record area construction. • **D** = Dynamic buffering. • **E** = Exchange buffering. • **R** = Record buffering. • **S** = Simple buffering.
BUFL(INTEGER)	Buffer length; must be 0 to 32760.
BUFNO(INTEGER)	Number of buffers; must be 0 to 255.
BUFOFF(INTEGER)	Block prefix length; must be 0 to 99.

Allocation Keyword	Description
DSORG(VALUE)	Dataset Organization. Legitimate values are: • **DA** = Direct Access. • **DAU** = Direct Access Unmovable. • **PO** = Partitioned Organization. • **POU** = Partitioned Organization Unmovable. • **PS** = Physical Sequential. • **PSU** = Physical Sequential Unmovable.
EROPT(VALUE)	Error Option. Legitimate values are: • **ABE** = Abnormal End-Of-Task. • **ACC** = Accept block causing error. • **SKP** = Skip block causing error.
KEYLEN(INTEGER)	Key length; must be 0 to 255.
LIMCT(INTEGER)	Number of blocks or tracks to be searched for a block or available space; must be 0 to 32760.
LRECL(VALUE)	Logical Record Length. Legitimate values are: • 0 to 32760. • Character "X". • 1 to 16384 with K-multiplier. Where: • **X** = LRECL value exceeds 32756 for variable length spanned records processed under QSAM. • **K** = LRECL value is a multiplier of 1024.
NCP(INTEGER)	Maximum number of read or write macros before a check; must be 0 to 255. If you are running TSO/E on MVS/ESA SP 4.2.2 or earlier, the maximum value is 99.
OPTCD(VALUE)	Optional Services Codes. Legitimate values are: • **A** = Actual device addresses presented in read and write macro instructions. • **B** = End-Of-File recognition disregarded for tapes. • **C** = Chained scheduling is to be used. • **E** = Extended search for block or available space. • **F** = Feedback can be requested in read and write macro instructions. • **Q** = ANSI translate. • **R** = Requests relative block addressing. • **T** = Requests user totaling facility. • **W** = Requests a validity check for write operations on direct access devices. • **J** = Indicates that the character after the carriage control character is to be interpreted as a table reference character .
PROCOPT(VALUE)	File processing option. Legitimate values are: • **INPUT** = Specifies that the data set is to be processed for input only. • **OUTPUT** = Specifies that the data set is to be processed for output only .
EXPDT(VALUE)	Dataset expiration date (YYDDD OR YYYY/DDD).
RETPD(INTEGER)	Dataset retention period (NNNN).

Allocation Keyword	Description
FCB(IMAGE_ID)	Forms Control Image (Buffer) to be used to print an output data set. **IMAGE_ID** specifies a 1 to 4 alphanumeric or national characters which identify the image to be loaded into the Forms Control Buffer.
COPIES(NNN)	Number of copies of a dataset to print. An integer from 1 to 255.
COPYGROUPS(GROUP_ VALUE1 GROUP_VALUE2 ...GROUP_VALUE8))	Used with the **COPIES** option to specify the number of times each page is to be printed. Up to eight group values can be specified with a sum not exceeding the number specified in the **COPIES** parameter.
PROTECT	Specifies that the DASD data set or tape volume containing a tape data set is to be RACF protected.
ACCODE(VALUE)	ANSI accessibility code (A through Z).
OUTBIN(VALUE)	Specifies the output bin on the IBM 3800 laser printer. Legitimate values are: • **BURST** = Specifies that the data set should be sent to the burster/trimmer bin on the IBM 3800 laser printer. • **NOBURST** = Specifies that the data set should be sent to the continuous feed bin on the IBM 3800 laser printer.
CHARS(CHAR_TABLE1 ...CHAR_TABLE4)	Specifies the character table that is to be used for printing. One to four character tables can be specified.
FLASH(NAME COPIES)	Provides the ability to print a form, grid, design, or constant data on paper as it is being processed through the 3800 printer. Legitimate values are: • **NAME** = Specifies the name of the forms overlay to be used. • **COPIES** = Specifies the number of copies on which the forms overlay is to be used. ***Note:*** Separate parameters by spaces. Do not use commas.
FORMS(VALUE)	Specifies the specific print form to be mounted.
OUTDES(OUTPUT_ DESCRIPTOR_NAME ...)	Specifies a list of output descriptors that will be associated with the sysout data set. These descriptors are created by //OUTPUT JCL statements in the Shadow Web Server or Shadow Direct procedure. ***Note*:** Separate parameters by spaces. Do not use commas.
UCS(UCS_NAME)	Specifies the universal character set (font name) to be used when processing a print data set in the absence of a 'CHARS' specification.
WRITER(EXTERNAL_ WRITER_NAME)	Specifies the member name of a program in the system library that is to write the sysout data set. This program will be used instead of JES2 or JES3.
STORCLAS(STORAGE_CLASS)	The name of the storage class which is used to specify the service level for the data set.
MGMTCLAS (MANAGEMENT_CLASS)	The management class which is used to specify management criteria for the data set.
DATACLAS(DATA_CLASS)	The name of the data class which is used as an allocation template for the data set.

Allocation Keyword	Description
RECFM(OPTION1 OPTION2 ...OPTION5)	Record Format. Legitimate option values are: • **A** = ASA PRINTER CHARACTERS. • **B** = BLOCKED. • **D** = VARIABLE LENGTH ASCII RECORDS. • **F** = FIXED. • **M** = MACHINE CONTROL CHARACTER. • **S** = STANDARD BLOCKS OR SPANNED. • **T** = TRACK OVERFLOW. • **U** = UNDEFINED. • **V** = VARIABLE. ***Note:*** Combinations of these options can be selected. Each selection must be separated by a space. Review your MVS JCL Reference Manual for legitimate combinations.
RECORG(ORGANIZATION)	Dataset Organization. Legitimate values are: • **KS** = VSAM Cluster (KSDS). • **ES** = VSAM Entry Sequenced (ESDS). • **RR** = VSAM Relative Record (RRDS). • **LS** = VSAM Linear Space (LDS).
KEYOFF(OFFSET)	Key Offset.
REFDD(DDNAME)	The DDNAME of a data set whose properties specified on the JCL statement and in the data class are to be used to allocate the new data set. The following properties are copied from the referenced DD statement: • Dataset Organization (RECORG). • Size. • Directory blocks. • Logical Record Length (LRECL). • Record Format (RECFM). • Key Length (KEYLEN). • Key Offset (KEYOFF).
SECMODEL(MODEL_NAME)	The name of a "model" profile which RACF should use in creating a discrete profile for the data set.
DSNTYPE(DSNTYPE)	DATA SET NAME TYPE. Legitimate values are: • **LIBRARY** = A partitioned data set in PDSE format. • **PDS** = A partitioned data set in record format. • **PIPE** = A data pipe. • **HFS** = An HFS (Hierarchical File System) file.
RLS(RLS_VALUE)	Record Level Sharing. Legitimate values are: • **CR** = Consistent Read. • **NRI** = No Read Integrity.
FILEDATA(VALUE)	How the system converts between record format and byte-stream format. Currently meaningful only if path also is coded and the program uses BSAM or QSAM. Legitimate values are: • **TEXT** = Data consists of records that are separated by a delimiter. Currently it is EBCDIC newline (x'15'). • **BINARY** = Data does not contain record delimiters. In the current release the default is binary when creating the file. If you do not code PATHOPTS(OCREATE), then FILEDATA temporarily overrides the creation value.

Allocation Keyword	Description
PATH(PATHNAME)	Identifies an HFS file. A pathname consists of the names of the directories from the root to the file being identified, and then the name of the file. The form is /NAME1/NAME2/.../NAMEn. A pathname begins with a slash (/). The system treats any consecutive slashes like a single slash. The pathname can be 1 to 250 characters. A name can be 1 to 249 characters. Consists of printable characters from x'40' through x'FE'. A pathname is case sensitive. Thus, /usr/joe and /USR/joe define two different files. ***Note:*** *This is a required parameter unless you specify a DSN parameter.*
PATHDISP(NORMAL ABNORMAL)	Specifies the disposition of an HFS file upon normal and abnormal (conditional) session termination. Normal termination disposition: Indicates the disposition of the HFS file upon normal session termination. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted. Abnormal (conditional) termination disposition: indicates the disposition of the HFS file upon abnormal (conditional) session termination. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted. ***Note:*** The default for datasets allocated with the PATH parameter is **PATHDISP(KEEP KEEP)**.
PATHMODE(FILE_ACCESS_ ATTRIBUTE ...)	Specifies the file access attributes when the PATHOPTS operand specifies OCREAT. A FILE ACCESS ATTRIBUTE is one of the following: • SIRUSR • SIWUSR • SIXUSR • SIRWXU • SIRGRP • SIWGRP • SIXGRP • SIRWXG • SIROTH • SIWOTH • SIXOTH • SIRWXO • SISUID • SISGID You can specify up to 14 FILE ACCESS ATTRIBUTES. The system treats duplicate specifications of FILE ACCESS ATTRIBUTES as a single specification.

Allocation Keyword	Description
PATHOPTS(FILE_OPTION ...)	Specifies the file access and status used when accessing a file specified on the path operand. A FILE OPTION can be in the access group or the status group and is one of the following: Valid Access Groups are: • ORDONLY • OWRONLY • ORDWR Valid Status Groups are: • OAPPEND • OCREAT • OEXCL • ONOCTTY • ONONBLOCK • OSYNC • OTRUNC You can specify up to 8 FILE OPTIONs. The system treats duplicate specifications of FILE OPTIONs as a single specification. Code the FILE OPTIONs as follows: • Specify only one FILE OPTION from the access group. if you specify more than one access group file-option, the system ignores them and uses ORDWR as the option. • Specify up to 7 FILE OPTIONs from the status group. You can specify any combination of FILE OPTIONs from the status group.
SEGMENT(INTEGER)	The number of pages produced for a sysout data set before they are processed for printing. Must be 1 to 99999
SPIN(VALUE)	Specifies when a sysout data set is printed. Legitimate values are: • **UNALLOC** = Makes the dataset available for printing immediately after the dataset is unallocated from an explicit unallocation or at the end of the session. • **NO** = Makes the dataset available for printing at the end of the session.
MESSAGE(VALUE)	Specifies whether or not to display dynamic allocation failure messages on the system console. This value overrides the user-specifiable system default (See FILEMESSAGES). Legitimate values are: • **YES** = Display dynamic allocation failure messages. • **NO** = Do not display dynamic allocation failure messages.
MOUNT(VALUE)	Specifies whether to allow or not allow a volume to be mounted in order to satisfy a dynamic allocation request. This value overrides the user-specifiable system default. (See FILEMOUNT). Legitimate values are: • **YES** = Allow the system to mount a volume to satisfy a dynamic allocation request. • **NO** = Do not allow the system to mount a volume to satisfy a dynamic allocation request.

Allocation Keyword	Description
RECALL(VALUE)	Specifies whether to allow or not allow the system to recall a migrated dataset in order to satisfy a dynamic allocation request. This value overrides the user-specifiable system default. (See FILERECALL). Legitimate values are: • **YES** = Allow the system to recall datasets to satisfy a dynamic allocation request. • **NO** = Do not allow the system to recall datasets to satisfy a dynamic allocation request.

File Access Attributes for PATHMODE parameter

Sub-Parameter	Definition
SIRUSR	Specifies permission for the file owner to read the file.
SIWUSR	Specifies permission for the file owner to write the file.
SIXUSR	Specifies permission for the file owner to search, if the file is a directory, or to execute, for any other file.
SIRWXU	Specifies permission for the file owner to read, write, and search, if the file is a directory, or to read, write, and execute, for any other file. this value is the bit inclusive or of SIRUSR, SIWUSR, and SIXUSR.
SIRGRP	Specifies permission for users in the file group to read the file.
SIWGRP	Specifies permission for users in the file group to write the file.
SIXGRP	Specifies permission for users in the file group to search, if the file is a directory, or to execute, for any other file.
SIRWXG	Specifies permission for users in the file group to read, write, and search, if the file is a directory, or to read, write, and execute, for any other file. This value is the bit inclusive or of SIRGRP, SIWGRP, and SIXGRP.
SIROTH	Specifies permission for users in the file other class to read the file.
SIWOTH	Specifies permission for users in the file other class to write the file.
SIXOTH	Specifies permission for users in the file other class to search, if the file is a directory, or to execute, for any other file.
SIRWXO	Specifies permission for users in the file other class to read, write, and search, if the file is a directory, or to read, write, and execute, for any other file. this value is the bit inclusive or of SIROTH, SIWOTH, and SIXOTH.
SISUID	Specifies that the system set the user id of the process to be the same as the user id of the file owner when the file is run as a program.
SISGID	Specifies that the system set the file group of the process to be the same as the group id of the file owner when the file is run as a program.

File Option descriptions for PATHOPTS parameter

Sub-Parameter	Definition
ORDONLY	Specifies that the program can open the file for reading.
OWRONLY	Specifies that the program can open the file for writing.
ORDWR	Specifies that the program can open the file for reading and writing. Do not use this option for a FIFO special file; the result is undefined.
OAPPEND	Specifies that the system sets the file offset to the end of the file before each write, so that data is written at the end of the existing file.
OCREAT	Specifies that the system is to create the file. If the file already exists, the operation will fail if OEXCL is specified, and will open existing file if OEXCL is not specified.
OEXCL	Specifies that, if the file already exists, then HFS open file processing will fail. ***Note:*** *The system ignores OEXCL if OCREAT is not also specified.*
ONOCTTY	Specifies that, if the patterning of the file will not make the terminal device the controlling a terminal device, then op
ONONBLOCK	Specifies the following, depending on the type of file. For FIFO special files: • With ONONBLOCK specified and ORDONLY access: an open() function for reading-only returns without delay. • With ONONBLOCK not specified and ORDONLY access: an open() function for reading-only blocks (waits) until a process opens the file for writing. • With ONONBLOCK specified and OWRONLY access: an open() function for writing-only returns an error if no process currently has the file open for reading. • With ONONBLOCK not specified and OWRONLY access: an open() function for writing-only blocks (waits) until a process opens the file for reading. For character special files: • If ONONBLOCK is specified: an open() function returns without blocking (waiting) until the device is ready or available. Device response depends on the type of device. • If ONONBLOCK is not specified: an open() function blocks (waits) until the device is ready or available. Specification of ONONBLOCK has no other effects.
OSYNC	Specifies that the system is to move data from buffer storage to disk (or other permanent storage) before returning control from a callable service that performs a write.
OTRUNC	Specifies that the system is to truncate the file length to zero if all of the following are true: • The file specified on the path operand exists. • The file is a regular file. • The file successfully opened with ORDWR or OWRONLY. The system does not change the mode and owner. OTRUNC has no effect on FIFO special files or terminal device files.

PL/I Example

```
%INCLUDE SPCPHD
       .
       .
DCL  COMMAND         CHAR(80)       /* ALLOCATE COMMAND       */
       INIT('DSN(SWS.INPUT.DATA) DDN(INFILE) DISP(SHR)');
DCL  CMDLEN          FIXED BIN(31); /* COMMAND LENGTH         */
DCL  RC              FIXED BIN(31); /* RETURN CODE            */
CMDLEN = LENGTH(COMMAND);           /* SET COMMAND LEN        */
                                    /* DYNAMICALLY ALLOCATE
                                          AN INPUT FILE       */
CALL SWSALLOC(CMDLEN,               /* COMMAND LENGTH         */
              COMMAND,              /* COMMAND                */
              SWSASB);              /* ALLOCATION STATUS BLOCK*/
RC = PLIRETV();                     /* GET RETURN CODE        */
IF RC ¬= SWS_SUCCESS THEN           /* EXIT PROGRAM IF BAD RC */
 EXIT;
```

C Example

```
SWS_ALLOCATION_STATUS_BLOCK swsASB; /* response area          */
      .
      .
long RC;                            /* return code            */
char szCommand[ ] = "DSN(SWS.INPUT.DATA) DDN(INFILE) DISP(SHR)";
                                    /* Dynamically allocate an
                                       input file             */
rc = SWSALLOC(strlen(szCommand),    /* Command Length         */
                     szCommand,     /* Command                */
                     swsASB);       /* Response area          */
if (rc ¬= SWS_SUCCESS)
 do
     printf(swsASB.Error_Message);
     return rc;
 end
```

COBOL Example

```
*     NEON API COPY BOOK
    COPY SBCPHD.
              .
              .
              .
              .
      77   COMMAND-LENGTH       PIC S9(5) COMP
      77   COMMAND              PIC X(80)
      VALUE 'DSN(SWS.INPUT.DATA) DDN(INFILE) DISP(SHR)'.
*     DYNAMICALLY ALLOCATE AN INPUT FILE
      MOVE 80 TO COMMAND-LENGTH.
      CALL SWSALLOC
      USING COMMAND-LENGTH,
 COMMAND,
 SWS-ALLOCATION-STATUS-BLOCK.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS
 DISPLAY 'INFILE ALLOCATION FAILED.' UPON CONSOLE
 DISPLAY SWSASB-ERROR-MESSAGE UPON CONSOLE
 GOBACK.
```

The Shadow Web Server Allocation Status Block

	Can **not** be used in Shadow/REXX.
	Can **not** be used from Other REXX interpreters.
	High-level language interface available through the use of the SWSFREE and SWSALLOC APIs.

This structure allows the user to retrieve any error messages generated during dynamic allocation and de-allocation. It will also allow the user to obtain the system-generated DDName or DSName if one is not specified during the invocation of the SWSALLOC API.

PL/I Layout

```
/*-------------------------------------------------------------*/
/* THE FOLLOWING DEFINES THE ALLOCATION STATUS BLOCK           */
/* WHERE DYNAMIC ALLOCATION INFORMATION IS RETURNED TO THE     */
/* CLIENT PROGRAM.   THIS CONTROL BLOCK IS USED WITH THE       */
/* SWSALLOC AND SWSFREE HIGH-LEVEL LANGUAGE INTERFACE.         */
/*-+----1----+----2----+----3----+----4----+----5----+----6---*/
  DCL 1 SWSASB,                  /* ALLOCATION STATUS BLOCK AREA*/
        2 ASBINFO  FIXED BIN(31),/*  DAIR INFO CODE            */
        2 ASBRSN   FIXED BIN(31),/*  DAIR REASON CODE          */
        2 ASBDDN   CHAR(8),      /*  ASSIGNED DDNAME           */
        2 ASBDSN   CHAR(44),     /*  ASSIGNED DSNAME           */
        2 ASBMSG   CHAR(256);    /*  ERROR MESSAGE             */

     This layout is available when you specify:
          %INCLUDE   SPCPHD;
```

C Layout

```
/*-------------------------------------------------------------*/
/*  The following structure is used as a feedback area         */
/*  for dynamic allocate and de-allocation requests.            */
/*--+----1----+----2----+----3----+----4----+----5----+----6--*/
typedef struct SWS_ALLOCATION_STATUS_BLOCK{                    */
  long   Info_Code;          /* DAIR Info Code                */
  long   Reason_Code;        /* SVC 99 Reason Code            */
  char   Assigned_DDName[8]; /* Assigned DDName from SWSALLOC */
  char   Assigned_DSName[44];/* Assigned DSName from SWSALLOC */
  char   Error_Message[256]; /* Dynamic (de)allocation rrormsg*/
} SWS_ALLOCATION_STATUS_BLOCK;/* request string structure     */
```

This layout is available when you specify:

```
#include  "sccphd.h"
```

COBOL Layout

```
019105*************************************************************
019110* THE SWSALLOC/SWSFREE API INTERFACES USE THE FOLLOWING AREA
019115*************************************************************
019120  01 SWS-ALLOCATION-STATUS-BLOCK.
019125  03 SWSASB-INFO-CODE           PIC S9(5)   COMP.
019130  03 SWSASB-REASON-CODE         PIC S9(5)   COMP.
019135  03 SWSASB-ASSIGNED-DDNAME     PIC X(8).
019140  03 SWSASB-ASSIGNED-DSNAME     PIC X(44).
019145  03 SWSASB-ERROR-MESSAGE       PIC X(256).
```

This layout is available when you specify:

```
COPY  SBCPHD.
```

SDBALLOC/SWSALLOC Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level Language Interface available.

The REXX-language SDBALLOC/SWSALLOC built-in function can be used to dynamically allocate datasets. Datasets allocated using the SWSALLOC built-in function should use the SWSFREE built-in function.

> ***Note:***
> Because of comparable functionality of SWSALLOC to IBM's ALLOC function, this documentation is similar to IBM's TSO/E online help.

Syntax

The general form for a REXX-language invocation of SDBALLOC/SWSALLOC is:

```
rc = SDBALLOC/SWSALLOC("STRING")
```

The format of this command is similar in features and functions to the TSO/E Alloc command. If an error occurs, Shadow WebServer variable ALLOC.MESSAGE will be populated with a descriptive error message.

If you do not specify a DDN parameter, a system generated DDNAME will be created and the variable ALLOC.DDNAME will contain the generated DDNAME. The system generated DDNAME will be in the form of SYS followed by a five (5) digit number.

If you do not specify a DSN parameter, a system generated DATASET NAME will be created and the variable ALLOC.DSNAME will contain the generated DATASET NAME.

In the event of an error, the DAIR return code can be obtained from the ALLOC.INFOCODE and the reason code can be obtained from the ALLOC.REASON.

Valid Arguments

DSN(DSNAME)	Specifies the name of the dataset to be allocated. You can only specify a single dataset name. Dataset names must be fully qualified as there will be no prefix appended to the supplied name. ***Note:*** This is a required parameter unless you specify a PATH parameter.
DDN(DDNAME)	Specifies the DDNAME to associate with the allocated file. If you do not specify one, one will be dynamically generated for you. The generated DDNAME can be obtained from the variable ALLOC.DDNAME.
DEST(DESTINATION/NODE USERID)	Remote destination or a User at a specified node to which SYSOUT data sets are to be routed.
DISP(STATUS NORMAL ABNORMAL)	Specifies the disposition of file upon normal and abnormal (conditional) session termination. Status disposition: Indicates the disposition of the file upon normal session termination. • **SHR** = Dataset exists and exclusive control is not required. • **OLD** = Dataset exists and exclusive control is required. • **MOD** = Additions are to be made to the dataset. • **NEW** = Dataset is to be created. Normal termination disposition: Indicates the disposition of the file upon normal session termination. • **UNCATALOG** = Specifies that the file should be uncatalogued. • **CATALOG** = Specifies that the file should be catalog. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted. Abnormal (conditional) termination disposition: indicates the disposition of the file upon abnormal (conditional) session termination. • **UNCATALOG** = Specifies that the file should be uncatalogued. • **CATALOG** = Specifies that the file should be catalog. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted.
SYSOUT(CLASS)	Dataset is to be a system output dataset.
VOLUME(SERIAL(s))	Volume(s) on which the dataset resides or is to reside.
BLKSIZE(VALUE)	Blocksize; must be 0 to 32760.
BLOCKS(PRIMARY SECONDARY)	Space is to be allocated by BLOCKS. *Note:* Requires the BLKSIZE parameter.
TRACKS(PRIMARY SECONDARY)	Space is to be allocated by tracks.
CYLINDERS(PRIMARY SECONDARY)	Space is to be allocated by cylinders.
DIR(INTEGER)	Number of Directory Blocks required.

LIKE(MODEL_DATASET_NAME)	The model dataset is a dataset whose attributes are to be used to allocate a new dataset. The following attributes are copied from the model dataset: • Primary and Secondary space quantities (SPACE). • Directory space quantity (DIR). • Dataset Organization (DSORG). • Record Format (RECFM). • Optional Services Codes (OPTCD). • Logical Record Length (LRECL). • Key Length (KEYLEN). • Blocksize (BLKSIZE). • Volume Sequence Number (VSEQ). • Expiration Date (EXPDT). If SMS is active the following attributes are not copied: • Optional Services Codes (OPTCD). • Blocksize (BLKSIZE). • Volume Sequence Number (VSEQ). • Expiration Date (EXPDT). Any attribute(s) of the model data set can be overridden by explicitly specifying the appropriate keyword(s) on the allocate command.
HOLD	Dataset is to be placed on a hold queue upon de-allocation.
UNIT(UNIT_TYPE)	Device type to which a file or data set is to be allocated.
UCOUNT(COUNT)	Maximum number of devices to which a file or data set can be allocated.
PARALLEL	One device mounted for each volume specified on the volume parameter.
MAXVOL(VOL_COUNT)	Maximum number of volumes a data set can use.
PRIVATE	A volume which is not permanently resident or reserved is to be assigned the private volume use attribute.
RELEASE	Unused space is to be deleted when the data set is closed.
VSEQ(VOL_SEQ_NUM)	Which volume of a multi-volume data set to begin processing with.
ROUND	Allocated space should be equal to one or more cylinders.
BFALN(VALUE)	Buffer boundary alignment. Legitimate values are: • **D** = Double word boundary. • **F** = Full word boundary.
BFTEK(VALUE)	Type of buffering. Legitimate values are: • **A** = Automatic record area construction. • **D** = Dynamic buffering. • **E** = Exchange buffering. • **R** = Record buffering. • **S** = Simple buffering .
BUFL(INTEGER)	Buffer length; must be 0 to 32760.
BUFNO(INTEGER)	Number of buffers; must be 0 to 255.
BUFOFF(INTEGER)	Block prefix length; must be 0 to 99.

DSORG(VALUE)	Dataset Organization. Legitimate values are: • **DA** = Direct Access. • **DAU** = Direct Access Unmovable. • **PO** = Partitioned Organization. • **POU** = Partitioned Organization Unmovable. • **PS** = Physical Sequential. • **PSU** = Physical Sequential Unmovable.
EROPT(VALUE)	Error Option. Legitimate values are: • **ABE** = Abnormal End-Of-Task. • **ACC** = Accept block causing error. • **SKP** = Skip block causing error.
KEYLEN(INTEGER)	Key length; must be 0 to 255.
LIMCT(INTEGER)	Number of blocks or tracks to be searched for a block or available space; must be 0 to 32760.
LRECL(VALUE)	Logical Record Length. Legitimate values are: • 0 to 32760. • Character "X". • 1 to 16384 with K-multiplier . Where: • **X** = LRECL value exceeds 32756 for variable length spanned records processed under QSAM. • **K** = LRECL value is a multiplier of 1024.
NCP(INTEGER)	Maximum number of read or write macros before a check; must be 0 to 255. If you are running TSO/E on MVS/ESA SP 4.2.2 or earlier, the maximum value is 99.
OPTCD(VALUE)	Optional Services Codes. Legitimate values are: • **A** = Actual device addresses presented in read and write macro instructions. • **B** = End-Of-File recognition disregarded for tapes. • **C** = Chained scheduling is to be used. • **E** = Extended search for block or available space. • **F** = Feedback can be requested in read and write macro instructions. • **Q** = ANSI translate. • **R** = Requests relative block addressing. • **T** = Requests user totaling facility. • **W** = Requests a validity check for write operations on direct access devices. • **J** = Indicates that the character after the carriage control character is to be interpreted as a table reference character.
PROCOPT(VALUE)	File processing option. Legitimate values are: • **INPUT** = Specifies that the data set is to be processed for input only. • **OUTPUT** = Specifies that the data set is to be processed for output only.
RETPD(INTEGER)	Dataset retention period (NNNN).
FCB(IMAGE_ID)	Forms Control Image (Buffer) to be used to print an output data set. IMAGE_ID specifies a 1 to 4 alphameric or national characters which identify the image to be loaded into the Forms Control Buffer.
COPIES(NNN)	Number of copies of a dataset to print; an integer from 1 to 255.

