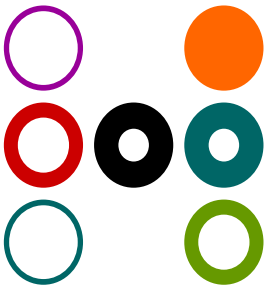
Essexer User Guide And Reference

Version 2.20 - Production

Igor Urisman



Hyperion®

/doc/gd_ref20.fm[.pdf]

Revision History

Date	Author	Revision
20 Jun 1997	Igor	Creation. Extracted from the QA Process Improvement document.
02 Sep 1997	Igor	Latest information; reference -> 2.13.
05 Oct 1997	Igor	Naming conventions.
10 Jan 1998	Igor	Update for 2.15.
10 Feb 1998	Igor	More updates for 2.15: cin, cout, clone.
22 Sep 1998	Igor	Clean-up, Update for 2.18, Hyperion stuff.
25 Jan 1999	Misha	Reprinted, man pages updated, pagination fixed, pdf created
01 Jul 1999	Misha	Man pages added and updated for 2.20, readme updated

Table of Contents

1 Introduction

Essexer is the standard Hyperion Essbase testing platform which consists of three main parts. Most of its functions are accessible through a set of commands which all begin with the *sxr* prefix followed by the actual command with a list of arguments. For example, to create a database we use this command:

```
sxr newdb sample basic
```

which will create new database named *Basic* in the existing application *Sample*.

Framework Repository

Throughout this document we will refer to Essexer programmatic facilities as the *Framework*. In addition to being a framework, Essexer is also a *repository* of tests which contains tens of thousands of files organized in thousands of tests. These tests can only be manipulated via the Essexer framework.

Paradigm

In addition to being a framework and a repository, Essexer is also a method, a *paradigm*, of developing tests which take full advantage of its programmatic facilities and deliver predictable and consistent test behavior. For instance, all tests must follow the basic self-verification model and verify their own results by producing output log files and comparing them with the base line output log files which are known to be correct and are stored in the repository.

The Essexer framework is inseparable from the Essexer paradigm. Indeed, the framework provides the means to the test developer to help her abide by the paradigm. In fact, there is nothing in the framework that a clever test designer could not do without and still follow the paradigm.

1.1 The Essexer Paradigm

1.1.1 Location Transparency

Any computer on the company's network, regardless of its operating system, can run any Essexer test. Furthermore, no matter what the test, the running of it requires no setup time, no code transfer, and minimal disk space. Here are some examples when location transparency is beneficial:

- Any participant of the product development cycle needs to be able to run any test on her own computer with her own local build. She can use an existing test to verify her latest code changes before checking them in, or run a bug case against a production build to debug the defect.
- Tests are stored centrally and are brought to a local computer at execution time. This allows for a standard revision control system to be implemented as the back store for Essexer's test repository.

To achieve location transparency, tests do not refer to files by their absolute path names, only by their short names. Instead, it is the Essexer framework that carries out its own file name expansion rules and finds the right path to the file at run time. Usually, this path leads across the network to the Essexer repository.

1.1.2 Platform Independence

Essexer provides mechanisms to develop platform independent tests which run on all operating systems supported by the Essbase server. This portability is based on the use of Korn shell, which is available on all Unix platforms natively and as a third party emulator on PC platforms.

In addition to using Korn shell, Essexer encapsulates all platform variations, presenting the test with a uniform interface. For instance, if a test needs to know the host name it is running on, it uses the environment variable SXR_HOST, maintained by the framework, instead of issuing OS specific commands. These commands are different from platform to platform and it would be very costly to expect tests to implement the clunky logic needed to find out the host name each time they need it. Instead, the framework does it once, and sets the SXR_HOST environment variable which can be used in an OS independent fashion.

Tests also do not use Essbase services directly because of the variance in the Essbase interface from platform to platform. For instance, to run an ESSCMD script, a test uses the `sxr esscmd` command, which hides all platform variations as well as performs the script file name expansion mentioned above. Essexer usually also provides some valuable extra services, unavailable in the original interface. Thus the `sxr esscmd` command supports substitution variables, a facility unavailable in ESSCMD.

1.1.3 Atomicity of Tests

Essexer provides simple ways to efficiently create applications and databases as part of a test's execution. Therefore, test setup becomes part of the test, eliminating the need to keep around any objects between two executions of the test. In other words, the Essexer paradigm requires and the Essexer framework provides simple means for developing tests which are atomic in that their execution does not depend on pre-existence or non-existence of any objects.

Following the principle of atomicity provides the following benefits:

- A test can be executed from anywhere without any setup.
- Correct execution of a group of tests does not depend on their order.
- User does not have to keep anything on her computer in case she will need to run the test again.

1.1.4 Self-Verification

A key way to limit the cost of finding and fixing bugs is limiting the cost of testing. This includes many activities, most notably the cost of test development, test execution and test analysis. The principle of self-verification strives to limit the cost of test analysis.

Generally speaking, any test which is deterministic in its outcome can be made verify its own results. To do so, a test produces various forms of output, e.g. ReportWriter reports, error files, export files, etc., which are then compared with previously stored reference copies of these files which are believed to be correct.

Essexer framework provides a convenient comparison mechanism, `sxr diff`, which, in addition to the regular file comparison utility diff(1), also implements the following useful logic:

- A successful comparison causes the creation of a success file (.suc).
- An unsuccessful comparison cases the creation of a failure file (.dif), which contains the difference between the test and the baseline logs in the form of the diff(1) output.
- If the test log wasn't found, the failure file contains a message to that effect.

1.1.5 Distributed Support

Prior to version 2.18, a test always assumed that the database was local., i.e. running on the same computer as the test. This gave test designer direct file system access to the physical database structures, as well as to the Essbase Agent.

Beginning with version 2.18, Essexer provides means to design tests that can run against remote database servers. For instance, the `sxr newapp` and the `sxr newdb` commands honor the SXR_DBHOST environment variable, which may be set, at run time, to an arbitrary host name. By default, the framework sets the SXR_DBHOST variable to the same host name as SXR_HOST, i.e. the computer on which the test is executing. However, prior to the test execution, it may be explicitly set to refer to any other host name.

1.1.6 Mobile Support

Essexer supports a model for operating in the disconnected mode, e.g. on a laptop computer temporarily disconnected from the company's network. For instance, it provides a simple way to clone an arbitrary test down onto a local computer, where it could be executed at a later time when it is disconnected from the network. The user of the test does not need to know what files constitute the test and where on the network they are located.

The cloning of tests is facilitated by the DNA files, which are produced by Essexer automatically during test execution. Each DNA file contains the file reference "trace" of a particular test, which is the list of files the test references during its execution.

1.2 A Case Study

To illustrate Essexer's main ideas, let us now consider a simple example, while deferring the more formal discussion until Chapter 2. We begin by looking at the Essexer repository, were all test files are stored. Its location is maintained by the

framework in the environment variable SXR_BASE.

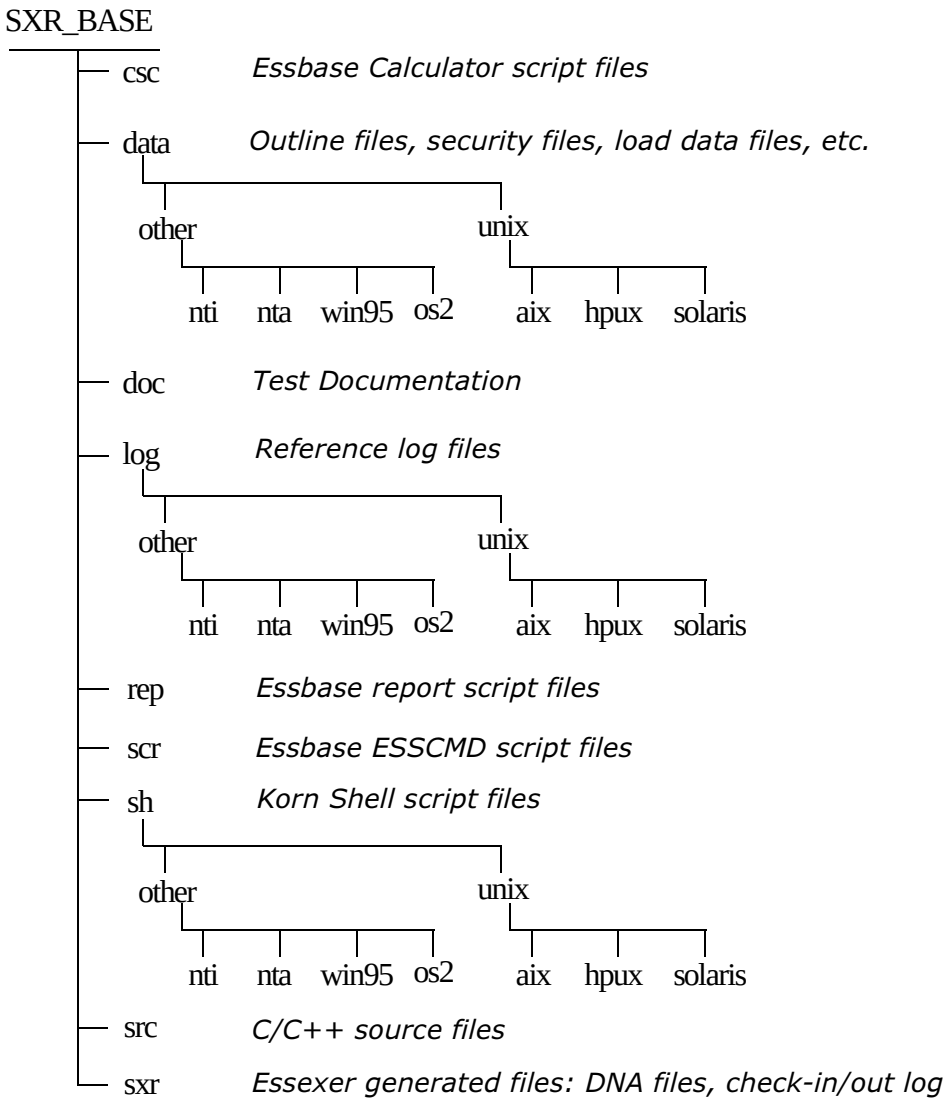


Fig 1. Directory structure of the Essexer repository.

Some of the directories in Fig. 1 contain a hierarchy of platform subdirectories designed to support platform-dependent versions of the same file. For example, if a test running on an IBM RS/6000 computer makes a reference to a log file named `accbdump.log`, Essexer will first look in the most platform dependent location, i.e. `$SXR_BASE/log/unix/aix` and then continue to `$SXR_BASE/log/unix`, and finally to the most platform generic directory `$SXR_BASE/log`.

Note, that we treat the DEC Alpha-based Windows NT as a separate platform (nta) from the Intel-based Windows NT (nti).

Here are the steps one takes to run an Essexer test, provided that Essexer framework been installed on the computer:

Sec 3.1.
Installing Essexer

sxr goview /home/views/beta2

Sec. 4.1. Essexer Views

Enter an Essexer view. `Sxr` is the common prefix for all Essexer command, and `goview` is the Essexer command used to enter an existing view, in this case located in `/home/views/beta2`. A new view is created with the `newview` Essexer command.

sxr shell accbmain.sh

Run test `accbmain.sh`. `Shell` is the Essexer command used to run executable shell scripts. Running of shell scripts which contain Essexer commands directly under the control of shell is not supported. In other words, simply typing `accbmain.sh` on the command line is a bad idea.

Sec. 3.5. Essexer Execution Logs...
Sec. 4.8. Analyzing Essexer Tests

Upon completion of the test, the `work` subdirectory of the view, where all test generated files are saved, will also contain the log of the run, named `accbmain.sog`, containing both the standard output and the standard error streams generated by the run. This file can be used to analyze the test run after it is completed.

Most tests are expected to verify their own results by comparing the output files they produce with their reference copies via the `sxr diff` command. Each such comparison produces either a difference (`.dif`) or a success (`.suc`) file, which is also stored in the work subdirectory of the current view.

2 The Concepts

2.1 Essexer Views

Most of Essexer commands are only available from an Essexer view. Entering a view initializes its context and establishes addressability to the Essexer repository. Formally, an Essexer view is

- A directory structure
- A set of environment variables
- A sub-shell

The directory structure of a view is similar to the structure of the base and is presented below in Fig. 2. The environment variables are recreated each time a view is re-entered and are destroyed when the view is exited. For a cleaner handling of the environment, `sxr goview` command creates a new sub-shell outside of which its Essexer environment variables are not visible. Exiting a view is achieved by terminating the sub-shell with the `exit` shell command.

Sec. 3.6. Essexer Environment Variables

The directory structure in Fig.2 constitutes the persistent part of the view and is created by the `sxr newview` command. As opposed to the repository directory structure (Fig. 1), there are no platform subdirectories in the view, because it resides on a particular computer and only serves tests running within it. Directories `bin` and `work` only exist in views, and contain files generated during test execution.

