dSERVE™

Business Basic File Services for the Rest of the World

Published under license by Synergetic Data Systems Inc.

Version 2.0

Table of Contents

Introduction

dSERVE opens up Business Basic data to non-Business Basic programs through TCP/IP sockets and live Business Basic tasks which act as proxies for other programs. Traditionally, Business Basic data files have been proprietary structures that are accessible only to the native Business Basic language that creates and manipulates them. In recent years, Business Basic vendors have supplied ODBC connectivity to their data files, but often this imposes severe performance penalties, and also restricts the use of that data to Windows applications that support ODBC.

The goal of dSERVE is to provide fast, native access to Business Basic data files from non-Business Basic environments. Examples include Visual Basic, Java, C++, and Perl. In fact, any language that can use TCP/IP sockets, from any machine that can connect to the server, can interact with dSERVE and therefore gain direct read access to the data.

For Windows users, we have developed a dynamic link library (DLL) and an ActiveX DLL, so the management of a connection and access to the data files can be performed without the need to write an interface to the server.

For Unix and Linux users, we have developed PHP and Perl modules that can be used to connect to and operate a dSERVE session.

The possible uses for this technology are endless. For example, company sales representatives on the road could open an Excel spreadsheet and run a macro to quickly retrieve recent orders from a large history file, over the Internet. A Perl or PHP programmer from your ISP could read live inventory status for your web site. A Visual Basic or Excel programmer could write reports using the DLL to access live, formatted data, and server-based calculations.

In this guide, we have documented both the server access model and the DLL functions. Depending on your programming environment, you can choose which method to use to obtain access to the Business Basic data.

Application Architecture

dSERVE is a client/server technology. The server is the machine running Business Basic tasks, while the client is any machine, including the server machine itself, using local access, that can open a TCP/IP socket connection to the server. Clients can be developed in any TCP/IP-aware language. In Windows, the client is usually an application designed to use the sdServe.dll ActiveX DLL provided with dSERVE. In Unix, both Perl and PHP clients are provided. In other environments, the server interface has been documented so that a programmer can develop an interface.

The server is the machine running the dSERVE server software. It can be any machine with TCP/IP networking support and with PRO/5 Revision 2.2 (and up) or ProvideX installed. The dSERVE server is a background process that listens on a specific TCP/IP port for client connections.

As connection requests come in, the server validates the client IP address against security and license count tables, and then starts an alternate process to handle the client's requests. The alternate process can be opened on a random port or on one of a set of pre-defined ports (defined with the procports value in dserve.ini). The client is instructed which port to use, and opens a second connection to the alternate process. Note that this use of a secondary port imposes additional configuration requirements if a firewall or network address translation (NAT) is used on a network. A firewall must allow not only the main listening port, but any configured procports as well. Similarly, if NAT is used, then port forwarding must be configured for both the listening port and the procports.

The client then issues requests to the alternate process to open files and dictionaries, read files by key or sequentially, and retrieve whole records, fields, or expressions. For most applications, the client software will open files via a data dictionary, then sequentially read some or all records in a file, sometimes reading related records from one or more secondary files as well. For applications that require updating of the Business Basic files, dSERVE can be configured to allow writes and removes from files. This access can be controlled at a file level from either the server or the client application.

When the client application is complete, it closes the connection to release the server process.

Business Basic File Concepts

With the computer industry focused on relational databases, it is easy to forget that b-tree and ISAM file structures still serve the purpose of record retrieval in a huge base of legacy and modern business applications. Business Basic has offered variations of these structures since the early 1970's, and nearly all files used in Business Basic applications offer fast, low-overhead, keyed access to data records. dSERVE is designed to work with keyed file structures, known as Mkeyed, Direct, and Sort files in the BBx and ProvideX world.

Keyed files associate a unique key value with a record. Applications can use a key value, such as an inventory item code, an employee's social security number, or a customer ID code, to read a specific record in a file. This first key is known as the Primary Key to the file. In the case of multi-keyed files, one or more alternate keys can also be used to retrieve records. Alternate keys are often used to provide sort paths to a file. For example, a customer file might contain alternate keys by name and zip code, so an application could retrieve customer records in name or zip code order, in addition to the customer ID order of the primary key.

Records in a keyed file, analogous to a row in a SQL database, can be retrieved randomly if the read function is given a key value to read with. This random access returns a single record very quickly, and can be used to set a position in the file. The position in a keyed file is called a key pointer, similar to a cursor in a SQL database. The key pointer is simply a position in the file, in primary or secondary key order. The key pointer can be moved at will by a program, so sequential processing of records can be started and ended at any point in the file.

Business Basic records are generally composed of fields, like a SQL row is composed of columns. Normally all records in a file are structured of the same fields, but this is not a requirement, and many applications store different record structures in the same file. Fields can contain either numeric or string data. The use of a particular field can vary, and its data can be interpreted as numeric or string at the time it is read and processed. In cases where the records in a file can contain different structures, it is possible to define automatic filtering so that specific types of records are associated with a particular dictionary definition.

Many Business Basic applications support a data dictionary that gives names to the fields in a record. If your application provides a dictionary, then the effort required to identify and parse records is simplified, and dSERVE can return and manipulate files and fields based on their dictionary names.

dSERVE offers functions for random reading of records, as well as forward and backward sequential reading. Records can be retrieved in either record style or field style, meaning the client application can parse fields on its own, or can use dSERVE, and if available, a dictionary, to parse fields and even return expressions.

Server Installation and Setup

Unix Servers

From CD:

- Login as root.
- Mount the CD in a manner that provides lowercase filenames. The CD is produced in ISO9660 format. Example: **mount –o lower /dev/cd0 /cdrom**.
- cd to the dserve/unix directory in the cd mount directory.
- Execute the install script: **./install.sh**.
- View and accept the license agreement presented.
- When prompted, enter the directory where dSERVE will be installed.
- The install script will install dSERVE in the selected directory, and then invoke the setup script, "./dsetup.sh".
- dsetup.sh will prompt for the directory where PRO/5 Revision 2.2 or ProvideX is installed. Once a valid directory is entered, the script "/usr/bin/ds20d" is created, which is used to manage the dSERVE server.

From a download:

- Login as root.
- Create a directory for dSERVE. Example: **mkdir /usr/dserve**.
- Move the download file dserve.tar.Z to the dSERVE directory.
- Uncompress and extract the files: **uncompress dserve.tar.Z; tar xvf dserve.tar**.
- Execute the setup script: **./dsetup.sh**.
- dsetup.sh will prompt for the directory where PRO/5 Revision 2.2 or ProvideX is installed. Once a valid directory is entered, the script "/usr/bin/ds20d" is created, which is used to manage the dSERVE server.

You can verify that the server is properly installed by entering the command **ds20d –v**. If a version information response prints, then the server is properly installed.

To start the server, enter the command **ds20d start**. To stop the server, use **ds20d stop**. To have the server start automatically when you boot your system, you can add these commands to your /etc/rc directory structure, or the /etc/initab file, sometime after TCP/IP has started. The exact structure varies between operating systems.

If this is the first time dSERVE has been installed, then it is operating in demonstration mode. See the Activation chapter for information about activating dSERVE in live mode.

Windows CD or Download

- For a CD install, locate and run setup.exe in the dserve2/win directory on the CD. For a download, just run the download .exe file. This will guide you through a typical Windows setup. When complete, the Start/Programs menu will have a dSERVE 2.0 Server menu.

- Run dSERVE Setup from the dSERVE menu. This will prompt you for the location of a Visual PRO/5 or ProvideX executable, and store that information so the server can start.

The server can be installed as a service on Windows NT, 2000, or XP, using the Install as Service menu option. Then it will start automatically when the system is booted. It can also be started manually from the dSERVE Status Monitor.

The status monitor provides a view of the active log file, as well as manual start and top options.

As with Unix, if this is the first time dSERVE has been installed, then it is operating in demonstration mode. See the Activation chapter for information about activating dSERVE in live mode.

Activation

Upon installation, the server will run in Demo Mode. In this mode, the server will randomly return a repeating string "*Demo" in place of text values. To activate the product, you need an activation key for the PRO/5 or ProvideX serial number that you installed dSERVE with. Contact your reseller or SDSI to purchase an activation key. Activation keys enable a certain number of client connections at one time. For example, a 5-connection license would enable five concurrent tasks to be connected to the server at one time.

On Unix, use the command **ds20d –act**, which will display the serial number and prompt for an activation key. Entry of a valid key will activate dSERVE. Stop and restart the server to enable live operation. If the runtime engine is licensed for a single user, you will probably have to stop the server before attempting to activate.

On Windows, use the dSERVE Activation option from the Start/Programs/dSERVE Server menu. To stop and re-start the server, use the Windows Control Panel Services applet if dSERVE has been installed as a service, or the status monitor if it is running as an application.

Bundled Installations

For users of BBx4 or PRO/5 up to revision 2.0, a special version of dSERVE is available that includes a run-time PRO/5 with socket support. This simplifies the installation of PRO/5 and the Basis License Manager, and allows dSERVE to operate with a stand-alone copy of PRO/5 that will not interfere with the regular applications running under older versions of PRO/5 or BBx4 (or even earlier releases). You can tell if you have installed a bundled version by the presence of a directory called "/usr/lib/sdsi/rt". The rt directory contains the minimal set of PRO/5 and BLM executables required, along with a license file, called "pro5.lic". Initially, the pro5.lic file will contain a demo license for PRO/5.

If you choose to purchase a bundled version of dSERVE, you will need to license the run-time before you can activate dSERVE. When you receive your activation key from SDSI, you will also receive a serial number and authorization code in order to obtain a live license file from BASIS International. Instructions are provided with this information on how to obtain your license file from SDSI via email.

The only licensing method supported by the bundled dSERVE installation is electronic, where you receive a license file by email, and you replace the pro5.lic file noted above with the live license file.

Note that the number of users required in your run-time PRO/5 license is not the same as the number of connected users. You need one PRO/5 user per port the server listens on. If you only setup one dSERVE server, so a single port is listened to for connections, you only need a 1-user PRO/5 run-time.

For Windows users, the bundled version installs Visual PRO/5 and the license manager in a directory called c:\sdsi\rt. The license file, like on Unix, is called pro5.lic, and in addition, a pointer license file called basis.lic is also created. The Start menu provides an option to request a permanent license, which will prompt you for your assigned serial number and authorization code, and submit a request automatically or produce a request file that can be emailed. More complete instructions are provided with your activation key.

Server Configuration

Startup Configuration

dSERVE stores its settings in the dserve.ini file in the dSERVE directory. You can edit this file using any text editor to make changes to its configuration, then manually stop and restart the server for the changes to take effect. Here are the startup elements you can configure, found in the [defaults] section:

```
[defaults]
port=8227
logfile=dserve.log
logdetail=0
timeout=3600
dict=demodict.ini
gzip=1
```

port specifies the port the server will listen on, and to which clients must connect. This defaults to 8227, but you can define any port that is not already assigned to another service. On Unix, ports below 1025 are only available to servers running as root.

logfile specifies a log file name. If this is blank, no event logging will take place. Otherwise, this file will be created and log records written to it as events take place.

logdetail must be set to 0 or 1, to indicate the logging level. If detail is 0, then connections and significant errors are logged. If detail is set to 1, then transaction detail is also logged. This, of course, has no effect if there is no log file specified.

timeout specifies the number of seconds that a processing task will wait for any activity before closing down. If you set this to 0, then the processing task will wait forever, and only a terminated connection will close it down.

dict defines the name of the extension dictionary, if any. The extension dictionary allows you to specify calculated fields, override dictionary field characteristics, and normalize files with multiple record types. See the Data Dictionary chapter for more information.

gzip is a flag that indicates the server has gzip installed and can use it to support the dsGzip function. This feature is useful when using the dSERVE export functions, in order to transfer smaller data file sback to the client for local processing. The value can be 0, indicating gzip is not available, or 1, indicating that it is. If your server does not have gzip, it is free software that can be obtained from the gzip.org website. The Windows version of dSERVE includes gzip.exe as part of its distribution.

Server Security

dSERVE provides two levels of security: address filtering and file location filtering. Each is controlled by a line in the dserve.ini file, in the [security] section.

allow defines address filtering by specifying the IP addresses allowed to connect to dSERVE.

This line consists of one or more IP addresses or IP address wildcards, delimited by semicolons. In addition to IP addresses, you can name a file or use the word "any". If a file is named, that file is read for a list if IP addresses delimited by white space (spaces, tabs, carriage returns). If the word "any" is encountered, then all IP addresses may connect to the server.

The following example would allow any IP address starting with 100.100.100., plus the specific address 222.223.224.225, to interact with the server. All other addresses are ignored.

allow=100.100.100.*;222.223.224.225

prefixmode controls how dSERVE looks for files or dictionaries. If prefixmode=0, then files and dictionaries can be opened anywhere on the server's file system. If prefixmode=1, then the server Business Basic prefix is always used, and absolute pathnames for files or dictionaries are suppressed. To define the prefix in a PRO/5 environment, use a PREFIX line in the config.bbx file in the dSERVE directory. In ProvideX, you can edit the program "start.pv" to establish the prefix as soon as a connection is initiated.

Note that if a file is opened via a dictionary, the dictionary will be searched via the prefix, but any pathnames referenced by the dictionary will not be adjusted, and may be absolute or relative paths.

procports allows you to specify a list of ports that dSERVE will use when connections are established and alternate processes are launched. Normally, alternate process connections are assigned a random port designated by the operating system. However, if you need to specify the ports that dSERVE can use, set procports to a comma-separated list of port numbers. Each port must be an integer from 1024 through 65535 (some socket implementations may have an upper limit of 32767). When a connection is established, dSERVE will try these ports in sequence until it successfully opens an available port. Note that you should have at least as many procports as licensed users. For example, procports=9000,9001,9002,20001,20002 will only establish alternate process connections on the five listed ports.

write turns on or off write capability for connected clients. If write=0, then write and remove operations will return an error 13. If write=1, then write and remove operations will be attempted, but will be subject to the file permissions granted the user that the dSERVE server operates under.

erase turns on or off erase capability for connected clients. If erase=0, then the dsErase command returns an error. If erase=1, then the dsErase command is supported (and will honor the prefixmode setting).

Further file-level or connection-level security can be obtained by establishing global strings either at the server level, in start.bb|pv, or from the client program using the Set Global String function. These global strings can be used to demote a write-enabled connection to a read-only connection. To disable write access to all files for a connection, set the global string "$ro-*" to any non-null value. To disable write access to a specific file for a connection, set the global string "$ro-*filename*" to any non-null value. dSERVE tests the *filename* setting first, and then tests "$ro-*" if the string $ro-*filename* is not defined. The following table outlines the tests performed.

Global String	**Value**	**Write Access?**
$ro-*filename*	not defined	$ro-* tested
	null	allowed
	non-null	not allowed
$ro-*	not defined	allowed
	null	allowed
	non-null	not allowed

The tests are performed at the time files are opened via the Open File or Open File via Dictionary functions are performed. When opening the file via the dictionary, it is the dictionary alias name that is tested rather than the physical file pathname. If prefixmode is set, then only the base filename is tested; however, when opening files with full paths with prefixmode off, be aware of the 32-byte limit on global string names. You may find it necessary to open files with their base name, using a search prefix setting to allow the run-time engine to locate the file.

dServe disallows the use of the following verbs and functions in expressions issued by client access functions:

BBx: SCALL()
ProvideX: SYS() and INVOKE

dServe also will not open files starting with the >, <, or | characters.

Date Configuration

As date fields are not intrinsic data types in Business Basic, dSERVE provides a method to help the server interpret fields as dates and correctly parse the internal data to produce a human-readable date. Internally, date fields are first converted from a native format into a julian number, then are returned with a DATE() or DTE() function. In addition, international formats and conventions are specified here. In the absence of specific calculations or string template user attributes, dSERVE will use the date configuration details in the [dates] section to return date fields properly. Here is a sample:

```
[dates]
format=%Mz/%Dz/%Yz
suffix=_date
style=jul
century=2000
cdate=mdy
```

format is a date format string used when returning date fields. The parameters for this string are specified in the PRO/5 DATE() function and the ProvideX DTE() function.

suffix defines a field name suffix that dSERVE will look for in dictionary definitions. Any field that ends with this string will be treated as a date, in the style specified by the **style** value.

style defines the default date type. Currently supported styles include “aon” for AddonSoftware, “ssi” for SSI’s FACTS, jul for a julian integer, or combinations of yyyy, yy, mm, dd for 4-digit years, 2-digit years, 2-digit months, and 2-digit days.

century is used to define the century assumed when a 2-digit year is found while converting a date to a julian value.

cdate is set to “mdy” or “dmy” to indicate the parsing order that that cdate() function uses when looking at its text argument. The cdate() function can be used in expressions, such as filters, to convert a human readable date to a julian value.