COPYGROUPS(GROUP_ VALUE1 GROUP_VALUE2 ...GROUP_VALUE8))	Used with the COPIES option to specify the number of times each page is to be printed. Up to eight group values can be specified with a sum not exceeding the number specified in the COPIES parameter. PROTECT Specifies that the DASD data set or tape volume containing a tape data set is to be RACF protected.
ACCODE(VALUE)	ANSI accessibility code (A through Z).
OUTBIN(VALUE)	Specifies the output bin on the IBM 3800 laser printer. Legitimate values are: • **BURST** = Specifies that the data set should be sent to the burster/trimmer bin on the IBM 3800 laser printer. • **NOBURST** = Specifies that the data set should be sent to the continuous feed bin on the IBM 3800 laser printer.
CHARS(CHAR_TABLE1 ...CHAR_TABLE4)	Specifies the character table that is to be used for printing. One to four character tables can be specified.
FLASH(NAME COPIES)	Provides the ability to print a form, grid, design, or constant data on paper as it is being processed through the 3800 printer. Legitimate values are: • **NAME** - Specifies the name of the forms overlay to be used. • **COPIES** - Specifies the number of copies on which the forms overlay is to be used. ***Note:*** Separate parameters by spaces. Do not use commas.
FORMS(VALUE)	Specifies the specific print form to be mounted.
OUTDES(OUTPUT_ DESCRIPTOR_NAME ...)	Specifies a list of output descriptors that will be associated with the sysout data set. These descriptors are created by //OUTPUT JCL statements in the Shadow WebServer or Shadow Direct procedure. ***Note:*** Separate parameters by spaces. Do not use commas.
UCS(UCS_NAME)	Specifies the universal character set (font name) to be used when processing a print data set in the absence of a 'CHARS' specification.
WRITER(EXTERNAL_ WRITER_NAME)	Specifies the member name of a program in the system library that is to write the sysout data set. This program will be used instead of JES2 or JES3.
STORCLAS(STORAGE_CLASS)	The name of the storage class which is used to specify the service level for the data set.
MGMTCLAS (MANAGEMENT_CLASS)	The management class which is used to specify management criteria for the data set.
DATACLAS(DATA_CLASS)	The name of the data class which is used as an allocation template for the data set.

RECFM(OPTION1 OPTION2 ...OPTION5)	Record Format. Legitimate option values are: • **A** = ASA PRINTER CHARACTERS. • **B** = BLOCKED. • **D** = VARIABLE LENGTH ASCII RECORDS. • **F** = FIXED. • **M** = MACHINE CONTROL CHARACTER. • **S** = STANDARD BLOCKS OR SPANNED. • **T** = TRACK OVERFLOW. • **U** = UNDEFINED. • **V** = VARIABLE. ***Note:*** Combinations of these options can be selected. Each selection must be separated by a space. Review your MVS JCL Reference Manual for legitimate combinations.
RECORG (ORGANIZATION)	Dataset Organization. Legitimate values are: • **KS** = VSAM Cluster (KSDS). • **ES** = VSAM Entry Sequenced (ESDS). • **RR** = VSAM Relative Record (RRDS). • **LS** = VSAM Linear Space (LDS) .
KEYOFF(OFFSET)	Key Offset.
REFDD(DDNAME)	The DDNAME of a data set whose properties specified on the jcl statement and in the data class are to be used to allocate the new data set. The following properties are copied from the referenced DD statement: • Dataset Organization (RECORG). • Size. • Directory blocks Logical Record Length (LRECL). • Record Format (RECFM). • Key Length (KEYLEN). • Key Offset (KEYOFF).
SECMODEL(MODEL_ NAME)	The name of a "model" profile which RACF should use in creating a discrete profile for the data set.
DSNTYPE(DSNTYPE) DATA SET NAME TYPE	Legitimate values are: • **LIBRARY** = A partitioned data set in pdse format. • **PDS** = A partitioned data set in record format. • **PIPE** = A data pipe. • **HFS** = An HFS (Hierarchical File System) file.
RLS(RLS_VALUE)	Record Level Sharing. Legitimate values are: • **CR** = Consistent Read. • **NRI** = No Read Integrity.
FILEDATA(VALUE)	How the system converts between record format and byte-stream format. Currently meaningful only if path also is coded and the program uses BSAM or QSAM. Legitimate values are: • **TEXT** = Data consists of records that are separated by a delimiter. Currently it is EBCDIC newline (x'15'). • **BINARY** = Data does not contain record delimiters. In the current release the default is binary when creating the file. If you do not code PATHOPTS(OCREATE), then FILEDATA temporarily overrides the creation value.

PATH(PATHNAME)	Identifies an HFS file. A pathname consists of the names of the directories from the root to the file being identified, and then the name of the file. The form is /NAME1/NAME2/.../NAMEn. A pathname begins with a slash (/). The system treats any consecutive slashes like a single slash. The pathname can be 1 to 250 characters. A name can be 1 to 249 characters. Consists of printable characters from x'40' through x'FE'. A pathname is case sensitive. Thus, /usr/joe and /USR/joe define two different files. ***Note:*** This is a required parameter unless you specify a DSN parameter.
PATHDISP(NORMAL ABNORMAL)	Specifies the disposition of an HFS file upon normal and abnormal (conditional) session termination. Normal termination disposition: Indicates the disposition of the HFS file upon normal session termination. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted. Abnormal (conditional) termination disposition: indicates the disposition of the HFS file upon abnormal (conditional) session termination. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted. ***Note:*** The default for datasets allocated with the PATH parameter is PATHDISP(KEEP KEEP).
PATHMODE(FILE_ ACCESS_ATTRIBUTE ...)	Specifies the file access attributes when the PATHOPTS operand specifies OCREAT. A FILE_ACCESS_ATTRIBUTE is one of the following: • SIRUSR • SIWUSR • SIXUSR • SIRWXU • SIRGRP • SIWGRP • SIXGRP • SIRWXG • SIROTH • SIWOTH • SIXOTH • SIRWXO • SISUID • SISGID You can specify up to 14 FILE_ACCESS_ATTRIBUTES. The system treats duplicate specifications of FILE_ACCESS_ATTRIBUTES as a single specification.

PATHOPTS(FILE_ OPTION ...)	Specifies the file access and status used when accessing a file specified on the path operand. A file_option can be in the access group or the status group and is one of the following: Valid Access Groups are: • ORDONLY • OWRONLY • ORDWR • Valid Status Groups are: • OAPPEND • OCREAT • OEXCL • ONOCTTY • ONONBLOCK • OSYNC • OTRUNC You can specify up to 8 FILE_OPTIONs. The system treats duplicate specifications of FILE_OPTIONs as a single specification. Code the FILE_OPTIONs as follows: • Specify only one FILE_OPTION from the access group. if you specify more than one access group file-option, the system ignores them and uses ORDWR as the option. • Specify up to 7 FILE_OPTIONs from the status group. You can specify any combination of FILE_OPTIONs from the status group.
SEGMENT(INTEGER)	The number of pages produced for a sysout data set before they are processed for printing; must be 1 to 99999
SPIN(VALUE)	Specifies when a sysout data set is printed. Legitimate values are: • **UNALLOC** = Makes the dataset available for printing immediately after the dataset is unallocated from an explicit unallocation or at the end of the session. • **NO** = Makes the dataset available for printing at the end of the session.
MESSAGE(VALUE)	Specifies whether or not to display dynamic allocation failure messages on the system console. This value overrides the user-specifiable system default (See FILEMESSAGES). Legitimate values are: • **YES** = Display dynamic allocation failure messages. • **NO** = Do not display dynamic allocation failure messages.
MOUNT(VALUE)	Specifies whether to allow or not allow a volume to be mounted in order to satisfy a dynamic allocation request. This value overrides the user-specifiable system default. (See FILEMOUNT). Legitimate values are: • **YES** = Allow the system to mount a volume to satisfy a dynamic allocation request. • **NO** = Do not allow the system to mount a volume to satisfy a dynamic allocation request.
RECALL(VALUE)	Specifies whether to allow or not allow the system to recall a migrated dataset in order to satisfy a dynamic allocation request. This value overrides the user-specifiable system default. (See FILERECALL). Legitimate values are: • **YES** = Allow the system to recall datasets to satisfy a dynamic allocation request. • **NO** = Do not allow the system to recall datasets to satisfy a dynamic allocation request.

File Access Attributes for PATHMODE parameter

Sub-Parameter	Definition
SIRUSR	Specifies permission for the file owner to read the file.
SIWUSR	Specifies permission for the file owner to write the file.
SIXUSR	Specifies permission for the file owner to search, if the file is a directory, or to execute, for any other file.
SIRWXU	Specifies permission for the file owner to read, write, and search, if the file is a directory, or to read, write, and execute, for any other file. this value is the bit inclusive or of SIRUSR, SIWUSR, and SIXUSR.
SIRGRP	Specifies permission for users in the file group to read the file.
SIWGRP	Specifies permission for users in the file group to write the file.
SIXGRP	Specifies permission for users in the file group to search, if the file is a directory, or to execute, for any other file.
SIRWXG	Specifies permission for users in the file group to read, write, and search, if the file is a directory, or to read, write, and execute, for any other file. This value is the bit inclusive or of SIRGRP, SIWGRP, and SIXGRP.
SIROTH	Specifies permission for users in the file other class to read the file.
SIWOTH	Specifies permission for users in the file other class to write the file.
SIXOTH	Specifies permission for users in the file other class to search, if the file is a directory, or to execute, for any other file.
SIRWXO	Specifies permission for users in the file other class to read, write, and search, if the file is a directory, or to read, write, and execute, for any other file. this value is the bit inclusive or of SIROTH, SIWOTH, and SIXOTH.
SISUID	Specifies that the system set the user id of the process to be the same as the user id of the file owner when the file is run as a program.
SISGID	Specifies that the system set the file group of the process to be the same as the group id of the file owner when the file is run as a program.

File Option descriptions for PATHOPTS parameter

Sub-Parameter	Definition
ORDONLY	Specifies that the program can open the file for reading.
OWRONLY	Specifies that the program can open the file for writing.
ORDWR	Specifies that the program can open the file for reading and writing. Do not use this option for a fifo special file; the result is undefined.
OAPPEND	Specifies that the system sets the file offset to the end of the file before each write, so that data is written at the end of the existing file.
OCREAT	Specifies that the system is to create the file. If the file already exists, the operation will fail if OEXCL is specified, and will open the existing file if OEXCL is not specified.

Sub-Parameter	Definition
OEXCL	Specifies that, if the file already exists, then HFS open file processing will fail. ***Note:*** The system ignores OEXCL if OCREAT is not also specified.
ONOCTTY	Specifies that, if the patenting of the file will not make the terminal device the controlling a terminal device, then op ONONBLOCK Specifies the following, depending on the type of file: For fifo special files: • With ONONBLOCK specified and ORDONLY access: an open() function for reading-only returns without delay. • With ONONBLOCK not specified and ORDONLY access: an open() function for reading-only blocks (waits) until a process opens the file for writing. • With ONONBLOCK specified and OWRONLY access: an open() function for writing-only returns an error if no process currently has the file open for reading. • With ONONBLOCK not specified and OWRONLY access: an open() function for writing-only blocks (waits) until a process opens the file for reading. For character special files: • If ONONBLOCK is specified: an open() function returns without blocking (waiting) until the device is ready or available. Device response depends on the type of device. • If ONONBLOCK is not specified: an open() function blocks (waits) until the device is ready or available. Specification of ononblock has no other effects.
OSYNC	Specifies that the system is to move data from buffer storage to disk (or other permanent storage) before returning control from a callable service that performs a write.
OTRUNC	Specifies that the system is to truncate the file length to zero if all of the following are true: • The file specified on the path operand exists. • The file is a regular file. • The file successfully opened with ordwr or owronly. The system does not change the mode and owner. OTRUNC has no effect on fifo special files or terminal device files.

SDBALLOC/SWSALLOC Examples

```
To allocate an input file with shared control:
rc = SWSALLOC(DSN(SWS.INPUT.FILE) DISP(SHR))
To allocate a new file as output
rc = SWSALLOC("DSN(SWS.OUTPUT.FILE) DDN(OUTFILE)
       DISP(NEW CATALOG DELETE) SPACE(1 5) TRACKS
       LRECL(80) BLKSIZE(3120) RECFM(F B)
       DSORG(PS)")
To allocate a sysout file:
rc = SWSALLOC(DDN(PRTFILE) SYSOUT(A) DEST(RMT3))
```

High-Level Language Interface SDBFREE (SDCPFR) SWSFREE (SWCPFR) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPFR/SWCPFR** .

SDBFREE/SWSFREE is used to de-allocate datasets. The SDBFREE/SWSFREE API dynamically de-allocates a data set and/or an HFS file. You can also change the output class of a sysout dataset, making it immediately available for processing by an output writer, while de-allocating them. The original dataset disposition, set when the dataset was dynamically allocated, can be overridden during de-allocation.

The format of this command is similar in features and functions to the TSO/E Free command. A text string is used as input in order to provide the parameters necessary to de-allocate the specified dataset.

CALL Arguments

The SDBFREE/SWSFREE (SDCPFR/SWCPFR) function arguments are described in the table which follows.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the de-allocation command string. If the length is longer than the actual command, trailing nulls or blanks will be ignored. If the length is less than the actual command string, the de-allocation command string will be truncated and possibly cause execution errors. The maximum string length is 32768 bytes.
2	CHAR *	PIC X(nnnnn)	CHAR (nnnn)	Input	The de-allocation command string. See Supported dynamic de-allocation keywords below.
3	SWSASB *	Usage pointer	PTR	Output	The Shadow WebServer Allocation Status Block. This is an required argument that provides information concerning the status of the de-allocation request.

Return Values

SDBFREE/SWSFREE always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed. Generally this indicates that the file was not allocated. There will be an error message in the Allocation Status Block describing the error.

Supported dynamic de-allocation keywords

The SDBFREE/SWSFREE (SDCPFR/SWCPFR) interface supports the following dataset de-allocation request parameters:

Allocation Keyword	Description
DSN(DSNAME)	Specifies the name of the dataset to be de-allocated. You can only specify a single dataset name. Dataset names must be fully qualified as there will be no prefix appended to the supplied name.
PATH(PATHNAME)	Identifies an HFS file to be de-allocated. A pathname consists of the names of the directories from the root to the file being identified, and then the name of the file. The form is /NAME1/NAME2/.../NAMEn. A pathname begins with a slash (/). The system treats any consecutive slashes like a single slash. The pathname can be 1 to 250 characters. A name can be 1 to 249 characters. Consists of printable characters from x'40' through x'FE'. A pathname is case sensitive. Thus, /usr/joe and /USR/joe define two different files.
DDN(DDNAME)	Specifies the DDNAME, of the datasets, to be de-allocated. This is a required field.
DEST(DESTINATION/NODE USERID)	Remote destination or a User at a specified node to which SYSOUT data sets are to be routed.
OUTDES(OUTPUT_DESCRIPTOR_NAME ...)	Specifies a list of output descriptors that will be associated with the sysout data set. These descriptors are created by //OUTPUT JCL statements in the Shadow WebServer or Shadow Direct procedure. ***Note:*** Separate parameters by spaces. Do not use commas.

Allocation Keyword	Description
SYSOUT(CLASS)	Dataset is to be a system output dataset
DISP(NORMAL)	Specifies the disposition of file upon de-allocation. Legitimate values are: • **UNCATALOG** = Specifies that the file should be uncatalogued. • **CATALOG** = Specifies that the file should be catalog. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted.
HOLD	Dataset is to be placed on a hold queue upon de-allocation.
NOHOLD	Dataset is not to be placed on a hold queue upon unallocation.
PATHDISP(NORMAL)	Specifies the disposition of an HFS file upon normal session termination. Use one of the following dispositions: • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted.
SPIN(VALUE)	Specifies when a sysout data set is printed. Legitimate values are: • **UNALLOC** = Makes the dataset available for printing immediately after the dataset is unallocated from an explicit unallocation or at the end of the session. • **b** = Makes the dataset available for printing at the end of the session.
MESSAGE(VALUE)	Specifies whether or not to display dynamic allocation failure messages on the system console. This value overrides the user-specifiable system default (See FILEMESSAGES). Legitimate values are: • **YES** = Display dynamic allocation failure messages. • **NO** = Do not display dynamic allocation failure messages.
MOUNT(VALUE)	Specifies whether to allow or not allow a volume to be mounted in order to satisfy a dynamic allocation request. This value overrides the user-specifiable system default. (See FILEMOUNT). Legitimate values are: • **YES** = Allow the system to mount a volume to satisfy a dynamic allocation request. • **NO** = Do not allow the system to mount a volume to satisfy a dynamic allocation request.
RECALL(VALUE)	Specifies whether to allow or not allow the system to recall a migrated dataset in order to satisfy a dynamic allocation request. This value overrides the user-specifiable system default. (See FILERECALL). Legitimate values are: • **YES** = Allow the system to recall datasets to satisfy a dynamic allocation request. • **NO** = Do not allow the system to recall datasets to satisfy a dynamic allocation request.

PL/I Example

```
%INCLUDE SPCPHD
       .
       .
DCL  COMMAND CHAR(80)               /* ALLOCATE COMMAND       */
     INIT('DDN(INFILE)');
DCL  CMDLEN  FIXED BIN(31);         /* COMMAND LENGTH         */
DCL  RC      FIXED BIN(31);         /* RETURN CODE            */
CMDLEN = LENGTH(COMMAND);           /* SET COMMAND LEN        */
/*DYNAMICALLY ALLOCATE AN INPUT FILE                          */
CALL SWSFREE(CMDLEN,                /* COMMAND LENGTH         */
             COMMAND,               /* COMMAND                */
             SWSASB);               /* ALLOCATION STATUS BLOCK*/
RC = PLIRETV();                     /* GET RETURN CODE        */
IF RC ¬= SWS_SUCCESS THEN           /* EXIT PROGRAM IF BAD RC */
    EXIT;
```

C Example

```
SWS_ALLOCATION_STATUS_BLOCK swsASB; /* response area          */
         .
         .
         .
         .
long RC;                           /* return code            */
char szCommand[ ] = "DDN(INFILE)"; /* Dynamically de-allocate
                                      a dataset by it's DDName  */
rc = SWSFREE(strlen(szCommand),    /* Command Length         */
                    szCommand,     /* Command                */
                     swsASB);      /* Response area          */
if (rc ¬= SWS_SUCCESS)
         do
           print(swsASB.Error_Message);
           return rc;
         end
```

COBOL Example

```
*     NEON API COPY BOOK
        COPY SBCPHD.
             .
             .
             .
             .
        77   COMMAND-LENGTH       PIC S9(5) COMP.
        77   COMMAND              PIC X(80) VALUE
             'DDN(INFILE)'.
       *     DYNAMICALLY DE-ALLOCATE A DATASET BY IT'S DDNAME
MOVE 80 TO COMMAND-LENGTH.
        CALL SWSFREE
            USING COMMAND-LENGTH,
                  COMMAND,
                  SWS-ALLOCATION-STATUS-BLOCK.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
        IF NOT SWS-SUCCESS
            DISPLAY 'INFILE DE-ALLOCATION FAILED.' UPON CONSOLE
            DISPLAY SWSASB-ERROR-MESSAGE UPON CONSOLE
            GOBACK.
```

SDBFREE/SWSFREE Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level Language Interface available.

The REXX-language SDBFREE/SWSFREE built-in function can be used to dynamically unallocate datasets.

Syntax

The general form for a REXX-language invocation of SDBFREE/SWSFREE is:

```
rc = SDBFREE/SWSFREE("STRING")
```

The format of this command is similar in features and functions to the TSO/E Free command. If an error occurs, the variable **ALLOC.MESSAGE** will be populated with a descriptive error message.

In the event of an error, the DAIR return code can be obtained from the **ALLOC.INFOCODE** and the reason code can be obtained from the **ALLOC.REASON**.

Valid Arguments

DSN(DSNAME)	Specifies the name of the dataset to be de-allocated. You can only specify a single dataset name. Dataset names must be fully qualified as there will be no prefix appended to the supplied name.
PATH(PATHNAME)	Identifies an HFS file to be de-allocated. A pathname consists of the names of the directories from the root to the file being identified, and then the name of the file. The form is /NAME1/NAME2/.../NAMEn. A pathname begins with a slash (/). The system treats any consecutive slashes like a single slash. The pathname can be 1 to 250 characters. A name can be 1 to 249 characters. Consists of printable characters from x'40' through x'FE'. A pathname is case sensitive. Thus, /usr/joe and /USR/joe define two different files.
DDN(DDNAME)	Specifies the DDNAME,of the datasets, to be de-allocated.
DEST(DESTINATION/NODE USERID)	Remote destination or a User at a specified node to which SYSOUT data sets are to be routed.

OUTDES(OUTPUT_ DESCRIPTOR_NAME ...)	Specifies a list of output descriptors that will be associated with the sysout data set. These descriptors are created by //OUTPUT JCL statements in the Shadow Web Server or Shadow Direct procedure. ***Note:*** Separate parameters by spaces. Do not use commas.
SYSOUT(CLASS)	Dataset is to be a system output dataset
DISP(NORMAL)	Specifies the disposition of file upon de-allocation. Use one of the following dispositions: • **UNCATALOG** = Specifies that the file should be uncatalogued. • **CATALOG** = Specifies that the file should be catalog. • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted.
HOLD	Dataset is to be placed on a hold queue upon de-allocation.
NOHOLD	Dataset is not to be placed on a hold queue upon unallocation.
PATHDISP(NORMAL)	Specifies the disposition of an HFS file upon normal session termination. Use one of the following dispositions: • **KEEP** = Specifies that the file should be kept. • **DELETE** = Specifies that the file should be deleted.
SPIN(VALUE)	Specifies when a sysout data set is printed. Legitimate values are: • **UNALLOC** = Makes the dataset available for printing immediately after the dataset is unallocated from an explicit unallocation or at the end of the session. • **NO** = Makes the dataset available for printing at the end of the session.
MESSAGE(VALUE)	Specifies whether or not to display dynamic allocation failure messages on the system console. This value overrides the user-specifiable system default (See FILEMESSAGES). Legitimate values are: • **YES** = Display dynamic allocation failure messages. • **NO** = Do not display dynamic allocation failure messages.
OUNT(VALUE)	Specifies whether to allow or not allow a volume to be mounted in order to satisfy a dynamic allocation request. This value overrides the user-specifiable system default. (See FILEMOUNT). Legitimate values are: • **YES** = Allow the system to mount a volume to satisfy a dynamic allocation request. • **NO** = Do not allow the system to mount a volume to satisfy a dynamic allocation request.
RECALL(VALUE)	Specifies whether to allow or not allow the system to recall a migrated dataset in order to satisfy a dynamic allocation request. This value overrides the user-specifiable system default. (See FILERECALL). Legitimate values are: • **YES** = Allow the system to recall datasets to satisfy a dynamic allocation request. • **NO** = Do not allow the system to recall datasets to satisfy a dynamic allocation request.

SDBFREE/SWSFREE Examples

```
To free a specific DSName:
    rc = SWSFREE("DSN(MYPFX.DSNAME)")
To free a specific DDName:
    rc = SWSFREE("DDN(INFILE)")
To route a file to sysout:
    rc = SWSFREE("DDN(PRTFILE) SYSOUT(A)")
```

High-Level Language Interface SDBVALUE (SDCPVL) SWSVALUE (SWCPVL) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPVL/SWCPVL**.

SDBVALUE/SWSVALUE is used to fetch or set transaction run-time variable values. The HLL API can operate upon the following variable types:

Variable Type	Fetch Existing Values	Assign New/Changed Value
GLOBAL	Yes	Yes
GLVEVENT	Yes	Yes
Event-Related (See also, WWW.)	Yes	No

CALL Arguments

The SDBVALUE/SWSVALUE function call requires either six or seven arguments. The seventh argument is required for value fetch requests; It must be omitted for value update operations.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A four-byte flag-word indicating the sub-function to be performed. One of the following manifest constants should be used to indicate the desired operation. The values are mutually exclusive; only one can be used. • **SWS_VALUE_OBTAIN** fetches the current value of the named variable. If the variable is not initialized, return an error (SWS_NO_DATA_FOUND) to the caller. • **SWS_VALUE_VALUE** fetches the current value of the named variable. If the variable is not initialized, the upper-case name of the variable is returned as its current value. (This matches the behavior of REXX-language procedures, where the value of an un-initialized variable is the name of the variable, itself.) • **SWS_VALUE_UPDATE** replaces the value of an already named variable with the new value specified by the caller. If the variable is un-initialized, create the variable and assign the specified value to it.
3	UCHAR *	PIC X(nnn)	CHAR (nnn)	Input	The character name of the variable to be acted upon. The variable name string can be null-terminated, or the size can be explicitly specified by the 'SVASZ' argument. The maximum length of any variable name passed to the SWSVALUE API can not exceed 50 bytes.
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the variable name specified by the third argument. This argument can be an integer fullword value in the range 5-to-50. You can also use the manifest constant, SWS_NTS, to specify that variable name is a null-terminated string.
5	PTR	PIC X(nnn)	CHAR (nnn)	InOut	For the SWS_VALUE_OBTAIN or SWS_VALUE_VALUE sub-function, this argument is the address of the data buffer which will receive the fetched variable value. Fetched values are always returned as a null-terminated string, even if the value must be truncated to fit within the supplied buffer space. For the SWS_VALUE_UPDATE sub-function, this argument specifies the address of the value data to be assigned to the new/updated variable. ***Note:*** The implementation maximum size for the value of any variable is 32,000 bytes; 8,000 if used in HTML extension substitution processing.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
6	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	For the SWS_VALUE_OBTAIN or SWS_VALUE_VALUE sub-function, this argument specifies the total size of the buffer area (argument five). For the SWS_VALUE_UPDATE sub-function, this argument specifies the size of the variable value, given by the argument five, which is to be assigned to the variable.
7	SDWORD*	PIC S9(5) COMP	FIXED BIN(31)	Output	For the SWS_VALUE_OBTAIN or SWS_VALUE_VALUE sub-function, this argument receives the actual size of the variable's value. • If the return value is shorter than the supplied buffer area, this argument receives the actual number of bytes used to store the value within the output buffer. • If the number of bytes required to store the fetched value is greater than or equal to the return buffer area size, then the fetched value is truncated, and a null terminator is placed in the last buffer position. This argument receives the count of bytes actually needed to save the entire value. For the SWS_VALUE_UPDATE sub-function, C-language callers should code NULL for this argument. COBOL and PL/I callers should omit this argument.

Return Values

SDBVALUE/SWSVALUE always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded. The variable's value was fetched or updated as requested.
SWS_SUCCESS_WITH_INFO, SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set for fetch operations when the returned variable value has been truncated.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error. For instance, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.