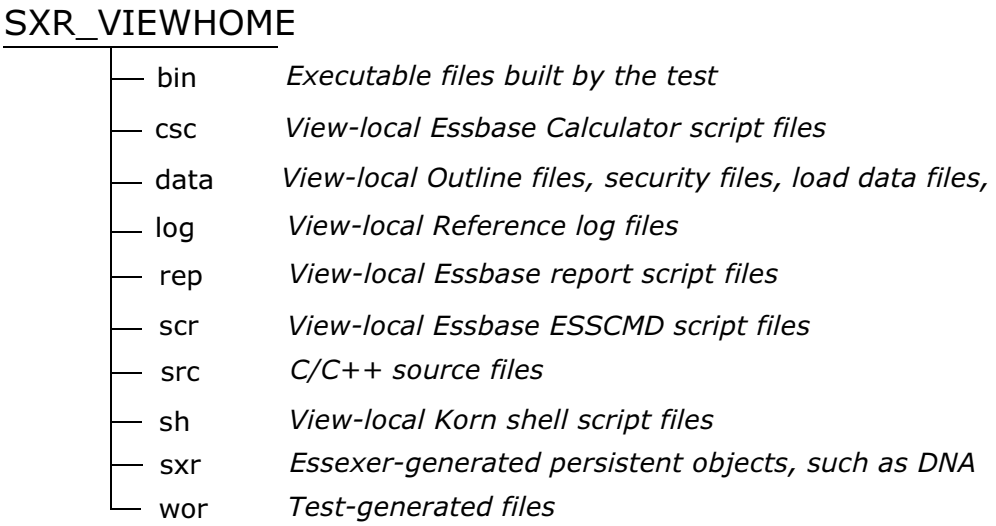


Fig 2. Directory structure of an Essexer view.

There are several reasons why we may wish to have a particular test file locally in the view, rather than in the repository. Most importantly, the repository is source-

Sec. 3.3. Test Source Control System

controlled and it's not possible to change files directly in the base. Rather, one has to first check the necessary file(s) out into a local view using the `sxr cout` command. Once the editing of the files has been completed, they can be checked back into the source control system using the `sxr cin` command. Therefore, the local view becomes a test development environment where one may develop new tests or modify existing ones without affecting the rest of the users.

For instance, we may wish to make changes to file `accxcalc.scr`, one of the ESSCMD script files used by the `accxmain.sh,` the Extended Acceptance test. To do that, we need to check out the file's latest revision into a local view by issuing the following command:

```
sxr cout -lock accxcalc.scr
```

Once the editing of the file has been completed, it can be validated by running the `accxmain.sh` test in the local view because the view-local file shadows the similarly named file in the base. Once the new revision of `accxcalc.scr` has been validated, it is checked back into the source control system:

```
sxr cin accxcalc.scr
```

`Cin` and `cout`, like most other Essexer commands, can only be issued from inside of an Essexer view.

Sec. 3.9. Mobile Support

Another reason for having test files in a local view, rather then running them directly off the Essexer repository, is mobility. By copying all of a test's files down into a local view, one eliminates the need to have the base location mounted on the local computer at the time of the test execution.

2.2 View Initialization File .sxrrc

Sec. 3.6. Essexer Environment Variables

When a user enters an Essexer view, by using the `sxr goview` command, Essexer creates a new sub-shell and initializes a set of environment variables which provide the run-time context to the tests and the Essexer framework. Many of these variables accept user-defined values, e.g. SXR_DBHOST, the host name of the computer running the Essbase server. While regular shell initialization files, like `.cshrc`, may be useful in automating such initialization, it is often advantageous to have view-specific environment customization such as, for instance, to define a specific trigger to only apply within the scope of a particular view. To aid in the administration of such view-specific environments, the view initialization file `.sxrrc` may be used.

Sec. 3.7. Essexer Post-action Triggers

The .sxrrc file is sourced[1] during the course of the `sxr goview` command, the `.sxrrc` file must reside in the root directory of the view and contain correct Korn shell syntax. The table below presents the list of user-definable environment variables, that will be honored by the subsequent Essexer environment initialization. The *Scope* column indicates whether a variable can be reset within an Essexer view

1. "Sourcing" is the Unix jargon for executing within the same Shell, as opposed to the regular execution, which conventionally takes place in a sub-shell which preserves the calling shell's environment from seeing any changes made by the sub-shell.

session to take effect for the next Essexer command, or if its value is only considered during the execution of the `sxr goview` command.

Variable	Default	Scope	Meaning
SXR_USER	`essexer`	agtctl, newapp, newdb	The essbase super-user.
SXR_PASSWORD	`password`	agtctl, newapp, newdb	The password of the above user.

SXR_ESSBASE	Unix:
	`ESSBASE password -b &`
	PC:
	`start essbase password -b`
		agtctl	The agent startup command.

SXR_ESSCMD	`ESSCMD`	esscmd	Alternate ESSCMD interpreter.
SXR_BASE	`$SXR_HOME/../base`	View session	Location of test repository.
SXR_DIFF_LOGDEL	Unset	diff	Causes all test log files which compared successfully against their base lines via the "sxr diff' command be removed.
SXR_DIFF_TRIGGER	Unset	diff	On-diff trigger. Fires once per 'sxr diff" command.
SXR_ESSCMD_TRIGGER	Unset	esscmd	On-esscmd trigger. Fires once per 'sxr esscmd' command..
SXR_ESSINIT_SECFILE	Unset	essinit command	Default security file to be used with 'sxr essinit' without the -sec flag.
SXR_GOVIEW_TRIGGER	Unset	View session	On-goview trigger. Fires once just after the view has been entered. Ignored if '-notrig' flag used with 'sxr goview'.
SXR_HOME	Must be set by user	N/A	Location of the Essexer framework installation.
SXR_DBHOST	`$SXR_HOST`	newapp, newdb, fcopy	The host name of the computer running the Essbase server.
SXR_NEWAPP_TRIGGER	Unset	newapp	On-newapp trigger. A separate trigger is fired for each application.

Variable	Default	Scope	Meaning
SXR_NEWDB_TRIGGER	Unset	newdb	On-newdb trigger. A separate trigger is fired for each database.
SXR_SHELL_TRIGGER	Unset	shell	On-shell trigger. Fires once for interactively invoked Shell scripts only.

2.3 File Name Expansion Based on Search Paths

Essexer provides means for development of location transparent tests, which is tests that can be run anywhere without any code modifications. The principal idea behind location transparency is to require tests to only reference files by their short names and let Essexer figure out the path to the file at the test execution time. To do that it uses path-based file name expansion model to translate short file names into fully qualified path names.

Sec. 3.5. Running ESSCMD scripts

Essexer uses explicit file typing. In other words, it categorizes files by their intended use. For instance, if we want to execute an ESSCMD script file `accxcalc.scr`, we use the following Essexer command:

```
sxr esscmd accxcalc.scr
```

From the context of the command Essexer understands that `accxcalc.scr` is an ESSCMD script file and knows to search for it in the appropriate directory list.

Essexer supports eight file types, and therefore has to maintain eight separate directory lists to search through. These search paths are maintained by Essexer in environment variables named SXR_PATH_*, which are initialized automatically upon entry into a view. These paths are:

Variable	Flag	Contents
SXR_PATH_CSC	-csc	Calculator script files.
SXR_PATH_DATA	-data	Various files not contained in other directories, such as outline files, load data files, security files, etc..
SXR_PATH_LOG	-log	Reference log files.
SXR_PATH_REP	-rep	Essbase report script files.
SXR_PATH_SCR	-scr	Essbase ESSCMD script files.
SXR_PATH_SRC	-src	C/C++ source files.
SXR_PATH_SH	-sh	Shell scripts, usually tests.
SXR_PATH_SXR	-sxr	Essexer-generated persistent files, such as the DNA files and the checkout log.

The "flag" column indicates the modifier that has to be given to the `sxr which` command which prints the full path to a file, as seen from a specific view. For instance, to find the path to the file `accxcalc.scr`, we type

```
sxr which -scr accxcalc.scr
```

Here's how this path based file name expansion works in conjunction with the platform specific subdirectories in the base. If, for instance, a certain test file, say a log file, is not identical across platforms and has a base version valid for all platforms except nti and an nti specific version, then we place the generic version in `$SXR_BASE/log/` and the nti-specific version in `$SXR_BASE/log/other/nti/`. When an Essexer view is entered on an NT computer, Essexer sets the `SXR_PATH_LOG` environment variable such that the log directories are scanned in the following order:

```
$SXR_VIEWHOME/log
$SXR_BASE/log/other/nti
$SXR_BASE/log/other
$SXR_BASE/log
```

Therefore, unless there is a view-local copy of the file, the nti specific version will be found before the generic one. Alternatively, on a Solaris computer the same path will look this way:

```
$SXR_VIEWHOME/log
$SXR_BASE/log/unix/solris
$SXR_BASE/log/unix
$SXR_BASE/log
```

Consequently, the generic version of the same file will be seen by exact same test running under Solaris.

Similarly to the shell path mechanism, no path lookup is performed for fully qualified file names, i.e. the command

```
sxr which -sh d:/some/shell/file
```

will attempt no name expansion and return precisely the file name itself, provided that it exists.

Essexer's path based name expansion also supports file name patterns. This is a powerful concept used by most Essexer commands. For instance, ESSCMD script `accbrep.scr` expects 37 ReportWriter scripts named `Accbc01.rep` through `Accbc37.rep` to be present in the database directory. The test can ensure that the files are copied there and then execute the ESSCMD script by doing the following:

```
sxr fcopy -in -rep 'Accb*.rep' -out !!!!Accept!Basic!
sxr esscmd Accbrep.scr
```

The fcopy command in this example locates all files matching the pattern on the SXR_PATH_REP path and copies them into the database directory *Basic* in application *Accept*. We are using a special Essexer command, rather then just copying the 37 files to `$ARBORPATH/app/Sample/Basic`, because the server is not

guaranteed to be running on the local computer. The sxr fcopy command uses Essbase's PutObject/GetObject mechanism whenever SXR_HOST and SXR_DBHOST refer to different host names.

Sec.3.7. Developing Distributed Tests

2.4 Essexer Substitution Variables

Sec. 1.3. The Essexer Paradigm

As a way of insuring location transparency, the tests do not refer to any files by their absolute path names. Instead of having to know where a file resides, the test lets the Essexer framework figure out the absolute file name at run time.

However, in cases when file references are made from inside an ESSCMD or ReportWriter scripts, additional facilities are needed to resolve absolute file name references at run time. Consider, for instance, an ESSCMD script that loads data into the database from file `stbbdata.txt`. When using the `loaddata` command we have a choice of either placing the file in the database directory where the Essbase server will be able to find it, or by specifying its absolute pathname. The latter is outright unacceptable as a violation of the location independence principle. The problem with the former is more subtle. Indeed, we could have the following code fragments:

- **Calling Shell script:**

```
# Create application and database. Both the newapp and
# the newdb commands honor the value of the SXR_DBHOST
# variable.
sxr newapp stbb_app
sxr newdb -otl stbb.otl stbb_app stbb_db

# Locate file stbbdata.txt on the data path and move it
# to the database directory:
sxr fcopy -in -data stbbdata.txt \
        -out !stbb_app!stbb_db!

# Load data into stbb_db:
sxr esscmd sxbbdtld.scr
```

- **`sxbbdtld.scr` ESSCMD script:**

```
Login "local" "essexer" "password"
Select "stbb_app" "stbb_db"
LoadData 2 "stbbdata.txt"
```

While this example will work, it may not be feasible if, for instance, `stbbdata.txt` is a large file and the computer running the test does not have enough space to receive it. To work around this problem we take advantage of

Essexer's substitution variables to pass the absolute file name to the ESSCMD script at run time:

- **Calling Shell script:**

```
# Create application and database. Both the newapp and
# the newdb commands honor the value of the SXR_DBHOST
# variable.
sxr newapp stbb_app
sxr newdb -otl stbb.otl stbb_app stbb_db

# Load data into stbb_db
sxr esscmd sxbbdtld.scr\
   %FNAME=$(sxr which -data stbbdata.txt)
```

- **`sxbbdtld.scr` ESSCMD script:**

```
Login "Local" "essexer" "password"
Select "stbb_app" "stbb_db"
LoadData 3 "%FNAME"
```

In this example the `sxr which` command is evaluated at run time resulting in the correct absolute path name being substituted in the containing `sxr esscmd` command[1]. Therefore, even though no absolute path is hard-coded in the ESSCMD script, at the time it is executed it will contain the absolute path name of file `stbbdata.txt` as seen from the view, substituted in place of the "%FNAME" string[2].

Sec. 3.3. File Name Expansion...

There is no limit on the number of substitution variables in a script or the number of times a certain variable may appear in the script. In fact, there is a number of improvements we could do to the above example to make it even more generic. For instance, we may want to make the host name and the Essbase user name and password parameters of the test. Here's how it may be done:

1. I am making a use of the `$(...)` korn Shell notation which denotes in-line evaluation. In other words, the string inside the parenthesis is executed first and its output is substituted. The back quotes is an alternate notation for the same thing.
2. Behind the scene, Essexer will create a temporary file in the work directory of the executing view named like the source script file with the process ID attached at the end (e.g. `stbbdtld.scr.$$`) with all the substitution variables expanded. This file is then passed to the ESSCMD processor. It can be examined at the end of the test run.