Data Dictionaries and Dictionary Extensions

dSERVE supports both the BASIS and ProvideX data dictionary. In the case of ProvideX, dSERVE can interpret both the providex.ddf keyed dictionary and the .ini dictionary format. The only exception is that dSERVE does not support the ProvideX "flattened file" dictionary format, which uses the *RECTYPE definition.

To use a dictionary, simply open files using the Open Via Dictionary function. This will open the file you specify, and also create an associated string template so subsequent field-style operations can use field names. In addition, you can list the files in the dictionary using the List Dictionary function. These functions use a dictionary name as a parameter.

The dictionary name for PRO/5 is a pathname to a .tpm file, which itself contains a line "DICTIONARY=*directory*". The *directory* is the directory path that contains the BASIS data dictionary files FILE.1, FIELD.1, etc. The dictionary name for ProvideX is a pathname either to providex.ddf or a dictionary .ini file. In both cases, if the security prefixmode is set to 1, then the pathname must be found via the prefix. Note that in the case of a ProvideX .ini dictionary, dSERVE supports the MUSTBE parameter for automatic filtering of data.

Note that if the security setting for prefixmode is 1 in the dserve.ini file, the ONLY way to use a providex.ddf dictionary is to modify start.pv to correctly set the search prefix to find your providex.ddf file rather than the demo providex.ddf file.

In addition to the field names identified in the dictionary, you can specify extensions and override characteristics in the extension dictionary. The extension dictionary is named in the [defaults] section of the dSERVE.ini file, with the "dict=*pathname*" line. The *pathname* text file is read for a section matching a dictionary file or alias name each time a file is opened via the dictionary. Sections may contain three types of lines:

*autofilter=*filter expression* is used to normalize the data in a file, by automatically selecting certain types of records. For example, if a file contains header and detail lines, where the header line is "000" and details lines are >"000", then two dictionary files would be created for the same physical file. The header record dictionary would have a dictionary extension *autofilter=line_number="000", while the detail record dictionary would have *autofilter=line_number>"000".

onopen=@program*(*arg1, arg2, …,argn*) names a program and arguments to CALL as soon as the file is opened via the dictionary. You can use this feature to establish parameters for later use in calculations. For example, *onopen=@params("customers") would cause dSERVE to execute "CALL "params","customers" when the file is opened. This program could define STBL or GBL values for use in calculation expressions, or could be used for other purposes, such as logging file opens.

name=tpldef,expression defines a calculated field and adds it to the string template returned by the Open File Via Dictionary function. The *name* can be any valid variable name, starting with a letter, followed by up to 31 letters, digits, or underscores. *tpldef* is a string template fragment that defines the characteristics of the data. For example, the fragment "c(10*)" defines a 10-byte, delimited character string. "n(8):prec=2:" defines an 8-byte, fixed length number with the user attribute "prec" set to 2. *expression* is a Business Basic expression that returns valid string or numeric data, depending on the date type defined in *tpldef*. Optionally, the expression may also specify a CALLed program and arguments, using the syntax *@program*(*arg1, arg2,…,argn*). One of the arguments should be val$ or val, to return string or numeric data, respectively, as the *tpldef* requires.

If *name* is already defined in the dictionary, then this specification overrides the characteristics of the field. Be careful not to accidentally redefine the file layout; for example, you must keep fixed length fields as the same length, and delimited fields must remain delimited with the same terminator.

If no *expression* is present, then only the template changes, and no expression is calculated for the field.

All expressions are calculated for each field-style read. If there are any dependencies (expressions that rely on other calculated fields), be sure the fields are named in top-down order so any calculated fields required in another calculation are listed first.

This example specifies a numeric calculation field: ytd_profit=n(10*):prec=2:,ytd_sales-ytd_cost.

This example uses a custom mask for a field: id:c(5*),str(id:stbl(“custmask”)).

This example calls a program “format” to return a new value (note the use of val$ to return a value): id:c(5*),@format(id,”custmask”,val$).

BASIS SQL Engine Support

If properly configured, dSERVE can provide access to data via the BASIS SQL engine, which adds support for table joins via the opening of SQL select statements, and if writing is enabled, support is also added for any SQL statement, including update and insert commands.

To configure the SQL engine, first you need to ensure the correct version of PRO/5 is installed. There are two unbranded pro5 engines: pro5b and pro5s. When you first install pro5, you choose which to install based on whether or not you want SQL support. If your pro5 is the same size as pro5b, you do not have SQL support and you would need to re-install and select the engine with SQL support (pro5s).

Next, add a SQL=*ini file* command into the config.bbx file in the dSERVE directory. The file this points to must be in a specific format, with a case-sensitive header line of [BASIS Data Sources], followed by lines that themselves point to other sections that contain CONFIG=*tpm file* lines. Here is a simple example:

In dSERVE's config.bbx:

```
SQL=dssql.ini
```

In dssql.ini:

```
[BASIS Data Sources]
Demo Data

[Demo Data]
CONFIG=dserve.tpm
```

This will enable the "Demo Data" data source to be used with the dictionary supplied in the DICTIONARY entry of dserve.tpm. You can then use the "sql:" prefix in two dSERVE commands to instruct the server to use SQL rather than native file I/O methods:

dsOpenDict ("sql: Demo Data", "select id,name,invoice_no,invoice_date from customers, invoices where customers.id = invoices.cust_id", chan%, pathname$, tmpl$)

Now there will be access to dictionary query functions on chan%, such as dsListFlds, and to the forward read functions, dsReadFldNext and dsReadRecNext.

In addition, you can retrieve a list of tables in the database with this function:

dsListDict ("sql: Demo Data",files$,descs$)

Note that most SQL errors are reported as a generic error 77. In order to find the actual error message associated with this error, you must have detail logging turned on, and look in the log file for the sqlerr= line following the log entry for the SQL command.

Windows Client DLL Installation

The Windows client software provided with dSERVE is designed for use in Windows 98 or higher systems, and development environments that can work with ActiveX in-process DLLs.

If you are installing from CD, then locate the dSERVE/client directory on the CD, and run the setup.exe program. Follow the simple prompts to install dSERVE on the local hard disk.

If you installed dSERVE from a download, then a directory was created under the dSERVE directory, called client. In the client directory is a Windows setup.exe program and associated files. If your client PCs have access to this directory, they can install directly from there. Otherwise, you can copy all the files in the client directory to a diskette and install from that.

The client installation will install and register both the ActiveX DLL and the native dserve2.dll. The object dSERVE will then be available for use by development environments such as VB6, VB.NET, C#, and others. A second object, dSERVEVBS, is available for VBScript operation. All ByRef method arguments in dServeVBS are of Variant type, and any lists are returned as delimited strings (Chr(0) delimiter) rather than arrays, due to problems of passing non-Variants and arrays as ByRef arguments.

Perl and PHP Clients

The Perl client supplied with dSERVE is a Perl module called dServe.pm. It can be copied to any of your Perl library directories, or used from the local directory. A Perl script can simply specify **use dServe;** and all the dSERVE functions will be exported and are available as commands.

The PHP client is the file dsphp.inc, which can be used in a PHP script with the **require *path*/dsphp.inc;** command, or just **require dsphp.inc** if the file is copied to one of your include_files paths. This is a PHP class file that provides for class instance creation using the command **$d=new dserve;**.

The function names in the Perl and PHP clients match those of the Windows ActiveX DLL client.

Error Codes

If an error occurs while the dSERVE server is processing a request, an error response is sent indicating the error number. Error numbers can range from 0 to 999, and indicate the type of problem that occurred. In some cases, the errors will be expected and can be ignored, or can indicate a condition the client software can base its actions on.

Errors are returned somewhat differently depending on the client. The ActiveX DLL functions all return an error number directly, with a –1 response indicating success. The PHP and Perl client functions all return True on success, False on error, and the variables $dserr and $dserrmsg contain the last error number and description.

If an error occurs on the server, and detailed logging is turned on, then the log file will contain complete error information, including the internal error number, O/S error code, and the line number in the processing program. This information can be helpful for support purposes.

The following table describes some of the possible errors.

Error Number	Description
Errors reported by the server	
0	A file or record is locked by another process and is unavailable. This will typically occur when opening a file. Since dSERVE turns on advisory locking on the server, you should not get an error 0 when reading a record. Applications often retry a set number of times, then allow the user to continue to retry or exit.
1	A record is too long for the file definition. This normally occurs on a write to a file, but can occur on a field style read if the number of variables exceeds the number available in the record.
2	End of file error. On a write operation, this means the file is limited to a certain number of records, and it is full. On a next or previous sequential read operation, the end or beginning of the file was reached.
3	Disk read or write error. This generally indicates a hardware problem.
4	Disk not ready error. This can occur if removable media is not available, or a drive is down. On Windows-based servers, this frequently occurs because an unavailable drive letter has not been disabled. dSERVE attempts to disable unavailable drives on startup. In a PRO/5 or Visual PRO/5 environment, you can edit dSERVE's config.bbx file to add **dsksyn *drive*:** commands to disable drives before startup.
7	Key corruption in the data file. This indicates a problem that should be corrected by your application programmers.
10	Invalid file name.
11	Missing key. This is often an expected error, since it is common to position a key pointer at some point in a file without regard to the existence of a key. It can also indicate a referential integrity problem in your files, such as an invoice record for a deleted customer.
12	File not found. This can occur in a dsOpen or dsOpenDict function, and indicates the file name isn't on the server. It can also mean the server is running in prefix mode, and the path being searched for isn't in the prefix search list.
13	Improper access to a file. Often this is due to permissions.
14	Improper file channel usage. This may indicate an attempt to open an unavailable device, which likely means the filename is invalid and wouldn't be a

Error Number	Description
	file type supported by dSERVE.
15	Disk full error.
16	Out of resources. This can indicate PRO/5 or Visual PRO/5 resources, or operating system resources. It most often occurs when trying to open too many files. In a PRO/5 or Visual PRO/5 environment, resource allocations can be edited in dSERVE's config.bbx file. No matter what the file allocations are, dSERVE will report this error if more than 255 files are opened.
17	Referencing an invalid disk.
18	Permissions error.
20	Syntax error. This may occur when setting an invalid filter with the dsFilter function, an invalid template with the dsTmpl function, or in an invalid ***=expression*** value in a field style read.
21	Invalid line reference. This may indicate a user defined function in the file functions.txt contains a goto or err= branch to an invalid statement.
24	Multi-line function error. This may indicate a syntax problem or illegal verb in a function defined in the functions.txt file.
25	Missing or invalid user defined function reference.
26	String/numeric mismatch. This indicates a syntax error in a function or expression, where a string or a number is required, but the opposite is supplied.
27	A return, retry, or looping error, generally caused by an unexpected wend, until, swend, or return statement.
28	A for/next loop error, where a next variable doesn't match a preceding for variable.
29	A mnemonic error, generally caused by using single quotes rather than double quotes in a string literal. In Business Basic, a single quote indicates a mnemonic value, such as 'FF' or 'LF' (form feed and line feed, respectively).
30	Corrupt or invalid program. This can occur trying to execute a PRO/5 program that has been encrypted using the PRO/5 Revision 2.2 SAVEP algorithm with a pre-2.2 PRO/5.
31	Out of workspace memory. PRO/5 and Visual PRO/5 work with a fixed memory allocation, which is indicated by the "-m*pages*" command line argument. If your application exceeds this value, this error occurs. You can increase the –m parameter in the startup script or shortcut.
32	Stack overflow, caused by too many nested functions or parentheses.
33	System memory overflow.
34	System buffer overflow.
36	Call/enter argument mismatch. This indicates a programming problem, most likely in a user-defined program "startup", which is called with a single numeric argument when the server is started.
38	Invalid verb in a called program.
40	Number overflow error, usually caused by a divide by zero.
41	Invalid integer, such as a negative length, or non-integer value in a function that requires an integer.
42	Array subscript bounds error.
43	A mask overflow. Normally not reported by dSERVE, as mask overflows are set to return a non-masked version of the data.
46	Invalid string size. The most common cause of this error is a key that is longer than the defined key length of the file.

Error Number	Description
47	Substring error, where the position and/or length of a substring reference (*stringvar$*(*start,length*)) exceeds the bounds of the string. To avoid this, use the left(*stringvar$,length*), mid(*stringvar$,start,length*), or right(*stringvar$,length*) built in functions.
49	Global string reference error, when using a STBL or GBL function without an error branch.
60	A general I/O error occurred that did not result in another, more specific error code.
70, 71, 72	These errors indicate network problems.
997	Client address not granted server access.
998	License count exceeded during connection request.
999	Unable to start alternate server process during a connection request. The server might be out of resources.
Errors reported by Windows clients	
1024	Device timeout error.
1057	Device I/O error. The server may be down, or TCP/IP communications may be down.
1068	Device unavailable error.
1070	Device permissions error.

Client Function Declarations and Usage

We have documented the server interface, so that any TCP/IP-aware language could be used to develop client applications. However, dSERVE is supplied with three clients for various platforms:

- A Windows In-Process ActiveX DLL, sdServe.dll, which contains SDSI dSERVE objects
- A Perl module, dServe.pm
- A PHP class file, dsphp.inc

Each of these clients provides the functionality to open and close a connection, and process all functions supported by the dSERVE server. The names of the functions are the same, and the arguments differ only in that the ActiveX and PHP classes contain the connection handle when instantiated, so its function arguments do not include the connection handle, whereas the Perl client functions do include the connection handle. Any other differences are noted, if applicable, in the function's documentation.

The Perl and PHP functions all return True on success, and False if an error occurs. If an error occurs, then the client program can check the dserr and dserrmsg class variables for the error code and error description, respectively.

The ActiveX DLL functions return –1 on success, and an error number 0 or above if an error occurs. The DLL also raises error events, and has two properties, LastErrorNumber and LastErrorMessage, that contain the error number and message, respectively. All functions also have associated completion events, if the class is instantiated "With Events", and if the programmer wishes to use events in his or her programming style. The DLL installation includes a help file that documents all the methods (functions), properties, and events.

For purposes of consistent documentation, all the following function references use the nomenclature of BASIC for variable types based on their suffix:

- $ for strings
- & for long integers
- % for short integers

Open a Connection

DLL: dsConnect(hostid$, portnum&,maxtime&)
PHP: dsConnect(hostid$, portnum&,maxtime&)
Perl: dsConnect (hostid$, portnum&, maxtime&, tcphandle&)

The first step in any program is to establish a connection with the dSERVE server. Your client must have a live TCP/IP connection to the machine running the dSERVE server, or the connection will fail. In addition, the client and server must be able to communicate over the socket identified in portnum&, and to open a secondary connection once the initial connection is established.

hostid$	This is the IP address or hostname of the server machine. The client machine must have access to the server via TCP/IP.
portnum&	This is the port (or socket) number the server listens on. The server defaults to 8227, but it can be any number the installer selects in the

	server's configuration or startup.
tcphandle&	Returned as a connection handle that is used for all other functions. Once this is set, do not change it!
maxtime&	This is the maximum time in milliseconds that the function will wait for a connection. If no connection is established in this amount of time, then the function returns an error. If you set this to 0, no timeout is used, and system defaults will apply.

Note the following connection related errors:

997 means the server filtered the client's IP address and did not accept it, based upon its "allow" setting in the configuration .ini file.

998 means the number of active connections is currently at the license limit.

999 means there was a problem launching the secondary process to handle this connection, possibly meaning there is a resource limit being reached on the server.

Close a Connection

DLL: dsDisconnect
PHP: dsDisconnect
Perl: dsDisconnect (tcphandle&)

When your activities on the connection are complete, you should disconnect in order to release the server process handling your connection.

tcphandle&	This is the handle returned by the dsConnect function.

Open a File

DLL: dsOpen (fl$,chan%)
PHP: dsOpen (fl$,chan%)
Perl: dsOpen (tcphandle&,fl$,chan%)

Before you can access files, they must be opened and assigned a file channel. You can open files directly, using the dsOpen function, or you can open files indirectly via the data dictionary with the dsOpenDict function. Either way, you are provided a file channel that is used to read data in the file.