Return Value	Description
SWS_NO_DATA_FOUND, SQL_NO_DATA_FOUND	For a SWS_VALUE_OBTAIN sub-function request, the named variable is not initialized.
SWS_INVALID_HANDLE, SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR/SWSERROR

PL/I Example

```
DCL  SCONN  PTR;                    /* Connection Handle      */
DCL  SVANA  CHAR(50);               /* variable name          */
DCL  SVASZ  FIXED BIN(31);          /* variable name size     */
DCL  SBUFF  CHAR(256);              /* buffer area            */
DCL  SBFSZ  FIXED BIN(31) INIT(256);/* Buffer size            */
DCL  SRTSZ  FIXED BIN(31);          /* Fetched value size     */
DCL  RC FIXED BIN(31);              /* return code            */
DCL  DMHX   FIXED BIN(31) BASED;    /* Dummy Handle field     */
ADDR(SCONN)->DMHX = 0;              /* Clear Connection Handle*/
SVANA    = 'WWW.VAR.FORMFIELD';     /* Set variable name      */
SVASZ    = 17;                      /* set variable name length*/
CALL SWSVALUE( SCONN                /* get the variable value */
       SWS_VALUE_VALUE,
       SVANA,
       SVASZ,
       SBUFF,
       SBFSZ,
       SRTSZ );
RC = PLIRETV();                     /* get return code        */
IF (RC ^= SWS_SUCCESS &             /* exit program if bad RC */
    RC ^= SWS_SUCCESS_WITH_INFO) THEN
EXIT;

SVANA    = 'GLVEVENT.FORMFIELD';    /* Set variable name      */
SVASZ    = 18;                      /* set variable name length*/
SBUFF    = 'Html Form Field Value'; /* set variable value data*/
SBFSZ    = 21;                      /* length of value data   */
CALL SWSVALUE( SCONN                /* create GLVEVENT variable*/
SWS_VALUE_UPDATE,
  SVANA,
  SVASZ,
  SBUFF,
  SBFSZ );
RC = PLIRETV();                     /* get return code        */
IF (RC ^= SWS_SUCCESS &             /* exit program if bad RC */
    RC ^= SWS_SUCCESS_WITH_INFO) THEN
  EXIT;
```

C Example

```
HDBC sConn   = NULL;               /* Connection Handle        */
char sVana[] = "WWW.VAR.FORMDATA"; /* variable name            */
char sVana2[] = "GLVEVENT.DATA";   /* variable name            */
char sBuff[256];                   /* return buffer area       */
SDWORD sRtsz;                      /* return variable size     */
long RC;                           /* return code              */

rc = SWSValue( &sConn,             /* get query variable value */
               SWS_VALUE_VALUE,    /* subfunction = retrieve   */
               sVana,              /* null-terminated name     */
               SWS_NTS,            /* indicate null-terminated */
               sBuff,              /* return buffer address    */
               sizeof(sBuff),      /* maximum buffer size      */
               &sRtsz );           /* actual size return area  */
if (rc ^= SWS_SUCCESS) return;     /* exit program if bad RC   */

rc = SWSValue( &sConn,             /* set new variable value   */
               SWS_VALUE_UPDATE,   /* subfunction = set value  */
               sVana2,             /* null-terminated name     */
               SWS_NTS,            /* indicate null-terminated */
               sBuff,              /* value information buffer */
               sRtsz,              /* size of value data       */
NULL);                             /* Must be NULL for UPDATE  */

if (rc ^= SWS_SUCCESS) return;     /* exit program if bad RC   */
```

COBOL Example

```
 77  SCONN               USAGE IS POINTER.
 77  SBUFF              PIC X(80).
 77  SBFSZ              PIC S9(5) COMP VALUE 80.
 77  SVANA              PIC X(50) VALUE 'WWW.INPUTURL'.
 77  SVANA2             PIC X(50) VALUE 'GLVEVENT.ABC'.
 77  SVASZ              PIC S9(5) COMP VALUE 50.
 77  SRTSZ              PIC S9(5) COMP.
*
*    Obtain input URL value
*
 CALL 'SWCPVL' USING SCONN,
               SWS-VALUE-VALUE,
               SVANA,
               SVASZ,
               SBUFF,
               SBFSZ,
               SRTSZ.
 MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
 IF NOT SWS-SUCCESS GOBACK.
*
*    Set GLVEVENT.ABC to the same value
*
 MOVE SRTSZ TO SBFSZ.
 CALL 'SWCPVL' USING SCONN,
               SWS-VALUE-UPDATE,
               SVANA2,
               SVASZ,
               SBUFF,
               SBFSZ.
 MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
 IF NOT SWS-SUCCESS GOBACK.
```

SDBVALUE/SWSVALUE Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters (see SWSVALUE For other REXX interpreters).
✔	High-level language interface available.

Using this function, you can manipulate Global Variables in ways which are not possible in standard Shadow/REXX. For example, the SDBVALUE/SWSVALUE function lets you use compound symbols as a kind of data base.

This Shadow/REXX intrinsic function has been exported for use by other REXX interpreters (see Invoking other REXX interpreters for additional information).

However, the exported functionality differs considerably from the functionality described here, by allowing other REXX interpreters to access WWW Event-Related and GLVEVENT. Variables.

Consult the page SDBVALUE/SWSVALUE For other REXX interpreters for information on using this function from other REXX interpreters. The information on this page applies only to the Shadow/REXX implementation!

Use this function to access only Global Variables. Do not use this function to access WWW. Event-Related Variables or GLVEVENT. Temporary Variables.

You need use this function only when some special interrogation or serialization processing is required by your Shadow/REXX procedure. Under normal circumstances, you can reference or set the value of a Global variable, simply by using it within a normal REXX-language statement, as in the following:

```
SAVENAME = GLOBAL.COMPANY.NAME
GLOBAL.COMPANY.NAME = "NEON Systems, Inc."
GLVEVENT.MYDATA = "ABC"
```

You should not create too many global variables under a single global variable stem. If you do, you will no longer be able to view them using the SWS/IPSF 7.1 Option, nor access them using the SWSVALUE function.

The absolute product limit is 32,768 variables under a single global variable stem. However, in practice, we strongly recommend that no more than 10,000 global variables exist at any given instant under a single global variable stem.

Syntax

The general form for invocation of SDBVALUE/SWSVALUE is:

```
var = SDBVALUE/SWSVALUE(derivedname, actioncode, newval, oldval)
```

Valid Arguments

derivedname	Gives the name of the symbol to be acted on. When you use this argument without quotation marks, simple symbols (which are case sensitive) following the stem are replaced by their values.
Actioncode	Specifies the action to be taken on the symbol. The table below indicates what actions are taken.
newval	Supplies the new value (if any) to assign to the symbol.
oldval	Fetches the value of the symbol before the action takes place.

Return Values

SDBVALUE/SWSVALUE returns a value from the function call, and, in the case of some action codes, also places information in the external data queue.

The Chart below shows what actions are performed for each of the action codes values, and what values are returned.

Action Code	Description
A (Add)	Adds a number specified by increment, to the existing compound symbol given by derivedname. Returns the sum of the compound symbol and the increment Does not change the external data queue All references to the compound symbol are serialized during the ADD operation. That is, you can use this function safely to increment a counter that is set by concurrent tasks. **Syntax Example:** `val = SWSVALUE(derivedname,'A',increment)`
C (Compare and Update)	Updates a compound symbol after verifying its current value. Safely updates Global symbols shared by more than one event procedure or symbols that multiple copies of the same rule might access and update. Does not change the Shadow/REXX external data queue. Returns the REXX "true" value (1), if the comparison found the symbol's pre-action value to be equal to old value and the compound symbol was updated, or the REXX "false" value (0), if the comparison found unequal values and therefore did not update the value of the compound symbol. Serializes the compare and update operations for global variables. **Syntax Example:** `val = SWSVALUE(derivedname,'C',newval,oldval)`

Action Code	Description
D (Drop)	Performs the Shadow/REXX DROP operation on the compound symbol specified by derivedname. The compound symbol is reset to its "uninitialized" value; that is, its derived name. If derivedname is the name of a stem, then all compound symbols belonging to that stem are not just dropped, but also rendered "nonexistent" and the virtual storage allocated to them is released. Returns the value of derivedname. Does not change the external data queue. All other references either see the compound symbol as it existed before the DROP operation began, or as it is after the DROP operation completes. **Syntax Example:** `val = SWSVALUE(derivedname,'D')`
E (Existence)	Checks to see whether a given global variable exists. Does not change the Shadow/REXX external data queue. Returns the status of a given global variable as one of these characters: • **I** for Initialized. • **U** for Uninitialized. • **N** for Does not exist. **Syntax Example:** `val = SWSVALUE(derivedname,'E')` ***Note:*** For normal REXX symbols N and U would have interchangeable meanings. However, for global variables, N means that no storage exists for a variable; and U means that the variable exists in storage, but is uninitialized and so is set to the value of its name.
F (Find)	Checks to see if a given global variable exists. The F action is more efficient and more reliable than using the E and O functions together. Returns the status of a given global variable as one of these characters: • **I** for Initialized. • **U** for Uninitialized. • **N** for Does not exist. When the returned value is not N (meaning that the derived name exists), the value of the node is returned on the external data queue. The maximum length of a string pulled from the external data queue is 350 bytes. Longer values are truncated. **Syntax Example:** `val = SWSVALUE(derivedname,'F')`

<table>
<tr><th>Action Code</th><th>Description</th></tr>
<tr><td>I (Information)</td><td>
Returns to the external data queue information about all of the immediate subnodes of the derivedname.

The derivedname value must be a compound symbol node. The return value is the number of immediate subnodes that exist. The external data queue contains two lines per subnode: the first line contains the next segment of the derived name, and the second line contains statistics

about the derived name. The second line returned for each derived name contains the information shown below:

<table>
<tr><th>Word</th><th>Length</th><th>Description</th></tr>
<tr><td>1</td><td>8</td><td>Number of subnodes under this subnode</td></tr>
<tr><td>2</td><td>8</td><td>Create date (in the form yy/mm/dd)</td></tr>
<tr><td>3</td><td>8</td><td>Create time (in the form hh:mm:ss)</td></tr>
<tr><td>4</td><td>17</td><td>Create event procedure or program name</td></tr>
<tr><td>5</td><td>8</td><td>Create jobname, taskname, or TSO ID</td></tr>
<tr><td>6</td><td>8</td><td>Last modification date</td></tr>
<tr><td>7</td><td>8</td><td>Last modification time</td></tr>
<tr><td>8</td><td>17</td><td>Last modification event procedure or program name</td></tr>
<tr><td>9</td><td>8</td><td>Last modification jobname, taskname or TSO ID</td></tr>
<tr><td>10</td><td>8</td><td>Number of access to this node</td></tr>
<tr><td>11</td><td>8</td><td>Number of updates to this node</td></tr>
</table>

Returns the number of subnodes listed in the external data queue.

Places two lines per subnode in the external data queue.

Returns no partially-updated symbol names.

Syntax Example:
<code>val = SWSVALUE(derivedname,'I')</code>
</td></tr>
</table>

Action Code	Description
L (List)	Lists the derived names of all the immediate subnodes of derivedname by placing them on the external data queue. The results of this action illustrate the difference between dropped symbols (processed by action D) and removed symbols (processed by action R). Dropped symbols still exist, so the List action can find them. The List action does not return removed symbols. Returns the number of subnodes listed in the external data queue. Places a list of subnodes of the specified nodes in the external data queue. **Syntax Example:** `val = SWSVALUE(derivedname,'L')`
O (Obtain)	Obtains the value of a global variable. If the global variable does not exist, Shadow/REXX returns an error. Does not change the external data queue. **Syntax Example:** `val = SWSVALUE(derivedname,'O')`
R (Remove)	Removes the node specified by derivedname and all of its subnodes. Once a node is removed, it ceases to exist. Returns the number of subnodes removed. Does not change the external data queue. Does not allow other accessors of compound symbols to see partially-updated symbols. **Syntax Example:** `val = SWSVALUE(derivedname,'R')`
S (Subtree)	Lists the derived names of all the subnodes of derivedname in the external data queue. Action code S is similar to code L with two differences: • Shadow/REXX places the entire global variable name in the external data queue. • All subnodes of the derived name are listed. Returns the number of subnodes listed in the external data queue. Places the entire global variable name in the external data queue. Returns no partially-updated symbol names **Syntax Example:** `val = SWSVALUE(derivedname,'S')`

Action Code	Description
T (Subtree/Info)	Returns to the external data queue information on all the subnodes of the derivedname The derivedname value parameter must be a compound symbol node. The return value is the number of subnodes that exist. The external data queue contains two lines per subnode: the first line contains the next segment of the derived name, and the second line contains statistics about the derived name. The second line contains information in the format shown for the I actioncode. Action code T resembles code I with three differences: The entire global variable name goes into the external data queue. All subnodes of the derived name are listed. The "Number of Subnodes" field on the second line of pair of messages in the external data queue for each node always contains zero. Returns the number of subnodes listed in the external data queue Places in the external data queue two lines per subnode and the entire Global Variable name. Returns no partially-updated symbol names **Syntax Example:** `val = SWSVALUE(derivedname,'S')`
U (Update)	Assigns newvalue as the value of the compound symbol specified by derivedname. If the compound does not exist, Shadow/REXX creates it and gives it the new value. Returns the variable specified by newvalue. Does not change the external data queue Prevents others accessing compound symbols from seeing partially-updated symbols **Syntax Example:** `val = SWSVALUE(derivedname,'U',newval)`
V (Value)	Returns the current value of the node specified by derivedname. If the node does not exist, Shadow/REXX creates it but assigns it no value (giving the symbol the same value as its name). Returns the value of the specified compound symbol. Does not change the external data queue. Prevents the issuer of SWSVALUE from seeing partially-updated symbols. **Syntax Example:** `val = SWSVALUE(derivedname,'V')`

High-Level Language Interface SQLTOKEN (SDCPTK) SWSTOKEN (SWCPTK) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPTK/SWCPTK.**

The SQLTOKEN/SWSTOKEN service provides a means of saving and restoring transaction-oriented data using a server-created token value.

Transaction data can be saved before generating an out-bound response to a Web transaction, and then be restored (using the token value) when the next transaction arrives.

The token service allows you to create complex, inter-active Web transactions which need a scratch-pad area to save state information between Web transaction boundaries.

All tokens have a timeout associated with them at creation time. If the token is not accessed within the timeout period the Server automatically deletes the token (along with the associated data).

CALL Arguments

The SQLTOKEN/SWSTOKEN service takes from three to eight arguments, depending on the sub-function being requested.

The Create Sub-function

The create sub-function causes the Server to create a new token and save an initial data value. The service returns the 24-byte token identifier. The token identifier is used to request other operations against the token.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	SWS_TOKEN_CREATE			Input	A four-byte flag-word indicating the type of operation to be performed. The constant shown invokes the create sub-function.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
3	UCHAR*	PIC X(25)	CHAR(25)	Output	The buffer area where the service returns the 24-byte token identifier. The 24-byte token ID is returned with a 1-byte null terminator value in the 25th position.
4	PTR	PIC X(nnn)	CHAR(nnn)	Input	The buffer area containing the initial data to be saved when the token is created. You can not specify a null-terminated string for this argument. The length of the data must be explicitly given by the fifth argument.
5	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the initial data to be saved for the token. You must explicitly provide the size of the forth argument; SWS_NTS is not a valid argument value.
6	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The timeout value for the token, expressed in seconds. The token service deletes un-referenced tokens after some defined time period. The time period for auto-delete can be expressed using this argument.
7	PTR	PIC X(nnn)	CHAR(nnn)	Input	Optional argument. If used, this argument is the user data value to be associated with the token. The value can be a null-terminated string.
8	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	Optional argument. If used, this argument is the length of the user data given by the 7th argument. You may specify SWS_NTS to indicate that the user data value is a null terminated string.

The Delete Sub-function

The delete sub-function causes the Server to delete a token.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	SWS_TOKEN_ DELETE			Input	A four-byte flag-word indicating the type of operation to be performed. The constant shown invokes the delete sub-function.
3	UCHAR*	PIC X(24)	CHAR(24)	Intput	The buffer area where the service obtains the 24-byte token identifier of the token to be deleted.
4	PTR	PIC X(nnn)	CHAR (nnn)	Input	The buffer area containing the initial data to be saved when the token is created. You can not specify a null-terminated string for this argument. The length of the data must be explicitly given by the fifth argument.

The Get Sub-function

The get sub-function retrieves the data associated with a token.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	SWS_TOKEN_ GET			Input	A four-byte flag-word indicating the type of operation to be performed. The constant shown invokes the get sub-function
3	UCHAR*	PIC X(24)	CHAR(24)	Input	The buffer area where the service obtains the 24-byte token identifier of the token to be referenced.
4	PTR	PIC X(nnn)	CHAR(nnn)	Output	The buffer area where the service returns the data value associated with the token. If the actual token data is larger than this area, the returned value will be truncated to fit within this area.
5	SDWORD	PICS9(5) COMP	FIXED BIN(31)	Input	The size of the return data area buffer, specified by the forth argument.
6	SDWORD *	PICS9(5) COMP	FIXED BIN(31)	Output	The token service returns the actual size of the data value associated with the token into this area.

The Put Sub-function

The put sub-function updates part or all of the data value associated with a token. Note that the put sub-function cannot be used to alter the length of the data value associated with a token; only to re-write some or all of it.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	SWS_TOKEN_ PUT			Input	A four-byte flag-word indicating the type of operation to be performed. The constant shown invokes the put sub-function .
3	UCHAR*	PIC X(24)	CHAR(24)	Intput	The buffer area where the service obtains the 24-byte token identifier of the token to be updated.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
4	PTR	PIC X(nnn)	CHAR(nnn)	Output	The buffer area where the service obtains the new data value to be written for the token. If the size of this area is smaller than the actual token data, this data will overwrite only the leading portion of the token's data value; the remainder will be un-changed. If this data value is larger than the actual token data, the service rewrites only the portion corresponding to the existing actual token data length. (I.E. This service cannot be used to enlarge the data area associated with a token.)
5	SDWORD	PICS9(5) COMP	FIXED BIN(31)	Input	The size of the data area given by the forth argument.

The Replace Sub-function

The replace sub-function re-writes the data value associated with a token. The original data value is purged, and the new value becomes associated with the token. Use this sub-function (not the put sub-function) to change the size of the data associated with a token.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage pointer	PTR	Input	The Web Server connection connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	SWS_TOKEN_ REPLACE			Input	A four-byte flag-word indicating the type of operation to be performed. The constant shown invokes the replace sub-function
3	UCHAR*	PIC X(24)	CHAR(24)	Input	The buffer area where the service obtains the 24-byte token identifier of the token to be replaced.
4	PTR	PIC X(nnn)	CHAR(nnn)	Output	The buffer area where the service obtains the new data value to be written for the token. The new data value completely replaces the existing data value.
5	SDWORD	PICS9(5) COMP	FIXED BIN(31)	Input	The size of the data area given by the forth argument.

Return Values

SWSTOKEN/SQLTOKEN always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS, SQL_SUCCESS	The operation succeeded without error.
SWS_SUCCESS_WITH_INFO, SQL_SUCCESS_WITH_INFO	The operation succeeded. This return value is set when data is truncated for the get sub-function.
SWS_NO_DATA_FOUND, SQL_NO_DATA_FOUND	The requested token could not be found.
SWS_ERROR, SQL_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR/SQLERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error. For instance, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server may provide diagnostic information in the wrap-around trace.
SWS_INVALID_HANDLE, SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using SQLERROR/SWSERROR.

PL/I Example

```
DCL  SCONN PTR;                    /* Connection Handle      */
DCL  STKID CHAR(25);               /* token ID value         */
DCL  SDATA CHAR(256);              /* data value             */
DCL  SSIZE FIXED BIN(31);          /* data value size        */
DCL  STMOU FIXED BIN(31) INIT(300); /* timeout value        */
DCL  RC    FIXED BIN(31);          /* return code            */
DCL  DMHX  FIXED BIN(31) BASED;    /* Dummy Handle field     */
ADDR(SCONN)->DMHX = 0;             /* Clear Connection Handle*/
SDATA= 'Hello World!';             /* Set output area        */
SSIZE    = 12;                     /* set length            */
CALL SWSTOKEN( SCONN               /* create a token         */
       SWS_TOKEN_CREATE,
       STKID,
       SDATA,
       SSIZE,
       STMOU );
RC = PLIRETV();                    /* get return code        */
IF RC ^= SWS_SUCCESS THEN          /* exit program if bad RC */
 EXIT;
```

C Example

```
HDBC sConn      = NULL;              /* Connection Handle     */
char sData[]    = "Token Data Area"; /*data string definition */
char sTkid[25];                      /* Token ID return area  */
long RC;                             /* return code           */
rc = SWSTOKEN( &sConn,               /* create the token      */
     SWS_TOKEN_CREATE,
     sTkid,
     sData,
     sizeof(sData) );
if (rc ^= SWS_SUCCESS) return;       /* exit program if bad RC*/
```

COBOL Example

```
77  SCONN                   USAGE IS POINTER.
77  STKID                    PIC X(25).
77  SDATA                   PIC X(80).
77  SSIZE                   PIC S9(5) COMP.
77  STMOU                   PIC S9(5) COMP VALUE 300.
77  SUSDA                   PIC X(30) VALUE 'User Data Area'.
77  SUSLN                   PIC S9(5) COMP VALUE 30.
MOVE 'HELLO WORLD!'          TO SDATA.
MOVE 12                      TO SSIZE.
CALL 'SWCPTK' USING SCONN,
       SWS-CREATE-TOKEN,
       STKID,
       SDATA,
       SSIZE,
       STMOU,
       SUSDA,
       SUSLN );
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.
```

SDBTOKEN/SWSTOKEN Function

✔	Can be used in Shadow/REXX
✔	Can be used from other REXX interpreters
✔	High-level language interface available

The SDBTOKEN/SWSTOKEN built-in function provides a means of saving and restoring transaction-oriented data using a server-created token value.

Transaction data can be saved before generating an out-bound response to a transaction, and then be restored (using the token value) when the next transaction arrives.

The token service allows you to create complex, inter-active transactions which need a scratch-pad area to save state information between transaction boundaries.

All tokens have a timeout associated with them at creation time. If the token is not accessed within the timeout period the Server automatically deletes the token (along with the associated data).

Syntax

The general form for invocation of SDBTOKEN/SWSTOKEN is:

```
var = SWSTOKEN( func, arg2, arg3, arg 4 )
```

Valid Arguments

The first argument, func, to the SDBTOKEN/SWSTOKEN function specifies the sub-function to be performed. The values which can be coded for the func argument are:

CREATE	Create a new token and save the associated data.
GET	Retrieve data associated with a token value.
PUT	Update the data associated with an existing token value.
REPLACE	Replace the data associated with an existing token value.
DELETE	Delete a token and the associated data.

CREATE Service

The sub-function creates a new token value and saves data associated with the token. The data can later be retrieved or updated, using the token value.

Syntax Example

The CREATE service is invoked by coding:

```
newtok = SWSVALUE('CREATE',data,timeout,userdata)
```

The arguments to the CREATE service call are:

data	The data to be associated with the token. This operand is required. Once the token is created, the size of the data can only be altered using the REPLACE token service (PUT cannot be used to change the data size). When invoked from Shadow/REXX, the maximum size of the data area is limited to 32,000 bytes.
Timeout	A expiration timeout value to be associated with the token, specified in seconds. If the token value is un-accessed for this length of time, the token value (and associated data) is discarded by the system. Any access to the token value causes this expiration timer to be restarted. If this argument to the function call is omitted, the system uses the value set for the TOKENTIMEOUT product parameter.
Userdata	Specifies an optional character string which is associated with the token. If this operand is not specified, the user data value is set to the value of the original URL under which the token was created. This character string is displayed on the active tokens ISPF display, but has no other purpose. This argument is optional. .

Return Value

The token CREATE service always returns the 24-byte token value. A run-time error is generated if the token cannot be created.

GET Service

The sub-function retrieves the data associated with a previously created token.

Syntax Example

The GET service is invoked by coding:

```
data = SWSVALUE('GET',token)
```

The arguments to the GET service call are:

Token	The token value returned from the CREATE function.

Return Value

The token GET service returns the data associated with the token when it was created. The length of the data returned is always equal to the length of the data associated with the token when it was created.

If the input token value is unknown the function returns a NULL string. This can occur because another application has caused the token to be explicitly deleted or because the token timeout period has expired.

PUT Service

The sub-function updates the data associated with a previously created token.

Syntax Example

The PUT service is invoked by coding:

```
rc = SWSVALUE('PUT',token,newdata)
```

The arguments to the PUT service call are:

token	The token value returned from the CREATE function.
newdata	The new data value to be written to the token. If this data value is longer than the value originally written, it is truncated to the length of the original data value. If the new value is shorter than the previous value, it overlays only the front portion of the value. Note that the REPLACE service must be used to alter the size of the data associated with a token.

Return Value

The token PUT service returns one of the following numeric values:

Return Value	Description
0	The newdata value was saved.
100	The token value is unknown or invalid. For example, the token has been explicitly deleted or the timeout period has elapsed.

REPLACE Service

The sub-function replaces the data associated with a previously created token.

Syntax Example

The REPLACE service is invoked by coding:

```
rc = SWSVALUE('REPLACE',token,newdata)
```

The arguments to the REPLACE service call are:

token	The token value returned from the CREATE function.
newdata	The new data value to be written to the token. This data completely replaces the data associated with the token.

Return Value

The token REPLACE service returns one of the following numeric values:

Return Value	Description
0	The newdata value was saved.
100	The token value is unknown or invalid. For example, the token has been explicitly deleted or the timeout period has elapsed.

DELETE Service

The sub-function deletes a token and associated data and removes is from the system.

Syntax Example

The DELETE service is invoked by coding:

```
rc = SWSVALUE('DELETE',token)
```

The arguments to the DELETE service call are:

token	The token value returned from the CREATE function.

Return Value

The token DELETE service returns one of the following numeric values:

Return Value	Description
0	The token has been deleted.
100	The token value is unknown or invalid. For example, the token has been explicitly deleted or the timeout period has elapsed.

High-Level Language Interface SDBCONCT (SDCPCC) SWSCONCT (SWCPCC) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPCC/SWCPCC**.

SDBCONCT/SWSCONCT is used to concatenate multiple DDNames under a single DDName.

The format of this command is similar in features and functions to the TSO/E CONCAT command. A text string is used as input in order to provide the parameters necessary to define the files to be concatenated. Files can be "de-concatenated" using the SDBDECON/SWSDECON command.

CALL Arguments

The SDBCONCT/SWSCONCT (SDBPCC/SWCPCC) function arguments are described in the table which follows. Only two of the three arguments are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the concatenation command string. If the length is longer than the actual command, trailing nulls or blanks will be ignored. If the length is less than the actual command string, the concatenation command string will be truncated and possibly cause execution errors. The maximum string length is 32768 bytes.
2	CHAR*	PIC X(nnnnn)	CHAR (nnnnn)	Input	The concatenation command string. See Supported Concatenation keywords below.
3	SWSASB*	Usage pointer	PTR	Output	The Shadow Web Server Allocation Status Block. This is an optional argument that provides information concerning the status of the concatenation request. If you do not specify this argument, you will not have access to the reason code nor the DAIR code.

Return Values

SDBCONCT/SWSCONCT always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	The operation succeeded. The specified operation was performed.
SWS_ERROR	A parameter validation or run-time error was encountered. Error information is available using the SWSERROR function.
Any other value	The operation failed. Generally this indicates that the file was not concatenated. There will be an error message in the Allocation Status Block describing the error.