- **Calling Shell script:**

```
# Create application and database. Both the newapp and
# the newdb commands honor the value of the SXR_DBHOST
# variable.
sxr newapp stbb_app
sxr newdb -otl stbb.otl stbb_app stbb_db

# Load data into stbb_db
sxr esscmd sxbbdtld.scr          \
   %FNAME=$(sxr which -data stbbdata.txt) \
   %HOST=$SXR_DBHOST %USER=$SXR_USER\
   %PASSWD=$SXR_PASSWORD
```

- **`sxbbdtld.scr` ESSCMD script:**

```
Login "%HOST" "%USER" "%PASSWD" ;
Select "stbb_app" "stbb_db" ;
LoadData 3 "%FNAME" ;
```

2.5 Essexer Environment Variables

Essexer maintains and makes use of a number of environment variables. Some of them can be set by user and are honored by Essexer – these were presented earlier in Section 2.2. – while others are only set by Essexer and changing their values by user is forbidden. The following is the full list of this latter kind of Essexer environment variables. Together with the table in Section 2.2 it presents the complete enumeration all Essexer environment variables.

Variable	Meaning
SXR_BIN	Path to the view-local bin directory, i.e. same as `$SXR_VIEWHOME/bin`.
SXR_INVIEW	Name of the current Essexer view.
SXR_PATH_CSC	File name expansion path for calculator scripts.
SXR_PATH_DATA	File name expansion path for data files, outline files, filters, etc..
SXR_PATH_LOG	File name expansion path for reference log files.
SXR_PATH_REP	File name expansion path for ReportWriter scripts.
SXR_PATH_SCR	File name expansion path for ESSCMD command files.
SXR_PATH_SH	File name expansion path for directly executable files, e.g. shell and perl scripts.
SXR_PATH_SRC	File name expansion path for C/C++ source code files.

Variable	Meaning
SXR_PATH_SXR	File name expansion path for Essexer generated files.
SXR_PLATFORM	The host platform identifier: `nti` and `nta` for Windows NT on Intel and DEC Alpha respectively, `win95` for Windows 95, `os2` for IBM OS/2, `aix` for IBM AIX, `solaris` for Sun Solaris, and `hpux` for HP/UX.
SXR_SRC	Path to the view-local source file directory, i.e. same as `$SXR_VIEWHOME/src`
SXR_VERSION	Current Essexer version, e.g. "2.20 - Production"
SXR_VIEWHOME	Path to the root of the current Essexer view.
SXR_WORK	Path to the view-local work directory, i.e. `$SXR_VIEWHOME/work`

2.6 Essexer Post-action Triggers

Many Essexer commands can be extended by using triggers. A trigger is a shell command string which has been put in association with a certain Essexer command. Once the regular meaning of the command is carried out, the trigger is executed ("fired") automatically by Essexer. This fact that trigger(s) fire after the command's regular task is completed is reflected in this type of triggers being called "post-action" triggers, as opposed to the "pre-action" triggers which, as their name suggests, fire before the command's regular meaning. Essexer only support post-action triggers.

The principal idea behind triggers is that they are used not as permanent part of a test, but to alter its behavior at run time. For instance, triggers can be used to execute a test with any combination of Storage Manager parameters without making the test set them explicitly. In the following example we set the on-newdb trigger by creating an environment variable `SXR_NEWDB_TRIGGER`, which the Essexer framework understands to contain the command string to be executed with every `sxr newdb` command issued by the test:

```
export SXR_NEWDB_TRIGGER='            \\
   sxr esscmd msxxsdsi.scr %APPNAME=stbb %DBNAME=stbb \\
        %ITEM=5 %ARG1=200M            \
   sxr esscmd msxxsdsi.scr %APPNAME=stbb %DBNAME=stbb \\
        %ITEM=12 %ARG1=20M            \
   sxr esscmd msxxsdsi.scr %APPNAME=stbb %DBNAME=stbb \\
        %ITEM=18 %ARG1=1 %ARG2=Y %ARG3=0'
```

The first script will set the data cache size to 200 Mb, the next will set the index cache size to 20 Mb, and the last script will set the isolation to committed access with pre-image reader's access and immediate time-out.

It doesn't matter when and how the `SXR_NEWDB_TRIGGER` environment variable is set, as long as the `sxr newdb` command detects it in the environment.

In most cases the natural place to set it is the view initialization file `.sxrrc`.The table below lists the Essexer commands which support triggers. Found in the second column is the name of the environment variable whose value constitutes the definition of the trigger.

Command	Variable	Comments

diff	SXR_DIFF_TRIGGER	Fires once.
		May be used to perform additional tasks in case of a miscompare:
		`if ( test -f $SXR_WORK/${SXR_TRIGGER_ARG1%.*}.dif ) then`
		`  # we've got a dif, special processing here`
		`fi`
		Arguments:
		`SXR_TRIGGER_ARG1` - the name of the test generated log file.
		`SXR_TRIGGER_ARG2...` - the names of the baselines.

esscmd	SXR_ESSCMD_TRIGGER	Fires once. May be used for <missing uniform> Arguments: `SXR_TRIGGER_ARGC` - number of arguments passed to 'sxr esscmd' command, starting with the script name and including all the substitutions. `SXR_TRIGGER_ARG1` - the name of the script being executed. `SXR_TRIGGER_ARG2...` - rest of arguments to 'sxr esscmd'.
goview	SXR_GOVIEW_TRIGGER	Fires once. May be used to kick-off a certain test, say `accxmain.sh`, immediately upon entering the view: `export SXR_GOVIEW_TRIGGER='sxr sh accxmain.sh'` Arguments: `SXR_TRIGGER_ARGC` - always set to 1 `SXR_TRIGGER_ARG1` - the name of the view being entered.

newapp	SXR_NEWAPP_TRIGGER	Fires once for each application.
		May be used to set certain application-wide settings, which may not be a regular part of the test:
		`export SXR_NEWAPP_TRIGGER='sxr esscmd setappst.scr \`
		`%APPNAME=$SXR_TRIGGER_ARG1'`
		Arguments:
		`SXR_TRIGGER_ARGC` - always set to 1
		`SXR_TRIGGER_ARG1` - the name of the application

Command	Variable	Comments
newdb	SXR_NEWDB_TRIGGER	Fires once for each database. May be used to set certain database-wide settings, which may not be a regular part of the test: `export SXR_NEWDB_TRIGGER='sxr esscmd setdbsti.scr \ %APPNAME=$SXR_TRIGGER_ARG1' \ %DBNAME=$SXR_TRIGGER_ARG2` Arguments: SXR_TRIGGER_ARGC - always set to 2 SXR_TRIGGER_ARG1 - the name of the application SXR_TRIGGER_ARG2 - the name of the database
shell	SXR_SHELL_TRIGGER	Fires once, for the interactively invoked Shell script only. May be used to save off the results of a run to a backup location: `export SXR_SHELL_TRIGGER=\ 'cp -r $SXR_WORK/* x:/backups/'` Arguments: SXR_TRIGGER_ARGC - number of arguments passed to 'sxr shell' command, starting with the script name and including all its arguments. SXR_TRIGGER_ARG1 - the name of the shell script being run. SXR_TRIGGER_ARG2... - rest of arguments to 'sxr shell'.

Most triggers fire only once for the command they are associated with. For instance, the on-diff trigger fires once at the end of each `sxr diff` command and the on-goview trigger fires once right after the view is entered, already within the sub-shell. The two exceptions from this rule are the on-newapp and on-newdb triggers, and for a good reason – these commands take multiple arguments, and it makes sense to fire the trigger for each argument. In the case of the `newapp` command a separate trigger fires for each of the applications created and the `newdb` command fires a separate trigger for every database it creates.

To be useful, triggers need to be able to figure out their run-time context. As an Essexer command fires a certain trigger, it also sets up conventionally named environment variables which then can be used by the trigger to know the arguments that were passed to that command. `SXR_TRIGGER_ARGC` is the argument counter and `SXR_TRIGGER_ARG1, SXR_TRIGGER_ARG2`, etc. are the arguments. For example, the on-newdb trigger always gets two arguments – `SXR_TRIGGER_ARG1` is set to the name of the application and `SXR_TRIGGER_ARG2` is set to the name of the database for which the trigger is firing. Consequently, `SXR_TRIGGER_ARGC` is always set to 2.

3 Developing Essexer Tests

3.1 Installing Essexer

The following is a current copy of the installation readme file.

```
 Hyperion Solutions
                        Essexer Installation README file

The Essexer framework can be run both locally off your local disk, or
remotely directly off the Essexer server. Both methods have their
advantages and disadvantages.

When running Essexer remotely, you will not have to worry about
maintaining your Essexer environment.  As new releases of Essexer
become available, they will supersede the old ones without your
intervention.

When running Essexer locally, you won't depend on the network and
you'll be able to run on a stand-along laptop, or while the network is
down. Note that to run an existing test locally you will not only need
the Essexer framework, but also the test itself reside in a local
view.  Refer to the 'sxr clone' command for details on how you can
copy an existing test into your local view.

Prerequisites to installing Essexer on Windows 95/NT:
-----------------------------------------------------

These steps are required regardless of whether you are going to run
Essexer locally or remotely.

  1. MKS toolkit version 6.1 on NT 4.0 or Win95.

     To find out what release of MKS you're running, type "mksinfo".  You
     have to pay attention to two things which are known to have caused much
     grief:

    a. Make sure that MKS's bin directory appears on the path ahead of every-
       thing else that may interfere with it, such as the NT bin directory
       and the MSDEV bin directory which are both known to plagiarize Unix
       command names.

    b. Make sure that the '.sh' extension is deleted from the your NT registry.
       For that, type 'regedit' on your command prompt, then click on the
       HKEY_CLASSES_ROOT folder, highlight the '.sh' entry and push the
       Delete button.

  2. Hummingbird NFS client software (a.k.a. Hummingbird Maestro Solo).

     a) To install on an Intel-based Windows NT 4.0, run
        S:/Util/Nfs/Nt_intel/Software/setup.exe (S:/ is //Hq-fs2/Sys1)
        The Serial Number is MOMPIU-010840.  Leave the site license box clear.
```

To install on an Intel-based Windows 95, run
S:/Util/Nfs/Win95_98/Software/setup.exe (S:/ is //Hq-fs2/Sys1)
The Serial Number is MOMPIU-010840. Leave the site license box clear.

To install on a DEC Alpha-based Windows NT 4.0, run
S:/Util/Nfs/Nt_alpha/Software/Setup.exe (S:/ is //Hq-fs2/Sys1)
The Serial Number is MONTAU-000350.

b) After the installation is completed, there will be a few-minute delay, unaccompanied by any messages, during which the NFS client surveys the visible network. Once it is finished, the NFS Maestro for Windows NT - Client Configuration dialog box comes up. Click "OK".

c) Let the installation reboot your computer.

Cancel out of parmset (network diagnostic module). It has been noticed that NFS runs better with minimum default settings.

3. Mount the Essexer server on your local host. The host name of the Essexer server is STSUNSXR, and the file system you need is /essexer. You may mount it as any drive letter. (We currently suggest U:\ as it is not being used by Novell login scripts):

   a. Make sure you can ping stsunsxr (172.16.3.201).
   b. Use Hummingbird's Network Access application to mount the remote file system. Choose a local drive letter (U).
   c. Chose"\\stsunsxr\/essexer" as the network path (you can browse for it too.) If the network path comes back as invalid, use the Share Editor application to add Stsunsxr to Hummingbird's list of known hosts.
   d. Specify "essexer" as the remote userid and "password" as its password.
   e. Check the "permanent" box.
   f. Push the "preserve case" radio button.
   g. Click "connect" and acknowledge the status dialog box.
   h. Move to the "Register" tab and register essexer/password username. This will be used at the OS boot time, when Hummingbird attempts to remount the file system automatically, in place of your OS username/password.

To run Essexer remotely:

1. Define the SXR_HOME environment variable, which points to the location of the Essexer framework. Let's say you've mounted the Essexer server as drive Q. Then define SXR_HOME to be 'Q:\latest'.

2. Place $SXR_HOME/bin on your path.

To run Essexer locally:

1. Run Essexer installation script. Assuming you've mounted Essexer server as Q drive and want your local installation to go in C:/sxr, type

   Q:/install/install.sh C:/sxr

```
   2. Place C:/sxr/bin on your path.

   3. SXR_BASE variable, if defined, should not be pointing to the location
      that might become unavailable. Use sxr clone. See sxr man clone.

Prerequisites for running tests on UNIX boxes:
----------------------------------------------
  1. Determine which user is going to be running the tests.
  2. Mount stsunsxr /import/essexer to a local box:
              sun:  vi /etc/vfstab, add mount point
              aix:  vi /etc/filesystems
              hpux: vi /etc/fstab
         create mount point, mkdir /import/essexer
         mount /import/essexer
     You will have to be able to login as root for the above.

  You might be able to use automount instead of the above. Contact
     your unix system administrator for details (or hire one.)

  3. Make sure your .cshrc (or another init file for this user) contains:
         setenv ARBORPATH # for Essbase, actually, not sxr
         setenv LD_LIBRARY_PATH "/usr/lib:$ARBORPATH/bin" # for Essbase, too
         setenv SXR_HOME /import/essexer/latest # to run remotely
     The above format is for csh, use apropriate syntax for your shell.
     Make sure $ARBORPATH/bin is in the path # for Essbase
     Add $SXR_HOME/bin to the path


Post-installation steps:
------------------------

  1. Create users "essexer", "essexer1" through "essexer5", all with password
     "password" and with supervisor privileges.  Essexer does not currently
     support creation of user accounts programmatically.

  2. Make sure your server may be referred to as "local", i.e. the ESSCMD
     command "login local essexer password" will succeed. For this, edit:
              NT4: %windir%\system32\drivers\etc\hosts
              Win95: %windir%\hosts
              solaris: /etc/inet/hosts (read-only file owned by root)
     to contain:     127.0.0.1  local



Other tips:
-----------------------
  1. Make sure that .sh association does not exist on your machine. The
     association will be created by installing MKS. You can also accidently
     create association by double-clicking filename with extention sh and
     choosing program (Notepad?) to open it without unchecking
     "Always use this program to open this file." See 1b above for how to fix.

  2. Even though both forward and backslashes should be fine in path and
     sxr-releated variable names in PC environment, I seem to have better
     luck with forward slashes.
```

```
   3. If you are testing multiple Essbase installations on the same PC, your life
      might be easier if you put variables in the path rather than hard-code
      directories. For this: in addition to global Path environment variable,
      create user path variable and set it to:
      %ARBORPATH%/bin;%SXR_HOME%/bin and whatever else you need. If ARBORPATH and
      SXR_HOME are global, they will be substituted correctly in the path every
      time you change them and start a new cmd or Korn shell session.

More Essexer info is on

http://greenhouse/html/engineering/essexer/essmainw.htm


Last modified: 9/28/98 by Igor, 7/1/99 by Misha.
```

3.2 Name Space

Essexer tests share a common namespace. For instance, all shell files go into the `sh` directory on the base, and all log files go in the `log` directory. Therefore, special attention has to be payed to naming of the test files.