The server can be set to only open files found in the directory prefix search list. If the file is outside of that search list, this function will return an error. In addition, the user ID under which the server runs must have read access to open and read the file.

Once a file is opened, you can access full records or individual fields by number. Optionally, you can associate a string template with the channel using the dsTmpl function, and then use field names to access the data.

tcphandle&	The connection handle assigned by the dsConnect function.
fl$	The file name or pathname to open.
chan%	Returned as the file channel number if the file is successfully opened.

List Dictionary

DLL: dsListDict (dict$,wildcard$,array files$)
VBS: dsListDict (dict$,wildcard$, zfiles$,zdescriptions$)
PHP: dsListDict (dict$,wildcard$,array files$)
Perl: dsListDict (tcphandle&,dict$,wildcard$,array files$)

This function returns a list of files and associated descriptions from a dictionary.

tcphandle&	The connection handle assigned by the dsConnect function.
dict$	The pathname to the dictionary. On BBx, this must be a TPM file that contains a line DICTIONARY=*pathname*, which points to the directory where the BASIS data dictionary files are located. On ProvideX, this must be either the providex.ddf file or a dictionary .ini file. If prefixmode is set, then the path must be found via the directory search prefix. If the BASIS SQL engine is configured, you can specify dict$ as "sql: *data source*" to retrieve a list of tables in the *data source* name.
wildcard$	Can be set to return just a subset of file names from the dictionary. Use * to represent any set of characters, ? to represent a single character. AP*, for example, would return all files starting with "AP". *hist* would return all files containing the text "hist". Wildcard text is not case sensitive.
array files$	Returns a 2-dimensional array of file names and descriptions, such that for each file, the second dimension contains the name and description. files$(0,0)=first name, files$(0,1)=first description. In Perl and PHP, $files[0][0]=first name, $files[0][1]=first description. Note that the ProvideX dictionary is does not return descriptions, so those elements will be blank.
zfiles$	A Chr(0) delimited string, easily converted to an array with the VBScript Split function.
zdescriptions$	A Chr(0) delimited string, easily converted to an array with the VBScript Split function.

Open a File via the Data Dictionary

DLL: dsOpenDict (dict$,fl$,chan%,pathname$,tmpl$)
PHP: dsOpenDict (dict$,fl$,chan%,pathname$,tmpl$)
Perl: dsOpenDict (tcphandle&,dict$,fl$,chan%,pathname$,tmpl$)

This function provides a convenient way of opening a file defined in the data dictionary. The server resolves the dictionary name to the physical path, opens the file, and creates a string template for the file.

See the chapter about dictionary extensions for information about adding or replacing fields in the template returned by this function. For more information about string template formats and user attributes, see **Associate a String Template with a File**.

tcphandle&	The connection handle assigned by the dsConnect function.
dict$	The pathname of the dictionary definition file. In a BBx environment, this should be a TPM file, often called config.tpm. It stores values for DICTIONARY and other string constants that the server uses to locate the data dictionary files. In ProvideX, this should point to either a providex.ddf file or a dictionary .ini file. If prefixmode is set, then the file must be found through the prefix. If the Basis SQL engine is configured, then you can specify a data source name prefixed with "sql:", such as "sql: Demo Data".
fl$	This is the file name as it is stored in the dictionary. This may or may not match the physical disk file name that stores the data. If the Basis SQL engine is configured, and dict$ is a data source name prefixed with "sql:", then this can be a valid SQL command, such as "select * from customers".
chan%	Returns a file channel number, which is used for any operations on the file.
pathname$	Returns the physical path opened.
tmpl$	Returns a string template associated with the channel. There is no need to perform a dsTmpl function.

List File Fields

DLL: dsListFlds (chan%, array flds$)
VBS: dsListFlds (chan%,zflds$)
PHP: dsListFlds (chan%,array flds$)
Perl: dsListFlds (tcphandle&,chan%,array flds$)

This function returns an alphabetical list of field names from a channel that has a string template associated with it. Any file opened via the data dictionary will have such a template. The list is returned in alphabetical order. Fields with a user-attribute of "hide=1" will not be returned, allowing a developer to suppress unwanted fields from the list.

This function returns an error if used on a channel that has no string template associated with it.

tcphandle&	The connection handle assigned by the dsConnect function.
chan%	The channel number returned from a dsOpen or dsOpenDict function.
array flds$	Returns an array of field names.
zflds$	A Chr(0) delimited string, easily converted to an array with the VBScript Split function.

List a File's Indexes

DLL: dsGetIndexes (chan%,array indexes$)
VBS: dsGetIndexes (chan%,zindexes$)
PHP: dsGetIndexes (chan%,array indexes$)
Perl: dsGetIndexes (tcphandle&,chan%,array indexes$)

This function uses a file's template and file key structure information from the FID and FIN functions to return an index that shows descriptive names of each key in the file. Multi-keyed files have a primary key (element 0) and from 1 to 16 alternate keys (possibly more, depending on the Business Basic implementation).

tcphandle&	The connection handle assigned by the dsConnect function.
chan%	The channel number returned from a dsOpen or dsOpenDict function.
array indexes$	Returns an array of key descriptions, with element 0 being the primary key, and other elements matching their respective key chain numbers.
zindexes$	A Chr(0) delimited string, easily converted to an array with the VBScript Split function.

Close a File

DLL: dsClose (chan%)
PHP: dsClose (chan%)
Perl: dsClose (tcphandle&,chan%)

You can close a file using the channel number returned when it was opened. In most cases, you won't need to close files, as they are automatically closed when the connection is closed. However, if you open too many files at one time, the server may run out of resources. In such cases, you may need to close unneeded files using this function.

tcphandle&	The connection handle provided by the dsConnect function.
chan%	The file channel number provided by the dsOpen or dsOpenDict function.

Associate a String Template with a File

DLL: dsTmpl (chan%,tmpl$)
PHP: dsTmpl (chan%,tmpl$)
Perl: dsTmpl (tcphandle&,chan%,tmpl$)

String templates are often used as record descriptors in Business Basic. They allow a record to be parsed into named values, rather than forcing direct substring or field number references to data. When a file is opened via the data dictionary, using the dsOpenDict function, a string template is automatically associated with the file. However, any file channel can have a template associated with it by using this function, including one opened via dsOpenDict whose string template you wish to modify. Once a string template has been successfully associated with a file, the filter and field style read functions work with names defined in the template. If you do not have a template associated with a file channel, no filtering can be set for that channel, and all read operations must be performed by record or with field numbers.

A string template is a literal string that identifies field names, their data type and length, and user attributes. Each field is delimited with a comma, and its attributes are delimited with colons. The format of a string template is very concise, and missing (or extra) colons or commas result in errors.

Field names can have letters, digits, and underscores, and can be up to 32 bytes long. They must start with a letter. Field names are not case sensitive, and must be unique within the template.

A field can be represented as a 1-based array by appending [*count*] to the name.

Following the field is a colon and a type indicator. Type indicators are single letters that identify the type of data. Some indicators require a length, which is entered in parenthesis after the type letter. Others are fixed in length and don't accept a length value. Further, if the length is variable and the field ends in a field terminator, a * character after the length indicates the field data ends at a field separator, and a *=*val* indicates the field data ends at the field separator specified with the ASCII value *val*.

Valid field types, and their syntax, include:

Code	**Description**	**Syntax**
c	Fixed length string	c(*len*)
c	Variable length string with field terminator	c(*len**) or c(*len*=term*)
n	Fixed length decimal number	n(*len*)
n	Variable length decimal number	n(*len**) or n(*len*=term*)
I	Signed binary integer of *len* bytes	i(*len*)
u	Unsigned binary integer of *len* bytes	u(*len*)
f	8 byte IEEE floating point number	f
b	8 byte internal floating point number	b
d	8 byte BCD floating point number	d
x	4 byte local C float number	x
y	8 byte local C double float number	y

Following the type and length indicator are optional user-defined attributes. Attributes are given as *name=value* pairs, with each pair separated by one or more spaces. Colon characters must start and end the user attributes, so colons are invalid in attribute names or values. Spaces are also invalid, as they are interpreted as delimiters.

User attributes that have special meaning to dSERVE include:

caption	A caption that can be used as a column heading or field label. Remember that the value can't contain spaces, so you may wish to use underscores instead, and parse them in your application. Any spaces found in a dictionary caption are converted to underscores when a template is created automatically in the dsOpendict function. This value is returned by the dictionary parser, and may be returned to the client and used via the dsFattr function.
mask	A format mask for either strings or numbers. A format mask for a number will imply a precision value. The date returned by a field style read function will be formatted with this mask. Valid numeric mask characters include: #=space filled digit, 0=zero filled digit, ,=thousands separator position, .=decimal point position, -,(),CR=negative

	indicators. We aware that some masking formats may provide non-numeric data to a client application. For example, a negative number with a trailing minus sign is treated as a text field by Excel. Valid string mask characters include: X=any printable character, A=any alphabetic character converted to upper case, a=any alphabetic character, 0=any digit, Z=any digit or alphabetic character converted to upper case, and z=any digit or alphabetic character.
prec	A decimal precision to use in a default mask for a numeric field, if no mask is specified.
date	Specify a format and interpret the value as a date. The format of the date can include "aon" for AddonSoftware, "ssi" for SSI's FACTS, jul for a julian integer, or combinations of yyyy, yy, mm, dd for 4-digit years, 2-digit years, 2-digit months, and 2-digit days.

Here are some string template examples:

id:c(5),name:c(30),amount:n(10)
Three fixed length fields. ID is a 5-byte string, Name is a 30-byte string, and Amount is a 10 byte decimal number.
id:c(5*),name:c(30*),amount:n(10*):mask=##,###.00- prec=2:
The same as above, but each field is a variable length with a standard field separator indicating the end of the field. Additionally, the amount field contains two user-defined attributes, "mask" and "prec".
cust_num:c(6):mask=00-0000 caption=Customer_ID:, invoice_num:c(8):caption=Invoice_Number, invoice_date:i(3):date=jul:, amount:b:prec=2:
A 6-byte cust_num string field with a format mask and a caption, an 8-byte invoice_num field with a caption, 3-byte binary integer that stores a julian number to be returned as a human-readable date, and an 8-byte business math formatted amount field, returned with a default mask set to precision 2.

tcphandle&	The connection handle provided by the dsConnect function.
chan%	The file channel provided by the dsOpen or dsOpenDict function.
tmpl$	The string template definition text.

Set a Template Field Attribute

DLL: dsSetAttr (chan%, field$, attr$, value$, tmpl$)
PHP: dsSetAttr (chan%, field$, attr$, value$, tmpl$)
Perl: dsSetAttr (tcphandle&, chan%, field$, attr$, value$, tmpl$)

This function can be used to add or change a user attribute for a channel's template. Use it, for example, to override the mask or precision of a field after a file has been opened via a dictionary that doesn't provide the desired attribute setting.

tcphandle&	The connection handle provided by the dsConnect function.
chan%	The file channel provided by the dsOpen or dsOpenDict function.

field$	The field name whose user attribute will be updated. The field must be present in the template associated with the channel.
attr$	The name of the attribute, such as mask or prec. If the attribute doesn't exist, it will be added.
value$	The value to set for the attribute. This may be null, in which case the attribute will be created as a Boolean name. If it contains a value, the attribute will be created in the format name=value. Values cannot contain spaces or colons.
tmpl$	Returned as the new string template definition for the channel. This is simply for reference purposes. The channel's template is updated by this function, so there is no need to perform a dsTmpl function.

Filter Records From Sequential Reads

DLL: dsSetFilter (chan%,filter$)
PHP: dsSetFilter (chan%,filter$)
Perl: dsSetFilter (tcphandle&,chan%,filter$)

Sets a filter on a file channel so any sequential (next or previous) reading of the file returns just those records that match the filtering criteria. It is generally preferable to filter data on the server rather than in the client application, as that will reduce transmission overhead.

A filter is a logical expression clause that would apply to an IF *logical expression* THEN *action* statement. Expression fragments include field names or literal values, optionally manipulated with Business Basic or user-defined functions, logical operators, and comparison fields or literal values. Multiple expression fragments can be chained with **and** or **or**, and parentheses can be used to logically group expressions together.

In order for a filter to be set on a channel, a template must first be associated with the channel. Then any field names found in the template can be included in the filter. In addition to field names, you can refer to the whole record with the variable name **rec$** and the key with the variable **ky$**. Numeric fields must be compared with numeric values, string fields with string values.

Dates require special consideration. If a field is determined to be a date (see the section explaining Date Interpretation), then it is converted to a julian integer. This must be compared with another julian value in a filter expression. This can be another date field, of course, as both will be converted to julian format. If you want to compare a date field to a literal date, you should use the cdate() function, which converts a string argument to a julian number. The cdate() function uses the date entry format defined by the server's configuration or defined in the dsSetDate function. It can parse dates as 4, 6, or 8 digit strings ("12302001") or as delimited strings ("12/30/01"). You can also use the Business Basic JUL() function, which returns a julian number for a given year, month, and day argument, like JUL(2001,3,15). Note that JUL(0,0,0) returns the current date.

Filter examples, assuming a string template of:
id:c(5*),name:c(5),date:n(7):date=jul:,amount:b:prec=2:

Negative amounts:
amount < 0
Amounts at least 10,000, and dates before Jan 1, 2000:

amount >= 10000 and date < cdate(“1/1/2000”)
Amounts at least 10,000, and dates more than 30 days old (uses the internal jul() function):
amount >= 10000 and date +30 < jul(0,0,0)
A range of ID codes, or amounts less than 0
(id>=”10000” and id <=”19999”) or amount < 0
Names containing “inc.”, in any case, anywhere in the name (uses the internal pos() and cvs() functions):
pos(“inc.”=cvs(name,8))>0

tcphandle&	The connection handle provided by the dsConnect function.
chan%	The file channel provided by the dsOpen or dsOpenDict function.
filter$	The filter string.

Read Binary Data

DLL: dsReadBinary (chan%,offset&,bytes%,block$)
PHP: dsReadBinary (chan%,offset&,bytes%,block$)
Perl: dsReadBinary (tcphandle&,chan%,offset&,bytes%,block$)

If a text file (rather than a Business Basic keyed file) is opened on a channel, then blocks can be read as binary data, given an offset and number of bytes. The client is responsible for parsing the data, including line terminators in text files.

tcphandle&	The connection handle provided by the dsConnect function.
chan%	The file channel provided by the dsOpen or dsOpenDict function.
offset&	The byte offset, 0-based, to begin the block read. 0 is the first byte, 1023 is the 1024th byte, etc. If the value is past the end of the file, an error 2 is returned.
bytes%	The number of bytes to read. Note that if the end of file is reached, no error is returned, but the length of block$ will not match bytes%.
block$	Returns the block of data read from the file.

A common algorithm for reading a whole file is as follows (in VB syntax):

```
offset&=0
size%=1024
contents$=””
do while dsReadBinary(h&,chan%,offset&,size%,block$)=-1
      contents$=contents$+block$
      offset&=offset&+len(block$)
loop
```

Read a Full Record by Key

DLL: dsReadRec (chan%,key$,knum%,record$)
PHP: dsReadRec (chan%,key$,knum%,record$)

Perl: dsReadRec (tcphandle&,chan%,key$,knum%,record$)

This function will return a full record, if the specified key value is in the file. If not, an error 11 is returned. Note that often an error is expected, such as when doing a positioning read to some logical point within the file. You can also use this function to set the alternate key value for subsequent sequential reads of the file.

If non-ASCII storage methods are used, it is easier to use string templates and field style reads. Also, records can be very long and contain padded data, so in many circumstances performance is improved using field style reads, as less data will be transmitted from the server.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
key$	The key value used to read the file.
knum%	The alternate key number (0=primary key, 1-16 are possible alternate keys).
record$	Returns the full record.

Extract a Full Record by Key

DLL: dsExtractRec (chan%,key$,knum%,record$)
PHP: dsExtractRec (chan%,key$,knum%,record$)
Perl: dsExtractRec (tcphandle&,chan%,key$,knum%,record$)

This function is identical to the Read Record function, above, except that the EXTRACT function locks the record from EXTRACTs by other users. Use this function in preparation for a write record operation or write field operation.

Read Next Full Record

DLL: dsReadRecNext (chan%,ky$,record$)
PHP: dsReadRecNext (chan%,ky$,record$)
Perl: dsReadRecNext (tcphandle&,chan%,ky$,record$)

This function will return the next sequential key and record from the file channel. If a filter has been set for the channel, then the file is read until a record matching the filter criteria is found. When the end of the file is reached, the function returns 2.