Supported Concatenation Keywords

The SDBCONCT/SWSCONCT (SDBPCC/SWCPCC) interface supports the following dataset concatenation request parameters:

Concatenation Keyword	Description
DDN(DD1 DD2 DD3 ...)	Specifies a list of previously allocated ddnames to be concatenated together. ***Note:*** Separate parameters by spaces. Do not use commas. The datasets will be concatenated as a single DDName using the first DDName in the list.
PERM(VALUE)	Specifies whether or not to permanently concatenate these DDNames. Files that are permanently concatenated can not be "de-concatenated". Legitimate values are: • **YES** = Permanently concatenate these files. • **NO** = Do not permanently concatenate these files. ***Note:*** The default is **NO.**

PL/I Example

```
%INCLUDE SPCPHD
 .
 .
DCL COMMAND CHAR(80)                /* CONCATENATE CMD         */
           INIT('DDN(INFILE1 INFILE2)');
DCL CMDLEN FIXED BIN(31);           /* COMMAND LENGTH          */
DCL RC FIXED BIN(31);               /* RETURN CODE             */
CMDLEN = LENGTH(COMMAND);           /* SET COMMAND LEN         */
/* CONCATENATE THE INPUT FILES                                */
CALL SWSCONCT(CMDLEN,               /* COMMAND LENGTH          */
COMMAND,                            /* COMMAND                 */
SWSASB);                            /* ALLOCATION STATUS BLOCK*/
RC = PLIRETV();                     /* GET RETURN CODE         */
IF RC ^= SWS_SUCCESS THEN           /* EXIT PROGRAM IF BAD RC */
  EXIT;
```

C Example

```
SWS_ALLOCATION_STATUS_BLOCK swsASB; /* response area          */
       .
       .
       .
       .
long RC;                            /* return code           */
char szCommand[] = "DDN(INFILE1 INFILE2)";


/* Concatenate the input files                                */

rc = SWSCONCT(strlen(szCommand),    /* Command Length        */
              szCommand,            /* Command               */
              swsASB);              /* Response area         */

if (rc ^= SWS_SUCCESS)
       do
         printf(swsASB.Error_Message);
         return rc;
       end
```

COBOL Example

```
*     NEON API COPY BOOK
    COPY SBCPHD.
         .
         .
         .
         .
    77   COMMAND-LENGTH       PIC S9(5) COMP.
    77   COMMAND              PIC X(80)
         VALUE 'DDN(INFILE1 INFILE2)'.

  *     CONCATENATE THE INPUT FILES

    MOVE 80 TO COMMAND-LENGTH.
    CALL SWSCONCT
        USING COMMAND-LENGTH,
              COMMAND,
              SWS-ALLOCATION-STATUS-BLOCK.

    MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
    IF NOT SWS-SUCCESS
        DISPLAY 'FILE CONCATENATION FAILED.' UPON CONSOLE
        DISPLAY SWSASB-ERROR-MESSAGE
```

SDBCONCT/SWSCONCT Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level language interface available.

The REXX-language SDBCONCT/SWSCONCT built-in function can be used to concatenate multiple ddnames of previously allocated datasets. Datasets concatenated using the SWSCONCT built-in function can use the SDBDECON/SWSDECON built-in function to de-concatenate the DDName.

> ***Note:***
> Because of comparable functionality of SDBCONCT/SWSCONCT to IBM's CONCAT function, this documentation is similar to IBM's TSO/E online help.

Syntax

The format of this command is similar in features and functions to the TSO/E CONCAT command. If an error occurs, the REXX variable, **ALLOC.MESSAGE**, will be populated with a descriptive message. The DAIR return code can be obtained from **ALLOC.INFOCODE** and the reason code can be obtained from the **ALLOC.REASON**.

The general form for a REXX-language invocation of SDBCONCT/SWSCONCT is:

```
rc = SWSCONCT("STRING")
```

Supported Concatenation Keywords

The SDBCONCT/SWSCONCT (SDBPCC/SWCPCC) interface supports the following dataset concatenation request parameters:

Concatenation Keyword	Description
DDN(DD1 DD2 DD3 ...)	Specifies a list of previously allocated ddnames to be concatenated together. ***Note*:** Separate parameters by spaces. Do not use commas. The datasets will be concatenated as a single DDName using the first DDName in the list.

Concatenation Keyword	Description
PERM(VALUE)	Specifies whether or not to permanently concatenate these DDNames. Files that are permanently concatenated can not be "de-concatenated". Legitimate values are: • **YES** = Permanently concatenate these files. • **NO** = Do not permanently concatenate these files. ***Note*:** The default is **NO**

SDBCONCT/SWSCONCT Examples

To concatenate multiple ddnames

```
rc = SWSCONCT(DDN(INFILE1 INFILE2))
```

High-Level Language Interface SDBDECON (SDCPDC) SWSDECON (SWCPDC) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is SDCPDC/SWCPDC.

SDSBDECON/SWSDECON is used to "de-concatenate" a DDName that was previously concatenated using the SDBCONCT/SWSCONCT command.

The format of this command is similar in features and functions to the TSO/E DECONCAT command. A text string is used as input in order to provide the parameters necessary to define the file to be concatenated. Files can be concatenated using the SDBCONCT/SWSCONCT command.

CALL Arguments

The SDBCONCT/SWSCONCT (SDBPCC/SWCPCC) function arguments are described in the table which follows. Only two of the three arguments are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	LONG	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the de-concatenation command string. If the length is longer than the actual command, trailing nulls or blanks will be ignored. If the length is less than the actual command string, the de-concatenation command string will be truncated and possibly cause execution errors. The maximum string length is 32768 bytes.
2	CHAR*	PIC X(nnnnn)	CHAR (nnnnn)	Input	The de-concatenation command string. See supported de-concatenation keywords.
3	SWSASB*	Usage pointer	PTR	Output	The Shadow Web Server Allocation Status Block. This is an optional argument that provides information concerning the status of the de-concatenation request. If you do not specify this argument, you will not have access to the reason code nor the DAIR code.

Return Values

SDBDECON/SWSDECON always sets a signed numeric return code value. Possible values are:

SWS_SUCCESS

The operation succeeded. The specified operation was performed.

SWS_ERROR

A parameter validation or run-time error was encountered. Error information is available using the SWSERROR function.

Any other value

The operation failed. Generally this indicates that the file was not de-concatenated. There will be an error message in the Allocation Status Block describing the error.

Supported De-concatenation Keywords

The SDBDECON/SWSDECON (SDBPDC/SWCPDC) interface supports the following dataset concatenation request parameters:

De-concatenation Keyword	Description
DDN(DDNAME)	Specifies the ddname of the concatenated file.

PL/I Example

```
%INCLUDE SPCPHD
       .
       .
       .
       .
DCL  COMMAND         CHAR(80)       /* DE-CONCAT COMMAND      */
 INIT('DDN(INFILE)');
DCL  CMDLEN          FIXED BIN(31); /* COMMAND LENGTH         */
DCL  RC              FIXED BIN(31); /* RETURN CODE            */

CMDLEN = LENGTH(COMMAND);           /* SET COMMAND LEN        */


/* DE-CONCATENATE THE INPUT FILE                              */

CALL SWSDECON(CMDLEN,               /* COMMAND LENGTH         */
              COMMAND,              /* COMMAND                */
              SWSASB);              /* ALLOCATION STATUS BLOCK*/
     RC = PLIRETV();                /* GET RETURN CODE        */
    IF RC ^= SWS_SUCCESS THEN       /* EXIT PROGRAM IF BAD RC */
       EXIT;
```

C Example

```
SWS_ALLOCATION_STATUS_BLOCK swsASB; /* response area          */
      .
      .
      .
      .
long RC;                          /* return code            */
   char szCommand[] = "DDN(INFILE)";

/* De-concatenate the input file                             */

rc = SWSDECON(strlen(szCommand),    /* Command Length         */
              szCommand,            /* Command                */
              swsASB);              /* Response area          */

   if (rc ^= SWS_SUCCESS)
      do
        printf(swsASB.Error_Message);
        return rc;
      end
```

COBOL Example

```
 *     NEON API COPY BOOK
   COPY SBCPHD.
        .
        .
        .
        .
   77   COMMAND-LENGTH       PIC S9(5) COMP.
   77   COMMAND              PIC X(80)
        VALUE 'DDN(INFILE)'.

 *     DE-CONCATENATE THE INPUT FILE.

   MOVE 80 TO COMMAND-LENGTH.
   CALL SWSDECON
       USING COMMAND-LENGTH,
             COMMAND,
             SWS-ALLOCATION-STATUS-BLOCK.

   MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
   IF NOT SWS-SUCCESS
       DISPLAY 'INFILE DE-CONCATENATION FAILED.' UPON CONSOLE
       DISPLAY SWSASB-ERROR-MESSAGE UPON CONSOLE
       GOBACK.
```

SDBDECON/SWSDECON Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	High-level language interface available.

The REXX-language SDBDECON/SWSDECON built-in function can be used to de-concatenate a single ddname that was previous concatenated using the SDB-CONCT/SWSCONCT built-in function to concatenate the DDNames.

Note:
Because of comparable functionality of SDBDECON/SWSDECON to IBM's DECONCAT function, this documentation is similar to IBM's TSO/E online help.

Syntax

The format of this command is similar in features and functions to the TSO/E DECONCAT command. If an error occurs, the REXX variable **ALLOC.MES-SAGE** will be populated with a descriptive message. The DAIR return code can be obtained from **ALLOC.INFOCODE** and the reason code can be obtained from the **ALLOC.REASON**.

The general form for a REXX-language invocation of SDBDECON/SWSDECON is:

```
rc = SWSDECON(DDN(INFILE1))
```

Web Server REXX and SEF APIs

This section covers the following Web Server REXX and SEF APIs:

API Description	DIRECT	WEB	SEF	WEB/RX
Web Server REXX and SEF only APIs				
To clear REXX external data queue:	⊘	SWSClearQueue or SWCPQL	⊘	SWSCLEDQ
To perform security authorization processing:	⊘	⊘	SDBECURE	SWSECURE
To serialize usage of resources:	⊘	⊘	⊘	SWSENQ
"PARSE PULL" operation in Shadow/ REXX:	⊘	SWSGetQueue or SWCPQG	⊘	⊘
To set or display SWS product parameter values:	⊘	⊘	SDBPARM	SWSPARM
Equivalent to Shadow/REXX "Queue" (not "QUEUED()"):	⊘	SWSPutQueue or SWCPQP	⊘	⊘
Partly equivalent to Shadow/REXX built-in function "QUEUED()":	⊘	SWSQueryQueue or SWCPQQ	⊘	⊘
To create and write customized SMF records:	⊘	⊘	SDBSMF	SWSSMF
To transmit out-bound data to web server clients:	⊘	⊘	⊘	SWSXMIT

High Level Language SWSClearQueue (SWCPQL) Function

✔	Can be used in Shadow/REXX.
⊘	Not available from Other REXX interpreters.
✔	HLL entry point name is **SWCPQL.**

SWSClearQueue is the Web Server API function used to clear the external data queue associated with the current web transaction thread. Clearing the queue marks it as empty.

Normally, an external data queue is allocated and used only when executing Shadow/REXX procedures. However, a queue may now also be used from HLL programs. For HLL program executions, an external data queue can be pre-allocated by coding the QUEUESIZE( ) keyword. If one of the SWSxxxxxQueue HLL functions is invoked, an external data queue is created dynamically, using the default size, if one does not already exist.

CALL Arguments

The SWSClearQueue function takes one to three arguments; only the first is required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage Pointer	PTR	Input	The Web Server connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument is unused in the current release and must be set to zero.
3	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument is unused-used in the current release and must be set to zero.

Return Values

SWSClearQueue always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_INVALID_HANDLE	The connection handle is invalid. No error information is available.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
Any other value	The operation failed.

PL/I Example

```
DCL  TCONN     PTR;                    /* Connection Handle       */
DCL  RC        FIXED BIN(31);          /* return code             */
DCL  DMHX      FIXED BIN(31) BASED;    /* Dummy Handle field      */
DCL  FB00      FIXED BIN(31) INIT(0);/* Dummy argument            */

ADDR(TCONN)->DMHX = 0;                 /* Clear Connection Handle */

CALL SWSClearQueue( TCONN              /* clear the queue         */
                    FB00,
                    FB00);
    RC = PLIRETV();                    /* get return code         */
IF RC ^= SWS_SUCCESS THEN              /* exit program if bad RC  */
    EXIT;
```

C Example

```
HDBC    tConn   = NULL;              /* Connection Handle       */
SDWORD  tDummy = 0;                  /* dummy argument          */
long RC;                             /* return code             */

rc = SWSClearQueue( &tConn,          /* clear the queue         */
                    tDummy,
                    tDummy);
if (rc ^= SWS_SUCCESS) return;       /* exit program if bad RC  */
```

COBOL Example

```
77  TCONN                USAGE IS POINTER.
77  FB00                 PIC S9(5) COMP VALUE 0.


CALL 'SWCPQL' USING TCONN,
                    FB00,
                    FB00.
MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.
```

SWSCLEDQ Function

	Can be used in Shadow/REXX.
	Can not be used from other REXX interpreters.
	No high-level language interface.

The SWSCLEDQ built-in function provides a quick means of clearing the REXX external data queue. The function is used as a short cut in place of the following REXX coding:

```
DO WHILE QUEUED() > 0
    PARSE PULL X
END
```

Coding SWSCLEDQ

To code the SWSCLEDQ function, use the following format:

```
var = SWSCLEDQ()
```

Return Values

The function always returns 0 (zero) to the caller.

High-Level Language Interface SDBECURE (SDCPSC) SWSECURE (SWCPSC) Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreters.
✔	HLL entry point name is **SDCPSC/SWCPSC**.

SDBECURE/SWSECURE is a built-in function used to do security authorization processing. The function is divided into six different subfunctions each of which provide a different type of security authorization processing.

Call Arguments

The SDBECURE/SWSECURE function takes a varying number of arguments depending on the subfunction requested. The subfunction request itself is designated via the second entry in the parameter list passed to the SDBECURE/SWSECURE function.

Return Values

SDBECURE/SWSECURE returns both a numeric code and a character response if the subfunction call was a success. If the subfunction call failed, only a numeric code is returned. The actual text results are returned to the “output buffer” location specified on each call which is designated by the third entry in the parameter.

Return Value	Description
SWS_SUCCESS	The requested operation succeeded. The return value has been determined and placed into the buffer area. The actual size of the data is set into the fourth argument.
SWS_SUCCESS_WITH_INFO	The return buffer was not large enough to store the return value. The return value was truncated. The size of the return value, before truncation, is set in the fourth argument. For character data, a null termination byte is always placed into the last position of the output buffer area.
SWS_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error, for example, you invoked the API service outside of a web transaction procedure, or from outside the Server’s address space. The Server *may* provide diagnostic information in the wrap-around trace.

Return Value	Description
SWS_INVALID_HANDLE	The connection handle argument is invalid.

SDBECURE/SWSECURE Dataset Access Parameter List

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage Pointer	PTR	Input	Connection handle.
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A four byte binary integer indicating function to be performed. Must be the following: SWS_SECURE_VFYDSN.
3	UCHAR *	PIC X(nnn)	CHAR(nnn)	Input/ Output	The data buffer to receive the information.
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the output data buffer.
5	UCHAR *	PIC X(nnn)	CHAR(nnn)	Input	The dataset name.
6	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the dataset name.
7	UCHAR	PIC X(1)	CHAR(1)	Input	The access type: • **A** to verify Alter Access. • **C** to verify Control Access. • **R** to verify Read Access. • **U** to verify Update access.
8	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the access type (must be one).
9	UCHAR *	PIC X(nnn)	CHAR(nnn)	Input	The volser *(Optional).*
10	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the volser *(Optional).*

Return Values

The function returns the string `ALLOW` if the specified type of access to the data set is allowed. Otherwise, an error message is returned. The returned value for CA-ACF2 is a CA-ACF2 message; for RACF, the returned value is one of these messages:

```
RESOURCE NOT PROTECTED BY RACF
RESOURCE ACCESS DENIED BY RACF
```

SDBECURE/SWSECURE Fetching Logon ID Field Data Parameter List

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage Pointer	PTR	Input	Connection handle.
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A four byte binary integer indicating function to be performed. Must be the following: SWS_SECURE_USERINFO.
3	UCHAR *	PIC X(nnn)	CHAR(nnn)	Output	The data buffer to receive the information.
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the output data buffer.
5	UCHAR *	PIC X(nnn)	CHAR(nnn)	Input	The fieldname.
6	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the fieldname.

Return Values

Shadow/REXX does the following conversions depending on field format:

- Binary fields are converted to signed decimal values without leading zeros or blanks. The number zero is returned as 0.
- Character fields are returned as is, possibly truncated to the Shadow/REXX defined maximum valid string length.
- Date fields are converted to the form `yyyy/mm/dd` with leading zeros kept (so that the result is always exactly ten non-blank characters). A zero date field is returned as the string `****/**/**`.
- Bit fields are converted to a 0 (FALSE or off) or a 1 (TRUE or on).
- The group list field inquiry is handled separately. The function returns a integer count of the number of group entries found in the list. Each individual group name is returned as a separate entry in the external data queue.

SDBECURE/SWSECURE Requesting Security Product Information Parameter List

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage Pointer	PTR	Input	Connection handle.
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A four byte binary integer indicating function to be performed. Must be the following: SWS_SECURE_PRODINFO.
3	UCHAR *	PIC X(nnn)	CHAR (nnn)	Output	The data buffer to receive the information.
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the output data buffer.
5	UCHAR *	PIC X(nnn)	CHAR (nnn)	Input	The name constant.
6	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the name constant.

Return Values

The function returns a string with the requested information. If the information cannot be obtained, a NULL string is returned.

SDBECURE/SWSECURE Generalized Resource Rule Checks Parameter List

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage Pointer	PTR	Input	Connection handle.
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A four byte binary integer indicating function to be performed. Must be the following: SWS_SECURE_GENRES.
3	UCHAR *	PIC X(nnn)	CHAR(nnn)	Output	The data buffer to receive the information.
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the output data buffer.
5	UCHAR *	PIC X(nnn)	CHAR(nnn)	Input	The resource class name.
6	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the resource class name.
7	UCHAR *	PIC X(nnn)	CHAR(nnn)	Input	The resource entity name.
8	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the resource entity name.
9	UCHAR	PIC X(1)	CHAR(1)	Input	The access type: • **A** to verify Alter Access • **C** to verify Control Access • **R** to verify Read Access • **U** to verify Update access
10	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The length of the access type (must be one).

Return Values

For any of the three security products, if the specified access to the resource is allowed, Shadow/REXX returns the string `ALLOW`. Otherwise, Shadow/REXX returns an error message:

```
RESOURCE NOT PROTECTED BY RACF
RESOURCE ACCESS DENIED BY RACF
```

SDBECURE/SWSECURE Password Validation Parameter List

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage Pointer	PTR	Input	Connection handle.
2	UDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	A four byte binary integer indicating function to be performed SWS_SECURE_VALPSWD.
3	UCHAR *	PIC X(nnn)	CHAR(nnn)	Output	The data buffer to receive the information.
4	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the output data buffer.
5	UCHAR *	PIC X(nnn)	CHAR(nnn)	Input	The userid.
6	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the userid.
7	UCHAR *	PIC X(nnn)	CHAR(nnn)	Input	The password.
8	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the password.
9	UCHAR *	PIC X(nnn)	CHAR(nnn)	Input	The new password.
10	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	The size of the new password.

Return Values

For all three security products, if the password was correct (and a new password was assigned if specified), the returned value is the string `ALLOW`. Otherwise, Shadow/REXX returns a message:

```
INVALID SECURITY ENVIRONMENT
     USER PROFILE NOT DEFINED TO RACF
     PASSWORD IS NOT AUTHORIZED
     PASSWORD HAS EXPIRED
     USER NOT DEFINED TO THE GROUP
     REJECTED BY INSTALLATION EXIT
     ACCESS HAS BEEN REVOKED
     RACF IS NOT ACTIVE
     GROUP ACCESS HAS BEEN REVOKED
     NOT AUTHORIZED TO USE THIS TERMINAL
     INVALID DAY OR TIME OF DAY
     TERMINAL CANNOT BE USED
     NOT AUTHORIZED TO USE APPLICATION
```

Note:
For CA-ACF2, invalid password attempt calls increase the invalid password violation counter for the specified user ID.

PL/I Example

```
DCL SCONN    PTR                    /* Connection Handle      */
DCL SBUFF    CHAR(256)              /* Output Buffer          */
DCL SBFSZ    FIXED BIN(31)          /* Output Buffer length   */
DCL SUID     CHAR(8)                /* Userid                 */
DCL SUIDSZ   FIXED BIN(31)          /* Userid Length          */
DCL PSWD     CHAR(8)                /* Password               */
DCL PSWDSZ   FIXED BIN(31)          /* Password Length        */
ADDR(SCONN)->DMHX=0;                /* Zero connection  handle*/
    SUID="USERID";                  /* Set Userid             */
    SUIDSZ=6;                       /* Set Userid length      */
    PSWD="PASSWORD";                /* Set Password           */
    PSWDSZ=8;                       /* Set Password length    */
CALL SWSECURE(SCONN                 /* Call the function      */
                SWS_SECURE_VALPSWD,
                SBUFF,
                SFFSZ,
                SUID,
                SUIDSZ,
                SPSWD,
                SPWDSZ);
RC=PLIRETV();                       /* Get return code        */
IF RC ^=SWS_SUCCESS THEN            /* exit if bad RC         */
       EXIT;
```

C Example

```
HDBC sConn = NULL;                  /* Connection Handle      */
char         hBuff[80]              /* Output Buffer          */
SDWORD       hBfsz                  /* size of output buffer  */
char         hUid[] = 'USERID'      /* Userid                */
SDWORD       hUidsz                 /* size of userid         */
char         hPwd[] = 'PASSWORD     /* password              */
SDWORD       hPwdsz                 /* size of password       */
long RC                             /* return code            */
rc = SWSecure(&sConn,               /* send the response      */
              SWS_SECURE_VALPSWD,
              hBuff,
              hBfsz,
              hUid,
              hUidsz,
              hPswd,
              hPswdsz);
If(rc ^=SWS_SUCCESS) return;        /* exit if bad rc         */
```

COBOL Example

```
77 SCONN        USAGE IS POINTER.
77 HBUFF        PIC X(80).
77 HBUFFSZ      PIC S9(5) COMP.
77 HUID         PIC X(8)
77 HUIDSZ       PIC S9(5) COMP.
77 HPSWD        PIC X(8).
77 HPSWDSZ      PIC S9(5) COMP.
MOVE 80TO HBUFFSZ
MOVE 'USERID'TO HUID.
MOVE 6TO HUIDSZ.
MOVE 'PASSWORD'TO HPSWD.
MOVE 8TO HPSWDSZ.
CALL 'SWCPSC' USING SCONN,
                    SWS-SECURE-VFYDSN,
                    HBUFF,
                    HBUFFSZ,
                    BDATA,
                    BSIZE.
MOVE RETURN CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK/
```

The SDBECURE/SWSECURE Function

✔	Can be used in Shadow/REXX.
✔	Can be used from other REXX interpreter.
✔	High-level language entry point is **SDBCPSC/SWSCPSC.**

This function provides a set of subfunctions which perform six types of security authorization processing.

The SDBECURE/SWSECURE function has one required parameter, which must be followed with various other parameter arguments, depending on the requested function.

The Operation Argument

The first argument of the SDBECURE/SWSECURE function specifies the operation which is to be performed. The first argument can be one of the following values:

D	Used to verify data set access privileges.
F	Used to fetch information about a logged-on user.
I	Used to return information about the security product installed on your MVS system.
R	Used to request generalized resource rule validation.
P	Enables you to validate a Userid and password and/or set a new Userid.

Verifying Data Set Access

You can verify that the current user has authorization to access a data set. The current user, for Web transactions is the Effective Userid.

Coding Data Set Access Requests

Issue the version of SDBECURE/SWSECURE shown below to verify data set access privileges:

```
var = SDBECURE/SWSECURE ('D','dsname','accesstype','volser')
```

The arguments shown above, are coded as follows:

dsname	This argument specifies the dataset to be checked. The argument must be present or the function request is rejected.

Accesstype	Use the accesstype argument to specify the typ e of data set access you wish to check. If you omit the accesstype argument, Shadow/REXX uses a default of R (Read access). The access type argument can be specified as one of: • **A** to verify Alter access to a data set. • **C** to verify Control access to a data set. • **R** to verify Read access to a data set. • **U** to verify Update access to a data set.
Volser	The volser argument supplies the volser number to be validated. If you do not specify a volser, the argument is blank by default.

Return Values

The function returns the string `ALLOW` if the specified type of access to the data set is allowed. Otherwise, an error message is returned. The returned value for

CA-ACF2 is a CA-ACF2 message; for RACF, the returned value is one of these messages:

```
RESOURCE NOT PROTECTED BY RACF
RESOURCE ACCESS DENIED BY RACF
```

Fetching Logon ID Field Data

You can retrieve security subsystem information about the current user. The current user for Web transactions is the Effective Userid.

Note:

The Shadow Web Server uses the MVS SAF router interface for processing all security verification requests. This function will only operate correctly for field values which are available on the ACEE control block. Because many security products build only a skeletal ACEE block, some or all of the values may be unavailable if the underlying security subsystem product is other than RACF.

Coding Fetch Data Requests

Issue the version of SDBECURE/SWSECURE shown below to fetch field data from the current user's ACEE:

```
var = SDBECURE/SWSECURE('F','fieldname')
```

The fieldname argument is required. One of the following character constants can be coded:

Fieldname Constant	Format of Returned Data	Description
VERSION	Binary	ACEE Version code.
INSTALLATIONDATA	Character	Contents of the Installation Data field.
USERDATA	Character	Contents of the user data field.
USERID	Character	Contents of the ACEE Userid field .
GROUP	Character	Contents of the ACEE Group Field.
SPECIAL	Bit	Special Attribute.
AUTOMATIC	Bit	Automatic Attribute.
OPERATIONS	Bit	Operations Attribute.
AUDITOR	Bit	Auditor Attribute.
LOG	Bit	Logging on for most operations.
PRIVILEGED	Bit	Started task with privileged flag.
RACF	Bit	RACF defined user flag.
ALTER	Bit	Alter authority flag.
CONTROL	Bit	Control authority flag.
UPDATE	Bit	Update authority flag.
READ	Bit	Read authority flag.
NONE	Bit	None authority flag.
GROUPLISTCONTAINS	Bit	Group list contents flag.
DATE	Date	RACINIT Date.
STCNAME	Character	Started task name.
TERMINAL	Character	Terminal ID.
DEFINEUSERS	Bit	Authorized to define users.
PROTECTDASD	Bit	Authorized to protect DASD.
PROTECTTAPE	Bit	Authorized to protect tape.
PROTECTTERMINALS	Bit	Authorized to protect Terminals.
APPLICATIONLEVEL	Binary	Application Level.
PORTOFENTRYLEVEL	Binary	Port of entry level.
PORTOFENTRYDATA	Character	Port of entry data.
CLASSAUTHORIZATIONS	Binary	Class authorizations.
APPLICATION	Character	Application name.

Fieldname Constant	Format of Returned Data	Description
APPLICATIONDATA	Character	Application data.
USERNAME	Character	User name field.
SURROGATEUSERID	Character	Surrogate userid.
GROUPLIST	Group List	A list of groups.