The name space is organized hierarchically based on *themes* and *features* within themes. Themes usually refer to a collection of related features, such as Distributed Cube is the theme uniting the Transparent Partitions, Replicated Partitions, Linked Partitions and Outline Synchronization features.

Sometimes, a theme may not be directly related to any product feature, but is a collection of related tests. For example, the System Test theme refers to the Regression Suites feature, as well as the Bank of Boston, Sears and other customer applications.

Every file name begins with a two letter theme identifier, followed by a two-letter feature identifier. Refer to the table below for the current list of themes and features. Under each theme/feature combination, a special test name `????main.sh` is reserved to group together all of that feature's tests. For example, `dcrpmain.sh` runs all the tests there are to do with the Distributed Cube Replicated Partition feature.

Theme and Feature	ID
1. Schema definition	sd
1.1. Dimension build	sddb
1.2. Restructure	sdre
1.3. Schema manipulation	sdmn
1.4. Substitution variables	sdsv
1.5. Attributes	sdat

Theme and Feature	ID
1.6. Bugs	sdbg
2. Data definition	dd
2.1. Data loader	ddld
2.2. Export, Import	ddxi
2.3. Bugs	ddbg
3. SQL interface	sq
3.1. Dataload	sqld
3.2. Dimension build	sqdb
3.2. Bugs	sqbg
4. Data Manipulation	dm
4.1. Calculator	dmcc
4.2. Dynamic calculator	dmdc
4.3. Currency conversion	
4.4. Bugs	dmbg
5. Data extraction	dx
5.1. Report writer	dxrr
5.2. Spreadsheet extractor	dxss
5.3. Bugs	dxbg
6. *Storage manager*	sm
6.1. Transaction manager	smtx
6.2. Crash recovery	smcr
6.3. Backup	smbu
6.4. Allocation manager	smal
6.5. Index manager	smix
6.6. LRO	smlr
6.7. IBM RSM	smib
6.8. Bugs	smbg
7. Distributed cube	dc
7.1. Transparent partitions	dctp
7.2. Replicated partitions	dcrp
7.3. Mixed partitions	dcmp
7.3. Outline synchronization	dcos

Theme and Feature	ID
7.4. Bugs	dcbg
8. Agent	ag
8.1. Security	agsy
8.2. Login, logout.	aglg
8.3. Networking	agnt
8.4. Application manipulation	agap
8.5. Bugs	agbg
9. API's	ap
9.1. General API	apge
9.2. Outline API	apot
9.3. Grid API	apgd
9.4. Bugs	apbg
10. Acceptance tests	ac
10.1. Basic acceptance	accb
10.2. Extended acceptance	accx
10. Miscellaneous	ms
10.1. Common	msxx
10.2. ESSCMD	mssc
10.3. Performance	mspf
10.4. Bugs	msbg
11. System test	st
11.1 Regression Suites	strs
11.2 OLAP Benchmark	stob
11.3. Bank of Boston	stbb

3.3 The Test Source Control System

All test files reside in the Essexer repository, on a ClearCase VOB. Running of tests does not require any files to be checked out, but if a file needs modifying, or a new file has to be created, the check-in/check-out protocol must be followed.

ClearCase commands cannot be used directly for manipulating files in the Essexer test repository. Rather, Essexer's own `goview`, `cin` and `cout` commands carry out the appropriate ClearCase actions.[1]

Files are checked out of the source control with the `sxr cout` command.

Multiple file arguments are supported, as well as file patterns. The files are copied into the appropriate directory of the current view, where they can be modified. The `sxr cout` command must specify the path(s) of the file(s) being checked out, with respect to the current view. For instance, if the current view's home directory is `d:/views/my_view`, then the command

```
sxr cout d:/views/my_view/sh/accxmain.sh
```

will check out file `accxmain.sh` and bring it into the `sh` directory of the current view, but the command

```
sxr cout d:/views/not_my_view/sh/accxmain.sh
```

will fail because it does not reference the current view.

Of course, relative file names may also be used, as in the example blow:

```
sxr cout sh/accxmain.sh,
```

assuming our current directory is the home of the current view, or we can use the `SXR_VIEWHOME` environment variable:

```
sxr cout $SXR_VIEWHOME/sh/accxmain.sh.
```

Note, that for the check-out operation to succeed the files do not need to exist in the local directory, but merely their paths have to evaluate to one of local view's directories. Furthermore, if a file already exists in the current view, the `sxr cout` command will prompt the user before overriding it. A "no" answer will result in that file's being skipped, while the "yes" answer will cause the existing copy to be saved off in the `.backup` sub-directory before being overridden.

If a pattern is used with the `cout` command, all files matching the pattern will be checked out. For instance, to check out all ESSCMD and report script files that begin with "dctps", assuming the current directory to be the root of the current view:

```
sxr cout 'scr/dctps*.scr' 'rep/dctps*.rep'
```

or, without reliance on the current directory by using absolute file names:

```
sxr cout "$SXR_VIEWHOME/scr/dctps*.scr" \
       "$SXR_VIEWHOME/rep/dctps*.rep"
```

Note the double quotes around the file pattern arguments to prevent the calling shell from interpreting the wild cards (`*`). We could not use the single quotes here, like in the previous example, because we still want the calling shell to expand the variables (`$SXR_VIEWHOME`).

The check-in operation returns specified files into the Essexer repository. Similarly to the check-out, it takes file names or patterns which must evaluate to existing files in the current view. For example, to check-in all files in the current

1. At the time of this writing, Essexer test repository was still being migrated to ClearCase with some details remaining to be worked out. I will not spend any more time discussing ClearCase related specifics here, leaving it to the next revision of this document.

view, assuming the current directory to be view's home directory,

```
sxr cin sh/* csc/* data/* log/* scr/* src/*
```

Note that we are not using any quotes here because we do want the invoking shell to expand all the patterns before passing control to Essexer.

Each check-in and each check-out command, issued from a certain view is logged in file `$SXR_VIEWHOME/sxr/check.log`.

3.4 Creating Applications And Databases

Essexer provides the `newapp` and `newdb` commands to facilitate efficient creation of Essbase applications and databases. In their simplest form, they can be used as follows:

```
sxr newapp sample
sxr newdb sample basic
```

The only argument to the newapp command, `sample`, names an application name, which is created on the host pointed to by the SXR_DBHOST environment variable. The two arguments to the `newdb` command name an existing application in which the new database is to be created (`sample`) and the new database, `basic`. Again, the value of the SXR_DBHOST environment variable will be honored.

In most cases tests should not explicitly manipulate the SXR_DBHOST environment variable. It is set by a user at run time to the name of the host running the Essbase server. Together with the SXR_HOST they tell Essexer to alter the behavior of those of its commands, such as `newapp` and `newdb`, that depend on the Essbase server host name where the Essbase Agent is accepting login requests from clients (SXR_DBHOST) and the client host name where all of the tests's ESSCMD sessions originate (SXR_HOST).

Sec. 2.5. Essexer Environment Variables

This distributed client/server support by Essexer is sometimes referred to as three-tier support, or three-tier architecture. The name is due to the fact that in the most complicated case three computers are involved in the running of the test: the Essexer repository server, the Essbase client computer and the Essbase server computer. Another section in this chapter addresses in detail the issues of development of distributed tests. For now, let us conclude by noting that if the user does not set the value of SXR_DBHOST at run time, it defaults to that of SXR_HOST, indicating the Essbase server is running on the local machine.

Sec. 3.7. Developing Distributed Tests

In most cases tests will have to use more sophisticated variants of the newapp and newdb commands than in the previous example. Let's consider, for instance, what happens if application Sample already exists. The newapp command will get an error and the test will proceed to create the Basic database within an old application, which is unlikely to be desirable. The `-force` flag, supported by both `newapp` and `newdb` commands helps to remedy the situation. It directs Essexer to first drop the object it is about to create, and then proceed with creating it. In case the object did not exist in the first place, the only negative side effect will be the Essbase error to that effect.

Another useful feature of the newapp and newdb commands is that they let user create more multiple objects with just one command. For example statement

```
sxr newapp -force accxapp1 accxapp2
```

creates two applications, `accxapp1` and `accxapp2`. Note the "-force" flag which instructs Essexer to drop the applications before creating them. We can now proceed to creating the databases:

```
sxr newdb accxapp1 accxdb11 accxdb12 accxdb13
sxr newdb accxapp2 accxdb21 accxdb22 accxdb23
```

The first statement creates three databases, `accxdb11`, `accxdb12` and `accxdb13`, all in the application `accxapp1`. Similarly, the second statement creates databases `accxdb21, accxdb22` and `accxdb23` in the application `accxapp2`. Note that we did not use the "-force" flag with the newdb commands because we know that the applications are brand new.

The newdb commands we've considered in this section so far create an empty database, i.e. a database without an outline. It is often desirable to use an existing outline file, say `stxxbsic.otl`, to pass it to the newdb command so that the database it creates has that outline. This is accomplished with the "-otl" flag:

```
sxr newdb -otl msxxbsic.otl sample basic
```

This directs Essexer to perform the following steps:

Sec. 2.3. File Name Expansion...

- Locates the outline file on the SXR_PATH_DATA path. In most cases, where there is no view-local copy or an OS specific version of this file, it will be expanded to `$SXR_BASE/data/msxxbsic.otl`.
- Create an empty database as if no outline file was specified.
- Perform a file-system based restructuring with the outline file located on step 1.

When the "-otl" flag is supplied with newdb command which names multiple databases, all of them will be created with the same outline.

3.5 Running ESSCMD Scripts

Sec. 2.4. Essexer Substitution Variables

We have already seen examples of how the `sxr esscmd` command is used to run ESSCMD scripts when we talked about substitution variables. Let us now consider it in more detail.

Like other Essexer commands, the run time behavior of the `sxr esscmd` can be controlled by environment variables that are meaningful to Essexer, most importantly by SXR_ESSCMD. If set, it specifies the name of the alternate ESSCMD executable. This is useful, when testing a non-public version of ESSCMD, or when running a version of ESSCMD executable different from that of the server.

Two special cases of the alternate ESSCMD interpreter are supported grammatically. Supplying the "-q" flag to the `sxr esscmd` command will instruct

Essexer to run the ESSCMDQ executable, while the "-g" flag invokes the ESSCMDG Grid-API based variant of ESSCMDQ.

Each time Essexer encounters a command like `sxr esscmd stbbdltd.scr`, it does the following things:

- Locates the script file `stbbdltd.scr` on the SXR_PATH_SCR path. In most cases, when there is no view-local copy of this file, it will be expanded to `$SXR_BASE/scr/stbbdtld.scr`.
- A working copy of the script file is created in $SXR_WORK (the work directory of the current view) named `stbbdtld.scr.$$`, where "$$" stands for the process ID, intended to make the name unique.
- The working copy of the script file is scanned for substitution variables, which are substituted in-place. The substitution variables are passed to the `sxr esscmd` command as arguments, following the script file name, that have the format "string1=string2". String1 is understood by Essexer to be the name of the substitution variable, while string2 is the value it is replaced with.
- The replaced script file is passed as argument to the appropriate ESSCMD interpreter.

Sec. 2.3. File Name Expansion...

Sec. 3.8. Analyzing Essexer Tests

Sec. 2.4. Essexer Substitution Variables

Let us now go back to the example we have used when we talked about the substitution variables:

```
sxr esscmd stbbdtld.scr          \
   %FNAME=$(sxr which -data stbbdata.txt) \
   %HOST=$SXR_DBHOST %USER=$SXR_USER\
   %PASSWD=$SXR_PASSWORD
```

Here we instruct Essexer to execute the script file `stbbdtld.scr` while replacing four substitution variables, %HOST, %USER, %PASSWD and %FNAME[1]:

```
Login "%HOST" "%USER" "%PASSWD" ;
Select "stbb_app" "stbb_db" ;
LoadData 3 "%FNAME" ;
```

To follow exactly what happens between the executing shell encountered the above sxr esscmd statement and the replaced script file is passed to the ESSCMD interpreter by Essexer, we have to remember how Unix shells work. Before giving control to Essexer, shell performs all appropriate expansions. In our example, it is the variable expansion ($SXR_DBHOST) and the execution expansion $(...).