If non-ASCII storage methods are used, it easier to use string templates and field style reads. Also, records can be very long and contain padded data, so in many circumstances performance is improved using field style reads, as less data will be transmitted from the server.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
key$	Returns key value of the record. If the file is being read by an alternate key, as specified in a prior dsReadRec or dsReadFld function, then the value returned is that of the alternate key.
record$	Returns the full record.

Read Previous Full Record

DLL: dsReadRecPrev (chan%,ky$,record$)
PHP: dsReadRecPrev (chan%,ky$,record$)
Perl: dsReadRecPrev (tcphandle&,chan%,ky$,record$)

This function will return the prior sequential key and record from the file channel. If a filter has been set for the channel, then the file is read until a record matching the filter criteria is found. When the beginning of the file is reached, the function returns 2.

If non-ASCII storage methods are used, it easier to use string templates and field style reads. Also, records can be very long and contain padded data, so in many circumstances performance is improved using field style reads, as less data will be transmitted from the server.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
key$	Returns key value of the record. If the file is being read by an alternate key, as specified in a prior dsReadRec or dsReadFld function, then the value returned is that of the alternate key.
record$	Returns the full record.

Read Record Fields by Key

DLL: dsReadFld (chan%,ky$,knum%,flds$,array values$)
VBS: dsReadFld (chan%,ky$,knum%,flds$,zvalues$)
PHP: dsReadFld (chan%,ky$,knum%,flds$,array values$)
Perl: dsReadFld (tcphandle&,chan%,ky$,knum%,flds$,array values$)

This function will return an array of data for the fields specified in flds$, if the specified key value is in the file. If not, an error 11 is returned. Note that often an error is expected, such as when doing a positioning read to some logical point within the file. You can also use this function to set the alternate key value for subsequent sequential reads of the file.

The field specification can be defined in several ways. It can contain field names from a template associated with the channel, or field numbers, or expressions. Each field or expression is delimited with a comma.

User attributes from a template can affect the format of the data returned. This is of particular use with formatted strings or dates. See the dsTmpl function for a description of the user attributes that are used.

Field numbers are just decimal numbers that reference a physical field in the file. For example, a flds$ string of “1,3,5,6” would return the first, third, fifth, and sixth physical fields from the file.

Field names must be found in the template associated with the channel. An example might be: "id,name,slsp,mtd_sales,ytd_sales". Mask, date format, and precision settings found in the user attributes for the field are used when returning the data. Numerics are returned trimmed of any spaces. Strings are returned trimmed of trailing spaces, but will retain any leading spaces. Fields interpreted as dates are returned as human-readable date strings in the format defined by the server configuration or the dsSetDate function.

Expressions are Business Basic code fragments that return a value. If they resolve to numeric data, the server will convert them to decimal strings. The expression must start with an equal sign (=), and can contain references to template field names, the record itself as rec$, the key value as ky$, or a physical field as rec.fld$[*field-number*]. Field names in the expression are treated as their native type. A numeric field is a numeric value in the expression. Fields interpreted as dates are julian numbers in the expression. The expression can also contain any valid Business Basic function, any internal dSERVE function. Be sure to balance any quotes or parentheses in the expression. If the expression starts with "=raw:", then dSERVE returns data as stored, without converting numbers and dates to printable values. This can be useful when deriving keys to secondary files, if the keys contain binary-stored dates or numbers.

Internal dSERVE Functions	
cdate(*string*)	Converts *string*, assumed to be a delimited text date such as "1/30/2000", into a julian number. The order of elements (mdy or dmy) is determined by the date parameters. Any non-numeric delimiter may be used, or the date string can contain zero-filled digits, such as "01302000".
cnum(*string*)	Converts *string*, which may contain non-numeric data such as commas, into a number. cnum("1,255.30CR") will return -1255.3.
left(*string,length*)	Returns the left-most *length* characters of *string*, without an error 47 if the *string* isn't long enough. The result is right-padded with spaces if necessary.
mid(*string,position,length*)	Returns a sub-string from *string*, without an error 47.
right(*string,length*)	Returns the right-most *length* characters of *string*, without an error 47. The result is left-padded with spaces if necessary.

External programs can be written and called by the server, in order to return values that can't be resolved in a simple expression. To reference an external called program, use the syntax "*@progname*(*arg1, arg2,… argn*)". This is equivalent in Business Basic syntax to "CALL "*progname*",*arg1,arg2,…,argn*". See your Business Basic manual for information about writing and CALLing programs. When executing an external function, one of the arguments must be the variable VAL$, which is the value that will be returned to the client. In addition to that variable, you can use REC$ as the string templated data record, KY$ as the record key, and your own variables, literals, and expressions. To avoid variable conflicts with the server program, you should begin your variable names with TEMP, such as TEMPNAME$ or TEMP_AMOUNT. If you open any files in your CALLed program, be sure to use channel numbers assigned with UNT, and keep track of (and re-use) their channels with global strings, or close them before your program exits. Otherwise, you may have resource limit errors if your program is CALLed many times in a single during a single connection.

For example, to execute a program called "customdate", which accepted the record, a field name, and returned a decimal string of a julian number as a field, you could use this syntax:
@customdate(rec$,"order_date",val$).

There is a limit of 255 unique @call and =*expr* definitions per connection, and each @call and =*expr* statement is limited to 255 bytes of text.

Field names, expressions, and external functions can be intermixed in flds$. For example, you could return quantity, price, and extension like this: “quantity,price,=quantity*price”.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
key$	The key value used to read the file.
knum%	The alternate key number (0=primary key, 1-16 are possible alternate keys).
flds$	A list of fields to read, delimited by commas, along with optional @ and = expressions. If the value is null and a template is associated with the channel (i.e. the file was opened via dsOpenDict), then all fields are returned in template order.
array values$	Returns an array of values associated with array flds.
zvalues$	A Chr(0) delimited string, easily converted to an array with the VBScript Split function.

Extract Record Fields by Key

DLL: dsExtractFld (chan%,ky$,knum%,flds$,array values)
VBS: dsExtractFld (chan%,ky$,knum%,flds$,zvalues$)
PHP: dsExtractFld (chan%,ky$,knum%,flds$,array values)
Perl: dsExtractFld (tcphandle&,chan%,ky$,knum%,flds$,array values)

This function is identical to the Read Record Fields function, above, except that the EXTRACT function locks the record from EXTRACTs by other users. Use this function in preparation for a write record operation.

Read Next Record Fields

DLL: dsReadFldNext (chan%,flds$,ky$,array values)
VBS: dsReadFldNext (chan%,flds$,ky$,zvalues$)
PHP: dsReadFldNext (chan%,flds$,ky$,array values)
Perl: dsReadFldNext (tcphandle&,chan%,flds$,ky$,array values)

This function will return the next sequential key and field values from the file channel. If a filter has been set for the channel, then the file is read until a record matching the filter criteria is found. When the end of the file is reached, the function returns 2.

See the dsReadFld function for information about the format of flds$.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
flds$	A list of fields to read, delimited by commas, along with optional @ and = expressions. If the value is null and a template is associated with the channel (i.e. the file was opened via dsOpenDict), then all fields are returned in template order.
ky$	Returns the key of the record.
array values	Returns an array of values associated with array flds.
zvalues$	A Chr(0) delimited string, easily converted to an array with the VBScript Split function.

Read Previous Record Fields

DLL: dsReadFldPrev (chan%,flds$,ky$,array values)
VBS: dsReadFldPrev (chan%,flds$,ky$,zvalues$)
PHP: dsReadFld (chan%,flds$,ky$,array values)
Perl: dsReadFld (tcphandle&,chan%,flds$,ky$,array values)

This function will return the previous sequential key and field values from the file channel. If a filter has been set for the channel, then the file is read until a record matching the filter criteria is found. When the beginning of the file is reached, the function returns 2.

See the dsReadFld function for information about the format of flds$ and values$.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
flds$	A list of fields to read, delimited by commas, along with optional @ and = expressions. If the value is null and a template is associated with the channel (i.e. the file was opened via dsOpenDict), then all fields are returned in template order.
ky$	Returns the key of the record.
array values	Returns an array of values associated with array flds.
zvalues$	A Chr(0) delimited string, easily converted to an array with the VBScript Split

	function.

Get File Keys

DLL: dsGetKey (chan%, key$)
dsGetKeyFirst (chan%, key$)
dsGetKeyLast (chan%, key$)

PHP: dsGetKey (chan%, key$)
dsGetKeyFirst (chan%, key$)
dsGetKeyLast (chan%, key$)

Perl: dsGetKey (tcphandle&, chan%, key$)
dsGetKeyFirst (tcphandle&, chan%, key$)
dsGetKeyLast (tcphandle&, chan%, key$)

These functions return the current key, first key, and last key, respectively. The first and last keys are what one would expect: first key in the file, and last key in the file. The current key is whatever the current key pointer is set to in the channel. If the last operation was a read, then the current key is the key after the record just read. If the last operation was an extract, then the current key is the key extracted. If the file has just been opened, and no read operations performed, then the current key is the first key in the file. These functions also honor the last used key number in a read operation.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
key$	Returned current, first, or last key of the channel.

File Identification

DLL: dsInfoFid (chan%,fid$)
PHP: dsInfoFid (chan%,fid$)
Perl: dsInfoFid (tcphandle&,chan%,fid$)

The Business Basic FID function (FIB on ProvideX) provides information about a file's structure, such as its key size(s) and record length. This function returns the value of FID or FIB for any given file channel. Note that the format is determined by the language (PRO/5 or ProvideX) the server runs under.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
fid$	Returned as the FID() or FIB() value of the channel.

File Information

DLL: dsInfoFin (chan%,fin$)
PHP: dsInfoFin (chan%,fin$)

Perl: dsInfoFin (tcphandle&,chan%,fin$)

The Business Basic FIN function provides information about a file's structure, such as its key size(s) and record length. This function returns the value of that function for any given file channel. Note that the format is determined by the language (PRO/5 or ProvideX) the server runs under.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
fid$	Returned as the FIN() value of the channel.

Basic File Statistics

DLL: dsInfoStats (chan%,keysize%,recsize&,numrecs&)
PHP: dsInfoStats (chan%,keysize%,recsize&,numrecs&)
Perl: dsInfoStats (tcphandle&,chan%,keysize%,recsize&,numrecs&)

This function returns basic information about the file associated with the file channel.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
keysize%	Returns the primary key length.
recsize&	Returns the record length.
numrecs&	Returns the active record count.

Template Field Attributes

DLL: dsFattr (chan%,fld$,attr$,fattr$)
PHP: dsFattr (chan%,fld$,attr$,fattr$)
Perl: dsFattr (tcphandle&,chan%,fld$,attr$,fattr$)

This function will return information about a string template, or a field within the template, or a user attribute within a field. A template must have been associated with the file channel via the dsTmpl or dsOpenDict functions before this function can work.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
fld$	The field to return attribute information about. If null, then a list of fields delimited by line-feed (ASCII 10) characters is returned in fattr$.
attr$	A user attribute, such as "mask" or "date". If null, then a standard field specification is returned in fattr$. Otherwise, the value of the user-defined attribute is returned.
fattr$	Returns a value, depending on the settings of fld$ and attr$. If fld$ is null, then this holds a line-feed delimited list of field names.

	If fld$ is a field name, and attr$ is null, then this returns a binary string containing field attributes, as defined in the BBx FATTR or ProvideX XFA function. If fld$ is a field name and attr$ is a user attribute, then this returns the value of the user attribute.

Get Date Management Information

DLL: dsGetDate (mask$,suffix$,style$,century%,mdy$)
PHP: dsGetDate (mask$,suffix$,style$,century%,mdy$)
Perl: dsGetDate (tcphandle&,mask$,suffix$,style$,century%,mdy$)

Since Business Basic provides no specific date data type, dates can be stored in a variety of ways in a Business Basic data file. However, client software often has no way of interpreting this information or the algorithms would be very complex to write. The trick is to interpret the date and generate a usable format using the Business Basic engine in the server. In order to manage this format, and the process of interpreting the stored date structure, a set of parameters is used. This function returns those parameters to the client, while the companion dsSetDate function sets new parameters. Parameter defaults are defined in the server configuration.

tcphandle&	The connection handle returned by the dsConnect function.
mask$	Returns the date format mask used when sending date fields in read operations. The mask uses three elements and optional modiers: %M Decimal month, “3” %Mz Decimal month, zero filled, “03” %Ms Short month name, “Mar” %Ml Long month name, “March” %D Decimal day, “1” %Dz Decimal day, zero filled, “01” %Ds Short day name, “Sun” %Dl Long day name, “Sunday” %Y, %Yz 2-digit year, “00” %Ys, %Yl 4-digit year, “2000”
suffix$	Returns the field name suffix that indicates a field is a date. For example, if the suffix is “_date”, then any field whose name ends in “_date”, such as “order_date”, will be interpreted as a date field. Note that a template associated with a channel can also define a “date=*style*” user attribute to force date interpretation.
style$	Returns the date style, such as “aon”, “jul”, “mdy”, etc. Date styles are documented in the dsTmpl function description.
century%	Returns the default century used by the cdate() function when 2-digit years are provided. This would generally be 2000.
mdy$	Returns the date element order used by the cdate() function, either “mdy” or “dmy”.

Set Date Management Information

DLL: dsSetDate (mask$,suffix$,style$,century%,mdy$)
PHP: dsSetDate (mask$,suffix$,style$,century%,mdy$)
Perl: dsSetDate (tcphandle&,mask$,suffix$,style$,century%,mdy$)

tcphandle&	The connection handle returned by the dsConnect function.
mask$	Sets the date format mask used when sending date fields in read operations. Examples might be: "%Mz/%Dz/%Yz" for a standard mm/dd/yy US format, "%Ml %D, %Yl" for a long date, like January 1, 2001. See the dsGetDate function for all valid date mask parameters.
suffix$	Sets the field name suffix that indicates a field is a date. Note that a template associated with a channel can also define a "date=*style*" user attribute to force date interpretation.
style$	Sets the date style, such as "aon", "jul", "mdy", etc., used for default interpreted dates. Date styles are documented in the dsTmpl function description. Note that a user attribute "date=*style*" in the template associated with a file will override the default style.
century%	Sets the default century used by the cdate() function when 2-digit years are provided. This would generally be 2000.
mdy$	Sets the date element order used by the cdate() function, either "mdy" or "dmy". The default is "mdy".

Global String Set (STBL or GBL)

DLL: dsSetGbl (gblname$,value$)
PHP: dsSetGbl (gblname$,value$)
Perl: dsSetGbl (tcphandle&,gblname$,value$)

tcphandle&	The connection handle returned by the dsConnect function.
gblname$	The name of the global string. This can be any printable text value up to 32 bytes long. Avoid names starting with ! or $ to eliminate possible conflicts with the server environment.
value$	The value to assign to the named global string.

Global String Get (STBL or GBL)

DLL: dsGetGbl (gblname$,value$)
PHP: dsGetGbl (gblname$,value$)
Perl: dsGetGbl (tcphandle&,gblname$,value$)

tcphandle&	The connection handle returned by the dsConnect function.
gblname$	The name of the global string.
value$	Returns the value of the named global string. If the string has not been defined previously, either with a dsSetGbl function or by the server application, then an error 49 is returned (error 23 on ProvideX).

The following three functions update data records in the files accessed by dSERVE. Note that Business Basic does not provide for referential integrity; it is therefore the responsibility of the programmer to ensure that data consistency is maintained. For example, if a sort file is used to maintain a sort of customer records by name, then that sort file must be updated whenever the customer record is updated. A second example: before removing a customer record, you should verify that there are no open invoice records.

Before developing an application that updates files, make sure you fully understand the business rules and file relationships involved. Be sure to thoroughly test with trial data files any application that updates records.

Write Record

DLL: dsWriteRec (chan%,key$,record$)
PHP: dsWriteRec (chan%,key$,record$)
Perl: dsWriteRec (tcphandle&,chan%,key$,record$)

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
key$	The primary key to use when writing the record. Note that if the file is a true multi-keyed file, with all keys defined with record references, then this value is ignored during the write.
record$	The full record to be written. If the record contains field terminators, it is the writing application's responsibility to include the proper values. If there is a template associated with the channel, then dSERVE will issue a FIELD(REC$) function to ensure that any extraneous data is removed before the write. Otherwise, the record length must be less than or equal to the record length defined for the file, or an error 1 will be returned. dSERVE must be configured to allow write access for this function to work.