Return Values

Shadow/REXX does the following conversions depending on field format:

- Binary fields are converted to signed decimal values without leading zeros or blanks. The number zero is returned as 0.
- Character fields are returned as is, possibly truncated to the Shadow/REXX defined maximum valid string length.
- Date fields are converted to the form yyyy/mm/dd with leading zeros kept (so that the result is always exactly ten non-blank characters). A zero date field is returned as the string ****/**/**.
- Bit fields are converted to a 0 (FALSE or off) or a 1 (TRUE or on).
- The group list field inquiry is handled separately. The function returns a integer count of the number of group entries found in the list. Each individual group name is returned as a separate entry in the external data queue.

Requesting Security Product Information

To retrieve information about the security product itself (if any) on your system using this form of the SDBECURE/SWSECURE function:

Coding Information Requests

Issue the version of SDBECURE/SWSECURE shown below to retrieve information about the security subsystem in use:

```
var = SDBECURE/SWSECURE ('I', 'name')
```

The name argument is required. Code one of the constant values shown below:

Name Constant	Returned Value
PRODUCT	Return the name of the security product. (e.g. RACF, ACF2, or "UNKNOWN SECURITY PRODUCT").

Name Constant	Returned Value
MODE	Returns the CA-ACF2 operating mode: • QUIET • LOG • WARN • ABORT • OFF This value is only valid for systems running CA-ACF2.
RELEASE	Causes Shadow/REXX to return the release and version number of CA-ACF2, CA-TOP SECRET, or RACF. For RACF and CA-TOP SECRET systems, this is a three-character string of the form v.r where v is the version number and r is the release number.

Return Values

The function returns a string with the requested information. If the information cannot be obtained, a NULL string is returned.

Generalized Resource Rule Checks

You can request that the Effective Userid for a Web transaction be validated against a security subsystem generalized resource rule.

Coding Generalized Resource Rule Checks

To request generalized resource rule validation, use this form of SDBECURE/SWSECURE

```
var = SDBECURE/SWSECURE ('R',class,resource,requestcode)
```

This function call verifies that the current user has access to a generalized resource. Code each argument as follows:

class	The generalized resource class name (type name for ACF2). Note that SAF processing is used for all resource rule checks. If your security subsystem is ACF2, you must define the ACF2 resource type as a SAF class name.
resource	The 1-to-39 byte resource entity name.
requestcode	The requestcode argument specifies the type of access to be verified. If you omit this argument, Shadow/REXX uses the default value R (Read access). Systems. The request code can be specified as one of the following: • **A** to verify Alter access to a resource. • **C** to verify Control access to a resource. • **R** to verify Read access to a resource. • **U** to verify Update access to a resource.

Returned Values

For any of the three security products, if the specified access to the resource is allowed, Shadow/REXX returns the string `ALLOW`. Otherwise,

Shadow/REXX returns an error message:

```
RESOURCE NOT PROTECTED BY RACF
RESOURCE ACCESS DENIED BY RACF
```

Validating a Userid and Password

This form of request causes the indicated Userid and password to be verified and logged onto the system. If the userid and password are valid an ACEE is created and made active for the current subtask. It remains active until explicitly reset, as described in the warning box, below.

You can perform a SWSECURE logon operation without supplying a password on the request if all the following conditions are met:

- The SWSECURE request is being made either by an /*ATH rule, or by a /*WWW rule which is defined in the Master WWW Ruleset.
- The WWWRUNAUTHFORMATS start-up parameter is set to "ALL".
- The WWWRUNAUTHLOCATION start-up parameter is set to "ANYWHERE" or "MASTERONLY".
- You are not requesting that the password be changed (no new password is specified for the SWSECURE request).

Note:

If the SWSECURE Password Validation request fails for any reason (i.e. Userid unknown, password invalid, new password invalid, etc.) the server's **Web Transaction Default Userid is always made active for the current task. This applies even if some other userid was active before the failing password validation request.**

When you successfully perform a logon operation from within a web transaction task, the newly logged on Userid is handled as though any other client logon, with the following exception: the newly logged-on userid remains permanently in effect for the Web transaction (even across all URL RESCAN operations) until explicitly reset. The userid is only deactivated when:

Note, however, that logged-on userid will remain permanently in effect for the web transaction (even across all URL RESCAN operations) until explicitly reset. The userid is only deactivated when:

- Another SWSECURE Password Validation request is performed.
- A /*WWW rule is matched which has RUNAUTH(proxy-id).
- RUNAUTH(NONE) coded on the rule header.
- The web transaction ends.

Coding Password Validation Requests

To validate a userid and password, use this form of the SDBECURE/SWSECURE function:

```
var = SDBECURE/SWSECURE ('P','userid','password','newpassword')
```

The arguments shown above, are coded as follows:

userid	The Userid to be validated .
password	The password associated with the Userid.
newpassword	The new password value to be associated with the Userid.

If you omit the newpassword argument, SDBECURE/SWSECURE validates the Userid and password. If you specify newpassword, SDBECURE/SWSECURE changes the password.

Return Values

For all three security products, if the password was correct (and a new password was assigned if specified), the returned value is the string ALLOW. Otherwise, Shadow/REXX returns a message:

```
INVALID SECURITY ENVIRONMENT
USER PROFILE NOT DEFINED TO RACF
PASSWORD IS NOT AUTHORIZED
PASSWORD HAS EXPIRED
USER NOT DEFINED TO THE GROUP
REJECTED BY INSTALLATION EXIT
ACCESS HAS BEEN REVOKED
RACF IS NOT ACTIVE
GROUP ACCESS HAS BEEN REVOKED
NOT AUTHORIZED TO USE THIS TERMINAL
INVALID DAY OR TIME OF DAY
TERMINAL CANNOT BE USED
NOT AUTHORIZED TO USE APPLICATION
```

Note:

For CA-ACF2, invalid password attempt calls increase the invalid password violation counter for the specified user ID.

SWSENQ Function

	Can be used in Shadow/REXX.
	Can not be used from other REXX interpreters.
	No high-level language interface.

This function interacts with MVS's ENQ/DEQ services to serialize usage of resources. Any REXX procedure using the SWSENQ function should use a SIGNAL ON SYNTAX statement to dequeue resources if the program fails to run properly. (Leaving the resources queued can leave your system inoperable.)

Syntax

The general form for invocation of SWSENQ is:

```
var = SWSENQ( func, major, minor, type, scope, ret )
```

Valid Arguments

SWSENQ uses the following arguments:

func	This operand is required. Specify either • **E** to enqueue on a resource. • **D** to dequeue from a resource.
major	The major name of the resource, up to eight characters in length. The function converts this value to upper case. The operand can be omitted, in which case the value **SWS2** is used.
minor	The minor name of the resource, containing up to 255 characters. The function does not perform upper case conversion on this parameter value. If the operand is omitted, the function supplied a value based upon the name of the event procedure from which the request was issued.
type	This operand is optional for enqueue requests. If omitted, the enqueue service request enclusive access to the resource. Valid values for this argument are: • **E** for exclusive encase • **S** for shared enqueues. The operand should be omitted for dequeue requests. If other operands follow, code a comma to indicate its omission.
scope	This argument is optional. If omitted, the value **SYSTEM** is used by default. It specifies the scope of the request, which can be: • **STEP** for a jobstep-wide enqueue request • **SYSTEM** for a system-wide enqueue request • **SYSTEMS** for a systems-wide enqueue request

ret	The type of return value for the enqueue or dequeue request. This value is one of the following: • Specify **HAVE** to return control when the enqueue has been obtained. • Specify **NONE** to return control when the enqueue has been obtained. • Specify **TEST'**to test whether the desired enqueue is available immediately but does not enqueue on the resource. • Specify **USE** to enqueue the desired resource only if it is available immediately.

Return Values

The function returns a numeric value equivalent to the return code issued by the underlying MVS service.

High-Level Language SWSGetQueue (SWCPQG) Function

	Can not be used from other REXX interpreters.
	No high-level language interface.
	Can be used in Shadow/REXX.

SWSGetQueue is the Web Server API function used to read lines from the external data queue associated with the current web transaction thread. The function returns the next (FIFO order) queued data line, if any, to a buffer in the application.

Normally, an external data queue is allocated and used only when executing Shadow/REXX procedures. However, a queue can now also be used from HLL programs. For HLL program executions, an external data queue can be pre-allocated by coding the QUEUESIZE( ) keyword. If one of the SWSxxxxxQueue HLL functions is invoked, an external data queue is created dynamically, using the default size, if one does not already exist.

CALL Arguments

The SWSGetQueue function takes four arguments, all of which are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage Pointer	PTR	Input	The Web Server connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	PTR	PIC X(nnn)	CHAR(nnn)	Output	This argument points to the data buffer which will be filled with the next queued line, if any. The queued line is truncated if longer than the size of this area. The area is binary-zero-padded on the right.
3	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument gives the size of the data buffer pointed to by the 2nd argument.
4	SDWORD FAR*	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument receives the actual queued data line size, regardless whether it was truncated during retrieval.

Return Values

SWSGetQueue always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	The operation succeeded.
SWS_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error. For instance, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_SUCCESS_WITH_INFO	The queued line was longer than the data buffer provided and was truncated.
SWS_NO_DATA_FOUND	No more lines are queued (queue is empty).
SWS_INVALID_HANDLE	The connection handle is invalid. No error information is available.
Any Other Value	The operation failed.

PL/I Example

```
DCL  TCONN   PTR;                    /* Connection Handle     */
DCL  RC      FIXED BIN(31);          /* return code           */
DCL  DMHX    FIXED BIN(31) BASED;    /* Dummy Handle field    */
DCL  BUFFER  CHAR(752);              /* Buffer area           */
DCL  BUFFLEN FIXED BIN(31) INIT(752); /* Size of buffer       */
DCL  LINELEN FIXED BIN(31) INIT(0);  /* Actual line size      */

ADDR(TCONN)->DMHX = 0;               /* Clear Connection Handle*/

CALL SWSGetQueue( TCONN              /* read the queue        */
                  BUFFER,
                  BUFFLEN,
                  LINELEN );

 RC = PLIRETV();                     /* get return code       */
IF RC = SWS_NO_DATA_FOUND THEN       /* queue is empty        */
do something else
IF RC ^= SWS_SUCCESS THEN            /* exit program if bad RC*/
     EXIT;
```

C Example

```
HDBC    tConn     = NULL;        /* Connection Handle       */
char    tbuffer[752];            /* buffer area             */
SDWORD  tbufflen  = 752;         /* buffer length           */
SDWORD  tLineLen  = 0;           /* actual line length      */
long RC;                         /* return code             */

rc = SWSGetQueue( &tConn,        /* query the queue         */
                  &tbuffer,
                  tbufflen,
                  &tLineLen );

if (rc ^= SWS_SUCCESS) return;   /* exit program if bad RC  */
```

COBOL Example

```
77  TCONN                USAGE IS POINTER.
77  TBUFFER              PIC X(752).
77  TBUFFLEN             PIC S9(5) COMP VALUE 752.
77  TLINESIZE            PIC S9(5) COMP VALUE 0.

CALL 'SWCPQG' USING TCONN,
                    TBUFFER,
                    TBUFFLEN,
                    TLINESIZE.

MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF SWS-NO-DATA-FOUND THEN
     do something else
END-IF.
IF NOT SWS-SUCCESS GOBACK.
```

SDBPARM/SWSPARM Function

	Can be used in Shadow/REXX.
	Can not be used from other REXX interpreters.
	No high-level language interface.

You use the SDBPARM/SWSPARM function of Shadow/REXX to set or display the values of Shadow Web Server product parameters.

Syntax

The general form for invocation of SWSPARM is:

- To display parameters, use this format:

```
var = SWSPARM("SHOW", parmname, "INFO", "NAMES")
```

- To set parameters, use this format:

```
var = SWSPARM("SET", parmname, newvalue )
```

Valid Arguments

The first two arguments are required parameters. Other arguments are optional.

parmname	Specifies the name of the product parameter (for example, "TRACEHTML") to be displayed or set. This name can contain no more than 50 characters. For the Show function, this argument can be coded as 'GROUPS' to have a list of product parameters returned to the external data queue. You can also code a product parameter group name (such as 'PRODWWW') to display a list of the individual parameters defined within the group.
newvalue	Specifies the new value you are assigning to a parameter when you use the Set function. This argument is required for Set requests.
INFO	Use the INFO argument with the Show function to display the possible values the parameter can have.
NAMES	Use the NAMES argument with the Show function to display the names and modifiability of individual parameters.

Note:

The **'INFO'** and **'NAMES'** parameters must be coded on the function call as parameters 3 and 4, respectively. Use commas to indicate any omitted arguments which precede these two values.

Return Values

For all Show functions, SDBPARM/SWSPARM returns the results on the REXX external data queue. The external data queue is not used for Set function requests.

You can retrieve the result lines from the external data queue using code such as:

```
DO WHILE QUEUED() > 0
PARSE PULL QLINE
....perform some process against each line
END
```

In addition, the SDBPARM/SWSPARM function always returns one of these codes:

Return Value	Description
0	The SDBPARM/SWSPARM function completed successfully.
4	Authorization check failed.
16	The Web Server Subsystem is not active.
20	The parameter new value is not valid.
48	The parameter name specified is not valid.
52	Some type of abend occurred while processing your request.

Examples

You can see this function code in action by referring to the supplied sample PARMS Web Transaction.

Example 1

To display the address of a module, invoke the SWSPARM function as follows:

```
RetCode = SWSPARM("SHOW","OPWWWWPR")
Say "SWSPARM() return code is:" RetCode
Do While Queued() <> 0
Pull Data
Say Data
End
```

In response, the following information is displayed:

```
SWSPARM() return code is:     0
ADDRESS OF MODULE OPWWWWPR X'06E1B000'
```

Example 2

To display the current value of an individual parameter:

```
RetCode = SWSPARM("SHOW","TRACEHTML")
Say "SWSPARM() return code is:" RetCode
Do While Queued() <> 0
Pull Data
Say Data
End
```

In response, the following information is displayed:

```
SWSPARM() return code is:     0
WEB TRANSACTION OUTPUT TRACE DEFAULT NO
```

Example 3

To display the current value of an individual parameter, along with additional information:

```
RetCode = SWSPARM("SHOW","TRACEHTML","INFO,"NAMES")
Say "SWSPARM() return code is:" RetCode
Do While Queued() <> 0
Pull Data
Say Data
End
```

In response, the following information is displayed:

```
SWSPARM() return code is:                          0
WEB TRANSACTION OUTPUT TRACE DEFAULT              NO
TRACEHTML                                         Y PRODWWW
FIELD FORMAT                                      BD
FIELD LENGTH                                      013
FIELD GROUP                                       016
FIELD SUFFIX                                        *
WEB TRANSACTION OUTPUT TRACE DEFAULT FLAG OFF    NO
WEB TRANSACTION OUTPUT TRACE DEFAULT FLAG ON     YES
```

Example 4

To display the address of a module with information and name, invoke SWSPARM as follows:

```
RetCode = SWSPARM("SHOW","OPWWWWPR","INFO","NAMES")
do while QUEUED() > 0
pull data
say data
end
```

In response, the following information is displayed:

```
ADDRESS OF MODULE OPWWWWPR                X'06E1B000'
OPITQWFU                                  N PRODMODULES
FIELD FORMAT                              ND
FIELD LENGTH                              004
FIELD GROUP                               015
FIELD SUFFIX                               *
MODULE ORIGINAL ADDRESS                   X'06E1B000'
MODULE FINAL ADDRESS                      X'06E1B000'
MODULE VECTOR TABLE ENTRY ADDRESS         X'06F20390'
MODULE SIZE                               14160 BYTES
MODULE PROTECT KEY                        CODE (2)
MODULE VERSION                            02.01.00
MODULE PROGRAMMER NAME                    AI38LRM
MODULE ASSEMBLY DATE                      04/08/96
MODULE ASSEMBLY TIME                      10.04
MODULE IS ELIGIBLE FOR RELOAD             YES
```

High-Level Language SWSPutQueue (SWCPQP) Function

	See QUEUE operation.
	Not available from other REXX interpreters.
	HLL entry point name is *SWCPQP.*

SWSPutQueue is the Web Server API function used to write lines to the external data queue associated with the current web transaction thread. The function writes the next (FIFO order) queued data line from a buffer in the application.

Normally, an external data queue is allocated and used only when executing Shadow/REXX procedures. However, a queue can now also be used from HLL programs. For HLL program executions, an external data queue can be pre-allocated by coding the QUEUESIZE( ) keyword. If one of the SWSxxxxxQueue HLL functions is invoked, an external data queue is created dynamically, using the default size, if one does not already exist.

CALL Arguments

The SWSPutQueue function takes three arguments, all of which are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage Pointer	PTR	Input	The Web Server connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	PTR	PIC X (nnn)	CHAR(nnn)	Output	This argument points to the data buffer from which the queue line will be written. The data buffer length can not exceed the maximum line size for queue entries (752 bytes in this release). The buffer data can be a NULL terminated string.
3	SDWORD	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument gives the size of the data to be written to the queue. Specify SWS_NTS if the buffer contains a NULL terminated string.

Return Values

SWSPutQueue always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	The operation succeeded.
SWS_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error. For instance, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_INVALID_HANDLE	The connection handle is invalid. No error information is available.
Any Other Value	The operation failed.

PL/I Example

```
DCL  TCONN     PTR;                 /* Connection Handle     */
DCL  RC        FIXED BIN(31);       /* return code           */
DCL  DMHX      FIXED BIN(31) BASED; /* Dummy Handle field    */
DCL  BUFFER    CHAR(752);           /* Buffer area           */
DCL  BUFFLEN   FIXED BIN(31);       /* Data length           */

ADDR(TCONN)->DMHX = 0;              /* Clear Connection Handle*/

MOVE 'DATA TO WRITE' to BUFFER;     /* put some data there   */
MOVE 13 to BUFFLEN.                 /* set length of data    */
CALL SWSPutQueue( TCONN             /* write the data        */
                  BUFFER,
                  BUFFLEN);

RC = PLIRETV();                     /* get return code       */
IF RC ^= SWS_SUCCESS THEN           /* exit program if bad RC */
      EXIT;
```

C Example

```
 HDBC    tConn     = NULL;          /* Connection Handle      */
 long RC;                           /* return code            */

rc = SWSPutQueue( &tConn,           /* query the queue        */
                  "Hello World.",
                  SWS_NTS );

if (rc ^= SWS_SUCCESS) return;      /* exit program if bad RC */
```

COBOL Example

```
77  TCONN               USAGE IS POINTER.
77  TBUFFER            PIC X(752).
77  TBUFFLEN           PIC S9(5) COMP.

MOVE 'DATA TO WRITE' TO TBUFFER.
MOVE 13 to TBUFFLEN.
CALL 'SWCPQP' USING TCONN,
             TBUFFER,
             TBUFFLEN.

MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
    IF NOT SWS-SUCCESS GOBACK.
```

High-Level Language SWSQueryQueue (SWCPQQ) Function

	See QUEUED( ) built-in function.
	Not available from other REXX interpreters.
	HLL entry point name is **SWCPQQ**.

SWSQueryQueue is the Web Server API function used to query the external data queue associated with the current web transaction thread. The function returns information about the overall queue size, maximum size of each line, and number of queued lines.

Normally, an external data queue is allocated and used only when executing Shadow/REXX procedures. However, a queue can now also be used from HLL programs. For HLL program executions, an external data queue can be pre-allocated by coding the QUEUESIZE( ) keyword. If one of the SWSxxxxxQueue HLL functions is invoked, an external data queue is created dynamically, using the default size, if one does not already exist.

CALL Arguments

The SWSQueryQueue function takes four arguments, all of which are required.

Arg	HLL Argument Type			I/O	Description of Argument
	C	COBOL	PL/I		
1	HDBC	Usage Pointer	PTR	Input	The Web Server connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	SDWORD FAR*	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument receives the maximum size of each queued line. The value returned is 752 in the current release of the product.
3	SDWORD FAR*	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument receives a count of the maximum possible lines allocated to the queue; both in-use and free lines are included in the total.
4	SDWORD FAR*	PIC S9(5) COMP	FIXED BIN(31)	Input	This argument receives a count of the currently queued lines.

Return Values

SWSQueryQueue always sets a signed numeric return code value. Possible values are:

Return Value	Description
SWS_SUCCESS	The operation succeeded.
SWS_ERROR	A parameter validation or runtime error was encountered. Error information is available using the SWSERROR function.
SWS_ENVIRONMENT_ERROR	The request could not be processed because of a runtime environmental error. For instance, you invoked the API service outside of a web transaction procedure, or from outside the Server's address space. The Server *may* provide diagnostic information in the wrap-around trace.
SWS_INVALID_HANDLE	The connection handle is invalid. No error information is available.
Any Other Value	The operation failed.

PL/I Example

```
DCL  TCONN     PTR;                    /* Connection Handle      */
DCL  RC        FIXED BIN(31);          /* return code            */
DCL  DMHX      FIXED BIN(31) BASED;    /* Dummy Handle field     */
DCL  LINESIZE  FIXED BIN(31) INIT(0); /* Maximum Line Size      */
DCL  MAXLINES  FIXED BIN(31) INIT(0); /* Total Lines            */
DCL  QUEUED    FIXED BIN(31) INIT(0); /* Queued Lines           */

ADDR(TCONN)->DMHX = 0;                 /* Clear Connection Handle*/

CALL SWSQueryQueue( TCONN              /* query the queue        */
                    LINESIZE,
                    MAXLINES,
                    QUEUED );

RC = PLIRETV();                        /* get return code        */
IF RC ^= SWS_SUCCESS THEN              /* exit program if bad RC */
       EXIT;
```

C Example

```
HDBC    tConn     = NULL;          /* Connection Handle       */
SDWORD  tLineSize = 0;             /* Lines Size              */
SDWORD  tMaxLines = 0;             /* Total lines             */
SDWORD  tQueued   = 0;             /* Queued Lines            */
long RC;                           /* return code             */

rc = SWSQueryQueue( &tConn,        /* query the queue         */
                    &tLineSize,
                    &tMaxLines,
                    &tQueued );

if (rc ^= SWS_SUCCESS) return;     /* exit program if bad RC  */
```

COBOL Example

```
77  TCONN               USAGE IS POINTER.
77  TLINESSIZE         PIC S9(5) COMP VALUE 0.
77  TMAXLINES          PIC S9(5) COMP VALUE 0.
77  TQUEUED            PIC S9(5) COMP VALUE 0.

CALL 'SWCPQQ' USING TCONN,
                    TLINESIZE,
                    TMAXLINES,
                    TQUEUED.

MOVE RETURN-CODE TO WS-SWSAPI-RETURN-CODE.
IF NOT SWS-SUCCESS GOBACK.
```

SDBSMF/SWSSMF Function

	Can be used in Shadow/REXX.
	Can not be used from other REXX interpreters.
	No high-level language interface.

This function enables SEF event procedures to create and write customized SMF records.

Syntax

The general form for invocation of SDBSMF/SWSSMF is:

```
var = SWSSMF( subtype, data )
```

Valid Arguments

Both arguments to the SWSSMF function are required. The arguments are:

subtype	A numeric value between 1000 and 32767. This value becomes the SMF record subtype code.
data	The data argument is the part of the SMF record following the standard Shadow Web Server 40-byte header section. This data can contain at most 344 bytes. If records exceed this limit, Shadow/REXX truncates them and issues a warning message.

Return Values

The SDBSMF/SWSSMF function returns these values:

Return Value	Description
0	The SMF record was successfully written.
4	Shadow/REXX received a non-zero return code from the SMFWTM macro while trying to write the SMF record. This error can result when all SMF data sets are full.
8	The product's SMFNUMBER parameter is set to zero, preventing the product from creating SMF records.

SWSXMIT Function

	Can be used in Shadow/REXX.
	Can not be used from other REXX interpreters.
	No high-level language interface.
	See also SWSSEND Host Command Environment.

SWSXMIT is a built-in function used to transmit out-bound data to web server clients from REXX-Language event procedures. SWSXMIT can only be used from within WWW event procedures and will return an error if invoked from other event procedure types.

This function is obsolete, although it will continue to be supported for the foreseeable future. NEON Systems recommends you use the SWSSEND built-in function.

Syntax

The general form for invocation of SWSXMIT is:

```
z = SWSXMIT( arg1 {, arg2 {, arg3 {, arg4 }}} )
```

Valid Arguments

The SWSXMIT function takes from one to four arguments. The first argument is always required. All other arguments are optional.

The first argument always specifies the data to be transmitted to the web server client. A NULL string can be passed as the first argument, or the argument can be omitted entirely by coding a single comma in its place.

The second through fourth arguments are selected from the following string constants:

LF	Indicates that a Linefeed character should be appended to the data. This operand can also be coded as the constant CRLF.
ASCII	Indicates that the data should be translated from EBCDIC to ASCII before transmission.
FLUSH	Indicates that this data (and any data already in the out-bound buffers) should be written to the client immediately.
PURGE	Indicates that all data currently un-transmitted within buffers should be discarded.

The PURGE operand must not be coded in conjunction with any other arguments.

Return Values

The function returns a zero to the calling program if the function completed successfully. A non-zero value indicates that the out-bound communications session has failed.

Examples

The following call will buffer the HTML data for out-bound transmission. A Line-Feed character will be added following the data and the data will be translated to ASCII before transmission:

```
htmldata = "<h1>This is a Header</h1>"
z=SWSXMIT( htmldata, 'LF', 'ASCII' )
```

The following call will place the data into the out-bound buffer with no additional processing:

```
z=SWSXMIT( gifdata )
```

The following call will buffer the HTML data for out-bound transmission. It will then cause all buffered data to be sent to the client immediately.

```
htmldata = "<h1>This is a Header</h1>"
z=SWSXMIT( htmldata, 'ASCII', 'LF', 'FLUSH' )
```

The following call will purge all previously buffered data. Data which was flushed prior to this call, will have already been sent to the web client.

```
z=SWSXMIT( , 'PURGE' )
```

CHAPTER 8:
Shadow Enterprise Direct API Function Calls

This chapter describes all Shadow Enterprise Direct Host Application Program Interface (API) functions and ***applies specifically to Shadow Enterprise Direct.***

Direct API	API Description
NEONBindCol	To bind columns for result set.
NEONDescribeParam	To describe passed parameter.
NEONError	To get error information.
NEONGetInfo	To return information to ODBC CALL RPC.
NEONNumParams	To access number of parameters.
NEONResetParam	To reset parameters.
NEONReturnStatus	To return status to client.
NEONThrow	To return row to result set.
NEONTraceMsg	To write message to trace browser.

NEONBindCol

performs a bind column on behalf of an ODBC CALL RPC. This call is used to bind a column to return sets back to the client. The caller must provide information, which is used to build a description of the result-set column.

Syntax

The general form for invocation of NEONBindCol is:

```
rc=NEONBindCol (NULL,
                2,
                SQL_C_DEFAULT,
                SQL_INTEGER,
                sizeof(int),
                0,
                SQL_NULLABLE,
                vlsr,
                &alen,
                "Value",
                SQL_NTS);
```

Arguments

The NEONBindCol function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
1	LONG	hstmt	INPUT	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	LONG	icol	INPUT	Column number of the result data. Columns are numbered from the left, starting with 1.
3	LONG	fCType	INPUT	C data type of column data. Value must be SQL_C_DEFAULT at this time. This means that C type must match SQL type.
4	LONG	fSqlType	INPUT	SQL data type of column data.
5	LONG	cbColDef	INPUT	Precision of column. This value is primarily used for decimal and character string data.
6	LONG	ibScale	INPUT	Scale of column. This value is primarily used for decimal data.
7	LONG	fnullable	INPUT	Indicates if column can have null values. Possible values are SQL_NO_NULLS and SQL_NULLABLE.
8	LONG	rgbvalue	INPUT	Pointer to storage for data. Actual data must be at this location when NEONThrow function is called to send a row.