1. The use of the "%" sign is not mandatory. When substituting variables, Essexer merely replaces all occurrences of one string with another string. However, it is recommended the "%" sign is used in every symbolic variable name to increase readability of the script file by a human.

In the case of variable expansion, shell simply replaces the variable reference, e.g. $SXR_USER, with its current value, e.g. "essexer". Therefore, when Essexer gets control from shell, it sees "%USER=essexer" as the value of its third argument, which is exactly what we need.

In the case of execution expansion, shell threats the string inside of the $(...) notation as an executable command, which it runs in a temporary sub-shell, and replaces the entire $(...) construct with its output. In our example, "sxr which -data stbbdata.txt" is the command that gets executed in a subshell. Its output is the full path to file `stbbdata.txt`, which is what shell substitutes in place of the $(...). Therefore, when Essexer gets control from the shell, it sees its first argument as something like `%FNAME=Q:/base/data/stbbdata.txt`.

Sec 1.1. The Essexer Paradigm

The reason why scripts avoid hardcoding information, like the full path of a data file, is the location transparency principle. By deferring the exact path evaluation until run time, scripts let the Essexer framework take into the account such things as the host name of the computer and view name where the test is executing.

3.6 Running ReportWriter Scripts

ReportWriter scripts may be executed by Essexer either via the `sxr esscmd` facility described above or by a dedicated command `sxr runrept`. The first method provides more flexibility, but is longer to code and involves an extra ESSCMD script file which does nothing but logs in and runs a report. For most standalone ReportWriter scripts, using the `sxr runrept` is easier to code and faster to work. Additionally, the `sxr runrept` command supports substitution variables inside of report scripts.

In its simplest way, the `sxr runrept` command requires four arguments and looks like this:

```
sxr runrept accxapp accxdb accxcalc.rep accxcalc.log
```

The four required arguments are: the application name, the database name, the report script name and the output log file name.

Sec.3.5. Running ESSCMD Scripts

Substitution variables are supported for the report scripts, just like they were for the ESSCMD scripts. They are passed to Essexer as arguments following the mandatory parameters:

```
sxr runrept accxapp accxdb accxcalc.rep accxcalc.log \
        %DIMENSION=Product %MISSING=Y
```

In general, each time Essexer encounters an `sxr runrept` command, it does the following things:

- Locates the script file (say, `accxcalc.rep`, in the example above) on the SXR_PATH_REP path. In most cases, when there is no view-local copy of this file, it will be expanded to `$SXR_BASE/rep/accxcalc.rep`.

Sec. 2.3. File Name Expansion...

- A working copy of the script file is created in $SXR_WORK (the work directory of the current view) named `accxcalc.rep.$$`, where "`$$`" stands

Sec. 3.8. Analyzing Essexer Tests

for the process ID, intended to make the name unique.

Sec. 2.4. Essexer Substitution Variables

- The working copy of the script file is scanned for substitution variables, which are replaced in-place. The substitution variables are passed to the `sxr runrept` command as arguments that have the format "string1=string2". String1 is understood by Essexer to be the name of the substitution variable, while string2 is the value it is replaced with.
- The replaced report script file is passed as argument to a special purpose ESSCMD script file, maintained by Essexer, which runs it.

Normally, the sxr runrept command inherits the host name, user name and password from the run time environment, namely SXR_DBHOST, SXR_USER and SXR_PASSWORD variables respectively. Although this covers the majority of cases, these defaults can be overridden with the "-host", "-user" and "-password" flags.

Sec. 3.7 Developing Distributed Tests

3.7 Developing Distributed Tests

One of Essexer's strengths is its support of distributed testing. A test designer will find it rather simple to design a test which could be directed at execution time to either run against a local or a remote Essbase server, residing on an arbitrary host. Furthermore, a more sophisticated test designer will be able to build tests that involve complicated multi-host distributed architectures.

Let us, once again use the example from a previous section:

```
# Create application and database. Both the newapp and
# the newdb commands honor the value of the SXR_DBHOST
# variable.
sxr newapp stbb_app
sxr newdb -otl stbb.otl stbb_app stbb_db

# Load data into stbb_db
sxr esscmd stbbdtld.scr          \
   %FNAME=$(sxr which -data stbbdata.txt) \
   %HOST=$SXR_DBHOST %USER=$SXR_USER\
   %PASSWD=$SXR_PASSWORD
```

By default, Essexer sets the value of the SXR_DBHOST variable to be the same as SXR_HOST, i.e. the name of the computer where the test is running. However, should the user change the value of SXR_DBHOST to, say, "stnti7" before running this test[1], it will run against the remote server as is.

1. Once again, the view initialization file .sxrrc is recommended to set SXR_DBHOST. That was it will be set automatically upon entering the view.

In the case of the `newapp` and `newdb` commands, the test designer did not have to do anything to achieve this flexibility – these Essexer commands by design honor the value of the SXR_DBHOST variable and creates the databases on the right host. In the case of the ESSCMD script, the test designer had to parametrize the host name by making it a substitution variable and assigning it the value of SXR_DBHOST variable.

Let us now consider a more complicated example in which two databases are linked with a transparent partition. Let `stsetrgt` be the target database and `stsesrc` be the source. We want to develop a test which would be flexible enough so that at run time we could direct it to either

- create both databases on the same computer where the test is running or
- create both databases on one computer which is different from the computer where the test is running or
- create each database on a separate host.

Here's how we do that:

```
# If source and target hosts are not set, default both
# to SXR_DBHOST
export STSE_HOST_SRC=${STSE_HOST_SRC:-$SXR_DBHOST}
export STSE_HOST_TRGT=${STSE_HOST_TRGT:-$SXR_DBHOST}

# Create target application and database.
sxr newapp -host $STSE_HOST_TRGT -force stsetrgt
sxr newdb -host $STSE_HOST_TRGT -otl stsetrgt.otl \
        stsetrgt stsetrgt

# Create source application and database.
sxr newapp -host $STSE_HOST_SRC -force stsesrc
sxr newdb -host $STSE_HOST_SRC -otl stsesrc.otl \
        stsesrc stsesrc

# Dimension-build the target database
sxr esscmd stsedbt.scr %HOST=$STSE_HOST_TRGT

# Dimension-build the source database
sxr esscmd stsedbs.scr %HOST=$STSE_HOST_SRC

# Load data in the source database
sxr esscmd stselds.scr %HOST=$STSE_HOST_SRC\
        %DATA_FILE=$(sxr which -data stselds.txt)\
        %RULE_FILE=$(sxr which -data stselds.rul)\
        %ERR_FILE=$SXR_WORK/stselds.err

# Run some reports off the target site
sxr runrept -host $STSE_HOST_TRGT stsetrgt stsetrgt \
         stseprod.rep stseprod.log %DIMENSION=Product
```

Since there are two server sites in the test, there is potentially going to be three computers involved in the test:

- The client computer, where the test is running. As before, it's the SXR_HOST computer, and the test is not concerned with it. The only reference to a local computer is made at the very end of the test where we specify the error file destination to be the work directory of the current view. We let that reference be evaluated at run time by using the SXR_WORK environment variable which Essexer will have set to the right path.
- The source site server computer. We make its host name a test parameter

which we assume to be stored in the STSE_HOST_SRC environment variable. Each time the test needs to know the source site host name, it simply references this variable. Naturally, the test lets it default to SXR_DBHOST. That way, if at the execution time, the source site host name is not set explicitly, the test will still do a reasonable thing. Note, that SXR_DBHOST itself is a user-defined variable, which defaults to SXR_HOST. In other words, if user sets neither SXR_DBHOST, nor STSE_HOST_SRC, the former is defaulted to SXR_HOST (by Essexer), i.e. to the local host name, and the latter is defaulted to the former in the third line of the example above. If, on the other hand, the user sets SXR_DBHOST, then STSE_HOST_SRC will default to it, unless it is also explicitly set.

- The target site server computer. We make its host name a test parameter too and store it in STSE_HOST_TRGT. Its handling is identical to that of STSE_HOST_SRC described above. In the simplest case, when user sets neither the source, nor the target host names, they both default to SXR_DBHOST, which, in turn, defaults to the local host name if unset. In the most complex case, the user explicitly sets both the source and the target site host names, each pointing to its own computer.

3.8 Analyzing Essexer Tests

One of the objectives of the Essexer project has been the streamlining of the test results analysis. There are several things that the framework does automatically to aid in test analysis, while other things are left for the tests to implement. Ultimately, when there have been no failures, the analysis should take an engineer a matter of seconds and be completely automatable. Only failures should require special attention.

A number of auxiliary and log files are created by the Essexer during execution of a test which are helpful during the analysis of its results. The following is their complete list.

- **`sxr esscmd <base-name>.scr ...`**
  `<base-name>.scr.$$` contains the text of `<base-name>.scr` with all the substitution variables expanded[1]. This is the actual file passed by Essexer to the ESSCMD executable.
  `<base-name>.sog.$$` contains the stdout and stderr from the ESSCMD session. Essexer uses the extension "sog" to denote an Essexer-collected log file throughout the framework. Note that the PID's of both the script file and the sog file based on the common source script are the same and thus can be matched during the test analysis[2].

1. "$$" is the Korn shell notation for the process ID number (PID) of the current shell command.
2. Ideally, we want to use incrementally growing generation numbers instead of the process ID's. Implementing that requires some concurrent programming, which is not easily accomplished in Korn Shell, Essexer's programming platform, and therefore has been left out as a desirable enhancement.

- **`sxr runrept <app-name> <db-name> <base-name>.rep ...`**
  Currently, the `sxr runrept` command is implemented via the `sxr ess-cmd` command, which produces two work files as described above: `runrept.scr.$$` and `runrept.sog.$$`. Additionally, file `<base-name>.rep.$$` will contain the text of the source report script file with the all substitution variables expanded.
- **`sxr shell <base-name>.sh ...`**
  `<base-name>.sog` file contains the stdout and stderr from `<base-name>.sh` file. In the case of nested shell files, each `.sog` file contains the output from its corresponding Shell file and all the files it calls via the `sxr shell` command.
  Note that this naming convention will do poorly with recursive nesting of Shell scripts, resulting in sog files being overridden.

Generally, tests verify their own results by generating log files and comparing them with reference logs, or "baselines", which are generated once as part of the test creation and are checked-in to the log directory. The test therefore has to implement proper log collection throughout the course of its execution, so that any failure results in generation of a bad log file or one is not generated at all. The `sxr diff` command implements all the necessary logic to create a `.dif` (difference) file in case when the test log file differs from its baseline(s), or when it wasn't created, and a `.suc` (success) file when the test log file coincides with (one of) its baseline(s). Therefore, Essexer supports this log comparison based model by guaranteeing that each time the `sxr diff` command is called, either a `.dif` or a `.suc` file is created.

The first thing an engineer analyzing the test results should do is to change her working directory to `$SXR_WORK` (the current view's work directory) and find out what `.dif` and `.suc` files there are[1]. If there are no `.dif` files and the number of `.suc` files appears to be the correct one for this test, no further analysis is needed. Alternatively, if there is a `.dif` file, it will contain either the difference between the corresponding test-generated log file and its (last) baseline, or, if the test failed to generate the log file at all, a message to that effect. These difference files are a good starting point for localizing the problem.

Other things that may aid in the test analysis are the log files automatically collected by Essexer, which are also located in $SXR_WORK.

3.9 The Mobile Support

Essexer provides basic support for developing and executing tests on a computer that is (temporarily) disconnected from the Company network. There are basically two parts to running an Essexer test – the Essexer framework and the tests proper. As was discussed in the installation steps, the framework can be used either directly off the network or it can be installed locally. The test files usually reside in

1. By, for instance, typing "`ls *dif *suc`".

the base location on the network. To run autonomously, both the framework and the test files must reside on the laptop.

The installation of Essexer locally is simply a matter of running the install script and placing the appropriate directory on the Shell search path. The "cloning" of an existing test from the base into a local view is accomplished using the `sxr clone` command.

4 Reference

4.1 Resource Guide

- `stsunsxr:/essexer/doc` contains:
  This document in Acrobat format (gde_ref2.pdf).
  This document in FrameMaker format (gde_ref2.fm).
  The Style Guide in PostScript format (style_guide.ps).
  The Style Guide in Acrobat format (style_guide.pdf).
  The installation readme file.
- On-line help may be obtained on any Essexer command by typing
  `sxr help <command-name>`
  Additional on-line help is available with
  `sxr help path`
  `sxr help trigger`
- Korn Shell on-line help is available via regular man facility, which is also supported by the MKS toolkit.