Write Record Fields

DLL: dsWriteFld (chan%,key$,record$,flds$,array values)
VBS: dsWriteFld (chan%,key$,record$,flds$,zvalues$)
PHP: dsWriteFld (chan%,key$,record$,flds$,array values)

Perl: dsWriteFld (tcphandle&,chan%,key$,record$,flds$,array values)

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The file channel returned by the dsOpen or dsOpenDict function.
key$	The primary key to use when writing the record. Note that if the file is a true multi-keyed file, with all keys defined with record references, then this value is ignored during the write.
record$	The contents of the full record to be written. Within this record, the fields specified in flds$ will be updated. If this value is null, then only the values specified in flds$ will contain data. This will typically be set from a previous Extract Record function.
flds$	A comma-delimited list of field names to update, associated with the array of values.
array values	An array of values or expressions to assign to the associated field in the record. If a value contains binary data or line-feeds, you can use an expression format, such as '*=expression*' or '*@call-program*(*arg1*,*arg2*,…,val$). The expression can contain references to other fields in the record$ string in the format REC.*fld* or REC.*fld$*. When using the *@call* syntax, the value returned in the argument VAL$ will be assigned to the field in the record. For numeric fields, VAL$ should contain a simple numeric string, as dSERVE will resolve it as NUM(VAL$). After each field in flds$ is assigned the associated value from value$, the record is trimmed of extraneous data with a FIELD(REC$) function, and written to the file. dSERVE must be configured to allow write access for this function to work. There is a limit of 255 unique @call expressions per connection, and each @call statement is limited to 255 bytes of text. It is the application's responsibility to ensure that field data is properly formatted for the data files on the server. The server will un-mask any text fields that have been returned with mask values, and parse date fields from human-readable values back to internal formats. All other data must be set in a manner that the server and the host applications expect. Be sure to consult with someone knowledgeable in the file and field structures on the server before writing data back.
zvalues$	A Chr(0) delimited string of values, associated with the list in flds$, easily converted from an array with the VBScript Join function.

Remove Record by Key

DLL: dsRemoveRec (chan%,key$)
PHP: dsRemoveRec (chan%,key$)
Perl: dsRemoveRec (tcphandle&,chan%,key$)

tcphandle&	The connection handle returned by the dsConnect function.

chan%	The file channel returned by the dsOpen or dsOpenDict function.
key$	This is the key to remove from the file. If the key doesn't exist, an error 11 will be returned.

Create a Keyed File

DLL: dsCreateKeyed (pathname$,keydef$,recsize%)
PHP: dsCreateKeyed (pathname$,keydef$,recsize%)
Perl: dsCreateKeyed (tcphandle&,pathname$,keydef$,recsize%)

This function creates a new keyed file that can then be opened for reading and writing as any other Business Basic keyed file. If prefixmode is set, then the file is created in the temp/ directory under the server directory. Otherwise, the file is created in the path specified. An error 12 will occur if the file already exists.

The keydef$ is in a syntax that is useful to BBx or ProvideX, following the rules for the key specification portion of the MKEYED or KEYED verbs. Either single-keyed or multi-keyed files can be created, depending on the syntax supplied. For example, if keydef$ is "12", then a single-keyed file with a 12-byte primary key is created. If keydef$ is "[0:1:6],[0:7:20]+[0:1:6]", then a multi-keyed file is created with a primary and two-part secondary key. See your BBx or ProvideX documentation for the full syntax specification.

The recsize% option is also passed to the MKEYED or KEYED verb for the record size.

tcphandle&	The connection handle returned by the dsConnect function.
pathname$	The path (file) name to create on the server.
keydef$	The key specification, which must be a valid key setting for the BBx or ProvideX language under which dSERVE is running, based upon the MKEYED or KEYED command.
recsize%	Record size for the file.

Create a Text File

DLL: dsCreateText (pathname$)
PHP: dsCreateText (pathname$)
Perl: dsCreateText (tcphandle&,pathname$)

This function creates a new text file that can then be opened for reading and writing using dsReadBinary and dsWriteRec functions. If prefixmode is set, then the file is created in the temp/ directory under the server directory. Otherwise, the file is created in the path specified. An error 12 will occur if the file already exists.

tcphandle&	The connection handle returned by the dsConnect function.
pathname$	The path (file) name to create on the server.

Erase a File

DLL: dsEraseFile (pathname$)
PHP: dsEraseFile (pathname$)

Perl: dsEraseFile (tcphandle&,pathname$)

This function erases the pathname specified. If prefix mode is set, then the file must be found via the file search prefix set on the dSERVE server configuration. In addition, the erase option must be configured in the server configuration ini file ([security] section) to enable this function.

tcphandle&	The connection handle returned by the dsConnect function.
pathname$	The path (file) name to erase on the server.

Data Exporting Functions

Exporting is a feature of dSERVE version 2. Exporting provides a fast way of extracting data from a Business Basic data file and transferring it to a client for processing, by eliminating much of the overhead of reading individual records at a time. Exports can be run through completion to a local file, using the dsExport function, or can be executed in background on the server, using the dsExportStart function, so the client process need not wait for the export to finish before performing other tasks. When started with dsExportStart, the client task can monitor the status of the export with the dsExportStatus function.
Exports can be produced in four formats:

- tab delimited, a flat format where fields are delimited by tab characters (ASCII 9).
- csv, or comma-separated-values, where text fields are quoted and fields are separated by commas.
- xml, a simple xml format that can be easily parsed by xml tools.
- ado, a more complex xml format that can be opened directly as an ADO record set in Microsoft Windows development environments. The ActiveX DLL client offers a function to automatically return a record set object.

Since export files can become very large, a common bottleneck can be the time it takes to transfer a file back to the client across the network. For that reason, dSERVE also offers a function to execute the gzip command on the server to compress a file. The resulting compressed file can be transferred much more quickly than the original, then uncompressed on locally using gzip or gzip.exe. The dsExport function will attempt to use the gzip option automatically, and will fall back to using no compression if gzip fails on the server. Note that if gzip works on the server, the client must also have gzip in order to uncompress the file. The ActiveX DLL installation includes gzip.exe, but other client installs do not provide gzip, so if your system does not have it already, it must be obtained from gzip.org on the Internet.

Note that it is not always necessary to use dSERVE to copy a file from the server to the local system. For example, if dSERVE and a web server are on the same machine, then the export file can simply be copied using file system methods rather than the slower sockets. In this case, use the dsExportStart/dsExportStatus method, since dsExport automatically does a dSERVE binary copy from the server to the client.

Execute an Export

DLL: dsExport (chan%, format$, knum%, startkey$, endkey$, flds$, localpath$)
PHP: dsExport (chan%, format$, knum%, startkey$, endkey$, flds$, localpath$)
Perl: dsExport (tcphandle&, chan%, format$, knum%, startkey$, endkey$, flds$, localpath$)

This function is a wrapper that uses dsExportStart and dsExportStatus to start an export and retrieve its result into a local file. This function starts a background export process, which will export records from the file opened on channel chan%, honoring any filter specified using the dsFilter function. The format of the export file must be specified as “tab”, “csv”, “xml”, or “ado”. The result will be a tab-delimited file, a comma-separated value file, a simple XML file, or a Microsoft ADO XML file. The records in the range of startkey$ through endkey$ (in the key number specified in knum%) will be processed, and the fields listed in the comma-separated list in flds$ will be exported. The export files and status records are automatically removed on the server after 24 hours.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The channel, previously opened with the dsOpenDict , or the dsOpen function

	followed by a dsTmpl function. The channel must have a template associated with it.
format$	The format of the export: tab, csv, xml, or ado.
knum%	The key number to read chan% with.
startkey$	The starting key to begin exporting.
endkey$	The ending key to stop exporting.
fields$	
localpath$	The local path or file name to store the export data in once the export is complete.

Start an Export

DLL: dsExportStart (chan%, format$, knum%, startkey$, endkey$, flds$, id$)
PHP: dsExportStart (chan%, format$, knum%, startkey$, endkey$, flds$, id$)
Perl: dsExportStart (tcphandle&, chan%, format$, knum%, startkey$, endkey$, flds$, id$)

This function starts a background export process, which will export records from the file opened on channel chan%, honoring any filter specified using the dsFilter function. The format of the export file must be specified as “tab”, “csv”, “xml”, or “ado”. The result will be a tab-delimited file, a comma-separated value file, a simple XML file, or a Microsoft ADO XML file. The records in the range of startkey$ through endkey$ (in the key number specified in knum%) will be processed, and the fields listed in the comma-separated list in flds$ will be exported. Each export is started in background, and is given a 16-byte ID, returned in id$. The ID can be used for checking the status of the export (using the dsExportStatus function) and also for opening and reading the resulting data file. The file is created in the temp directory under the dSERVE server, as the name ID + “.exp”.

The export files and status records are automatically removed after 24 hours.

tcphandle&	The connection handle returned by the dsConnect function.
chan%	The channel, previously opened with the dsOpenDict , or the dsOpen function followed by a dsTmpl function. The channel must have a template associated with it.
format$	The format of the export: tab, csv, xml, or ado.
knum%	The key number to read chan% with.
startkey$	The starting key to begin exporting.
endkey$	The ending key to stop exporting.
id$	Returns the export ID, used for checking status and opening the resulting export file.

Check Export Status

DLL: dsExportStatus (id$, code$, errmsg$, maxrecs&, readrecs&)
PHP: dsExportStatus (id$, code$, errmsg$, maxrecs&, readrecs&)
Perl: dsExportStatus (tcphandle&, id$, code$, errmsg$, maxrecs&, readrecs&)

This function checks the current status of an export started by the dsExportStart function. Exports run in background on the server, so that the dsExportStart command can return immediately rather than having the client wait for a potentially long process to finish. This function provides on-demand status of a given export. It can be issued repeatedly until the code$ value is “9”, indicating completion, or “e”, indicating an error. If an

error occurred, then the errmsg$ value will contain a text description of the error. The maxrecs& value is the number of records in the channel that the export is executed on, and the readrecs& value is the current number of records that have been read, rounded to the nearest 100. Note that if a key range is specified, using startkey$ and endkey$ in the dsExportStart function, readrecs& may never reach maxrecs&.

Typically, this function will be executed within a loop that checks the status, then pauses, then checks again, until the function returns an error code, or code$ is "9" or "e". Once code$ is "9", then the export file "temp/"+id$+".exp" is completely built.

tcphandle&	The connection handle returned by the dsConnect function.
id$	The 16-character ID code returned by the dsExportStart function.
code$	A 1-character code that indicates the current status of the export. 0=not started, 1=started, 9=complete, e=error.
errmsg$	Returns an error message, if code$ is e.
maxrecs&	Returns the number of records in the channel specified in the dsExportStart function.
readrecs&	Returns the number of records read in the export process, rounded to the nearest 100. It can be used to estimate the progress of the export relative to maxrecs&, though if the dsExportStart function specified a key range, then the total number of records to be read will likely be less than maxrecs&.

Gzip a File

DLL: dsGzip (pathname$)
PHP: dsGzip (pathname$)
Perl: dsGzip (tcphandle&,pathname$)

This function will invoke gzip on a file to compress it. The gzip program must be present on the server, as the server simply invokes the command "gzip" or "gzip.exe". This function is useful when a large binary file is to be copied from the server to the client. The client can then "unzip" the file using a local copy of gzip or gzip.exe, using the –d (decompress) option. The primary purpose of this function in dSERVE is to support fast exporting and copying of data using the dSERVE export functions.

Note the gzip option must be set in the dSERVE configuration ini file in order for this function to operate.

tcphandle&	The connection handle returned by the dsConnect function.
pathname$	The path (file) name to gzip on the server.

Server Interface Specification

However, if your language or environment prevents the use of the DLL to communicate with the dSERVE server, this chapter is for you. Here we document how to communicate, via TCP/IP, with the dSERVE server.

Data Structures Used

Data is sent and received from dSERVE using various binary structures. The following table describes the data structures used by dSERVE.

String	Variable length ASCII strings, such as “John Smith”.
Numbers	Variable length strings containing numeric data, such as “123.45”.
Zero-filled Numbers	A fixed length numeric value with leading 0’s. For example, a 5-byte number 10 would be represented as “00010”.
4-byte Integer	A 4-byte integer value, “big endian” format, with the most significant byte to the left. In BASIC, here is an algorithm for calculating this value: c1%=INT(value&/256^3): m&=value& MOD 256^3 c2%=INT(m&/256^2): m&=m& MOD 256^2 c3%=INT(m&/256): m&=m& MOD 256 c4%=m& return CHR$(c1%)+CHR$(c2%)+CHR$(c3%)+CHR$(c4%)
2-Byte Integer	2-byte integer value, parsed left to right, from 0 to 65535. In generic BASIC syntax, generate a value with the algorithm: CHR(INT(*value*/256)) + CHR(MOD(*value*,256). To retrieve a value, use the algorithm: ASC(LEFT(*value*,1))*256 + ASC(RIGHT(*value*,1)).
1-Byte Integer	1-byte ASCII integer, from 0 to 255.

Sessions

Each time you wish to open and read data files with dSERVE, you begin by connecting to dSERVE and initiating a session. dSERVE will respond with a socket number on which to converse. This socket is reserved exclusively for your process to communicate with a dSERVE process on the server.

All commands for opening and reading files are sent during the lifetime of the session. When you are done, you simply close the socket, which will release the dSERVE process that was handling your connection.

To initiate a session:

1) Open a socket to the IP address and port on which the server listens for connections. The port defaults to 8227, but can be set by a server command line argument (-p) or the server’s .ini file.

2) Send a 1-byte integer 255 (0xff) to wake up the server. If your IP address passes dSERVE's security test, you will receive a 5-byte response. This will be either a 5-byte zero-filled integer or a letter "e" followed by a 4-byte zero filled integer.

An "e" style response indicates an error occurred trying to initiate the process to handle your session. The error number is stored in the remaining 4 bytes. An error of 999 indicates failure to start the session process. An error of 998 indicates you have exceeded the license limits of your installation.

A numeric response indicates a session process has been initiated and it is listening on a new port number. The number returned is that port. You should open a new socket to the new port, then close the socket to the original port. Your session is now started and the dSERVE process is awaiting your commands.

Sending and Receiving Data

All data is sent and received with a 5-byte zero-filled length prefix followed by the data. Data frequently contains binary characters, so you should perform your send and receive functions as binary reads and writes rather than line-oriented reads and writes. Line feeds and carriage returns should not be interpreted as end-of-lines, as they may very well occur in the middle of data. Beware of incomplete data. Some languages will return only available data, regardless of the length of data requested. In such a case, you should continue to read until all the data is returned.

When an error occurs during a function, dSERVE will respond with an error code rather than a length prefix and data. The error code is indicated by a letter "e" followed by a 4-byte integer error code. For example, if you perform a "read next record" function, and the key pointer is at the end of the file, dSERVE will respond with "e0002", indicating an error 2 (end-of-file) error.

When a command is sent or data received, the data will have a specific set of values in a define structure. Variable length data is prefixed with a 1- or 2-byte integer, while fixed length data is sent without a prefix.

For example, the following might be a response from a field style read, which returns a variable number of variable length data fields (<*n*> notation indicates byte values):

<0><5>00100<0><28>Synergetic Data Systems Inc.<0><2>CA

This response indicates a 5-byte field, a 28-byte field, and a 2-byte field. Each field is prefixed by a 2-byte integer indicating the length of the following data.

When this is received from the server, the whole response will be prefixed with a 5-byte, zero-filled integer indicating the length. In this example, 2+5+2+28+2+2, "00041".

Functions

After a session is started, all activity with the dSERVE server involves sending function commands and reading the response. The following describes the functions, their parameters, and the format of the dSERVE response. Remember that the 5-byte length and/or error prefixes are not included in these specifications.

Open a File

Send: "op"
1-byte length
file name

Receive: 1-byte file handle

Note: All file I/O commands require the file handle. Multiple files can be opened concurrently, up to the maximum number allowed by the operating environment of the server.

Close a File Handle

Send: "cl"
1-byte file handle

Receive: 0 bytes

Associate a String Template with a File Handle

Send: "tm"
1-byte file handle
2-byte length
template string

Receive: 0 bytes

Set Field Attribute

Sends: "sa"
1-byte channel
1-byte field name length
field name
1-byte attribute name length
attribute name
1-byte value length
value

Receives: New template string

This function can be used to add or change a user attribute for a channel's template. Use it, for example, to override the mask or precision of a field after a file has been opened via a dictionary that doesn't provide the desired attribute setting.