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
9	LONG	pcbvalue	INPUT	Pointer to storage for length of column data. Length is used when NEONThrow function is called to send a row.
10	CHAR*	szcolname	INPUT	Pointer to storage containing column name. Column name must be a valid DB2 column name.
11	LONG	cbcolname	INPUT	Length of column name string. This must be a valid DB2 column name length.

Return Values

NEONBindCol always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_ERROR	A parameter validation error was found.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using NEONError.

Diagnostics

When NEONBindCol returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling NEONError. The following table lists the SQLSTATE values commonly returned by NEONBindCol and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQL State	Error	Description
SS1000	General error	Invalid parameter list detected.
S1002	Invalid column number	Column number is zero.
S1002	Invalid column number	Column number exceeds maximum value.
S1010	Function sequence error	Result set has already been started.

Example

```
rc = NEONBindCol(NULL,            /* null statement handle    */
                 2,               /* column number            */
                 SQL_C_DEFAULT,   /* default C type           */
                 SQL_CHAR,        /* character SQL type       */
                 sizeof(vlsr)-1,  /* precision of column      */
                 0,               /* zero scale               */
                 SQL_NULLABLE,    /* column is nullable       */
                 vlsr,            /* pointer to column value  */
                 &alen,           /* pointer to column length */
                 "Value",         /* column name              */
                 SQL_NTS);        /* length indicates null-
                                          terminated        */
```

NEONDescribeParam

NEONDescribeParam is used to obtain information about a parameter passed from the client to the host.

Note:
The client can pass parameters to the host using both parameter markers (?) and parameter literals. Both types of client parameters are treated the same way on the host.

Syntax

The general form for invocation of NEONDescribeParam:

```
rc=NEONDescribeParam (NULL,
                      i,
                      &sqty,
                      &prec,
                      &scal,
                      NULL,
                      &paty,
                      &daad,
                      &daln);
```

CALL Arguments

The NEONDescribeParam function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
1	LONG	hstmt	Input	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	LONG	ipar	Input	Parameter number. All parameters including literals are numbered from the left starting at 1.00
3	LONG	fSqlType	Output	SQL data type of parameter data.
4	LONG	pcbColDef	Output	Precision of parameter. This value is primarily used for decimal and character string data.
5	LONG	pibScale	Output	Scale of parameter. This value is primarily used for decimal data.
6	LONG	pfNullable	Output	Indicates whether or not parameter allows null values.
7	LONG	pfparamtype	Output	Indicates input/output type of parameter. Parameters can be used to send data to host (input), receive data from host (output), or both (input/output).
8	LONG	prgbValue	Output	Pointer to storage for parameter. Parameter can be accessed and updated at this storage location.

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
9	LONG	PcbValue	Output	Actual length of column. Length will be same as precision except for variable length fields (character and binary). For variable length fields, length will be current length. For all types, this field may contain SQL_NULL_DATA.

Return Values

NEONDescribeParam always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_ERROR	A parameter validation error was found.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using NEONError.

Diagnostics

When NEONDescribeParam returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling NEONError. The following table lists the SQLSTATE values commonly returned by NEONDescribeParam and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQL State	Error	Description
SS1000	General error	Invalid parameter list detected.
S1000	General error	No room in buffer for column description.
S1004S1002	SQL data type out of range	SQL data type is invalid.
S1009	Invalid argument value	Column name address not set.
S1009	Invalid argument value	Column data length address not set.
S1009	Invalid argument value	Column data address is not set.
S1090	Invalid string or buffer length	Column name length is invalid.
S1090	Invalid string or buffer length	Column name length is not valid.
S1094	Invalid scale value	Decimal scale value is invalid.
S1099	Nullable type out of range	Nullable status value is invalid.

SQL State	Error	Description
S1104S1002	Invalid precision value	Decimal precision value is invalid.
S1104S1002	Invalid precision value	String or binary precision value is invalid.
S1C00S	Driver not capable	Data type is not SQL_C_DEFAULT.

Example

```
rc = NEONDescribeParam(NULL,    /* null statement handle       */
                       i,       /* column number               */
                       &sqty,   /* pointer to SQL type         */
                       &prec,   /* pointer to parameter's prec  */
                       &scal,   /* pointer to parameter's scale */
                       NULL,    /* nullable status not requested*/
                       &paty,   /* pointer to parameter's type  */
                       &daad,   /* data address (not used)     */
                       &daln);  /* data len address (not used)  */
```

NEONError

NEONError is the API function used to fetch information pertaining to the last Application Program Interface error detected for this transaction.

Syntax

The general form for invocation of NEONError is:

```
rc = NEONError (NULL,NULL, NULL,
                NULL,
                &ercd,
                (UCHAR FAR *),
                sizeof (ermg),
                NULL)
```

CALL Arguments

The NEONError function call requires eight arguments. None may be omitted from the function call.

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
1	HENV	henv	INPUT	The Web Server environment handle. The environment handle is an opaque, four-byte address pointer. The environment handle is currently not used, and must be set to zero (NULL).
2	HDBC	hdbc	INPUT	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
3	HSTMT	hstmt	INPUT	The Web Server statement handle. The statement handle is an opaque, four-byte address pointer. The statement handle is currently not used, and must be set to zero (NULL).
4	UCHAR*	szSqlState	OUTPUT	This argument should specify a character string buffer of at least 6 bytes in length. A state value, compatible in format with the ODBC specification is returned in this area, as a null terminated string.
5	SDWORD*	pfNativeError	OUTPUT	The 'native' error code is returned within this area. This is some value that describes the error condition.
6	UCHAR*	szErrorMsg	OUTPUT	The buffer area which receives the error message text. Note that the error message text will always be null-terminated. Room for the trailing null must be provided.
7	SDWORD	cbErrorMsgMax	INPUT	The total size of the error message buffer area supplied by the sixth argument. The error message will be truncated if it does not fit into this buffer, including room for the trailing null terminator.

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
8	SDWORD*	pcbErrorMsg	OUTPUT	The API returns the total size of the error message (excluding the null terminator). The returned size value will be larger than the buffer size if the error message has been truncated.

Return Values

NEONError always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The return values have been set.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_ERROR	A parameter validation error was found.
SQL_INVALID_HANDLE	One of the handle arguments is invalid.

Example

```
rc = NEONError (NULL,NULL,NULL,/* null env, conn, stmt handles  */
                NULL,          /* not interested in SQL state   */
                &ercd,         /* native error code             */
                (UCHAR FAR *), /* error message buffer          */
                sizeof (ermg), /* size of error message buffer */
                NULL)          /* not interested in real msg len*/
```

NEONGetInfo

NEONGetInfo is the API function used to fetch information about the current transaction execution environment and return it to the caller.

Syntax

The general form for invocation of NEONGetInfo is:

```
rc = NEONGetInfo( &sConn,
                  SWS_GET_IPADDRESS,
                  &sBuff[0],
                  sizeof(sBuff),
                  &sRTSZ );
```

CALL Arguments

The NEONGetInfo function takes five arguments. All five arguments must be specified on the call.

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
1	HDBC	hdbc	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	UDWORD	fInfoType	Input	A four-byte binary integer indicating the information item to be returned by the function. Specify any one of the manifest constants, shown in the table below, to indicate the data item to be fetched.
3	UCHAR *	rgbInfoValue	Output	The data buffer to receive the fetched information. Depending on the value of the second argument, the returned data may be a null-terminated string; a 16-bit integer value, a 32-bit flag-word value, or a 32-bit signed or unsigned integer.
4	SDWORD	cbInfoValueMax	Input	The size of the data buffer area given by the third argument.
5	SDWORD *	pcbInfoValue	Output	Return area receiving the total size, in bytes, of the requested information value, regardless of whether the fetched value could be completely stored within the buffer area. For character format data items, which are null terminated, this value does not include the null termination byte. *For requests which return character data:* If the total size of the requested information is greater than or equal to the size of the data buffer the returned character string is truncated, and a null terminationbyte is placed into the last available of the buffer area. *For requests which return any other data type:* The value given by the forth argument is ignored. The size of the return buffer area is assumed to be at least four bytes.

Return Values

NEONGetInfo always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The requested data has been fetched and placed into the buffer area. The actual size of the data is set into the sixth argument.
SQL_SUCCESS_WITH_INFO	The return buffer area was not large enough to store the fetched item. The fetched item was truncated. The size of the fetched item, before truncation, is returned to the sixth argument. For character data, a null termination byte is always placed into the last buffer position.
SQL_ERROR	A parameter validation error was found.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using NEONGetInfo.

Example

```
HDBC          sConn   = NULL;          /* Connection Handle   */
char          sBuff[256];              /* Return Buffer Area  */
SDWORD        sRTSZ;                   /* Return item size    */
long          RC;                      /* return code        */

 rc = NEONGetInfo( &sConn,
                   SWS_GET_IPADDRESS,
                   &sBuff[0],
                   sizeof(sBuff),
                   &sRTSZ );
```

NEONNumParams

NEONNumParams is used to obtain the number of parameters passed from the client to the host. This value will be zero or greater.

Note:
The client can pass parameters to the host using both parameter markers (?) and parameter literals. Both types of client parameters are treated the same way on the host.

Syntax

The general form for invocation of NEONNumParams is:

```
rc=NEONNumParams (&st,
                  &pacn)
```

CALL Arguments

The NEONNumParams function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
1	LONG	hstmt	Input	Statement handle. Since only one host RPC can execute at a time, for each host session, this value is ignored and must be zero.
2	LONG	pcpar	output	Number of RPC parameters passed to host from client. This argument is a pointer to a signed four-byte integer.

Return Values

NEONNumParams always sets a signed numeric return code value. Possible values are: The connection handle argument is invalid. No error information can be returned using NEONGetInfo.

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_ERROR	A parameter validation error was found.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using NEONError.

Diagnostics

When NEONNumParams returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling NEONError. The following table lists the SQLSTATE values commonly returned by NEONNumParams and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQL State	Error	Description
SS1000	General error	Invalid parameter list detected.
S1009S	Invalid argument value	Parameter count address not set.

Example

```
rc=NEONNumParams (&st,
                  &pacn)
```

NEONResetParam

NEONResetParam is used to reset the length of a parameter passed from the client to the host.

> ***Note:***
> The client can pass parameters to the host using both parameter markers (?) and parameter literals. This routine can only be used with parameter markers.

In practice, this routine is really only used to change null parameters to non-null parameters and vice versa.

Syntax

The general form for invocation of NEONResetParam is:

```
rc=NEONResetParam (hstmt,
                   rgbMsgText,
                   cbMsgText,
                   fOption)
```

CALL Arguments

The NEONResetParam function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
1	LONG	hstmt	Input	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	LONG	ipar	Input	Parameter number. All parameters including literals are numbered from the left starting at 1.
3	LONG	cbValue	INput	New parameter length value. Parameter becomes null if new value is SQL_NULL_DATA. Parameter becomes NON_NULL if new value is not SQL_NULL_DATA.

Return Values

NEONResetParam always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.

Return Value	Description
SQL_ERROR	A parameter validation error was found.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using NEONError.

Diagnostics

When NEONResetParam returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling NEONError. The following table lists the SQLSTATE values commonly returned by NEONResetParam and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQL State	Error	Description
SS1000	General error	Invalid parameter list detected.
S1000	General error	Trace message construction failed.
S1000	General error	Invalid option value detected.
S1009S	Invalid argument value	Message area address not set.
S1009	Invalid argument value	Trace message insertion failed.
S1090S1002	Invalid string or buffer length	Message area length is invalid.
S1090	Invalid string or buffer length	Message text length is not valid.

Example

```
rc=NEONResetParam (hstmt,
                   rgbMsgText,
                   cbMsgText,
                   fOption)
```

NEONReturnStatus

NEONReturnStatus is used to return status information to the client from an ODBC CALL RPC. The status data determines the return code from the SQLEXECDIRECT, SQLPREPARE, or SQLEXECUTE function that started the RPC. The client application can retrieve the status data (message and native code) by calling NEONError.

The actual return code returned to the ODBC application will be SQL_SUCCESS_WITH_INFO if this routine provides a positive return code and SQL_ERROR if this routine provides a negative return code. The return code provided by this routine is returned to the client application as the native error code (see the NEONError function description in the ODBC programmer's reference manual, not the NEONError function description here).

Syntax

The general form for invocation of NEONReturnStatus is:

```
rc=NEONReturnStatus (NULL,
                     ermg,
                     sizeof(ermg),
                     ercd);
```

CALL Arguments

The NEONReturnStatus function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
1	LONG	hdbc	Input	Connection handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	CHAR*	rgbMsgText	Input	Address of message text to be returned. Text must be set before function is called.
3	LONG	cbMsgText	Input	Length of message text to be returned. Value can be an actual length or can be specified as SQL_NTS if the message text is NULL_TERMINATED.
4	LONG	fNativeError	Input	Native error code. If value is negative, client return code will be SQL_ERROR. If value is positive, client return code will be SQL_SUCCESS_WITH_INFO. This field must not be zero.

Return Values

NEONReturnStatus always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_ERROR	A parameter validation error was found.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using NEONError.

Diagnostics

When NEONReturnStatus returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling NEONError. The following table lists the SQLSTATE values commonly returned by SQLReturnStatus and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQL State	Error	Description
SS1000	General error	Invalid parameter list detected.
S1000	General error	Invalid native error code detected.
S1009S	Invalid argument value	Message area address not set.
S1090S1002	Invalid string or buffer length	Message area length is invalid.
S1090	Invalid argument value	Message text length is not valid.

Example

```
NEONReturnStatus(NULL,          /* null statement handle       */
                 ermg,          /*error message buffer         */
                 sizeof(ermg),  /*size of error message buffer */
                 ercd);         /* error message               */
```

NEONThrow

NEONThrow is used to send a row from the host ODBC call RPC back to the client.

Note:
One or more columns must be bound before this routine is called.

NEONThrow is called for each row in the result set. When populating the result set, NEONThrow is called with a parameter of SQL_THROW_ROW. Once the result set is populated and you wish to send the result to the client, NEONThrow is called once again with a parameter of SQL_THROW_DONE.

Syntax

The general form for invocation of NEONThrow is:

```
rc= NEONThrow(NULL,
     SQL_THROW_DONE);
```

CALL Arguments

The NEONThrow function can be called by any ODBC CALL RPC and accepts the following arguments:

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
1	LONG	hstmt	INPUT	Statement handle. Since only one host RPC can execute at a time for each host session, this value is ignored and must be zero.
2	LONG	fOption	INPUT	Top of operation needed. This value is used to indicate that row is being provided or that result set is complete.

Return Values

NEONThrow always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified operation was performed.
SQL_SUCCESS_WITH_INFO	The operation partially succeeded. This return code value is set when the returned error message text has been truncated.
SQL_ERROR	A parameter validation error was found.
SQL_INVALID_HANDLE	The connection handle argument is invalid. No error information can be returned using NEONError.

Diagnostics

When NEONThrow returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling NEONError**.** The following table lists the SQLSTATE values commonly returned by NEONThrow and explains each one in the context of this function. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQL State	Error	Description
SS1000	General error	Invalid parameter list detected.
S1000S	General error	Invalid option value detected.
S1000	General error	Null data specified for a non-null column.
S1000	General error	Maximum row count limit has been reached.
S1001	Memory allocation failure	Buffer space allocation failed.
S1010S1002	Function sequence error	Result set has already been completed.
24000	Invalid cursor state	No columns have been bound so far.

Example

```
rc = NEONThrow(NULL,          /* null statement handle        */
      SQL_THROW_ROW);         /* throw the row to client      */
```

NEONTraceMsg

NEONTraceMsg is used to write a message into the wrap-around trace browse dataset. The message can contain any text desired. If the message is too long to fit within a trace browse record, it is truncated. Truncation is not considered an error.

Syntax

The general form for invocation of NEONTraceMsg is:

```
rc = NEONTraceMsg( &tConn,
          tData,
          SWS_NTS,
          0 );
```

CALL Arguments

The NEONTraceMsg function takes four arguments. All four arguments must be specified on the call.

Arg. No.	Arg. Type	Arg. Name	Input/ Output	Arg. Description
1	HDBC	henv	Input	The connection handle. The connection handle is an opaque, four-byte address pointer. The connection handle is currently not used, and must be set to zero (NULL).
2	PTR	rgbMsgText	Input	The data value which is to be written to the trace browse wrap-around dataset. You may specify a null terminated string, or explicitly provide the value length via the third argument. The maximum useable length for a trace browse record is approximately 730 bytes.
3	SDWORD	cbMsgText	Input	The size of the data value given by the second argument which is to be written to the trace record.
4	UDWORD	fOption	Input	This argument is currently not used, but may be in future releases. You must specify a zero value.

Return Values

NEONTraceMsg always sets a signed numeric return code value. Possible values are:

Return Value	Description
SQL_SUCCESS	The operation succeeded. The specified data was written to the product's wrap-around trace.
SQL_INVALID_HANDLE	The connection handle is invalid. No error information is available.
SQL_ERROR	A parameter validation or write error was encountered.
Any Other Value	The operation failed.

Example

```
HDBC tConn   = NULL;                  /* Connection Handle     */
char tData[] = "Null-terminated!";    /* Text string definition*/
long RC;                              /* return code           */

rc = NEONTraceMsg( &tConn,            /* output trace message  */
                   tData,
                   SWS_NTS,
                   0 );
```

CHAPTER 9:
Transaction Level Security (TLS)

This chapter covers Transaction Level Support (TLS). TLS was created to support the new and unique security requirements of Internet applications, while operating in the traditional enterprise computing environment. With TLS, web applications that access OS/390 data and transactions can be used by people who do not have mainframe userids. TLS can also be used with non-Internet applications. This chapter will provide a brief overview of TLS, as well as its implementation.
This chapter applies to Shadow Direct only.

What is TLS?

TLS is a mechanism that provides protection for business transactions on the Internet by allowing two entities to conduct a transaction with privacy and authentication.To accomplish this, TLS creates a separate, temporary security environment for each transaction that is sent over a connection from a middle-tier Web server or Application server to Shadow Direct on the host. The transaction is typically an RPC/stored procedure and the connection is a network session.

The Transaction Level Security environment can also be used to control access to the RPC, as well as perform such tasks as logging, tracing, auditing, etc.

Why use TLS?

The development of TLS grew from the need to replace traditional OS/390, UNIX, and NT security architecture, since it could not adequately handle the larger volumes of data associated with Internet applications and used by millions of people. In addition, traditional userids have become too costly to create and administer, and use too many machine resources required for logging onto the Internet.

TLS is based on the following assumptions:

- **Connections.** TLS assumes that each middle-tier Application Server (AS) or Web Server will initialize a small number of permanent connections to the host server.
- **Connection/Session/Thread Reuse.** TLS assumes that each of the relatively small number of connections will be shared across all of the Internet connections. Each connection can be serially reused an arbitrary number of times by a different Internet user each time.
- **Control Userids.** Given the lack of any relationship between the identity of an Internet user and host userid, and the continuous reuse of each of the pooled host connections, TLS assumes that all of the connections will be

established with a control userid that has sufficient resource access for all of the applications running on the AS/Web Server.

- **Transaction Security.** Because each connection is serially reused, each transaction for a given connection can be executed for a different Internet user. This means that each transaction must have a separate security environment associated with it, and must be appropriate for Internet application.
- **Performance.** It must be possible to establish and access a transaction security environment with essentially no or negligible overhead.
- **Resource Access.** The transaction security data must be available on the host side to control resource access on an application specific, selective basis. The transaction security data must also be available for auditing, logging, tracing, tracking, etc.

Implementing TLS

Shadow Direct implements TLS with a host of related new facilities. Together each of these facilities is an answer to the above assumptions.

All of the facilities are based on two new IDs:

- Generic ID
- Extended ID

These two IDs are provided in addition to the traditional userids supported by Shadow Direct. They are optional and can be used either together or separately.

In addition, the Generic and Extended ID values can be used for application debugging, logging, tracing, and auditing purposes. In many respects, they are similar to the User Parameter that can be set as part of the ODBC connection initialization, however, they have the advantage that they can be set and/or reset as many times as needed for each connection.

Generic ID

The Generic ID is an 8-byte string, which is automatically converted to uppercase and padded with blanks on the right. The Generic ID is made available to host applications, and is used for auditing, logging, tracing, tracking, etc. It is specified as an ASCII string on the client and is automatically converted to EBCDIC for host processing.

Extended ID

The Extended ID is a variable length 128 byte string. This string is passed from the client (ODBC) environment to the host. On the host side this ID is made available to host applications and is used for auditing, logging, tracing, tracking, etc. The Extended ID is assumed to contain some type of application specific user identifier such as an email address, Social Security Number, Login Name, Access ID, etc. Like the Generic ID, the Extended ID is specified as an ASCII string on

the client and is automatically converted to EBCDIC for host processing. This means that the Extended ID can not contain binary data such as a digital certificate.

Client Side Support

The Generic ID and the Extended ID are supported on the client side using the ODBC SQLSetConnectOption function. This function can be called at any time to set either value.

The option value for setting these IDs is as follows:

- Generic ID: `SQL_NEON_GENERIC_USERID`.
- Extended ID: `SQL_NEON_EXTENDED_USERID`.

Note:
Separate calls are required to set each value.

Examples

Generic ID

The following C example shows how the Generic ID values are set.

```
rc = SQLSetConnectOption(hdbc, SQL_NEON_GENERIC_USERID,
(UDWORD) “AI38KPO”);
if (rc != SQL_SUCCESS &&
    rc != SQL_SUCCESS_WITH_INFO)
  goto exlb;
```

Extended ID

The following C example shows how the Extended ID values are set.

```
rc = SQLSetConnectOption(hdbc, SQL_NEON_EXTENDED_USERID,
(UDWORD) “I am not a digital certificate”);
if (rc != SQL_SUCCESS &&
    rc != SQL_SUCCESS_WITH_INFO)
  goto exlb;
```

Both the Generic ID and Extended ID values are only transmitted over the network when they are set for the first time or when they are changed.

The Generic ID and/or Extended ID can only be used with a recent (as of 1999/06/02) build of the ODBC driver.

Note:
No new ODBC driver configuration is needed to use these new IDs.

Host Side Support

The Generic ID and the Extended ID are supported on the host side using several different mechanisms. Each of these mechanisms is optional and any can be used together. Several of these mechanisms are intended for application security, auditing, logging, tracing, tracking, etc. The choice of which host side mechanisms are used will be installation and application specific.

The host mechanisms are:

- APIs
- SMF Per-Transaction Recording
- Logging
- Trace Browse
- Remote Users

APIs.

The SQLGetInfo function can be used in host RPCs to access (but not update) the Generic ID and the Extended ID. The type values for the information are as follow:

- **C:** `SQL_GET_GENERICID` and `SQL_GET_EXTENDEDID`
- **Cobol:** `SQL-GET-GENERICID` and `SQL-GET-EXTENDEDID`
- **ASM:** `ODSQGIGN/ODSQGIEX`

Both are returned as null-terminated string values.

Note:
The output area for the Generic ID should be large enough for the 8-byte string and the one-byte null terminator. The output area for the Extended ID should be large enough for the 128-byte string and the one-byte null terminator.

SMF Per-Transaction Recording

Shadow Direct supports SMF recording on a per-transaction basis using SMO6 records. These records contain information about the current Generic and Extended IDs. The SMF Per-Transaction Recording is activated by setting the SMFTRANSACT parameter to YES.

Note:
The Extended ID area in the SMO6 record has room for only the first fifty bytes of the Extended ID. A new record format will be provided if the entire Extended ID is needed in the future.

Logging

Shadow Direct supports logging of SQL/transactions on a per-SQL basis using a DB2 table. The default table name for per-SQL logging is SHADOW.SQL-SOURCE, however this default can be changed using the LOGSOURCETABLE product parameter. Per-SQL logging is activated by setting the LOGSQL-SOURCE product parameter to YES. The Generic ID is stored in the GENERIC_USERID column and the Extended ID is stored in the EXTENDED_USERID column.

Note:
The EXTENDED_USERID column only has room for the first 254 bytes of the of Extended ID

Trace Browse

If a Generic ID exists, it will be contained in the USERID column of Trace Browse for SQL/RPC operations. The Generic ID replaces the standard userid in Trace Browse if the Generic ID has been set to a non-blank, non-zero value. This information is only provided for debugging, tracking, tracing, auditing, etc.

Note:
The standard userid will be stored in Trace Browse for non-SQL/ RPC operations (such as network I/O) even if the Generic ID is set. This means that both the Generic ID and the standard userid will normally appear in Trace Browse for one session.

Remote Users

The Remote Users display includes two new columns for the Generic ID and the Extended ID. These columns will contain their respective values if they have been set.

Passing Generic ID to SAF

Generic IDs can be passed to SAF to create an MVS security environment for running an RPC. To do this, the following requirements must be applied:

- The Generic IDs must be valid host userids.
- The IMPLEMENTTLS product parameter must be set to YES.

Note:
Setting IMPLEMENTTLS to YES will only affect the SAF processing of Generic IDs. All of the other features and facilities can be used even if the IMPLEMENTTLS is set to NO.

The MVS security environment created by passing the Generic ID to SAF is maintained for the duration of RPC execution and will influence what resources the RPC can access.

> ***Note:***
> The Generic ID MVS security environment will have no impact on SQL execution authority. The DB2 security environment is initialized when the DB2 thread is created and is not subsequently modified.

The Generic ID security environment will be used to determine the following:

- If the client is allowed to execute an RPC.
- If RPC authority checking has been activated by setting the CHECKRPCAUTHORITY product parameter to YES. RPC authority checking uses RACF Class/Entity Rules or ACF2 Generalized Resource Rules to determine if a client is authorized to execute an RPC.

> ***Note:***
> RPC authority checking can be used with or without Generic ID SAF processing and vice versa.

For performance reasons, the MVS security environments created by passing Generic IDs to SAF are cached. In other words, each Generic ID is passed to SAF only once and the MVS security environment is cached at the address space level. This approach allows use/reuse of Generic ID security environment with negligible overhead. Security environment caching is implemented by forcing the SHARERUNAUTHACEES product parameter to YES. As a consequence, this product parameter does not have to be set.

> ***Note:***
> There is no SEF processing of LOGONs for Generic IDs even if ATH Rules for LOGON have been enabled. The Generic ID MVS security environments are maintained in the cache until the main product address space terminates.

There is a possible security exposure associated with using Generic IDs with the IMPLEMENTTLS parameter set to YES. In this case, an MVS security environment will be created without a password. In addition, client applications will be able to use the Generic ID MVS security environment without providing a password. This means that only carefully controlled applications (running inside an Application Server/Web sever) should be allowed to connect to a copy of Shadow Direct that has IMPLEMENTTLS set to YES. This restriction can be enforced several ways including LOGON ATH Rules.

Note:

IMPLEMENTTLS defaults to NO and can only be set to YES using the Shadow Direct initialization script. IMPLEMENTTLS can not be set to YES after the main product address space initialization has been completed.

CHAPTER 10: SQLProcedure and SQLProcedure Columns

This chapter discusses how to create a pseudo DB2 stored procedure that contains necessary Meta data for input and output fields, as well as other required parameters for accessing CICS and IMS transactions.
This chapter applies to Shadow Direct only.