4.2 Essexer Command Reference

sxr agtctl

```
NAME
        sxr agtctl - Local Agent Control

SYNOPSIS
        >>- sxr agtctl -+-----------------------+--><
                        |                       |
                        +============== status =+
                        |                       |
                        +-+----------+- start --+
                        | |          |          |
                        | +- -force -+          |
                        +-------------- stop ---+
                        |                       |
                        +-------------- kill ---+

DESCRIPTION
        Controls local Essbase Agent, i.e. only meaningful when SXR_HOST and
        SXR_DBHOST environment variables are the same.  Otherwise, does
        nothing.

        If no arguments given, 'status' is assumed.  The '-force' flag is only
        meaningful with 'start'.

        'status' : The number of currently running Essbase Agents is returned.
```

```
        'start'  : The Agent is (re)started.  The following logic is carried
                   out:

                     if agent is not already running then
                       start it via essbase command w/ appropriate arguments
                     else if '-force' flag is given
                         do same as 'sxr agtctl kill'
                         start agent
                     fi

                   Note that no action is performed if the agent is already
                   running and '-force' flag was not supplied.

        'stop'   : The Agent is stopped via ShutdownServer ESSCMD command.

        'kill' : The Agent and all server processes are terminated
                   ungracefully via OS facilities, e.g. kill(1).  Note, that
                   similarly to 'stop', this option is also implemented
                   synchronously . After the OS 'kill' command is issued, the
                   list of active process is polled with 1-sec. internals
                   to account for possible OS's process aging latency.

ENVIRONMENT
        Must be in an Essexer view.

NOTES
```

sxr cin

```
NAME
        sxr cin - Check file(s) into source control from a local view.

SYNOPSIS
        >>- sxr cin -+----------+-+-+- <test-file-name> -+-+-->
                     |          | | |_____________________| |
                     +- -force -+ |                       |
                                  +- -dna <dna-file-name> -+

DESCRIPTION
        Checks into the source control system supplied files. Prints on its
        stdout the names of the files being checked-in.

        Currently, in the absence of the true source control system, the
        files are simply copied to the base location. If there is already an
        existing copy of the file, the command prompts for a confirmation
        to override the old copy.  A "y" reply causes the existing file to be
        overridden.  An "n" reply causes the current file to be skipped.
        Alternatively, if the file is not found in the source control system,
        it is placed in the most platform-generic directory (if applicable)
        for this file type.

        -force  Suppress prompts due to files' existence in the base.
```

Equivalent to replying “y” to all file override prompts.

-dna DNA files are not currently supported in this command.

<test-file-name> is either an absolute name or a relative name taken with respect to the current directory. It must be an existing file name within the current view. The file’s type is determined depending on its directory.

ENVIRONMENT

Must be in an Essexer view.

NOTES

See also sxr man cout.

sxr clone

NAME

sxr clone - Clone a test into the local view.

SYNOPSIS

```
>>- sxr clone -+----------+-+- <test-file-name> -----+------------><
               |          | |                        |
               +- -force -+ +- -dna <dna-file-name> -+
```

DESCRIPTION

Specified test is cloned into the local view according to the information in the test’s DNA file. A DNA file is a full list of all files that a test references during the course of its execution.

A test’s DNA file is produced during execution and is placed into the sxr directory of the current view. From there it may be checked into the Essexer repository using the ‘sxr cin’ command.

If a <test-file-name> is supplied, its DNA file name is assumed to have the same name and have the extension of ‘.dna’. This DNA file name is then expanded along the SXR_PATH_SXR path. Therefore, if the test has just been run in the current view, the DNA file which was created during that execution will take precedence over the base one.

Once the DNA file has been located, a single ‘sxr cout’ command is generated, which encompasses all the files. Should a certain file from this list already exist in the local view, the user will be prompted for a confirmation before that copy is overridden, unless the -force flag was supplied. See ‘sxr man cout’ for description of the -force flag.

Alternatively, an arbitrary <dna-file-name> can be supplied for direct cloning.

ENVIRONMENT

Must be in an Essexer view.

NOTES

See also sxr man cout, sxr man cin.

sxr cout

NAME

sxr cout - Check out file(s) from source control into a local view

SYNOPSIS

```
>>- sxr cout -+----------+-+---------+-+-+- <file-name-pattern> -+-+-><
              |          | |         | | |_____________________| |
              +- -force -+ +- -lock -+ |                         |
                                       +- -dna <dna-file-name> ----+
```

DESCRIPTION

Check out file(s) matching the pattern(s) and bring them into the respective directories in the local view. Prints on its stdout the names of the files being checked-out. If a file already exists in the local view, the command prompts for confirmation to override the existing file. A “y” reply causes the existing file to be saved in the .backup subdirectory, while an “n” reply cases the file to be skipped.

-lock : Lock files in the source control. Not implemented.

-force : Suppress prompts due to files’ existense in the local view. Equivalent to replying “yes” to all file override prompts.

-dna : DNA files are not currently supported in this command.

File name patterns use the usual Shell wild cards. Quoting may be needed to prevent expansion by the invoking Shell.

ENVIRONMENT

Must be in an Essexer view.

NOTES

See also sxr man cin.

sxr diff

NAME

sxr diff - Comparison of a test log file with (a set of) reference log file(s).

SYNOPSIS

```
>>- sxr diff -+--------------------------+---------------------->
              |                          |
              +-+-  -ignore <basic-RE> -+-+
                |_________________________|

    >- <test-file> -+----------------------+-------------------><
                    |                      |
                    +-+- <base-line-file> -+-+
                      |____________________|
```

```
DESCRIPTION
        Compares a test-generated output file with a (set of) base-line
        log file(s).  Optionally, one or more basic regular expressions
        may be supplied to direct the comparison to ignore differences which
        result from line segments matching these regular expressions.

       <basic-RE>
          A basic regular expression of the ed(1) style. Note, that in
          contrast with the full regular expression, basic RE syntax does not
          support combining multiple RE's with the "|" symbol.  To achieve the
          same effect, multiple -ignore may be used should be used.

        <test-file>
          The name of the test-generated output file to be compared.  Always
          assumed to reside in $SXR_WORK

        <base-line-file>
          One or more reference output files, whose exact location is
          determined by scanning the $SXR_PATH_LOG path. If no
          <base-line-file> name is given, same name as <test-file> is assumed.

          A successful comparison results in generation of the zero-length
          success file, named <test-file>.suc.  An unsuccessful comparison
          generates a difference file, named <test-file>.dif, which contains
          the output of the diff(1) command.

          If multiple <base-line-file>'s are supplied, the <test-file> is
          successively compared to each of them until either a successful
          comparison is performed or <base-line-file> list is exhausted. If
          multiple base-line files were specified and no match was found, the
          <test-file>.dif file will contain the output of the most recent
          comparison.


RETURN STATUS
        "0" A successful comparison was performed, i.e. a reference log
            matched the test log exactly.
        "1" An unsuccessful comparison was performed, i.e. (none of) the
            reference log file(s) matched the test log file.
        "2" A user error occurred, e.g. a test log file wasn't found.

TRIGGER
        A post-action trigger may be associated with this command by setting
        environment variable SXR_DIFF_TRIGGER to an Essexer-evaluatable command
        string.  Trigger's environment will contain variables SXR_TRIGGER_ARGC
        and SXR_TRIGGER_ARG1 through SXR_TRIGGER_ARG<n>, respectively set to
        the number <n> of the arguments passed to the diff command and all
        those arguments.  See 'sxr man trigger' for more information.

ENVIRONMENT
        Must be in an Essexer view.

        Ordinarily, the test log files are retained regardless whether or not
        a successful comparison was found.  This behavior can be overridden by
        setting environment variable SXR_DIFF_LOGDEL to any non-null value, in
```

which case successfully compared test log files will be removed at the end of ‘sxr diff’ command.

SEE ALSO

diff(1), regexp(5), ed(1), sxr man trigger.

sxr esscmd

NAME

sxr esscmd - Execute an ESSCMD script file.

SYNOPSIS

```
>>-- sxr esscmd -+------+- <script-file-name> ------------------->
                 |      |
                 +- -q -+

 >--+--------------------------------+------------------------><
    |                                |
    +-+- <old_string>=<new_string> -+-+
      |_______________________________|
```

DESCRIPTION

Execute an Essbase command file <script-file_name>. The file is searched for on the ESSCMD script file path ($SXR_PATH_SCR). Optionally, an unlimited number of substitution parameters may be supplied, in which case every occurrence of <old_string> be replaced with the corresponding <new_string>.

The ‘-q’ flag causes the default command file interpreter to be substituted with ESSCMDQ.

ENVIRONMENT

Must be in an Essexer view.

TRIGGERS

A post-action trigger may be associated with this command by setting environment variable SXR_ESSCMD_TRIGGER to a command string executable by Korn shell. The trigger’s environment will contain variables SXR_TRIGGER_ARGC and SXR_TRIGGER_ARG1 through SXR_TRIGGER_ARG<n>, respectively set to the number <n> of the arguments passed to the esscmd command and all those arguments. See ‘sxr man trigger’ for more information.

NOTES

The default Essbase command file interpreter is ESSCMD. It is overridden by the value of the $SXR_ESSCMD environment variable if it is set.

sxr essinit

NAME

sxr essinit - Initialize the local Essbase environment.

SYNOPSIS

```
>>-- sxr essinit -+----------------------------+----------------><
                 |                            |
                 +- -sec <security-file_name> -+
```

DESCRIPTION

(Re)initializes the local Essbase environment by performing the following steps:

- The current Essbase instance is terminated via ‘sxr agtctl kill’.
- The contents of the ${ARBORPATH}/app directory are recursively removed.
- Agent log file Essbase.log is removed.
- Security file ${ARBORPATH}/bin/essbase.sec is replaced with the <security-file-name>, if supplied, or with the default security file msxxnull.sec, if not.
- The Agent is restarted via ‘sxr agtctl start’ command.

ENVIRONMENT

Must be in an Essexer view.

NOTES

There is currently a limitation whereby the initial registration information cannot be passed to the essbase executable on the command line, and has to typed in interactively, which would not be allowable for the purpose of test automation. To alleviate this shortfall, Essexer supports the notion of an “empty” security file, i.e. the essbase.sec file that will be created by Essbase after launching it the very first time, typing in the installation owner information and quitting. This “empty” security file has, by convention, the name of msxxnull.sec and will be looked for if ‘-sec’ option is omitted, as if it was present. In other words, the command ‘sxr essinit’ is a shortcut for command ‘sxr essinit -sec msxxnull.sec’.

The name of the default empty security file is overridden by the value of SXR_ESSINIT_SECFILE environment variable if it is set.

Note, that the empty security files are not part of Essexer installation, but are looked for on the -data path. Note that Hyperion expressly states that security files are not portable between different platforms, and hence should be located in the appropriate platform directories of the -data path.

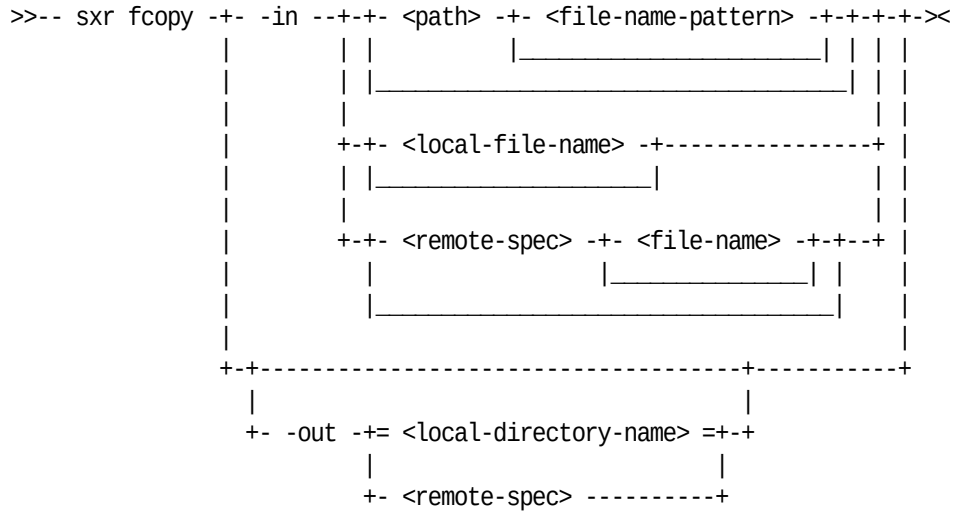
See also: sxr newapp, sxr newdb.

sxr fcopy

NAME

sxr fcopy - copy files between Essexer base, Essbase client and Essbase server machines.

SYNOPSIS



DESCRIPTION

A test, executing as an Essbase client, uses this command to transfer files between its own environment, that of an Essbase server, and the Essexer base repository. All of these are potentially different hosts. Multiple source files may be specified with the -in clause, but only a single target is allowed to follow the -out clause.

The -in clause must be always present. The -out clause may be omitted, in which case the work directory of the current view will be used as the target of the copy.

```
<path> ::= -+- -csc --+-
           +- -data -+
           +- -doc --+
           +- -log --+
           +- -rep --+
           +- -sh ---+
           +- -sxr --+
```

Indicates that what follows is a list of one or more file names (or file name patterns) of a certain file type as recognized by Essexer. Multiple path flags may follow the single -in flag.

<file-name>

A short file name. Used to specify the name of an database based object as identified by a remote spec in the -in clause.