Open a File via a Dictionary

Send: "od"
1-byte length
Dictionary name
1-byte length
File alias name

Receive: 1-byte file handle
1-byte length
file pathname
2-byte length
template string

Notes: This function opens and reads the dictionary, and uses that information to open the file and associate a string template with the file handle. The path to the physical file opened, as well as the string template, are returned. In BBx, the dictionary name should be a .tpm file. dSERVE locates the dictionary and data directories from the content of the .tpm file. In ProvideX, this should be the providex.ddf dictionary file or a dictionary .ini file. If prefixmode is set, then the dictionary file must be found via the server's directory search prefix. The physical file path is not affected by prefixmode, but if it is relative, it must be relative to the dSERVE home directory or be found in the directory search prefix.

If an extension dictionary exists for the file alias, then the template string, and any subsequent template-related operations, will return field names found in the extension dictionary in addition to those in the standard dictionary.

This operation can also establish automatic filtering of non-normalized data, and call an *onopen program defined in an extension dictionary.

List Files in a Dictionary

Send: "ld"
1-byte length
dictionary path string
1-byte length
wildcard string

Receive: 2-byte length
Delimited list of files (ASCII 10 trails each filename)
2-byte length
Delimited list of descriptions (ASCII 10 trails each description)

Notes: This function will return a list of files and associated descriptions from the dictionary named. In ProvideX, no descriptions are available in the dictionary. The dictionary must be a .tpm file for BBx, or a providex.ddf or dictionary .ini file for ProvideX. If prefixmode is set, then the dictionary pathname must be found via the server's directory search prefix. If a wildcard is specified, then only file names matching the wildcard are returned.

List Fields in File

Send: "lf"
1-byte file handle

Receive: 2-byte length
Delimited list of fields (ASCII 10 trails each field name)

Notes: This function works on files opened via a dictionary, or files with a manually associated string template. The fields are returned in alphabetical sequence. Fields with a template user attribute of "hide=1" are not returned.

Set Filter

Send: "sf"
1-byte file handle
2-byte length of filter string
filter string

Receive: 0 bytes

Notes: A filter is used by the sequential (next/previous) read functions to limit the data records returned for a given file handle. Filters can provide improved performance because only records that pass the filtering criteria will be sent across the network connection. See Filter Records From Sequential Reads section in the Client Functions documentation for details about filter formats.

Read Binary String

Send: "br"
1-byte file handle
4-byte integer offset
2-byte integer bytes

Receives: Bytes read, from 0 to bytes specified

This function is used to read data in blocks from a binary or text file on the server. The offset value is a 0-based value, with 0 being the first byte in the file. The function will return the number of bytes requested, unless the an end of file is reached, in which case the number of bytes up to the end of file are returned. If the offset is past the end of file, then an error 2 is returned.

Read Record by Key, Key Number

Send: "rr"
1-byte file handle
1-byte key number (0=primary key, 1-16 alternate keys)
1-byte key length

key string

Receive: record data string

Notes: You can use this function to retrieve a record, or to set the key pointer and/or alternate key number for later reads of the file handle. If the record doesn't exist, an error code 11 will be returned, but the key pointer and alternate key chain will be set. For all record style read operations, your application is responsible for parsing the data in the record into useful elements. If you want the dSERVE server to parse the record into fields, then use the field style read functions.

Extract Record by Key, Key Number

Send: "er"
1-byte file handle
1-byte key number (0=primary key, 1-16 alternate keys)
1-byte key length
key string

Receive: record data string

Notes: You can use this function to read and lock a record, or to set the key pointer and/or alternate key number for later reads of the file handle. Until another operation is performed on the channel, other users will be prevented from locking, writing, or removing the record, but not from reading the record. If the record doesn't exist, an error code 11 will be returned, but the key pointer and alternate key chain will be set. For all record style read operations, your application is responsible for parsing the data in the record into useful elements. If you want the dSERVE server to parse the record into fields, then use the field style read functions.

Read Next Record

Send: "rn"
1-byte file handle

Receive: 1-byte key length
key string
record data string

Notes: This will read the record at the key pointer, and advance the key pointer to the next record in preparation for the next read function.

Read Previous Record

Send: "rp"
1-byte file handle

Receive: 1-byte key length
key string
record data string

Notes: This will read the record before the last one returned by the last read operation.

Read Fields by Key, Key Number

Send: "fr"
1-byte file handle
1-byte key number (0=primary key, 1-16 are alternate keys)
1-byte key length
key string
2-byte field string length
field string

Receive: 2-byte field 1 length
field 1
2-byte field 2 length
field 2
…
2-byte field*n* length
field*n*

Notes: This function reads the record noted by the key and key number, then parses the record into fields and expressions specified in the field string. See the dsReadFld DLL function for a description of the field string. The number of fields received matches, and is associated with, the comma-delimited fields and expressions in the field string. It can also be used to set the key pointer and/or the alternate key number for subsequent reads.

Extract Fields by Key, Key Number

Send: "ef"
1-byte file handle
1-byte key number (0 for primary key, 1-16 for alternate keys)
1-byte key length
key string
2-byte field string length
field string

Receive: 2-byte field 1 length
field 1
2-byte field 2 length
field 2
…
2-byte field*n* length
field*n*

Notes: This function reads and locks the record noted by the key and key number, then parses the record into fields and expressions specified in the field string. Until another operation is performed on the channel, other users will be prevented from locking, writing, or removing the record, but not from reading the record. See the dsReadFld DLL function for a description of the field string. The number of fields received matches, and is associated with, the comma-delimited fields and expressions in the field string. It can also be used to set the key pointer and/or the alternate key number for subsequent reads.

Read Fields of Next Record

Send: "fn"
1-byte file handle
2-byte field string length
field string

Receive: 1-byte key length
key string
2-byte field 1 length
field 1
2-byte field 2 length
field 2
…
2-byte field*n* length
field*n*

Read Fields of Previous Record

Send: "fp"
1-byte file handle
2-byte field string length
field string

Receive: 1-byte key length
key string
2-byte field 1 length
field 1
2-byte field 2 length
field 2
…
2-byte field*n* length
field*n*

Get Key Functions

Send: "ky" for current key, "kf" for first key, "kl" for last key
1-byte channel

Receives: Key requested

Get FID Information

Send: "id"
1-byte file handle

Receive: Result of FID function for the file handle (FIB on ProvideX)

Notes: The FID function returns a binary string. The format of the string is defined by the language of the dSERVE server – BBx or ProvideX.

Get FIN Information

Send: "in"
1-byte file handle

Receive: Result of FIN function for the file handle

Notes: The FIN function returns a binary string. The format of the string is defined by the language of the dSERVE server – BBx or ProvideX.

Get Basic File Information

Send: "is"
1-byte file handle

Receive: 1-byte length
Key size as decimal string
1-byte length
Record size as decimal string
1-byte length
Number of records as decimal string

Get Field Attribute Information

Send: "if"
1-byte file handle
1-byte field name length
field name
1-byte attribute name length
attribute name

Receive: Result of FATTR function used with the string template associated with the file handle, the field name, and the optional attribute name

Notes: The FATTR function is used to return information about a string template. If no template is associated with the file handle, the function will return an error 47. If the field name is null, then the function returns a list of field names delimited by line feeds (ASCII 10's). If the field name is not null and the attribute name is null, then a binary string about the field is returned. If the field name and attribute names are provided, then the user attribute for the field is returned.

See your BBx or ProvideX reference manual for documentation about the FATTR format.

Get Indexes

Send: "ix"
1-byte file handle

Receive: 2-byte index 1 length
index 1 description
2-byte index 2 length
index 2 length
…
2-byte index *n* length
index *n* description

Notes: The first index description is for the primary key. Additional descriptions are for alternate keys 1 through 16. If a template is associated with the file handle, any index segments that match fields in the template will be shown as field names. Other index segments will be shown in the syntax "[*field*:*offset*:*length*]". This is information used when the file was originally created. If an index contains multiple segments, the segments will be delimited with "+" characters.

Get Date Settings

Send: "dg"

Receive: 1-byte date mask length
date mask
1-byte date suffix length
date suffix
1-byte date style length
date style
1-byte default century length
default century as decimal string
1-byte date entry format length
date entry format

Notes: See Set Date Management Information in the Client Functions documentation for details about each of these elements.

Set Date Settings

Send: "ds"
1-byte date mask length
date mask
1-byte date suffix length
date suffix
1-byte date style length
date style
1-byte default century length
default century as decimal string
1-byte date entry format length
date entry format

Receive: 0 bytes

Notes: See Set Date Management Information in the Client Functions documentation for details about each of these elements.

Set Global String (STBL or GBL)

Send: "gs"
1-byte name length
string name
2-byte value length
string value

Receive: 0 bytes

Notes: This implements the BBx STBL() function or ProvideX GBL() function, setting the global string name to a value. Avoid names starting with ! or $. Names can be up to 32 characters in length.

Get Global String (STBL or GBL)

Send: "gg"
1-byte name length
string name

Receive: Global string value is entire response.

Notes: This implements the BBx STBL() function or ProvideX GBL() function, setting the global string name to a value. A BBx error 49 or ProvideX error 23 will result if the name specified has not been defined previously.

The following three functions update data records in the files accessed by dSERVE. Note that Business Basic does not provide for referential integrity; it is therefore the responsibility of the programmer to ensure that data consistency is maintained. For example, if a sort file is used to maintain a sort of customer records by name, then that sort file must be updated whenever the customer record is updated. A second example: before removing a customer record, you should verify that there are no open invoice records.

Before developing an application that updates files, make sure you fully understand the business rules and file relationships involved. Be sure to thoroughly test with trial data files any application that updates records.

Write Record

Send: "wr"
1-byte file handle
1-byte key length
primary key
2-byte record length
data record

Receive: 0 bytes

Notes: This function will write the data record using the associated primary key. If the file is a multi-keyed file defined so that all the keys are determined by the record itself, then the key data supplied is ignored. Care must be taken to format the record properly, including binary storage constraints and field separators. dSERVE must be configured to be write-enabled for this function to work.

Write Fields

Send: "wf"
1-byte file handle
1-byte key length
primary key
2-byte record length
data record
2-byte field spec length
field names delimited by line-feed characters (chr(10))
2-byte value spec length
list of values or expressions associated with field names, delimited by line-feeds

Receives: 0 bytes

Notes: This function will update the named fields to the values specified and write the data record. If no data record is supplied, then only the fields specified will contain data. See Write Record Fields in the client documentation for more details. dSERVE must be configured to be write-enabled for this function to work.

Remove Record

Send: "rm"
1-byte file handle
1-byte key length
primary key

Receives: 0 bytes

Notes: This function will remove the record specified by the primary key from the file. dSERVE must be configured to be write-enabled for this function to work.

Create a Keyed File

Send: "ck"
1-byte file name length
file name
2-byte key specification length
key specification
2-byte record size

Receives: 0 bytes

This function creates a new keyed file that can then be opened for reading and writing as any other Business Basic keyed file. If prefixmode is set, then the file is created in the temp/ directory under the server directory. Otherwise, the file is created in the path specified. An error 12 will occur if the file already exists.

The key specification is in a syntax that is useful to BBx or ProvideX, following the rules for the key specification portion of the MKEYED or KEYED verbs. Either single-keyed or multi-keyed files can be created, depending on the syntax supplied. For example, if the key specification is "12", then a single-keyed file with a 12-byte primary key is created. If it is "[0:1:6],[0:7:20]+[0:1:6]", then a multi-keyed file is created with a primary and two-part secondary key. See your BBx or ProvideX documentation for the full syntax specification.

The record size is also passed to the MKEYED or KEYED verb for the record size.

Create a Text File

Send: "ct"
file name

Receives: 0 bytes

This function creates a new text file that can then be opened for reading and writing using dsReadBinary and dsWriteRec functions. If prefixmode is set, then the file is created in the temp/ directory under the server directory. Otherwise, the file is created in the path specified. An error 12 will occur if the file already exists.

Erase a File

Send: "dl"
file name

Receives: 0 bytes

This function erases the pathname specified. If prefix mode is set, then the file must be found via the file search prefix set on the dSERVE server configuration. In addition, the erase option must be configured in the server configuration ini file ([security] section) to enable this function.

Start an Export

Send: "e1"
1-byte channel
1 byte format length
format
1 byte key number
1 byte start key length
start key
1 byte end key length
end key
2-byte fields list length
fields list

Receives: 16-byte ID string

This function starts an export in background on the server, returning an ID code that can be used for subsequent export status functions, and to open and read the resulting export file when the export is complete.

Check Export Status

Send: "e2"
export ID (16-character code from former Export Start command)

Receives: 1-byte status code
4-byte maximum records
4-byte records read
remaining data is an error message string

This function is used to check the status of an export that was started with the Start Export function. The status code can be one of the following:

0 Export not yet started
1 Export started
9 Export complete

e Export had an error, and the error message string will contain data

Typically, this function will be called in a loop, that will check for errors, or a status code of 9 or e, before continuing with processing. Once the status code is 9, the export file is available for opening and reading. The file name is “temp/”+ID+”.exp”, where ID is the export ID returned in the Export Start function. The directory “temp” is under the dSERVE directory on the server, which itself is stored in the global string “$home”, and can be retrieved with the Get Global String function.

Compress a File using gzip

Send: “gz”
file name

Receives: 0 bytes

This function attempts to use gzip on the server to compress the file. The resulting file will be names *filename* + “.gz”, and when read back to the client will require gzip or gzip.exe to uncompress it. This function is primary intended for use with exports, where the file to be returned may be very large, and network bandwidth a bottleneck.

The gzip option must be set to 1 in the dSERVE configuration ini file in order for the server to honor this function. In addition, the gzip program must be available for use by the server, or an error is returned.

Samples

The following samples are provided with dSERVE to show how to use the Perl, PHP, and ActiveX DLL clients:

- sample.pl – a Perl program that uses the dServe.pm Perl Module and produces text output
- sample.php – a PHP script that uses the dsphp.inc class file and produces HTML output
- sample.vbs – a VBScript program that uses the dSERVE.DLL COM object in a Windows IIS web server environment and produces HTML output

Perl: sample.pl

To execute this sample, first copy the dServe.pm module to one of your system @INC include directories (try perl –V to get a list). Modify the $server and $port lines if needed (if dSERVE is running on a different machine on your network, or if the listening port has been changed from its default in the dserve.ini file on the server). Then try **perl sample.pl | more**.

```
# SETUP INSTRUCTIONS:
#
# Copy dServe.pm to a local Perl include directory.

# Modify the $server= value, and if necessary, the $port value, so that this
# script can connect to the dSERVE 2.0 server.  The default installation includes
# sample files and dictionaries used by this script.
#
```

The dServe.pm module needs to be loaded with the use or require statement. This module exports the dsXXX functions as well as the $dserr and $dserrmsg variables. All functions return true (1) on success, and false (0) on error. If an error occurs, you can check the $dserr and $dserrmsg variables for information.

```
# Include the dServe.pm module
use dServe;
```

Change these values, if required, to point to the correct dSERVE server and/or port.

```
# Change these values to point to the correct server/port, if necessary.
$server=localhost;
$port=8227;
$timeout=10;
```

The first step is always a dsConnect function, which establishes a handle to be used by all other functions. In this example, the variable $handle is this handle.

```
# Try to connect to the server
if (! dsConnect($server,$port,$timeout,$handle) ) {
     # Oops - an error trying to connect
     die "Error $dserr ($dserrmsg) connecting to server\n";
}
```

The following code is designed to open the correct sample dictionary, depending on which version of dSERVE is running: BBx or ProvideX. The value of the global string $sfx contains ".bb" for BBx or ".pv" for ProvideX. Given that information, the sample uses the correct dictionary in the variable $dict. In your own code, you would simply use the dictionary required by your application.

```
# Demo dictionary is based on the server interpreter (bb or pv)
# .bb=dserve.tpm, .pv=pvxdata/providex.ddf
dsGetGbl($handle,'$sfx',$bb);
$dict='dserve.tpm';
if ( $bb eq ".pv" ) {
     $dict='pvxdata/providex.ddf';
}
```

The dsListDict function returns an 2-dimensional array of file names and descriptions. It requires a handle, a dictionary name, a wildcard search string, and an array argument.

```
print "Files in Demo Dictionary\n";
print "=======================\n";
```

```
# List files in the demo dictionary
if ( dsListDict($handle,$dict,'',@files) ) {
     for $i ( 0 .. $#files ) {
          print "$files[$i][0] \t $files[$i][1]\n";
     }
}
```

The next section opens three files using the dsOpenDict function. This function uses the dictionary to open channel ($chan) to a physical disk file ($path), and associates a string template record definition with that channel ($tmpl). The record definition is derived from the dictionary. An array of fields can be returned with the dsListFlds function to assist in application development.