Introduction

Third party software that uses SQLProcedure and SQLProcedureColumns to invoke remote Stored Procedures can access IMS and CICS transactions. With Shadow, a user can define the necessary input and output for a CICS or IMS transaction and any necessary parameters that are required to execute the transaction and store this information in the DB2 catalog. This takes the form of a DB2 stored procedure definition. This pseudo-DB2 stored procedure enables a simpler and more flexible call for clients like Crystal Reports and Powerbuilder.

Syntax

The syntax for invoking this pseudo-procedure is as follows:

```
Call CICS.procedure-name(parm1,parm2,…) or

Call IMS.procedure-name(parm1,parm2,…)
```

Where

CICS or IMS
refers to the type of system on which the transaction will be executed. This constant is stored as the **PROC_OWNER** for the procedure.

Procedure-name
is the previously defined pseudo-procedure name that will invoke the transaction. This procedure is defined using the Shadow TSO ISPF interface.

Parm1,Parm2
are the parameters that are to be passed to the CICS or IMS transaction. These parms are defined using the map extraction procedure in the Shadow TSO ISPF interface. These parm descriptions are passed to the client program via the SQLProcedureColumns function call.

This syntax matches the IBM SQL stored procedure naming convention.

Stored Procedures

Although you can access CICS and IMS transactions with the Shadow_CICS and Shadow_IMS calls, these methods require the ability to supply parameters related to connecting to the desired CICS or IMS address space. This is no problem when using Visual Basic or C++ to code the call. However, this pseudo-stored procedure method becomes useful when these extra parameters cannot be supplied to a third party package that would otherwise be a useful client.

Also, if the user needs to supply complex input data types, this method allows the input to be mapped to pre-extracted definitions. Thus, the full range of data types supported by mainframe high level languages can be supported such as small integer, large integer, packed decimal and floating point.

In addition to this chapter, please see Chapter 3, “Running DB2 Stored Procedures” in this Guide for related information on stored procedures.

Preparing a Stored Procedure to Execute a CICS or IMS Transaction

The following steps must be completed to create a pseudo-stored procedure that can be used to execute a CICS or IMS transaction.

1. Define the DB2 table SHADOW.PROCEDURES, as instructed in optional Step 12 in Chapter 1 of the Shadow Installation Guide. Make sure that you specify a catalog prefix of SHADOW on your ODBC data source definition in order to access this table.

2. Create the input and output maps, using option 10.1, Map Extract, from the Shadow Primary Options Menu. For CICS programs, select Map ExtraCOBOL or PL/I. For IMS, select MFS. CICS maps are created from a compile listing from the CICS program that is to be executed. IMS maps are created directly from MFS source. For more information about this option, see Chapter 9, “Shadow Data Mapping Facility,” in the Shadow Server Users Guide.

3. Create the pseudo-stored procedure. This is done using the Stored Procedure option, option 10.8 from the Shadow Primary options menu. An example of this panel is shown in the following figure:

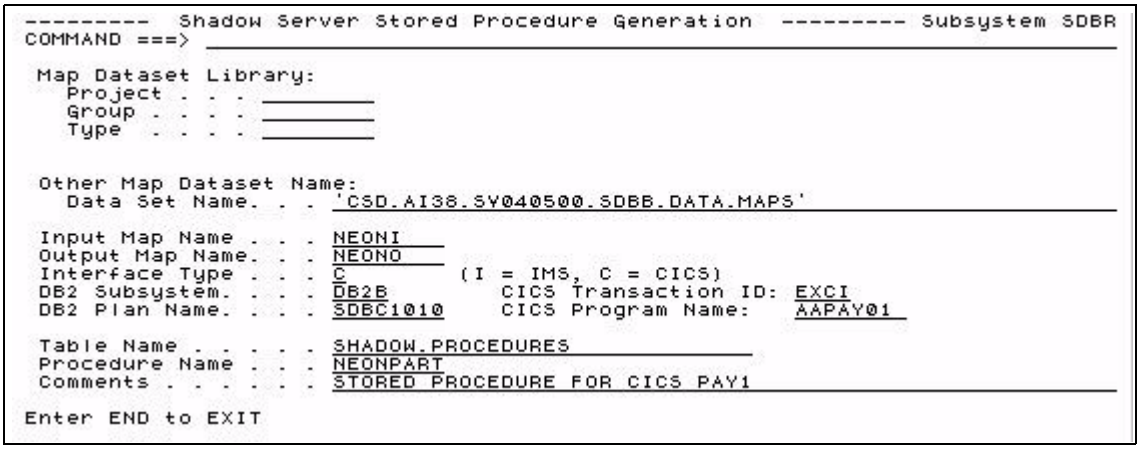


Figure 10–1.

a. Specify a procedure name. This will be written to the DB2 table – SHADOW.PROCEDURES.

b. Use transaction EXCI for CICS. This is used to communicate with the CICS address space.

c. Specify the name of the CICS program to be invoked. This program must have a valid PPT entry defined to CICS. For IMS, no transaction or program name is used. The MFS map must specify the IMS transaction or it must be entered in the 1st field of the map.

d. Specify the input and the output maps that were created in step 1.

The connection name and the target CICS or IMS address space are the default Shadow startup parameters. Related parms are:

For CICS: EXCICONNECTIONAME

For IMS: IMSPARTNERLU, IMSLOCALLU, IMSMODENAME, and IMSSECURITYTYPE

For more information, please see the Shadow Started Task Parameters in the Shadow Server and Shadow OS/390 Web Server Users' Guides.

4. To make the procedure active use the Refresh option, option 10.5 on the Shadow Mapping Facility options menu.

This pseudo-stored procedure name will now be returned by the ODBC function call SQLProcedures when directed at the Shadow address space. The Meta Data it contains will be returned by SQLProcedureColumns so that a SQLExecute can be used to invoke the procedure. Shadow will map all application parameters as mapped by the previous extract and pass them to the IMS or CICS program. All output will be passed back according to the previously mapped output map.

APPENDIX A: Shadow REXX

This appendix provides information about Shadow REXX, including its comparison to Standard REXX, its execution limits, elements, considerations, instructions, interfaces, and compiler error messages.

What Is Shadow/REXX?

Shadow/REXX is a proprietary implementation of standard REXX, an SAA-compliant programming language. A REXX-based approach was used in developing the original prototype of the Shadow OS/390 Web Server because of the many advantages the REXX language has over compiled and 4GL interpretive languages.

Shadow/REXX provides a simple but capable high-level language in which to write Web Server Transactions. Users who are completely unfamiliar with programming on a Mainframe platform can quickly learn to program in Shadow/REXX.

Shadow/REXX differs only slightly from standard REXX. Only those differences are explained here.

For more detailed information about standard REXX, refer to:

- *Modern Programming Using Rexx* by Robert P. O'Hara and David Roos Gomberg (Prentice Hall). This book includes many practical examples of REXX programming.
- IBM Hursley REXX Page (http://rexx.hursley.ibm.com/rexx/rexx.htm)
- REXX Language Associate Home Page (http://www.pvv.ntnu.no/RexxLA/)

Why Shadow/REXX?

TSO/E REXX was not used for the implementation, because:

- Shadow/REXX is specially enabled for processing within an OLTP environment. The Shadow/REXX interpreter contains facilities for limiting the overall resource utilization of any one procedure. This is important because the Shadow OS/390 Web Server may be executing hundreds of transactions simultaneously.

- Shadow/REXX is enabled for use in cross-memory environments. As the functionality of the Shadow OS/390 Web Server is extended into other areas, we can rely on it's ability to handle more complex operational environments.

- The Shadow/REXX interpreter runs faster than the TSO EXEC command for similar programs. When you use Shadow/REXX in the Web Server environment, all the code is pre-interpreted to speed processing.

Similarities Between Shadow/REXX and Standard REXX

Both Shadow/REXX and the standard REXX language:

- Enable you to issue commands to various host environments.
- Offer symbolic substitution that is simpler than that for other high-level languages.

The current version of Shadow/REXX supports these standard REXX features:

- All REXX programming structures.
- All standard SAA REXX functions with the exception of the I/O functions (CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, and LINES).
- Numbers with decimal points and exponents, as well as numeric digits with a precision up to 20 (default 9).
- Strings containing as many as 32,000 characters, including strings.

Differences Between Shadow/REXX and Standard REXX

There are several important differences between Shadow/REXX and standard REXX. These include:

PULL

- **REXX**. When a PULL instruction is executed and the external data queue is empty, a read is done from the “default character input stream.”
- **Shadow/REXX.** In Shadow Rexx, this is not practical. Within an event procedure, the only possible default character input stream is the Web Client program. It is not possible to prompt the browser program for input. This means a PULL on an empty external data queue results in a NULL (zero length) line being returned.

PUSH

- **Shadow REXX.** The PUSH instruction is not implemented in Shadow/REXX, but rather, returns the REXX error result number 64, the unimplemented feature error. However, the QUEUE instruction is implemented and can generally be used to accomplish the same results.

Shadow/REXX Execution Limits

Resource Use Monitoring

When Shadow/REXX executes a REXX program, it checks to see that a program does not consume an excessive amount of resources. This check is especially important for executing Web Transactions, since runaway procedures can degrade system performance significantly, causing poor response time and other problems.

Specifically, Shadow/REXX monitors:

- Program execution time.
- How many REXX clauses executed.
- How many REXX SAY instructions executed.
- How many host commands were issued.
- How many output lines the external data queue contains.

Parameters that Set Limits

Shadow OS/390 Web Server product parameters set limits for the execution values listed above. For any SEF Event Procedure:

This Parameter	Sets This Limit.....
SEFMAXSECONDS	Specifies the maximum time in seconds that an event procedure can execute for a given event.
SEFMAXCLAUSES	Specifies the maximum number of REXX clauses that a procedure can execute for a given event.
SEFMAXSAYS	Specifies the maximum number of SAY instructions that an event procedure can execute for a given event.
SEFMAXCOMMANDS	Specifies the maximum number of host commands that an event procedure can execute for a given event.
SEFMAXQUEUE	Specifies the maximum number of lines that a procedure can have in the external data queue for a given event. This limit can be overridden within an individual WWW transaction procedure using the QUEUESIZE() parameter.
SEFSIZE	Specifies the amount of storage an individual procedure can use for storage of REXX variable names and values, and evaluation and other work areas. This limit can be overridden within an individual WWW transaction procedure using the WORKSIZE() parameter.

Overriding Execution Limits

Shadow/REXX event procedures can override most of the execution limits by issuing the REXX OPTIONS instruction. However, once the the execution of a REXX program has begun, the maximum size of the external data queue and the amount of working storage space cannot be overridden. See WORKSIZE for a description of overrides allowed for WWW procedures.

Elements of Shadow/REXX

REXX Elements that Shadow/REXX Supports

Shadow/REXX implements all of the elements of the SAA standard REXX language except for the following:

- OPTIONS ETMODE
- PUSH
- The input/output functions CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, and LINES
- SCAN portion of TRACE.

Implementation Limits

The table below describes the minimum limits that SAA imposes and the maximum limits that Shadow/REXX imposes on REXX elements:

Item	SAA	Shadow/REXX
Literal Strings	100 Bytes	32000 Bytes
Symbol (variable name) length	250 Bytes	250 characters for local symbols; 50 characters for global symbols; 32 characters for function and routine name labels
Nesting control structures	100	Limited only by available stack space
Call Arguments	20	10
MIN and MAX function arguments	20	20
Queue entries	100	Controlled by product parameter; defaults to 3000
NUMERIC DIGITS value	1000	No maximum limit
Notational exponent value	999 999 999	999 999 999
Hexadecimal strings	250 bytes	32000 bytes
C2D input string	Limit is either 250 or the NUMERIC DIGITS value, divided by 2, whichever is smaller	4 (must be a positive number)
D2C output string	Limit is either 250 or the NUMERIC DIGITS value, divided by 2, whichever is smaller	256
X2D output string	Limit is either 500 or the NUMERIC DIGITS value minus 1	32000

Constants in Shadow/REXX

Shadow/REXX supports character strings up to 32,000 characters long. Numeric values can include decimal points and exponential notation.

Symbols in Shadow/REXX

Local symbols can contain up to 250 characters; global symbols, up to 50 characters, and labels used in function or subroutine calls, up to 32 characters.

Variable Values

Variables containing character strings can contain no more than 32000 bytes (or a lower value set by the REXXMAXSTRINGLENGTH parameter). This limitation also applies to intermediate results.

Compound Symbols

The limits on the symbol name (pre-substitution) and the derived name (post-substitution) of a compound symbol and on the value that a compound symbol takes are the same as the limits for the name and value of a regular variable.

You must treat the Web server product's environmental variables as compound symbols. For example, if you have defined the variable USER in a procedure and you refer to WWW.USER elsewhere, the USER part of WWW.USER is interpreted as the content of the previously defined variable of the same name.

Arithmetic Values and Operators

Shadow/REXX supports floating point arithmetic. Although you can set NUMERIC DIGITS higher than 9, higher values can impair performance.

Shadow/REXX Considerations

Before using Shadow/REXX, consider the following points.

- Shadow/REXX uses the following search order to locate external functions as it compiles a program:

  a. Built-in functions.

  b. Contents of the Event Procedure library in which the calling procedure resides.

  c. The SYSEXEC library concatenation.

  d. A LOAD issued for a module with the name of the external reference (that is, the standard MVS load module search mechanism is used).

- Avoid using “SWS” as the first characters of the names of REXX functions that you create.

Shadow/REXX Instructions

The following briefly explains the differences between Shadow/REXX instructions and those of standard REXX.

INTERPRET Instruction

Shadow/REXX supports INTERPRET instructions under these conditions:

- An INTERPRET instruction finds an external function or a load module only if another instruction refers to that function or module. Most product supplied built-in functions are dynamically located.
- When an INTERPRET instruction refers to global variables (for example, x = GLOBAL.A), the global variable stem must be used directly in another instruction (not another INTERPRET instruction) elsewhere in the program. You can use global variables accessed via the SWSVALUE function in an interpreted instruction.
- The total length of an interpreted string is limited by number of separate clauses to be executed and the amount of stack space available. *In general, you should not attempt to execute an INTERPRET statement which must interpret more than a few clauses.*

OPTIONS Instruction

The OPTIONS instruction under Shadow/REXX accepts the keywords shown below:

```
OPTIONS  MAXTIME=seconds | NOMAXTIME
         MAXSECONDS=seconds | NOMAXSECONDS
         MAXCLAUSES=clauses | NOMAXCLAUSES
         MAXSAYS=count | NOMAXSAYS
         MAXCMDS=count | NOMAXCMDS
         MAXCOMMANDS=count | NOMAXCOMMANDS
         MAXSTRING=bytes | NOMAXSTRING
```

Use.....	To.....
MAXTIME=seconds or MAXSECONDS=seconds	To change the limit on execution time.
NOMAXTIME or NOMAXSECONDS	To skip monitoring of execution time.
MAXCLAUSES=clauses	To change the limit on clauses executed.
NOMAXCLAUSES	To skip monitoring of clause execution.

Use.....	To.....
MAXSAYS=count	To change the limit on how many SAY instructions executed.
NOMAXSAYS	To skip limit checking for SAY instructions.
MAXCMDS=count or MAXCOMMANDS=count	To change the limit on host commands executed.
NOMAXCMDS or NOMAXCOMMANDS	To skip monitoring of host command execution.
MAXSTRING=bytes	To set maximum string length for all strings. (8)
NOMAXSTRING	To use the default maximum string length of 32000.

Format for OPTIONS Statements

Enclose all arguments except for variable names in single or double quotation marks. This prevents Shadow/REXX from parsing subclauses (such as MAXSAYS=5) before passing them to the OPTIONS statement processor.

Duration of OPTIONS Settings

Subroutines called by a REXX program inherit the OPTIONS settings, but the settings do not apply to any calling programs. If a subroutine changes a limit or skips monitoring, the previous limit is reinstated when the subroutine returns to its caller.

If you use multiple keywords referencing the same OPTIONS setting within the same REXX statement, Shadow/REXX uses the last keyword. For example, Shadow/REXX does not limit the number of host commands if you code this instruction:

```
OPTIONS "MAXCOMMANDS=100 NOMAXCMDS"
```

Sample Uses of OPTIONS

Example 1:

The following demonstrates using the OPTIONS instruction to override default execution limits for REXX programs:

```
CLAUSES = 30000
OPTIONS "MAXCLAUSES="CLAUSES
```

Example 2:

When inserted at the beginning of a REXX program, the following enables you to skip monitoring all execution limits in the program:

```
OPTIONS "NOMAXCLAUSES NOMAXTIME NOMAXSAYS NOMAXCMDS"
```

Shadow/REXX Built-in Functions

Shadow/REXX offers both standard REXX functions and a set of built-in functions designed for the Web Server environment. Refer to the Web Server API Function Index (either online or in the NEON Programming Guide) for more information.

Shadow Event Facility (SEF) Global Variables

Global variables are variables that can be shared by multiple SEF rules, Shadow/REXX programs, or both running in different address spaces.

Global variables are also compound symbols with any of the following stems:

```
GLOBAL.
GLOBALn. (The n is a single digit or letter.)
```

Use global variables as you would any other Shadow/REXX variable.

Shadow/REXX Interfaces

Shadow/REXX Interface with TSO - ADDRESS TSO

Shadow/REXX can be executed as a stand-alone command processor outside the Web Server main address space. It is used in this way to implement portions of the ISPF control interface. When used as a stand-alone command processor in a TSO/E address space, Shadow/REXX passes `ADDRESS TSO` commands to the local Terminal Monitor Program (TMP) for execution.

However, when executing Shadow Event Facility (SEF) procedures, each event is processed within the Shadow OS/390 Web Server address space. The Shadow OS/390 Web Server does not initialize itself as a TMP (Terminal Monitor Program). Therefore, **TSO/E is not available within the Shadow OS/390 Web Server's address space**.

Shadow/REXX implements an intercept which gains control when any command is issued to the **`ADDRESS TSO`** host command environment. This means all commands issued via the **`ADDRESS TSO`** environment are invalid, except the EXECIO host command. Shadow OS/390 Web Server implements its own version of EXECIO as an **`ADDRESS TSO`** host command. Therefore, you can use EXECIO, as documented for TSO/E REXX, in a Shadow/REXX program. Shadow/REXX's version of EXECIO:

- Does not support the LIFO option.
- Checks the syntax of the stem name.
- Prohibits use of global variable stems with the STEM option.
- Supports DISKRU only for true sequential data sets. You cannot use DISKRU against a partitioned data set member.

To compensate for this lack of TMP-based functionality within the **ADDRESS TSO** environment, Shadow REXXTOOLS is distributed as a separately licensed feature of the Shadow OS/390 Web Server. Shadow REXXTOOLS contains support for many functions which are not directly available from the ADDRESS TSO environment such as Dynamic Allocation requests.

A new feature of the Shadow OS/390 Web Server allows you to schedule TSO command procedures into an out-board TSO server address space using the **ADDRESS TSOSRV** host command environment.

Shadow/REXX Interface To Out-board TSO Servers - ADDRESS TSOSRV

TSO/E services are required when cusdtomers want to export their application as Web transactions. For this reason, the **ADDRESS TSOSRV** command environment was built. It passes TSO/E commands to a set of TSO server address spaces started and monitored by the main Web Server product.

WWW transaction procedures may execute TSO/E commands, within an out-board TSO/E address space by directing them to the **ADDRESS TSOSRV** command environment. The output of the TSO/E command is routed by the Web Server to the external data queue specified by the original WWW REXX procedure.

Example

```
/*WWW /LISTA
/*REXX

  ADDRESS TSOSRV
  "LISTA"        /* This command routed to out-board Server*/

  IF RC <> 0 THEN DO /* Time out or other scheduling error?*/
     ... take some recovery action
  END

  ADDRESS SWSSEND            /* Build HTTP Response        */
  "HTTP/1.0 200 OK"
  "Content-type: text/plain"
  ""
  DO WHILE QUEUED() > 0      /* While more result lines    */
     PARSE PULL LINE         /* Get result data line       */
     ADDRESS SWSSEND
     LINE                    /* Send result line as text   */
  END
```

Refer to Using TSO/E for Web Transaction Processing for information on the out-board TSO server facility.

Shadow/REXX Interface with SEF - ADDRESS SEF

You can use the ADDRESS SEF statement to pass host commands to the Shadow Event Facility (SEF). For example, the following statement tells SEF to enable the rule named WEBQUEST from a ruleset named WWW.

```
ADDRESS SEF
“ENABLE WWW.WEBQUEST”
```

Shadow/REXX Interface For Web Data Output - ADDRESS SWSSEND

You can use the **ADDRESS SWSSEND** environment to pass host data directly into an output buffer for transmission to a Web client’s browser program. Host “commands” presented to the **ADDRESS SWSSEND** environment become part of the data stream transmitted to Web Clients when the Web transaction ends.

This facility is available **only** within a Shadow/REXX language /*WWW transaction definition:

```
/*WWW  /SAMPLEDATA
/*REXX
ADDRESS SWSSEND
“HTTP/1.0 200 OK”
“Content-type: text/html”
“”
“<HTML><BODY>"
"<P>This is the response to a web transaction request."
"</body></html>"
```

Refer to the SWSSEND Host Command Environment for more information.

Compiler Error Messages

When the Shadow/REXX compiler finds syntax errors in an Shadow/REXX program or an SEF rule, the compiler generates a numbered error message. Because Shadow/REXX is an implementation of standard REXX, Shadow/REXX generates standard REXX error codes.

Non-Standard REXX Error Numbers used by Shadow/REXX

Code	Message Results
64	UNIMPLEMENTED FEATURE.

Code	Message Results
69	FUNCTION HAS TOO FEW ARGS.
70	FUNCTION HAS TOO MANY ARGS.
71	INCOMPATIBLE SHADOW/REXX CTL BLOCKS.
78	USER FUNCTION FAILED (CODE IN RC)
80	PULL FOUND EMPTY QUEUE.
86	INVALID SYMBOL.
91	INVALID OR MISPLACED OPTIONS STATEMENT The keyword(s) specified in the OPTIONS statement contain an error.
93	GLOBAL VARIABLE WORKSPACE OVERFLOW (size) The maximum amount of storage reserved for global variables (the value set by the GLOBALMAX parameter) was exceeded.
94	OVER seconds SECONDS USED FOR EXECUTION The program exceeded the maximum execution time for SEF rules (set via the SEFMAXTIME parameter) or REXX programs (set via the REXXMAXTIME parameter).
95	OVER count HOST COMMANDS ISSUED The program issued the maximum number of host commands for SEF rules or REXX programs.
96	OVER count "SAY" CLAUSES EXECUTED The program executed the maximum allowed SAY instructions for SEF rules or REXX programs.
97	OVER count CLAUSES EXECUTED The program executed the maximum allowed number of clauses for SEF rules or REXX programs.

Note:

The OPTIONS statement for an Shadow/REXX program can also generate error codes 94 through 97.

More Errors Detected: Because Shadow/REXX is a semi-compiler rather than a pure interpreter, its compile phase detects errors which other versions of REXX do not catch at execution time. This is especially true when converting programs to Shadow/REXX; you may encounter errors at compile time in supposedly error-free code. This can happen because many REXX interpreters do not detect errors in statements that do not execute.

Standard REXX Error Numbers Used by Shadow/REXX

Code	Message Results
4	PROGRAM INTERRUPTED
5	MACHINE RESOURCES EXHAUSTED
6	UNMATCHED “/*” OR QUOTE
7	WHEN OR OTHERWISE EXPECTED
8	UNEXPECTED THEN OR ELSE
9	UNEXPECTED WHEN OR OTHERWISE
10	UNEXPECTED OR UNMATCHED END
11	CONTROL STACK FULL
12	CLAUSE TOO LONG
13	INVALID CHARACTER IN PROGRAM
14	INCOMPLETE DO/SELECT/IF
15	INVALID HEXADECIMAL CONSTANT
16	LABEL NOT FOUND
17	UNEXPECTED PROCEDURE
18	THEN EXPECTED
19	STRING OR SYMBOL EXPECTED
20	SYMBOL EXPECTED
21	INVALID DATA ON END OF CLAUSE
22	INVALID CHARACTER STRING
24	INVALID TRACE REQUEST
25	INVALID SUB-KEYWORD FOUND
26	INVALID WHOLE NUMBER
27	INVALID DO SYNTAX
28	INVALID LEAVE OR REITERATE
29	ENVIRONMENT NAME IS TOO LONG
30	NAME OR STRING TOO LONG
31	NAME STARTS WITH NUMBER OR “.”
33	INVALID EXPRESSION RESULT
34	LOGICAL VALUE NOT 0 OR 1

Code	Message Results
35	INVALID EXPRESSION
36	UNMATCHED “(” IN EXPRESSION
37	UNEXPECTED “,” OR “)”
38	INVALID TEMPLATAE OR PATTERN
39	EVALUATION STACK OVERFLOW
40	INCORRECT CALL TO ROUTINE
41	BAD ARITHMETIC CONVERSION
42	ARITHMETIC OVERFLOW/UNDERFLOW
43	ROUTINE NOT FOUND
44	FUNCTION DID NOT RETURN DATA
45	NO DATA SPECIFIED IN FUNCTION RETURN
48	FAILURE IN SYSTEM SERVICE
49	INTERPRETATION ERROR

APPENDIX B: MVS Client Support

This appendix covers the ODBC interface in a COBOL client program.

Using the ODBC Interface In a COBOL Client Program

The ODBC interface provided by NEON is written in the C language and executes using the MVS C /LE runtime environment. The MVS LE runtime environment must be available on the host machine where the COBOL application is executing.

For this reason, in order to use the ODBC client interface from a MVS host COBOL application program, the execution environment of the COBOL program must first be determined.

If the COBOL program is compiled using COBOL MVS and uses the LE environment, then the client COBOL program must be statically linked with the SDBODBC module. If the execution environment is not LE, then the client COBOL program must be statically linked with SDBSQLI. SDBSQLI provides the entry points required by the COBOL program and in addition calls CEEPIPI (an IBM supplied module) to establish the LE environment for the SDBODBC module. Please note that SDBSQLI is not reentrant at this time and precautions must be taken accordingly.

The following table describes the available ODBC function calls and the entry point name that must be used to perform the function from a high level language.

ODBC Function	Entry Point	Description
SQLAllocEnv	SDODAE	Allocates memory for the environment handle and initializes the ODBC call level interface for use by the application.
SQLAllocConnect	SDODAC	Allocates memory for a connection handle with the environment identified by the environment handle passed.
SQLAllocStmt	SDODAS	Allocates memory for a statement handle and associates the statement handle with the connection specified by the connection handle passed.
SQLDriverConnect	SDODDC	Loads a driver and establishes a connection to a data source.
SQLExecDirect	SDODED	Executes a preparable statement.
SQLFetch	SDODFT	Fetches a row of data from the result set.

ODBC Function	Entry Point	Description
SQLGetData	SDODGD	Returns the result data for a single unbound column in the current row.
SQLDisconnect	SDODDI	Closes the connection associates with a specific connection handle.
SQLFreeStmt	SDODFS	Frees a statement handle and all associated memory.
SQLFreeConnect	SDODFC	Frees a connection handle and all associated memory.
SQLFreeEnv	SDODFE	Frees the environment handle and all associated memory.

A detailed description of the ODBC API can be found in the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*, available from Microsoft Press. The parameter list provided to the ODBC interface are all pointers by reference to allow calls to be made from COBOL.