<file-name-pattern>

A file name with the usual shell wild cards as described in

```
  man sh(1).  All files matching the pattern will be part of the copy
  operation. Note that in order to prevent expansion of the pattern by
  the current shell, it must be quoted.

<local-file-name>, <local-dir-name>

  A local file or a local directory. Follow the usual shell file
  naming convention.  A relative specification will be taken in
  relation to the $SXR_WORK directory of the current view.

<remote-spec> ::=
      !<host-name>!<user-name>!<password>!<app-name>!<db-name>!

  Denotes a particular user domain on a particular database on a
  particular Essbase server host.  Consists of five parts separated
  from one another with the "!", progressively defining the exact
  location of the file.  Only <app-name> and <db-name> are mandatory.
  If omitted, <host-name> defaults to $SXR_DBHOST, <user-name>
  defaults to $SXR_USER and <password> defaults to $SXR_PASSWORD.

  When used on the -in clause, must be followed by at least one
  <file-name> -- the name of the object to be coped.

EXAMPLES
        sxr fcopy -in -rep 'dxrrb*.rep' 'dxrrc*.rep' -data dxrrload.txt \
                  -out !Dxrrapp!Dxrrdb!

          Copies all reports matching the patterns and the dataload file into
          the server database directory on host $SXR_DBHOST.


        sxr fcopy -in '*.dnn' -out !stsun6!!!dctpmain!dctpsrc!

          Copies all .dnn files from the work directory of the current view to
          the database directory on host stsun6.  Note that even though we
          have used the default user name and password, we had to specify the
          expected number of exclamation points between the host name and the
          application name.

        sxr fcopy -in !staix5!user1!not2say!sample!basic!  \
                      Sddbhd1.rul Sddbhd2.rul Sddbhd3.rul  \
                  -out !staix7!!!Sddbapp!Sddbdb!

          Copies the specified three files from the database directory on host
          "staix5" to the database directory on host "staix7".  Note, that we
          are using non-default user name and password (user1/not2say) on
          staix5.

        sxr fcopy -in -data stxxlist.txt

          Copies stxxlist.txt file from the Essexer base to $SXR_WORK.


ENVIRONMENT
        Must be in an Essexer view.  For all remote string specified, an
        Essbase Agent must be running on each of the hosts.
```

```
NOTES
See also: sxr which, sxr path, sh(1).
```

sxr getaplog

```
NAME
        sxr getaplog - get application log into the view work diretory

SYNOPSIS
        >>- sxr getaplog --------<app-name> --+------------------------+--><
                                              |                        |
                                              +-+-    <file-name>   -+-+
                                        |_____________________|

DESCRIPTION
       Copies an Essbase application log file to a view work directory. This
       might be useful to verify that the expected thing happened at a given
       point rather than wait for the long test to complete to see failure
       caused by some error long time ago. Test developer might also be able
       to create more descriptive diff files, simplifying test analysis.

        <app-name>
          The name of the Essbase application which log is requested. This is
          a mandatory parameter.

        <file-name>
          Output file will be generated in $SXR_WORK directory. The name will
          be the second parameter, if specified, or the in the format
          $app-name.getaplog.$process_id.log
          Name in this format should be addressable and rather unique within
          a test suite environment.

          Only one log (app) at a time is supported in this release.

RETURN STATUS
        "0" operation completed successfully
        "1" Unsuccessful operation, e.g. application does not exist
        "2" A user error occurred, e.g. no application name supplied

TRIGGER
        No triggers supported before or after this command.
        See 'sxr man trigger' for more information.

ENVIRONMENT
        Must be in an Essexer view. ESSCMDQ must be available.

SEE ALSO
        sxr man view, sxr man trigger.
```

sxr goview

NAME

sxr goview - Enter an Essexer view.

SYNOPSIS

```
>>-- sxr goview -+------------------------+-+-----------+---->
                 |                        | |           |
                 +- -eval <shell-command> -+ +- -notrig -+

 >-- <directory-name> ---------------------------------------><
```

DESCRIPTION

Starts an Essexer view session in the view named (contained in) <directory_name>. The view has to exist. For making a newview, see 'sxr newview'.

A view session is a sub-shell and a set of Essexer specific environment variables, which are only initialized inside this view session and are destroyed as soon as the view session is terminated with 'exit' command. The value of SHELL environment variable is honored.

Essexer view initialization file .sxrrc, located in the root of the view, is sourced. This provides flexible, programmatically controlled, view-specific initialization settings.

-notrig : Firing of the on-goview trigger is suppressed.

-eval : Enters the view (and fires the on-goview trigger if set) and instead of giving control back to the interactive shell, evaluates <shell-command> and, subsequently terminates the view session. This option is helpful when an Essexer test is executed from a shell script.

ENVIRONMENT

Cannot be already in an Essexer view.

A post-action trigger may be associated with this command by setting environment variable SXR_GOVIEW_TRIGGER to an Essexer-evaluatable command string. Trigger's environment will contain variable SXR_TRIGGER_ARG1 set to the name of the view being entered. See 'sxr man trigger' for more information.

NOTES

See also: sxr man newview, sxr man inview, sxr man trigger.

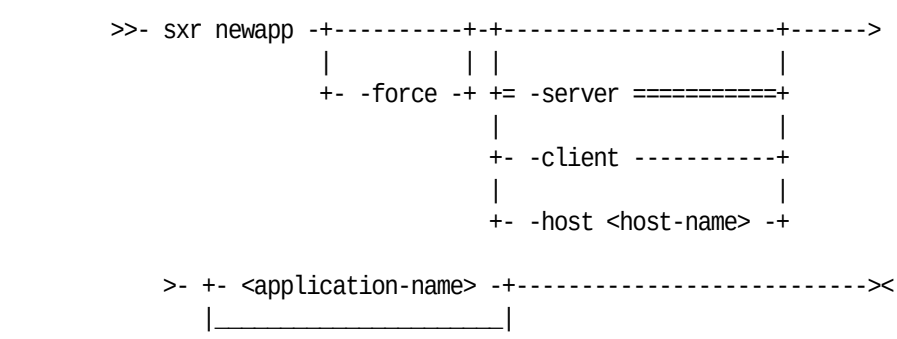
sxr inview

sxr newapp

NAME

sxr newapp - Create new Essbase applications.

SYNOPSIS



DESCRIPTION

Creates new Essbase applications named in the application name list. Flags, default or explicit, apply to all applications to be created.

-force : The following two septs are performed before each application is created: 1) all locked objects in any of the application's databases are unlocked; b) the application is deleted.

-server : The default. The applications are created on the server machine in directory $ARBORPATH/app. The server machine is assumed to be identified by the SXR_DBHOST environment variable.

-client : The applications are created on the client machine in directory $ARBORPATH/client. The client machine is assumed to be the local host, identified by the SXR_HOST environment variable.

-host : The subsequent parameter is treated as host name where the applications are to be created. This is especially useful for distributed tests where multiple server hosts are involved.

TRIGGERS

A set of post-action triggers may be associated with this command by setting environment variable SXR_NEWAPP_TRIGGER to a Shell command string. A separate trigger will fire for each of the newly created applications. Each trigger's environment will contain variable SXR_TRIGGER_ARG1 set to the application name for which it is firing.

ENVIRONMENT

Must be in an Essexer view.

NOTES

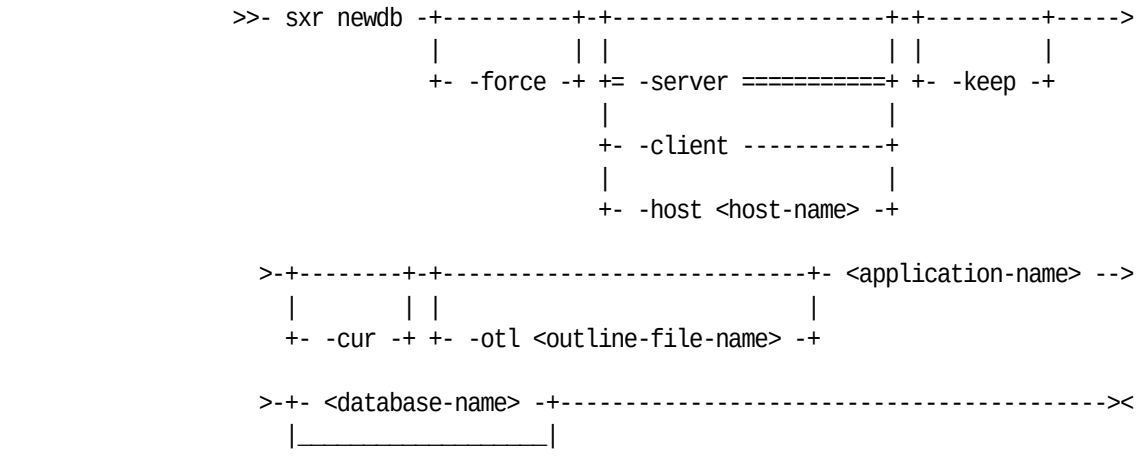
See also: sxr man essinit, sxr man newdb, sxr man trigger.

sxr newdb

NAME

sxr newdb - Create new Essbase databases.

SYNOPSIS



DESCRIPTION

Creates new Essbase databases named according to the supplied list within application <application-name>, which must exist. Any options given will apply to all databases to be created.

-force : The following two steps are performed before each database is created: 1) all locked objects in the database are unlocked; 2) the database is deleted. Incompatible with the -keep option.

-keep : No new databases are actually created. Instead, all supplied databases are restructured according to the new outline file, supplied with the '-otl' flag. Incompatible with the -force option, requires the -otl option.

-server : The default. The databases are created on the server machine in directory $ARBORPATH/app. The server machine is assumed to be identified by SXR_DBHOST environment variable.

-client : The databases are created on the client machine in directory $ARBORPATH/client. The client machine is assumed to be the local host, identified by the SXR_HOST environment variable.

-host : The subsequent parameter is treated as host name where the databases are to be created. This is especially useful for distributed tests where multiple server hosts are involved.

-cur : All named databases are created as currency databases.

-otl : The subsequent parameter is treated as an outline file, which is used to restructure each of the databases upon its creation. Required if -keep option is used.

```
TRIGGERS
        A set of post-action triggers may be associated with this command by
        setting environment variable SXR_NEWDB_TRIGGER to an
        Essexer-evaluatable command string.  A separate trigger will fire for
        each of the newly created databases.  Each trigger's environment
        will contain variables SXR_TRIGGER_ARG1 and SXR_TRIGGER_ARG2 set
        respectively to the application name and the database name for
        which it is firing.  See 'sxr man trigger' for more information.

ENVIRONMENT
        Must be in an Essexer view.

NOTES
See also sxr essinit, sxr newapp.
```

sxr newview

```
NAME
        sxr newview - Create a new Essexer view.

SYNOPSIS
        >>-- sxr newview <directory_name> --------------------------------><

DESCRIPTION
        Creates a new Essexer view named <directory_name>, which cannot be an
        existing directory. For example,

        Directory <directory_name> is created with the requisit Essexer view
        directory structure.  A sample view initialization file .sxrrc is
        placed in the root of the view.

        Recursive 'sxr goview' command is issued upon successful creation of a
        new view.

EXAMPLE
          sxr newview views/orion/v21

        Assumes existence of directory ./views/orion, but not directory v21
        under it.  Creates Essexer view directory structure in directory
        ./views/orion/v21.

ENVIRONMENT
        Cannot already be in an Essexer view.

NOTES
        See also: sxr goview, sxr inview.
```

sxr partner

```
NAME
        sxr partner - command line interface to QA Partner scripts

SYNOPSIS
        >>- sxr partner ---<argument> --+------------------------+--><
                                        |                        |
                                        +-+- <more-arguments> -+-+
                                    |_____________________|

DESCRIPTION
       Starts QA Partner scripts from the shell command line. Accepts the same
       arguments as partner.exe.

        <argument>
          A minimum of one mandatory parameter (script name).

        <more-arguments>
          Other arguments this and future releases of QA Partner support.


RETURN STATUS
        "0" operation completed successfully
        "1" Unsuccessful operation, e.g. partner.exe does not exist
        "2" A user error occurred, e.g. no application name supplied

TRIGGER
        No triggers supported before or after this command.

ENVIRONMENT
        Must be in an Essexer view. Valid QA Partner environment must be
        available (accessible).

SEE ALSO
        QA Partner documentation.
```

sxr perl

```
NAME
        sxr perl - Execute a perl script file.

SYNOPSYS
        >>-- sxr perl <file-name> -+------------------+------------------><
                                   |                  |
                                   +-+- <argument> -+-+
                                     |______________|

DESCRIPTION
        A directly executable perl script <file-name> is executed in the
        current view session.  The <file-name> is searched for on the shell
        path ($SXR_PATH_SHELL).
```

The standard output and standard error streams are merged and are sent to both the screen and a log file, whose name is constructed to have the base name of <file-name> and extention of '.sog'.

ENVIRONMENT
Must be in an Essexer view.

TRIGGERS
A post-action trigger may be associated with this command by setting environment variale SXR_PERL_TRIGGER to a shell-executable command string. Trigger's enviroment will contain variables SXR_TRIGGER_ARGC and SXR_TRIGGER_ARG1 through SXR_TRIGGER_ARG<n>, respectively set to the number <n> of the arguments passed to the shell command and all those arguments. Thus SXR_TRIGGER_ARG1 contains the name of the shell script passed to 'sxr perl'. See 'sxr man trigger' for more information.

NOTES
See also 'sxr man path'.

refresh

NAME
refresh - engineering Essbase installer

SYNOPSIS
refresh [-n] [-l] branch[:version] [debug | optimized] [client | server | both]

DESCRIPTION
Refresh script provides uniform and programmable interface for installing essbase builds across projects and platforms. Works in conjunction with a build tree.

ARGUMENTS:
With no arguments, refresh displays the usage information.
At least the name of the branch (e.g. eclipse or sunspot or 5.0.1) must be specified. If version is omitted, the latest available build for this branch will be installed. For example, "refresh 6.0.0" is the same as "refresh eclipse:latest both". Either internal (levi) or external (5.0.1) version can be specified. Instead of "latest" one can request a specific date of the build for the on-going releases or the patch level for the historic ones. Branch and version have to be separated by colon.