The channel returned by the dsOpenDict function is used for all I/O related to the opened file. For example, the $chan value is used for I/O for the CUSTOMERS file, and the $slspchan value is used for I/O for the SALEPRSN file.

Another function shown here is dsGetIndexes, which returns an array of descriptions of the primary and secondary keys found for the channel.

```
print "\n\n";
print "Files Opened For Sample\n";
print "======================\n";
# Open the three files needed for this sample
dsOpenDict($handle,$dict,'CUSTOMERS',$chan,$path,$tmpl);
dsOpenDict($handle,$dict,'SALESPRSN',$slspchan,$slsppath,$slsptmpl);
dsOpenDict($handle,$dict,'INVOICES',$invcchan,$invcpath,$invctmpl);

# Get a list of fields in each file
dsListFlds($handle,$chan,@flds);
dsListFlds($handle,$slspchan,@slspflds);
dsListFlds($handle,$invcchan,@invcflds);

@custidx=();
dsGetIndexes($handle,$chan,@custidx);

print "CUSTOMERS \t $chan \t $path \t ",join(",",@flds),"\n\n";
print "  indexes: ",join("; ",@custidx),"\n\n";
print "SALESPRSN \t $slspchan \t $slsppath \t ",join(",",@slspflds),"\n\n";

print "INVOICES \t $invcchan \t $invcpath \t ",join(",",@invcflds),"\n\n";
```

One of the most important concepts in Business Basic data files is that of the key pointer. A keyed file has a pointer that indicates the value of the next primary or secondary key that will be read. This is roughly analogous to a SQL cursor. This value can be changed with dsRead I/O functions, to move the pointer to a new location. There are also some functions to tell you where the current pointer is, as well as the values of the first and last keys in the file. These functions are shown here.

```
print "Primary Key Values of CUSTOMERS channel\n";
print "=======================================\n";
# Use key functions to show current, first, and last keys in CUSTOMERS
dsGetKey($handle,$chan,$curkey);
dsGetKeyFirst($handle,$chan,$firstkey);
dsGetKeyLast($handle,$chan,$lastkey);
print "Current $curkey\n";
print "First   $firstkey\n";
print "Last    $lastkey\n";
```

The next section of code performs a simple loop though the CUSTOMERS file. It performs an initial dsReadFld function to force the key pointer to the beginning of the file (key=’’, key number 0). It doesn’t care if an error is returned; in fact, and error 11 will occur and no data will be returned. The entire purpose is just to ensure that we start from the beginning of the file.

Next, a simple while loop is executed, until dsReadFldNext returns false, indicating an error occurred. This example assumes the error will be an end-of-file error (error 2). More complete code might be aware of different errors. The list of desired fields is provided in a comma-delimited string, and the last argument is an array that returns field values associated with the just-read record. The $ky variable returns the key of this record, which may be the primary key, or may be a secondary key if the last dsReadFld specified a key number other than 0.

```
print "\n";
print "Customer File Listing\n";
print "====================\n";
print "ID|Name|YTD Sales|Slsp\n";

# Start at the beginning of the file, by reading with a key of null, key number 0
@result=();
dsReadFld($handle,$chan,'',0,'',@result);
```

Launch Internet Explorer Browser.lnk

```
@flds=();
while ( dsReadFldNext($handle,$chan,$ky,'id,name,ytd_sales,slsp',@flds) ) {
     print "$ky: $flds[0]|$flds[1]|$flds[2]|$flds[3]\n"
}
```

The next section of code shows how to work with multiple files. There are three sample data files in use in this example: the main file is CUSTOMERS, which contains a salesperson ID that is the key to the SALESPRSN file. In addition, there is an INVOICES file whose primary key is the customer ID plus an invoice number. This code does a dsReadFld lookup on customer 00026, returning the ID, name, and slsp fields. It then uses the slsp field to do a dsReadFld lookup on the SALESPRSN file, returning the salesperson name. After that line is printed, it then uses the customer ID so process a range of records in the INVOICES file. Note the initial dsReadFld with a key of $custid. This will position the key pointer to the first invoice for customer 00026. The code then loops through the INVOICES channel until the key is no longer in the range of 00026… or an error occurs.

```
print "\n";
print "Customer Invoices, Using Multiple Files\n";
print "=======================================\n";

@custflds=();
@slspflds=();
@invcflds=();

# A sample key to use
$custid='00026';

if (! dsReadFld($handle,$chan,$custid,0,'id,name,slsp',@custflds)) {
     print "Error $dserr ($dserrmsg) reading $key from CUSTOMERS.\n";
} else {

     # Read saleperson record using slsp field from customer record
     if (! dsReadFld($handle,$slspchan,$custflds[2],0,'name',@slspflds)) {
          $slspflds[0]="$custflds{slsp} not in file";
     }

     print "$custflds[0] | ";
     print "$custflds[1] | ";
     print "$custflds[2] | ";
     print "$slspflds[0]\n";

     print "Invoice|Date|Amount\n";

     # Set invoice channel to key position 00026
     dsReadFld($handle,$invcchan,$custid,0,'',@invcflds);
```

```
     # Read invoice channel until end of file...
     READINVC: while ( dsReadFldNext($handle,$invcchan,$key,'invoice_no,invoice_date,amount',@invcflds) ) {

          # or the end of customer 00026.
          # The key (first 5 bytes) can be used to determine the customer.
          # You could also return the id field and check that.
          if ( substr($key,0,5) != $custid ) {last READINVC;}

          print "$invcflds[0]|$invcflds[1]|$invcflds[2]\n";
          next READINVC
     }
}
```

This last section of code demonstrates how to update a field in a file. This code first gets the current salesperson ID for customer 00013. It then does some simplistic logic to change this ID, and writes it back. Before writing it back, it does a dsExtractRec of the customer record. This serves two purposes: it locks the record, preventing another use from updating the record while this update is executed, and it also provides the balance of information for the record, so that the dsWriteFld function can update just the salesperson field and not be concerned with what happens to the other fields in the record. The dsWriteFld arguments include a primary key, a record string, a comma-delimited list of field names, and an associated array of field values. A write, as well as any other I/O to a channel, releases the lock that the Extract function placed on the record.

```
print "\n";
print "Update Salesperson Value for a Customer\n";
print "=======================================\n";

$key='00013';

# First, get the current slsp value

@custflds=(); #(slsp=>'');
if ( dsReadFld($handle,$chan,$key,0,'slsp',@custflds) ) {
     $origslsp=$custflds[0];
     print ">Current SLSP value for customer $key is $origslsp\n";

     # Rotate slsp code (hey, it's just a sample)
     if ($origslsp eq '100') {$newslsp='101';}
     if ($origslsp eq '101') {$newslsp='110';}
     if ($origslsp eq '110') {$newslsp='100';}

     # Extract (lock) full record
     $resp=dsExtractRec($handle,$chan,0,$key,$record);
     if ($resp) {
          print ">Extracted (locked) record for $key\n";

          #If successful, update slsp to new value
          @updatefelds=($newslsp);
          $update=dsWriteFld($handle,$chan,$key,$record,'slsp',@updatefelds);

          if ($update) {
               print ">Salesperson in $key changed from $origslsp to $newslsp\n";
               print ">Reloading page will reflect change in customer $key\n";
          } else {
               print ">Error dserr updating record\n";
          }
     } else {
          print ">Error dserr extracting/locking record\n";
     }
}
```

The last step should always be a dsDisconnect function, though if the program ends, the socket is automatically closed, and there is an effective disconnect.

```
dsDisconnect $handle;
```

PHP: sample.php

To execute this sample, first copy the dsphp.inc module to one of your system's php include_path directories (or change the sample.php script's **require** line to specify a full path). Modify the $server and $port lines if needed (if dSERVE is running on a different machine on your network, or if the listening port has been changed from its default in the dserve.ini file on the server). Then copy sample.php to a web server scripting directory and load it's URL in a web browser. If you have php configured for automatic invocation when a .php file is requested, then you should get an HTML page with data derived from the dSERVE sample files.

```
<html>

<?php

// SETUP INSTRUCTIONS:
// Copy this PHP script to an executable CGI directory on your webserver.  The
// server should be configured to run php when a file with a .php extension is
// specified in a browser location.  Also make sure to copy dsphp.inc to one of
// your local include_files paths, or else change the 'require' command to use
// a full path.
//
// Modify the $server= value, and if necessary, the $port value, so that this
// script can connect to the dSERVE 2.0 server.  The default installation includes
// sample files and dictionaries used by this script.
//
```

The dServe.pm module needs to be loaded with the use or require statement. This module exports the dsXXX functions as well as the $dserr and $dserrmsg variables. All functions return true (1) on success, and false (0) on error. If an error occurs, you can check the $dserr and $dserrmsg variables for information.

```
// Include the dsphp.inc class library from the include_files path
require 'dsphp.inc';

// Change these values to point to the correct server/port, if necessary.
$server=localhost;
$port=8227;
$timeout=10;
```

This code creates an instance of the dserve class ($d), and then connects to the server. If successful, the response variable $connected will be true, and $d->xxx will reference methods (functions) or properties (variables) found in the dserve class.

```
// Create a new dserve class and connect to the server
$d=new dserve;
$connected=$d->dsConnect($server,$port,$timeout);
```

The following code is designed to open the correct sample dictionary, depending on which version of dSERVE is running: BBx or ProvideX. The value of the global string $sfx contains ".bb" for BBx or ".pv" for ProvideX. Given that information, the sample uses the correct dictionary in the variable $dict. In your own code, you would simply use the dictionary required by your application.

```
if (!$connected) {
     // Oops - an error trying to connect
     echo "<h2>Error trying to connect: $d->dserr ($d->dserrmsg)</h2>\n";

} else {
     // Demo dictionary is based on the server interpreter (bb or pv)
     // .bb=dserve.tpm, .pv=pvxdata/providex.ddf
     $d->dsGetGbl('$sfx',$bb);
     $dict='dserve.tpm';
     if ( $bb == ".pv" ) {
          $dict='pvxdata/providex.ddf';
```

```
     }
}

?>

<h1>dSERVE 2 Sample PHP Program</h1>

<p><p>
<table border=1>
<tr><th colspan=2 bgcolor=#c0c0c0>Files in Demo Dictionary</th></tr>
```

The dsListDict function returns a 2-dimensional array of file names and descriptions. It requires a handle, a dictionary name, a wildcard search string, and an array argument.

```
<?php
if ($connected) {
     // List files in the demo dictionary
     $d->dsListDict($dict,'',$files);
     for ($i=0; $i<count($files); $i++) {
          echo "<tr><td> {$files[$i][0]} </td> <td> {$files[$i][1]} </td></tr>\n";
     }
}
?>
</table>

<p><p>

<table border=1>
<tr><th colspan=4 bgcolor=#c0c0c0>Files Opened for Sample</th></tr>
<tr> <th>File</th> <th>Channel</th> <th>Path</th> <th>Fields</th> </tr>
```

The next section opens three files using the dsOpenDict function. This function uses the dictionary to open channel ($chan) to a physical disk file ($path), and associates a string template record definition with that channel ($tmpl). The record definition is derived from the dictionary. An array of fields can be returned with the dsListFlds function to assist in application development.

The channel returned by the dsOpenDict function is used for all I/O related to the opened file. For example, the $chan value is used for I/O for the CUSTOMERS file, and the $slspchan value is used for I/O for the SALEPRSN file.

Another function shown here is dsGetIndexes, which returns an array of descriptions of the primary and secondary keys found for the channel.

```
<?php
if ($connected) {
     // Open the three files needed for this sample
     $d->dsOpenDict($dict,'CUSTOMERS',$chan,$path,$tmpl);
     $d->dsOpenDict($dict,'SALESPRSN',$slspchan,$slsppath,$slsptmpl);
     $d->dsOpenDict($dict,'INVOICES',$invcchan,$invcpath,$invctmpl);

     // Get a list of fields in each file
     $d->dsListFlds($chan,$flds);
     $d->dsListFlds($slspchan,$slspflds);
     $d->dsListFlds($invcchan,$invcflds);

     echo "<tr>\n";
     echo " <td valign=top>CUSTOMERS</td> <td valign=top>$chan</td> <td valign=top>$path</td>";
     echo " <td>",implode($flds,", "),"</td>";
     echo "</tr>\n";

     echo "<tr>\n";
     echo " <td valign=top>SALESPRSN</td> <td valign=top>$slspchan</td> <td valign=top>$slsppath</td>";
     echo " <td>",implode($slspflds,", "),"</td>";
     echo "</tr>\n";

     echo "<tr>\n";
     echo " <td valign=top>INVOICES</td> <td valign=top>$invcchan</td> <td valign=top>$invcpath</td>";
```

```
     echo " <td>",implode($invcflds,", "),"</td>";
     echo "</tr>\n";

}
?>
</table>

<p><p>

<table border=1)
<tr><th colspan=2 bgcolor=#c0c0c0>Primary Key Values of CUSTOMERS channel</th></tr>
<tr><th>Key</th><th>Value</th></tr>
```

One of the most important concepts in Business Basic data files is that of the key pointer. A keyed file has a pointer that indicates the value of the next primary or secondary key that will be read. This is roughly analogous to a SQL cursor. This value can be changed with dsRead I/O functions, to move the pointer to a new location. There are also some functions to tell you where the current pointer is, as well as the values of the first and last keys in the file. These functions are shown here.

```
<?php
if ($connected) {
     // Use key functions to show current, first, and last keys in CUSTOMERS
     $d->dsGetKey($chan,$curkey);
     $d->dsGetKeyFirst($chan,$firstkey);
     $d->dsGetKeyLast($chan,$lastkey);
     echo "<tr><td>Current</td><td>$curkey</td></tr>\n";
     echo "<tr><td>First</td><td>$firstkey</td></tr>\n";
     echo "<tr><td>Last</td><td>$lastkey</td></tr>\n";
}
?>
</table>

<p><p>

<table border=1 width=100%>
<tr><th colspan=4 bgcolor=#c0c0c0>Customer File Listing</th></tr>
<tr><th>ID</th><th>Name</th><th>YTD Sales</th><th>Slsp</th></tr>
```

The next section of code performs a simple loop though the CUSTOMERS file. It performs an initial dsReadFld function to force the key pointer to the beginning of the file (key=’’, key number 0). It doesn’t care if an error is returned; in fact, an error 11 will occur and no data will be returned. The entire purpose is just to ensure that we start from the beginning of the file.