For example, to call the SQLAllocEnv ODBC function, the following would be coded in the COBOL application:

```
WORKING STORAGE SECTION.
…
01 WS-HENV              PIC S9(08) COMP VALUE +0.
…
PROCEDURE DIVISION.
…
CALL 'SDODAE' USING WS-HENV.
…
```

A complete COBOL example program may be found in NEON.SV040100.SAMP, member ODBC#LE. A COBOL copybook, named SDBODCP, is also found in NEON.SV040100.SAMP. This copybook provides for constant values associated with the ODBC API.

Data Type	Picture Clause
HDBC	PIC S9(08) COMP or USAGE IS POINTER
HDBC FAR *	PIC S9(08) COMP or USAGE IS POINTER
HENV	PIC S9(08) COMP or USAGE IS POINTER
HENV FAR *	PIC S9(08) COMP or USAGE IS POINTER
HSTMT	PIC S9(08) COMP or USAGE IS POINTER
HSTMT FAR *	PIC S9(08) COMP or USAGE IS POINTER
HWND	PIC S9(08) COMP
PTR	PIC S9(08) COMP or USAGE IS POINTER
SDWORD	PIC S9(08) COMP

Data Type	Picture Clause
SDWORD FAR *	PIC S9(08) COMP or USAGE IS POINTER
SWORD	PIC S9(04) COMP
SWORD FAR *	PIC S9(08) COMP or USAGE IS POINTER
UCHAR FAR *	PIC S9(08) COMP or USAGE IS POINTER
UWORD	PIC S9(04) COMP
UWORD FAR *	PIC S9(08) COMP or USAGE IS POINTER

Glossary

The following list is compilation of some of the terms you will find used in NEON's documentation. If you do not find the term you are looking for, the best reference to turn to is the IBM publication: *Dictionary of Computing* (SC20-1699). You may also want to check the glossaries of the manuals listed in "Related Publications."

ACB — **Access Control Block.** A control block that links an application program (for example, a CICS system) to an access method (for example, VSAM or VTAM). In communication with DL/I, an ACB is used only when the underlying access method is VSAM.

ACEE — **Access Control Environment Element.** (CICS for MVS only.) In RACF, a control block containing details of the current user, including user ID, current connect group, user attributes, and group authorities. An ACEE is constructed during user identification and verification.

ACF — **Advanced Communication Function.** A group of IBM licensed programs that uses the concepts of Systems Network Architecture (SNA) including distribution of function and resource sharing.

ADABAS — **Adaptable Database System.** A type of database provided by Software AG.

address space — The range of addresses available to a computer program; the area of virtual storage available to a particular job or started task.

AMODE — **Addressing Mode.** An attribute in MVS and MVS/XA program that refers to the address length that a program is prepared to handle upon entry. In MVS/370, an addresses can be 24 bits in length. In the MVS/XA program, addresses can be 24 bits or 31 bits in length.

APF — **Authorized Program Facility.** A security feature of the MVS operating system that restricts the running of programs that make use of privileged machine instructions.

API — **Application Program Interface.** A set of routines provided in libraries that extends a language's functionality

APPC — **Advanced Program-to-Program Communication.** The general facility characterizing the LU 6.2 architecture and its various implementations in products.

application group name — In IMS/VS, a name that represents a defined group of resources (program specification blocks, transaction names, and logical terminal names).

APPN — **Advanced Peer-to-Peer Networking.** An extension to Systems Network Architecture (SNA). Extends the LU 6.2 peer orientation for end-user services to network control and supports multiple LU types, including LU 2, LU 3, and LU 6.2

ASCH — **Application Scheduler.** MVS application scheduler.

Auto-HTML — See Web Enabling.

Block Connection — A logical connection in which the connection resources (i.e., network session, threads etc.) are held for the lifetime of the connection and in which SQL operations may be grouped. Multiple sends may be issued on a physical network session. Each send is one or more SQL operations (a group). This type of connection is very efficient in network usage (only one connection made and network I/Os are reduced), efficient in CPU utilization (no overhead for multiple connections) but holds mainframe resources (TCBs, threads and TCP/IP sessions) over relatively long periods of time. The number of connections is limited by the number allowed for the scarcest resource.

BMP — **Batch Message Processing.** In IMS/VS, a batch processing program that has access to online databases and message queues.

BMS — **Basic Mapping Support.** Provides most of the input and output facilities required by application programs; allows you to separate the tasks of display design and CICS application programming. BMS interprets generalized application program output commands, and generates data streams for specific output devices. (Such data streams are said to be device dependent.) Conversely, it transforms incoming data streams to a form acceptable to application programs. It obtains information about the format of the data stream for the terminal from the terminal control table terminal entry (the TCTTE) for the task, not from the application program. The same BMS input or output commands in an application program can be used with different kinds of device.

CAF — **Call Attachment Facility.** The component of DB2 used by application programs in any address space to connect the application to DB2.

CCTL — **Coordinator Control Subsystem.** (CICS for MVS only.) In IMS/ESA, the transaction management subsystem that communicates with the DRA, which in turn communicates with DBCTL. In a CICS-DBCTL environment, the CCTL is CICS. The term is used in a number of IMS operator commands that apply to DBCTL, and in the IMS manuals.

CDRM — **Cross Domain Resource Manager.** The functions of the system services control point (SSCP) that control initiation and termination of cross-domain sessions.

CGI — **Common Gateway Interface.** An interface between a client (web browser) and internet connection server that receives input data from standard input, parses the data and translates the escaped characters back into real characters, performs any business process required, and sends a response to the client.

CICS — **Customer Information Control System.** A transaction processing extension to the operating system of IBM mainframe computers that makes it easier to write programs that enter, retrieve, and update data interactively from remote terminal services.

client/server — An application architecture where a remote system (the client) accesses data on a local system (the server).

CMOS — **Complementary Metal-Oxide Semiconductor.** A technology that combines the electrical properties of n-type semiconductors and p-type semiconductors.

COBOL — **Common Business-Oriented Language.** High-level programming language based on English, and used for business applications.

COMMAREA — **Communication Area.** A CICS area that is used to pass data between tasks that communicate with a given terminal. The area can also be used to pass data between programs within a task.

CORBA — **Common Object Request Broker Architecture.** A standard for distributed objects being developed by the Object Management Group. Provides the mechanisms by which objects transparently make requests and receive responses as defined by OMG's ORB. The CORBA ORB is an application framework that provides interoperability between objects built in different languages, running on different machines in heterogeneous distributed environments.

CP — **Control Program.** A computer program designed to schedule and supervise the execution of programs of a computer system.

CPI-C — **Common Programming Interface for Communications.** A type of API interface for LU 6.2.

CPU — **Central Processing Unit.** A processing unit. The part of a computer that includes the circuits controlling the interpretation and execution of instructions.

CS — **Cursor Stability.** An option used with block fetch allowing data changes to take place between the time the data is extracted and the time that it is actually used by the application.

CTDLI — A routine provided by IMS that processes DL/I calls from programs written in the C language.

DASD	**Direct Access Storage Device.** A device in which access time is effectively independent of the location of the data.
DB	**Database.** A collection of data with a given structure for accepting, storing, and providing, on demand, data for multiple users.
DB/DC	**DATABASE/Data Communication.** Type of IMS system that supports database as well as data communication access.
DB2	**DATABASE 2.** An IBM relational database management system. See DBMS.
DBA	**Database Administrator.** The person who maintains the database management system. Database Administration. The act of maintaining a database management system.
DBCTL	**Database Control.** (CICS for MVS only.) An interface between CICS for MVS and IMS/ESA that allows access to IMS DL/I full-function databases and to data entry databases (DEDBs) from one or more CICS systems without the need for data sharing. It also provides release independence, virtual storage constraint relief, operational flexibility, and failure isolation.
DBMS	**Database Management System.** System software for storing, accessing and removing information. A relational DBMS, such as DB2, permits a wide variety of views of the stored information without customer programming.
DBRM	**Database Resource Manager** (for example, DB2, IMS, Oracle, etc.)
DDF	**Distributed Data Facility.** The component of DB2 used to access databases and tables on remote nodes in the network.
ddname	**Data Definition Name.** The name of a data definition statement that corresponds to a data control block containing the same name.
DES	**Data Encryption Standard.** The National Institute of Standards and Technology Data Encryption Standard, adopted by the US Government, allowing only hardware implementations of the data encryption algorithm.
DFP	**Data Facility Products.** A group of IBM supplied access methods and utilities.
DL/I	**Data Language l.** In IMS/VS, the data manipulation language that provides a common high-level interface between a user application and IMS/VS. In VSE and CICS/VS, a database access language.
DMF	**Data Mapping Facility.** A feature of Shadow Direct that allows mapping from various sources. Data maps are created via a series

	of ISPF panels that allow the user to specify a dataset containing a listing of a program that contains a data definition.
DNS	**Domain Name Server.** In TCP/IP, a server program that supplies name-to-address translation by mapping domain names to internet addresses.
DRDS	**Dynamic Reconfiguration Data Set.** In VTAM, a data set used for storing definition data that can be applied to a generated communication controller configuration at the operator's request, or can be used to accomplish dynamic reconfiguration of NCP, local SNA, and packet major nodes.
DSNAME	**Dataset Name.** The term or phrase used to identify the data set.
DSA	**Dynamic Storage Area.** (CICS/VSE only.) System initialization parameter that pre-allocates the CICS dynamic storage area at system initialization.
DSN command	**Data source** (definitions); a DB2-supplied TSO command used to run DB2-based application programs and issue commands to DB2.
DTS	**Dynamic To Static Conversion Facility.** Also known as the Plan-Based Static SQL Conversion Facility. DTS converts dynamic SQL to plan-based static SQL. DTS fully supports plan-based security and is not subject to any restrictions with respect to COMMIT and ROLLBACK (including holding locks across a COMMIT or ROLLBACK).
DTSG	A utility developed by NEON System's UK office that provides an easier to use, graphical front end to the Dynamic to Static Analyzer (DSA) program. DTSG was developed using Visual Basic Version 4.0.
EBCDIC	**Extended Binary-Coded Decimal Interchange Code.** A coded character set of 256 8-bit characters.
ECF	**Enterprise Control Facility.** A management tool that is installed with the Enterprise Server and used to define monitoring and control parameters for the local Enterprise Server or any other Enterprise Server on the network.
EOV	End Of Volume.
ESTAE	**Extended Specify Task Abnormal Exit.** An MVS macroinstruction that provides recovery capability and gives control to the user-specified exit routine for processing, diagnosing an abend, or specifying a retry address.
event	A site-defined action, such as a SQL statement, or CICS, IMS or OS/390-MVS application program.

EXCI	**External CICS Interface.** Used by SHADOW_CICS to connect to the specified CICS region and execute the specified program.
EXEC	A TSO command for running REXX programs; a REXX program.
Fast Logon	A connection startup process where handshaking is kept to a minimum to reduce the number of network I/Os (from 2 to 1). Since assumptions are made about the level of code at each end, code level dependencies exist. If these assumptions are incorrect, the connection will fail.
FTP	**File Transfer Protocol.** A protocol used to request and receive files and file system directory information from another computer.
Group	A sequence of SQL operations that is collected and sent together as one block. A group is terminated by a resultset returning SQL operation (i.e., SELECT or CALL) or a logical unit of work termination (i.e., COMMIT - note: a ROLLBACK will cause the operations to be discarded). Only INSERTs, DELETEs and UPDATEs may be grouped. The maximum grouping allowed is determined at initialization time.
GUI	**Graphical User Interface.** A type of computer interface consisting of a visual metaphor of a real-world scene, often of a desktop.
HFS	**Hierarchical File System.** A disk-based filing system built on a hierarchy of special files called directories or folders. Descends from a main directory, called the root. Each lower level is a subsidiary.
HTML	**Hypertext Markup Language.** a simple markup language used to create hypertext documents that are platform independent. HTML documents are SGML (Standard Generalized Markup Language) documents with generic semantics that are appropriate for representing information from a wide range of domains. HTML markup can represent hypertext news, mail, documentation, and hypermedia; menus of options; database query results; simple structured documents with in-lined graphics; and hypertext views of existing bodies of information.
HTML Extension Facility	A run-time tailoring facility supported by Shadow OS/390 Web Server for text format data files. Provides extremely flexible and easy-to-use support for the creation of customized HTML forms and web pages. You make use of the HTML Extension Facility by including HTML-like statements directly within your source file. When the source file is to be transmitted out-bound, the Shadow OS/390 Web Server evaluates the HTML Extension statements and customizes the information that is actually sent.

HTTP	**Hypertext Transfer Protocol.** Protocol used by the World Wide Web. It allows the retrieval of virtually any digital file, in a format suitable for later rendering the file in its original text, audio, or visual media presentation form.
IDMS	Type of database management system supplied by Computer Associates.
IMS	**Information Management System.** An IBM hierarchical database management system.
Internet	A wide area network connecting many networks to allow the free flow of information between otherwise unconnected and often very incompatible computer systems.
Intranet	A closed subnetwork, based on Internet technology. It operates the same way as the global Internet, but usually exists within the confines of a single organization using private communication pathways. An intranet is used to disseminate information to "authorized" users, such as those within the organization, while preventing some or all access from outside the organization.
IP	**Internet Protocol.** A protocol used to route data from its source to its destination in an Internet environment.
IP Address	**Internet Protocol Address.** A two part address, used by TCP/IP to route information packets from one node in the network to another. Within a TCP/IP network IP addresses must be unique.
I/O	**Input/Output.** Pertaining to input, output, or both; or pertaining to a device, process, or channel involved in data input, data output, or both.
IPCS	**Interactive Problem Control System.** A component of VM (virtual machine) that permits online problem management, interactive problem diagnosis, online debugging for disk-resident CP abend dumps, and problem tracking and reporting.
ISPF	**Interactive System Productivity Facility.** An IBM-licensed program that serves as a full-screen editor and dialogue manager; used for writing application programs, it provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user.
ISO	International Standards Organization.
IUCV	**Inter-User Communications Vehicle.** An API used by Shadow Server to communicate with IBM TCP/IP.
JCL	**Job Control Language.** A control language used to identify a job to an operating system and to describe the job's requirements.

LAN	**Local Area Network.** A computer network located on a user's premises within a limited geographical area.
LPA	**Link Pack Area.** An area of main storage containing re-enterable routines from system libraries. In OS/VS2, an area of virtual storage containing re-enterable routines that are loaded at IPL time and can be used concurrently by all tasks in the system.
LRECL	**Logical Record Length.** In CICS/VS, the length of a logical record, which is a data record sent by one transaction program to another. In VSAM, the length of a unit of information normally pertaining to a single object.
LU	**Logical Unit.** A type of network accessible unit that enables end users to gain access to network resources and communicate with each other.
LU 6.2	**Logical Unit 6.2.** An SNA defined protocol for communication between two applications.
LUOW	**Logical Unit Of Work.** In IMS/VS, the processing unit that a program performs between synchronization points.
LZ	**Lempel Ziv.** A type of compression based on repeated characters in the data.
Message Connection	A logical connection in which the connection resources (i.e., network session, threads etc.) are held for the duration of a LUOW in which SQL operations may be grouped. Only one send may be issued on a physical network session. Each send must be a LUOW. INSERTs, DELETEs and UPDATEs cannot be mixed with SELECTs or CALLs without intervening COMMITs or ROLLBACKs.). This type of connection increases network usage (multiple connections) and CPU utilization (multiple connections) but releases mainframe resources (TCBs, threads and TCP/IP sessions) after relatively short periods of time. This is the most efficient mode as far as holding of mainframe resources is concerned. The network usage is greater than or equal to that for Transblock mode due to sessions being terminated after each send. The number of connections may exceed the number of actual resources.
MFS	**Message Format Services.** In IMS/VS, an editing facility that allows application programs to deal with simple logical messages instead of device-dependent data, thus simplifying the application development process.
MIB	Management Information Block.
MIME	**Multimedia Internet Mail Extension.** A type of Internet file supported by Shadow OS/390 Web Server.

MQ Series	Middleware which focuses on reliable and guaranteed delivery by continually retrying to send the message even if there has been gateway failure or a network outage. It even survives a restart of the queue manager.
MRO	**Multiregion Operation.** Communication between CICS systems in the same processor without the use of SNA network facilities.
MTS	**Multithreaded Server.** A type of transactional and object broker server.
MUNIX	Combination of UNIX and 0S/390-MVS knowledge.
MVS	**Multiple Virtual Storage.** An operating system for IBM System 370 hardware. Each user of the system is provided a "virtual" address space equal in size to the addressing limit of the machine. Also shorthand notation for MVS/XA (MVS/Extended Architecture) and MVS/ESA (MVS/Enterprise Systems Architecture).
NDS	**NEON Data Stream.** An ODBC-optimized protocol, implemented between the driver and the server components. NDS interacts with the network at the transport layer, thus avoiding the overhead inherent in higher-level network APIs. It also enhances performance in a variety of ways, including compressing the data, minimizing the number of client-to-server round trips, and increasing the network buffer size.
NLS	**National Language Support.** The modification or conversion of a US English product to conform to the requirements of another language or country.
NT	Network Terminal.
OC	Open Client. Type of API. Not supported by Shadow Direct.
ODBC	**Open Database Connectivity.** An API created by Microsoft that allows applications to access data in database management systems (DBMS) using SQL as a standard. This interface was designed to allow a single application to access many different database management systems.
OE	Open Edition.
OLTP	Online-Transaction-Processing.
OS/2	**Operating System/2.** An IBM supplied operating system for IBM personal computers; has many features, such as multitasking, similar to those of mainframe operating systems.
PCB	**Program Communication Block.** An IMS control block that describes an application program's interface to and view of an IMS

	database or, additionally for message processing and batch message processing programs, to the source and destinations of messages. PCBs are defined by the user during PSB generation.
PDS	**Page Data Set.** A method of storing several programs, such as REXX programs, as members of a single data set. In System/370 virtual storage systems, a data set in external page storage in which pages are stored.
Permanent Connection	A logical connection in which the connection resources (i.e., network session, threads etc.) are held for the lifetime of the connection and each SQL operation is transmitted and executed separately (i.e., no grouping). Multiple sends may be issued on a physical network session. Each send is an individual SQL operation. This type of connection is efficient in network usage (only one connection made) and CPU utilization (no overhead for multiple connections) but holds mainframe resources (TCBs, threads and TCP/IP sessions) over relatively long periods of time. The number of connections is limited to the number allowed for the scarcest resource.
PGP	**Pretty Good Privacy.** Allows companies to perform Electronic Data Interchange (EDI) over the Internet with privacy, authentication, and convenience; combines the convenience of the Rivest-Shamir-Adleman (RSA) public key cryptosystem with the speed of conventional cryptography, message digests for digital signatures, data compression before encryption, good ergonomic design, and sophisticated key management.
PL/I	**Programming Language One.** A programming language designed for numeric scientific computations, business data processing, systems programming and other applications.
PO	**Partitioned Organized.** Type of dataset organization.
Port	A 16-bit number used along with IP address to uniquely identify an application on a node within a TCP/IP network.
PSB	**Program Specification Block.** The control block that describes databases and logical message destinations used by an application program. A PSB consists of one or more PCBs.
PTF	**Program Temporary Fix.** A temporary solution or by-pass of a problem diagnosed by IBM as resulting from a defect in a current unaltered release of the program.
PU	**Physical Unit.** The component that manages and monitors the resources associated with a node, as requested by an SSCP via an SSCP-PU session. This term applies to type 2.0, type 4 and type 5 nodes only.

QMF Query Management Facility.

RACF **Resource Access Control Facility.** An IBM-licensed program that provides for access control by identifying and verifying the users to the system, authorizing access to protected resources, logging the detected unauthorized attempts to enter the system, and logging the detected accesses to protected resources.

RC **Return Code.** A code used to influence the execution of succeeding instructions; a value returned to a program to indicate the results of an operation requested by that program.

RDBMS **Relational Database Management Systems.** A type of database management system that stores information in tables – rows and columns – and conducts searches by using data in specified columns of one table to find additional data in another table.

RDT **Resource Definition Table.** In VTAM, a table describing the characteristics of each node available to VTAM, and associating each node with a network address.

REXX **Restructured Extended Executor.** An interpretive language used to write command lists.

RFC Request for Comments.

RPC **Remote Procedure Calls.** Allows a client to execute a program on a server, with the program being remote to the client.

RR **Repeatable Read.** An option used with block fetch, allowing many more pages to be locked for update, especially if the number of rows normally extracted by the query is small.

RSA **Rivest-Shamir-Adleman.** A scheme for public key cryptography.

RSP Remote Stored Procedures.

SAA **Systems Application Architecture.** A set of guidelines promoted by IBM for standardizing the design of large pieces of software. It includes a set of user interface guidelines called Common User Access (CUA), as well as guidelines for data communications, programming languages, and procedure libraries.

SAF **System Authorization Facility.** An MVS facility for routing authorization requests to RACF or equivalent system security packages.

SAM **Shadow Activity Monitor.** Provides a workstation-based tool for viewing and reporting the Shadow Server logs. SAM functions as a standard Shadow Direct ODBC client.

SDF	**Shadow Diagnostic Facility.** An ISPF-based application, allowing the administrator to view summary and detail information related to connectivity and to take actions to correct connectivity problems. All of the diagnostic, monitoring, and control information can be accessed and updated through the SDF.
SEF	**Shadow Event Facility.** A comprehensive and flexible mechanism for controlling the overall Shadow Direct client/server environment; allows each installation to tailor the execution characteristics of Shadow Direct to whatever level of detail (per-user, per-group, by time-of-day, etc.) is required.
SID	Site ID.
SMF	**System Management Facility.** An optional control program feature of OS/VS that provides the means for gathering and recording information used to evaluate system usage.
SNA	**Systems Network Architecture.** A layered scheme for communication between devices and applications in a network. Applies mainly to IBM networks.
SNMP	Simple Network Management Protocol.
SPUFI	**SQL Processor Using File Input.** An interactive component of DB2, used to query and maintain DB2 databases.
SQL	**Structured Query Language.** A non-procedural language for creating, querying, and maintaining relational databases.
SRB	**Service Request Block.** (CICS for MVS only.) An MVS dispatchable unit.
SRM	**System Resources Manager.** A group of programs that controls the use of system resources in order to satisfy the performance objectives of the installation.
SSL	**Security Socket Layers.** Encryption for the highest client/server security standard in practical use today.
Table	A named DB2 object, consisting of a specific number of columns and zero or more unordered rows of data.
TCB	**Task Control Block.** In CICS for MVS, an MVS control block. A TCB is created for each MVS task. Several TCBs are created for CICS management programs. All CICS application programs and all non-reentrant CICS code run under a single quasi-reentrant TCB.
TCP/IP	**Transmission Control Protocol/Internet Protocol.** A protocol specifically designed to facilitate communications between heterogeneous networks.

Thread An individual unit of work in OS/390-MVS used for authorization, data access, transaction access, monitoring and control.

TIB Terminal Information Block.

TMP **Terminal Monitor Program.** In TSO, a program that accepts and interprets commands from the terminal and causes the appropriate command processors to be scheduled and executed.

TNUF **Table Name Utilization Facility.** A feature of Shadow Direct that allows on-the-fly modification of table names on a user-by-user basis.

TP **Transaction Program.** A program that processes transactions in an SNA network.

TPL **Transport Parameter List.** An API used by Shadow Server to communicate with Interlink TCP/IP.

Transaction Connection A logical connection in which the connection resources (i.e., network session, threads, etc.) are held for the duration of each LUOW and each SQL operation is transmitted and executed separately (i.e., no grouping). Multiple sends may be issued on a physical network session. Each send is an individual SQL operation. The physical network connect is terminated at the end of a LUOW (i.e., COMMIT or ROLLBACK). This type of connection increases network usage (multiple connections) and CPU utilization (multiple connections) but releases mainframe resources (TCBs, threads and TCP/IP sessions) after relatively short periods of time. The number of connections may exceed the number of actual resources.

TransBlock Connection A logical connection in which the connection resources (i.e., network session, threads etc.) are held for the duration of each LUOW and in which SQL operations may be grouped. Multiple sends may be issued on a physical network session. Each send is one or more SQL operations (a group). This type of connection increases network usage (multiple connections) and CPU utilization (multiple connections) but releases mainframe resources (TCBs, threads and TCP/IP sessions) after relatively short periods of time. The network usage is less than or equal to that for Transaction mode due to grouping of sent data. The number of connections may exceed the number of actual resources.

TSO **Timesharing Option.** The interactive timesharing component of the MVS operating system that supports timesharing terminals.

TSS **Time Sharing System.** A programming system that provides users with conversational online access to a computing system with one or more processing units and simultaneously processes batched jobs.

UDP	**User Datagram Protocol.** In TCP/IP, a packet-level protocol built directly on the Internet protocol layer. Used for application-to-application programs between TCP/IP host systems.
UNIX™	An operating system developed by Bell Laboratories that features multiprogramming in a multi-user environment. The UNIX operating system was originally developed for use on minicomputers but has been adapted for mainframes and microcomputers.
URL	Uniform Request Locator.
VCF	**Virtual Connection Facility.** Allows sharing of OS/390-MVS connectivity resources across a larger user population by transparently switching connections between “real” and “virtual” as the application shifts from active to idle, and vice versa.
Virtual Storage	An operating system technique for providing more addressable storage to programs than is actually available on the hardware.
VM	**Virtual Machine.** A virtual data processing system that appears to be at the exclusive disposal of a particular user, but whose functions are accomplished by sharing the resources of a real data processing system.
VSAM	**Virtual Storage Access Method.** A type of data set maintained by TSO's Access Method Services program. VSAM datasets may be accessed sequentially and randomly.
VTAM	**Virtual Telecommunications Access Method.** IBM mainframe software that implements portions of the Systems Network Architecture (see SNA).
Web Enabling	The execution of online IMS transactions and commands converted by Shadow OS/390 Web Server into HTML format.
WLM	**Work Load Manager.** A component of the OS/390 operating system, first introduced in MVS/ESA 5.1. It is a policy driven manager system of resources that is intended to allow a user to define system performance goals in the same terms that would be used in a service level agreement.
WWW	World Wide Web.
Work Station	A powerful microcomputer typically used for scientific and engineering calculations. A workstation typically has more than four megabytes of RAM, more than 100 megabytes of disk capacity, and a screen with graphics resolution of at least 800 by 1000. Examples are the Sun Sparcstation and IBM RS/6000.

Wrap-Around Trace Facility A Shadow Server tool, designed to record critical events in the life of each individual transaction process. In addition, the tool is designed to record critical internal information which can be used to debug and correct problems within the Server itself. The wrap-around trace consists of a large block of virtual storage, which can optionally be backed by a data-in-virtual linear dataset. This block of virtual storage is sub-divided into a *status area, a configurable number of event blocks, and a series of vector tables.*

Index

Symbols

A

B

C

D

E

F

G

M

N

O

P

R

S

T

U

V

W

Reader's Comment Form

At NEON Systems, Inc. we are always looking for good ideas. If you have a suggestion or comment regarding any of our publications, please complete this form, and mail or fax it to us at the following address. Thank you.

Please complete the following information, or attach your business card here.

Your Name: _____

Phone Number: _____

Your Company: _____

Address: _____

_____

Publication Name: _____

Version *and* Edition Numbers (see page ii)**:** _____

Suggestion/Request: _____

_____

_____

_____

_____

_____

_____

Please mail or fax this page to:

NEON Systems, Inc.
14100 SW Freeway, Suite 500
Sugar Land, Texas 77478, U. S. A.

Fax Number: (**281) 242-3880**