"debug" or "optimize". Not all builds have both debug and optimized builds available. Patches generally have only optimized versions. Error will be returned if the specifically requested version does not exist, even if the other is there. The default is "optimized".

"both" is the default option. Client option is meaningful only for MS Windows. Under UNIX option both(again, default) will install server only, option client will install nothing. If both is chosen, server installs after client, so that server dlls take precedence.

```
The refresh script will preserve your security file and the contents
of the app directory. bin and locale directories are completely re-created
as part of refresh, so that no “leftover” files remain from previous installs.

OPTIONS:
-n returns branch:version string if branch:version exists, empty string if
does not. In NT it also translates branch:latest into branch:date format.

-l branch returns all versions of SERVER available for install in format
branch:version server opt

Neither -n nor -l option will remove or copy any files on the machine.


REQUIREMENTS for using refresh script are as follows:

1. Windows

1.1. MKS version 6.1 or above.
1.2. NFS client installed and configured
1.3. NFS-mount the remote file system containing build trees
1.4. BUILD_ROOT environment variable defined, pointing to the above mount point
Let’s assume that you’ve mounted the builds filesystem as letter V:/ on your
system.  Then the value of BUILD_ROOT should be V:/builds.

1.5. ARBORPATH variable defined and directory exists

2. UNIX
Steps 1.3 though 1.5 above. Substitute the drive letter notation for share
name path.


NOTES:
Refresh works in conjunction with the build tree.
Globalc version 3 is installed for eclipse (solaris and NT as of April 99).
Levi installs only 16 bit client for now.
Last of the mutually exclusive options takes precedence.
It is a bad idea (but a bug in refresh) to use refresh while essbase
is running. If you are installing in UNIX for the first time and
your default shell is csh, do rehash before running essbase.


EXAMPLES:
refresh eclipse:99_01_19 client

refresh 5.0.2:ga server

refresh -l eclipse (will take a while)

refresh $(refresh -l levi | grep 11) will be a problem if both debug
                                                and optimize exist
```

sxr runrept

```
NAME
        sxr runrept - Run a Essbase Report Writer script file.

SYNOPSIS
        >>-- sxr runrept -+--------------------+-+--------------------+-->
                          |                    | |                    |
                          +- -host <host-name> -+ +- -user <user-name> -+

         >----------------+-----------------------+---------------------->
                          |                       |
                          +- -password <password> -+

         >--------------- <application-name> <database-name> -------------->

         >--------------- <script-file-name> <output-file-name> ----------->

         >-------------+--------------------------------+---------------><
                       |                                |
                       +-+- <old_string>=<new_string> -+-+
                         |_______________________________|

DESCRIPTION
        <host-name>
              The name of the host at which the report is to be executed.
              If omitted, $SXR_DBHOST is assumed.

        <user-name>
              Essbase user name in whose security domain the report is to be
              executed. If omitted, $SXR_USER is assumed.

        <password>
              <user-name>'s password. If omitted, $SXR_PASSWORD is assumed.

        <application-name>
              The name of the application to be selected before the report is
              run.

        <database-name>
              The name of the database to be selected before report is run.

        <script-file-name>
              The name of the report script file. This name is expanded along
              the '-rep' path.

        <output-file-name>
              The name of the output file to be created in $SXR_WORK directory.

        <old_string>
              Arbitrary character string to be replaced in the
              <script-file-name> with the value of <new-string>.

ENVIRONMENT
        Must be in an Essexer view.
```

NOTES

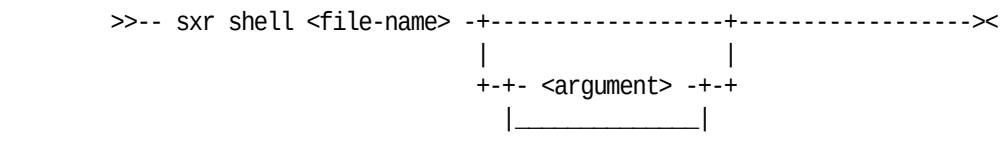
See also “sxr man path”.

sxr shell

NAME

sxr shell - Execute a Shell script file.

SYNOPSYS



DESCRIPTION

A directly executable <file-name> is executed in the current view session. The is searched for on the shell path ($SXR_PATH_SHELL).

The standard output and standard error streams are merged and are sent to both the screen and a log file, whose name is constructed to have the base name of <file-name> and extention of ‘.sog’.

ENVIRONMENT

Must be in an Essexer view.

TRIGGERS

A post-action trigger may be associated with this command by setting environment variale SXR_SHELL_TRIGGER to a shell-executable command string. Trigger’s enviroment will contain variables SXR_TRIGGER_ARGC and SXR_TRIGGER_ARG1 through SXR_TRIGGER_ARG<n>, respectively set to the number <n> of the arguments passed to the shell command and all those arguments. Thus SXR_TRIGGER_ARG1 contains the name of the shell script passed to ‘sxr shell’. See ‘sxr man trigger’ for more information.

NOTES

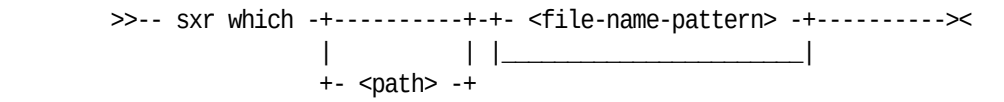
See also ‘sxr man path’.

sxr which

NAME

sxr which - Expand file pattern(s) along an Essexer path.

SYNOPSIS



```
DESCRIPTION
        Takes a list of file names or name patterns and successively attempts
        to locate them on the directory list defined by the '-<path>' flag.

        <path> ::= -+- -csc --+-
                   +- -data -+
                   +- -doc --+
                   +- -log --+
                   +- -rep --+
                   +- -sh ---+
                   +- -sxr --+

        <file-name-pattern>

        Uses the usual shell file name expansion wild cards, as described in
        sh(1).  Quoting is usually needed to prevent expansion by the invoking
        shell.

        For every element of the file name pattern list the following logic is
        carried out:

        If <file_name_pattern> begins with '/', then
            No path lookup is performed; all the files matching the pattern
            are displayed (example A).

        otherwise
            For every directory on the specified path from left to right:
                Expand the current <file_name_pattern>;
                For every full file name thus obtained:
                    If the non-qualified file name has not yet been located,
                    then print it.


        Return status:

        0 : File was found
        1 : File was not found, but no error occurred
        2 : Error occurred, e.g. wrong path flag passed.

EXAMPLES
        A) Let the contents of /home/sports/ be:
              foo
              football
              soccer
              foobar/

           and the contents of /home/sports/foobar/ be:
              baseball
              handball
              hockey

            The command
              sxr which "/home/sports/foo*" "/home/sports/foobar/*ball"

            will return:
```

```
            /home/sports/foo
            /home/sports/football
            /home/sports/other/baseball
            /home/sports/other/handball

      B) Let the contents of SXR_PATH_SCR be:
         /home/views/v1/scr; /net/base/scr

         and the contents of /home/views/v1/scr be:
           football
           handball

         and the contents of /net/base/scr be:
           football
           baseball

         The command
            sxr which -scr “foo*” “*ball”

          will return:
           /home/views/v1/scr/football
           /home/views/v1/scr/handball
           /net/base/scr/baseball

ENVIRONMENT
      Must be in an Essexer view.

NOTES
      You must quote file name patterns to prevent their expansion by the
      calling Shell.
```

4.3 Additional Topics

path

```
NAME
      path - Essexer’s path-oriented paradigm.

SYNOPSYS
      None.

DESCRIPTION
      Essexer uses path-based file handling paradigm, whereby users can
      only reference files by their unqualified names and Essexer performes
      the appropriate translation based on the search path environment
      variables.

      As a general rule, as many search path environment variables are
      supported as there are subdirectories in the basic view structure.
      These paths are:
```

```
        Environment    sxr   Description
        variable       flag
        ------------   ----  ------------------------------------------------
        SXR_PATH_CSC   -csc  Calc scripts.
        SXR_PATH_DATA  -data Various files not contained in other directories,
                             such as outline files, security files, load data
                             files, etc.
        SXR_PATH_LOG   -log  Reference log files.
        SXR_PATH_REP   -rep  Essbase reports.
        SXR_PATH_SCR   -scr  Essbase ESSCMD scripts.
        SXR_PATH_SH    -sh   Shell scripts, usually tests.

        Simiraly to the regular Shell path variables, each of the above
        contains a list of directories separated by a semicolon.  When looking
        for a file, Essexer scans the appropreate list from left to right until
        it finds the file in the directory from the path list.  Also simiraly
        to the Shell path mechanism, no path lookup is performed for fully
        qualified file names, i.e. the command

          sxr which -sh /some/not/even/necessarily/a/shell/file

        will return precisely the file itelf,
         /some/not/even/necessarily/a/shell/file, provided that it exists.

        In contrast with the regular Shell path handling, Essexer supports not
        just file names, but file name patterns.  See 'sxr man which' for more
        information.

ENVIRONMENT
        Usually, to take advantage of the paths you have to be in an
        Essexer view.
NOTES
        See also: sxr which, sxr ffetch.
```

trigger

```
NAME
        trigger - Essexer post-command trigger

DESCRIPTION
        The following commands may have post-action triggers to be associated
        with them at run time: diff, newapp, newdb, goview, esscmd, and shell.

        Triggers are defined by setting the appropreate environment variable to
        a string which is evaluated as a subshell via 'ksh -c' at the time
        the trigger is firing.  All triggers fire after all the usual work has
        been performed by the respective command.  For example, the on-newdb
        trigger fires after the database has been created.

        Triggers are made aware of the firing context via the following
        environment variables:

        SXR_TRIGGER_ARGC   - Number <n> of arguments passed to the trigger.
        SXR_TRIGGER_ARG1 through SXR_TRIGGER_ARG<n>
                           - Arguments as passed to the firing command.
```

```
Commands newdb and newapp fire a separate trigger for every database
or application they create respectively.

The following is the summary table of the triggers functionality:

ON-DIFF    :
defined by : SXR_DIFF_TRIGGER environment variable
fires      : once
arguments  : SXR_TRIGGER_ARGC - number of arguments passed to 'sxr
                 diff' command, starting with the test log file name.
             SXR_TRIGGER_ARG1 - test log file name.
             SXR_TRIGGER_ARG2... - reference log file names as passed
                 to 'sxr diff' command.
sample use : May be used to do additional logging if a difference
             was found:
             export SXR_DIFF_TRIGGER=\
                  'if (test -f "${SXR_TRIGGER_ARG1%%.*}.dif")     \
                   then                                          \
                       echo Failed to compare $SXR_TRIGGER_ARG1  \
                            >> mylogfile.log                     \
                   fi'

ON-GOVIEW  :
defined by : SXR_GOVIEW_TRIGGER environment variable
fires      : once
arguments  : SXR_TRIGGER_ARGC - always set to 1
             SXR_TRIGGER_ARG1 - the name of the view being entered.
sample use : May be used to kick-off a certain test, say accxmain.sh,
             immediately upon entering the view:
             export SXR_GOVIEW_TRIGGER='sxr sh accxmain.sh'

ON-ESSCMD  :
defined by : SXR_ESSCMD_TRIGGER environment variable
fires      : once
arguments  : SXR_TRIGGER_ARGC - number of arguments passed to 'sxr
                 esscmd' command, starting with the script name and
                 including all the substitutions.
             SXR_TRIGGER_ARG1 - the name of the script being executed.
             SXR_TRIGGER_ARG2... - rest of arguments to 'sxr esscmd'.
sample use : ??
             export SXR_ESSCMD_TRIGGER=''

ON-NEWAPP  :
defined by : SXR_NEWAPP_TRIGGER environment variable
fires      : for each application
arguments  : SXR_TRIGGER_ARGC - always set to 1
             SXR_TRIGGER_ARG1 - the name of the application
sample use : May be used to set certain application-wide settings,
             which may not be a regular part of the test:
             export SXR_NEWAPP_TRIGGER=\
                 'sxr esscmd setappst.scr %APPNAME=$SXR_TRIGGER_ARG1'

ON-NEWDB   :
defined by : SXR_NEWDB_TRIGGER environment variable
fires      : for each database
```

```
arguments  : SXR_TRIGGER_ARGC - always set to 2
             SXR_TRIGGER_ARG1 - the name of the application
             SXR_TRIGGER_ARG2 - the name of the database
sample use : May be used to set certain database-wide settings,
             which may not be a regular part of the test:
             export SXR_NEWDB_TRIGGER=\
                 'sxr esscmd setdbsti.scr %APPNAME=$SXR_TRIGGER_ARG1'

ON-SHELL   :
defined by : SXR_SHELL_TRIGGER environment variable
fires      : once, for the top level shell script only.  If a shell
             script file invokes further shell scripts via
             'sxr sh ...', those will not cause the on-shell trigger
             to fire.
arguments  : SXR_TRIGGER_ARGC - number of arguments passed to 'sxr
                 shell' command, starting with the script name and
                 including all its arguments.
             SXR_TRIGGER_ARG1 - the name of the shell script being run.
             SXR_TRIGGER_ARG2... - rest of arguments to 'sxr shell'.
sample use : May be used to save off the results of a run to a backup
             location:
             export SXR_SHELL_TRIGGER='cp -r $SXR_WORK/* /backups/'
```