Next, a simple while loop is executed, until dsReadFldNext returns false, indicating an error occurred. This example assumes the error will be an end-of-file error (error 2). More complete code might be aware of different errors. The list of desired fields is provided in a comma-delimited string, and the last argument is an array that returns field values associated with the just-read record. The $ky variable returns the key of this record, which may be the primary key, or may be a secondary key if the last dsReadFld specified a key number other than 0.

```
<?php
if ($connected) {
     // Start at the beginning of the file, by reading with a key of null, key number 0
     $d->dsReadFld($chan,'',0,'',$result);

     // Loop through the file, getting four named fields
     // An error, such as end-of-file, will return False, terminating the while loop
     while ( $d->dsReadFldNext($chan,'id,name,ytd_sales,slsp',$key,$vals) ) {

          echo "<tr> ";
          echo " <td>",$vals[0],"</td>";
          echo " <td>",$vals[1],"</td>";
          echo " <td align=right>",$vals[2],"</td>";
          echo " <td>",$vals[3],"</td>";
          echo "</tr>\n";
```

```
     }
}
?>
</table>

<p><p>

<table border=1 width=100%>
<tr><th colspan=4 bgcolor=#c0c0c0>Customer/Invoice Listing For Customer 00026</th></tr>
<tr><th>ID</th><th>Name</th><th>Slsp ID</th><th>Slsp Name</th></tr>
```

The next section of code shows how to work with multiple files. There are three sample data files in use in this example: the main file is CUSTOMERS, which contains a salesperson ID that is the key to the SALESPRSN file. In addition, there is an INVOICES file whose primary key is the customer ID plus an invoice number. This code does a dsReadFld lookup on customer 00026, returning the ID, name, and slsp fields. It then uses the slsp field to do a dsReadFld lookup on the SALESPRSN file, returning the salesperson name. After that line is printed, it then uses the customer ID so process a range of records in the INVOICES file. Note the initial dsReadFld with a key of $custid. This will position the key pointer to the first invoice for customer 00026. The code then loops through the INVOICES channel until the key is no longer in the range of 00026… or an error occurs.

```
<?php
if ($connected) {
     // Read customer key 00026
     if (! $d->dsReadFld($chan,'00026',0,'id,name,slsp',$custflds)) {
          $custflds=array("Error $d->dserr","","");
     }

     // Read saleperson record using slsp field from customer record
     if (! $d->dsReadFld($slspchan,$custflds[2],0,'name',$slspflds)) {
          $slspflds=array("$custflds[2] not in file");
     }

     echo "<tr>";
     echo " <td>$custflds[0]</td>";
     echo " <td>$custflds[1]</td>";
     echo " <td>$custflds[2]</td>";
     echo " <td>$slspflds[0]</td>";
     echo "</tr>\n";

     echo "<tr><td colspan=4>\n";
     echo " <table border=1>\n";
     echo "  <tr> <th>Invoice</th> <th>Date</th> <th>Amount</th> </tr>\n";

     // Set invoice channel to key position 00026
     $d->dsReadFld($invcchan,'00026',0,'',$trash);

     // Read invoice channel until end of file...
     while ( $d->dsReadFldNext($invcchan,'invoice_no,invoice_date,amount',$key,$invcflds) ) {

          // or the end of customer 00026.
          // The key (first 5 bytes) can be used to determine the customer.
          // You could also return the id field and check that.
          if ( substr($key,0,5) != '00026' ) break;

          echo "  <tr>";
          echo "   <td>$invcflds[0]</td>";
          echo "   <td>$invcflds[1]</td>";
          echo "   <td align=right>$invcflds[2]</td>";
          echo "  </tr>\n";
     }

     echo " </table>\n";
}
?>
</table>

<p><p>
```

This last section of code demonstrates how to update a field in a file. This code first gets the current salesperson ID for customer 00013. It then does some simplistic logic to change this ID, and writes it back. Before writing it back, it does a dsExtractRec of the customer record. This serves two purposes: it locks the record, preventing another use from updating the record while this update is executed, and it also provides the balance of information for the record, so that the dsWriteFld function can update just the salesperson field and not be concerned with what happens to the other fields in the record. The dsWriteFld arguments include a primary key, a record string, a comma-delimited list of field names, and an associated array of field values. A write, as well as any other I/O to a channel, releases the lock that the Extract function placed on the record.

```
<?php
if ($connected) {
     // This section updates the salesperson in customer 00013
     echo "<ul>\n";
     $key='00013';

     // First, get the current slsp value
     $d->dsReadFld($chan,$key,0,'slsp',$vals);
     $origslsp=$vals[0];
     echo " <li>Current SLSP value for customer $key is $origslsp\n";

     // Rotate slsp code (hey, it's just a sample)
     if ($origslsp == '100') $newslsp='101';
     if ($origslsp == '101') $newslsp='110';
     if ($origslsp == '110') $newslsp='100';

     // Extract (lock) full record
     $resp=$d->dsExtractRec($chan,0,$key,$record);
     if ($resp) {
          echo " <li>Extracted record for $key\n";

          //If successful, update slsp to new value
          $update=$d->dsWriteFld($chan,$key,$record,'slsp',array($newslsp));

          if ($update) {
               echo "<li>Salesperson in $key changed from $origslsp to $newslsp\n";
               echo "<li>Reloading page will reflect change in customer $key\n";
          } else {
               echo "<li>Error $d->dserr updating record\n";
          }
     } else {
          echo "<li>Error $d->dserr extracting/locking record\n";
     }

     echo "</ul>\n";
}
?>

<p><p>
```

The last step should always be a dsDisconnect function, though if the program ends, the socket is automatically closed, and there is an effective disconnect.

```
<?php
if ($connected) {
     $d->dsdisconnect();
}
?>

</html>
```

VBScript: samplevbs.htm

To execute this sample, first install the dSERVE 2 COM object on the Windows client machine(s) where you will browse the sample with Internet Explorer, and copy the samplevbs.htm file to a directory served by a local web server. Optionally, you can also simply open file sample off the file system in Internet Explorer, rather than serving it from a web server.

Modify the varServer and varPort settings if necessary, in order for the sample to connect to the correct machine where dSERVE's server is running.

```
<html>
<head>

<meta name="VI60_defaultClientScript" content="VBScript">
<meta name="GENERATOR" content="Microsoft FrontPage 5.0">
<meta name="ProgId" content="FrontPage.Editor.Document">

<script language="vbscript">

dim dSERVEVBS, varConnected
dim varServer, varPort, varTimeout, varDictionary
dim varChannel, varPathname, varTemplate, varFields
dim varSlspChannel, varSlspPathname, varSlspTemplate, varSlspFields
dim varInvChannel, varInvPathname, varInvTemplate, varInvFields

varServer = "localhost"
varPort = 8227
varTimeout = 10000
varDictionary = "dserve.tpm"

</script>

</head>

<body>
```

This code creates a dSERVE2.dSERVEVBS object, and then connects to the server. If successful, the response variable varConnected will be –1. If an error occurs, varConnected with be >= 0. There are two primary objects in the dSERVE2 ActiveX DLL: dSERVE and dSERVEVBS. The primary difference is in the support of Variant data types in the arguments of dSERVEVBS. In some cases, methods return arrays in dSERVE and delimited strings in dSERVEVBS.

After connecting, this section of code also determines if the server is running BBx or ProvideX by looking at the global string "$sfx", and sets the proper dictionary for the sample.

```
<script language="vbscript">

    set dSERVEVBS = CreateObject("dSERVE2.dSERVEVBS")
    varConnected = dSERVEVBS.dsConnect(varSERVER,varPort,varTimeout)

    If Not varConnected = -1 Then
        document.write "<h2>Error trying to connect: " & dSERVEVBS.LastErrorNumber & ": "
& dSERVEVBS.LastErrorMessage
```

```
    else
        dim varValue
        dSERVEVBS.dsGetGbl “$sfx”,varValue
        dSERVEVBS.Dictionary = varDictionary
        If varValue = ".pv" Then dSERVEVBS.Dictionary = "pvxdata/providex.ddf"
    End If

</script>

<h1>sdSERVE.dSERVEVBS Class Sample</h1>

<p><p>
<table border=1>
<tr><th colspan=2 bgcolor=#c0c0c0>Files in Demo Dictionary</th></tr>
```

The dsListDict function returns two delimited strings (this is one of the differences between the dSERVEVBS and dSERVE objects; dSERVE woule return a 2-dimensional array of file names and descriptions). It requires a handle, a dictionary name, a wildcard search string, and two ByRef arguments. The files and descriptions are returned as associated strings delimited with Chr(0) values, so it is easy to use Split functions to parse them into more usable arrays.

```
<script language="vbscript">

    if varConnected = -1 then

        'List files/tables in the dSERVE.tpm demo dictionary
        dim varFiles, arrFiles, varDescriptions, arrDescriptions
        dim result, i

        result = dSERVEVBS.dsListDict(varDictionary,"",varFiles,varDescriptions)

        arrFiles = split(varFiles,chr(0),-1,1)
        arrDescriptions = split(varDescriptions,chr(0),-1,1)

        for i = 0 to ubound(arrFiles)
            document.write "<tr>"
            document.write "<td width='50%'>" & arrFiles(i)& "</td>"
            document.write "<td width='50%'>" & arrDescriptions(i)& "</td>"
            document.write "</tr>"
        next

    End if

</script>
</table>

<table border=1>
<tr><th colspan=4 bgcolor=#c0c0c0>Files Opened for Sample</th></tr>
<tr> <th>File</th> <th>Channel</th> <th>Path</th> <th>Fields</th> </tr>
```

The next section opens three files using the dsOpenDict function. This function uses the dictionary to open channel (varChannel) to a physical disk file (varPathname), and associates a string template record definition with that channel (varTemplate). The record definition is derived from the dictionary. A list of fields can be returned with the dsListFlds function to assist in application development.

The channel returned by the dsOpenDict function is used for all I/O related to the opened file. For example, the $chan value is used for I/O for the CUSTOMERS file, and the $slspchan value is used for I/O for the SALEPRSN file.

```
<script language="vbscript">

    dim arrFields, strFields, i

    'Open the three files needed for this sample
    dSERVEVBS.dsOpenDict varDictionary,"CUSTOMERS",varChannel,varPathname,varTemplate
    dSERVEVBS.dsOpenDict
varDictionary,"SALESPRSN",varSlspChannel,varSlspPathname,varSlspTemplate
    dSERVEVBS.dsOpenDict
varDictionary,"INVOICES",varInvChannel,varInvPathname,varInvTemplate

    'Get a list of fields in each file
    dSERVEVBS.dsListFlds varChannel,varFields
    dSERVEVBS.dsListFlds varSlspChannel,varSlspFields
    dSERVEVBS.dsListFlds varInvChannel,varInvFields

    'Customer Fields
    strFields = ""
    arrFields = split(varFields,chr(0),-1)
    strFields = join(arrFields,", ")

    document.write "<tr>"
    document.write " <td valign=top>CUSTOMERS</td> <td valign=top>" & varChannel & "</td>
<td valign=top>" & varPathname & "</td>"
    document.write " <td>" & strFields & "</td>"
    document.write "</tr>"

    'Salesperson Fields
    strFields = ""
    arrFields = split(varSlspFields,chr(0),-1)
    strfields = join(arrfields,", ")

    document.write "<tr>"
    document.write " <td valign=top>SALESPRSN</td> <td valign=top>" & varSlspChannel &
"</td> <td valign=top>" & varSlspPathname & "</td>"
    document.write " <td>" & strFields & "</td>"
    document.write "</tr>"

    'Salesperson Fields
    strFields = ""
    arrFields = split(varInvFields,chr(0),-1)
    strFields = join(arrFields,", ")

    document.write "<tr>"
    document.write " <td valign=top>INVOICES</td> <td valign=top>" & varInvChannel &
"</td> <td valign=top>" & varInvPathname & "</td>"
    document.write " <td>" & strFields & "</td>"
    document.write "</tr>"

</script>

</table>

<table border=1)
<tr><th colspan=2 bgcolor=#c0c0c0>Primary Key Values of CUSTOMERS channel</th></tr>
<tr><th>Key</th><th>Value</th></tr>
```

One of the most important concepts in Business Basic data files is that of the key pointer. A keyed file has a pointer that indicates the value of the next primary or secondary key that will be read. This is roughly analogous to a SQL cursor. This value can be changed with dsRead I/O functions, to move the pointer to a new location. There are also some functions to tell you where the current pointer is, as well as the values of the first and last keys in the file. These functions are shown here.

```
<script language=vbscript>
<!--
    'Retrieve Primary Key Values
    Dim varPrimaryKey, varFirstKey, varLastKey

    dSERVEVBS.dsGetKey varChannel,varPrimaryKey
    dSERVEVBS.dsGetKeyFirst varChannel,varFirstKey
    dSERVEVBS.dsGetKeyLast varChannel, varLastKey

    Document.write "<tr><td>Current</td><td>" & varPrimaryKey &
"</td></tr><tr><td>First</td><td>" & varFirstKey & "</td></tr><tr><td>Last</td><td>" &
varLastKey & "</td></tr>"

-->
  </script>
</table>


<table border=1 width=100%>
<tr><th colspan=4 bgcolor=#c0c0c0>Customer File Listing</th></tr>
<tr><th>ID</th><th>Name</th><th>YTD Sales</th><th>Slsp</th></tr>
```

The next section of code performs a simple loop though the CUSTOMERS file. It performs an initial dsReadFld function to force the key pointer to the beginning of the file (key=””, key number 0). It doesn't care if an error is returned; in fact, an error 11 will occur and no data will be returned. The entire purpose is just to ensure that we start from the beginning of the file.

Next, a simple while loop is executed, until dsReadFldNext returns false, indicating an error occurred. This example assumes the error will be an end-of-file error (error 2). More complete code might be aware of different errors. The list of desired fields is provided in a comma-delimited string, and the last argument is a string that returns Chr(0) delimited field values associated with the just-read record. The varKey variable returns the key of this record, which may be the primary key, or may be a secondary key if the last dsReadFld specified a key number other than 0.

```
<script language="vbscript">

    dim varValues, arrValues, varKey

    dSERVEVBS.dsReadFld varChannel,"",0,"",varValues

    Do While
dSERVEVBS.dsReadFldNext(varChannel,"id,name,ytd_sales,slsp",varKey,varValues) = -1
        arrValues = split(varValues,chr(0),-1)
        document.write "<tr> "
        document.write " <td>" & arrValues(0) & "</td>"
        document.write " <td>" & arrValues(1) & "</td>"
        document.write " <td align=right>" & arrValues(2) & "</td>"
        document.write " <td>" & arrValues(3) & "</td>"
        document.write "</tr>"
    loop
```

```
</script>
</table>

<table border=1 width=100%>
<tr><th colspan=4 bgcolor=#c0c0c0>Customer/Invoice Listing For Customer 00026</th></tr>
<tr><th>ID</th><th>Name</th><th>Slsp ID</th><th>Slsp Name</th></tr>
```

The next section of code shows how to work with multiple files. There are three sample data files in use in this example: the main file is CUSTOMERS, which contains a salesperson ID that is the key to the SALESPRSN file. In addition, there is an INVOICES file whose primary key is the customer ID plus an invoice number. This code does a dsReadFld lookup on customer 00026, returning the ID, name, and slsp fields. It then uses the slsp field to do a dsReadFld lookup on the SALESPRSN file, returning the salesperson name. After that line is printed, it then uses the customer ID so process a range of records in the INVOICES file. Note the initial dsReadFld with a key of $custid. This will position the key pointer to the first invoice for customer 00026. The code then loops through the INVOICES channel until the key is no longer in the range of 00026… or an error occurs.

```
<script language="vbscript">

    dim varCustFields,arrCustFields'Customers File
    dim varSlspFields,arrSlspFields'Salespersons File
    dim varInvFields,arrInvFields'Invoices File
    dim varTrash

    'Read customer key 00026
    if Not dSERVEVBS.dsReadFld(varChannel,"00026",0,"id,name,slsp",varCustFields) = -1
Then
        arrCustFields = array("Error dSERVEVBS.dserr","","")
    else
        arrCustFields = split(varCustFields,chr(0),-1)
    end if

    document.write var
    'Read saleperson record using slsp field from customer record
    if Not dSERVEVBS.dsReadFld(varSlspChannel,arrCustFields(2),0,"name",varSlspFields) =
-1 Then
        arrSlspFields = array("varCustFields(2) not in file")
    else
        arrSlspFields = split(varSlspFields,chr(0),-1)
    end if

    document.write "<tr>"
    document.write " <td>" & arrCustFields(0) & "</td>"
    document.write " <td>" & arrCustFields(1) & "</td>"
    document.write " <td>" & arrCustFields(2) & "</td>"
    document.write " <td>" & arrSlspFields(0) & "</td>"
    document.write "</tr>"

    document.write "<tr><td colspan=4>"
    document.write " <table border=1>"
    document.write "  <tr> <th>Invoice</th> <th>Date</th> <th>Amount</th> </tr>"

    'Set invoice channel to key position 00026
    dSERVEVBS.dsReadFld varInvChannel,"00026",0,"",varTrash

    'Read invoice channel until end of file...
```

```
    do while
dSERVEVBS.dsReadFldNext(varInvChannel,"invoice_no,invoice_date,amount",varKey,varInvField
s) = -1

        arrInvFields = split(varInvFields,chr(0),-1)

        'or the end of customer 00026.
        'The key (first 5 bytes) can be used to determine the customer.
        'You could also return the id field and check that.
        if Not mid(varKey,1,5)="00026" Then Exit Do

        document.write "  <tr>"
        document.write "   <td>" & arrInvFields(0) & "</td>"
        document.write "   <td>" & arrInvFields(1) & "</td>"
        document.write "   <td align=right>" & arrInvFields(2) & "</td>"
        document.write "  </tr>"
    loop

    document.write " </table>"

    dSERVEVBS.dsDisconnect
    Set dSERVEVBS = nothing

</script>

</td>  </tr> </table>
</table>

</body>

</html>
```