SCSI TOOLBOX, LLC

VCPSSL Threaded Functions - Advanced

APIs to Run Different Test Sequences to Different Drives

Normal testing on a drive involves running the exact same test on all drives under test – for example running a Write Test on all drives. But at times you want to run a Write test on some drives, while running a Read test on others.

As an overview of how to accomplish running different tests on different drives, you will basically follow the following steps:

- Add tests to test-sequence-0
- Add tests to test-sequence-1
- …….
- Add tests to test-sequence-N

- Add all the devices you will be testing via VCSCSIAddDiskDeviceToBeTested

- Associate disk-drive-0 to a particular test-sequence
- Associate disk-drive-1 to a particular test-sequence
- ……
- Associate disk-drive-M to a particular test sequence

- Start the test sequence on each disk-drive

API: VCSCSISetActiveTestSequenceNumber

```
BOOL VCSCSISetActiveTestSequenceNumber(int nTestSequenceNumber)
```

The above API tells the library which test sequence you will be adding tests to. It is important to create your test sequences sequentially, that is to create test-sequence-0, then test-sequence-1, and so forth. That is call VCSCSISetActiveTestSequenceNumber(0), then add your tests (which will add them to test-sequence-0), then call VCSCSISetActiveTestSequenceNumber(1), then add your tests (which will add them to test-sequence-1), and so forth.

Return Value:
TRUE – the library successfully has made nTestSequenceNumber the active test sequence. All tests added from this point on will be added to this test sequenc (until the next call to VCSCSISetActiveTestSequenceNumber).

FALSE – the library was NOT successful in making nTestSequenceNumber the active test sequence. The cause of this is you did not create your test sequences sequentially (i.e. you skipped a test sequence); another cause is that a drive is currently executing the specified test sequence.

API: VCSCSIGetActiveTestSequenceNumber

```
int VCSCSIGetActiveTestSequenceNumber()
```

The above API simply returns the active test sequence number. If you have not called VCSCSISetActiveTestSequenceNumber, then it will return 0 (which is the default test sequence number).

API: VCSCSIRemoveAllTestsFromTestSequence

```
BOOL VCSCSIRemoveAllTestsFromTestSequence(
                                        int nTestSequenceNumber)
```

This API removes all the tests from test-sequence nTestSequenceNumber.

Return Value:

TRUE – the library removed all the tests from the test sequence

FALSE – the library did NOT remove all the tests from the test sequence. Cause of this is there is NO test sequence nTestSequenceNumber; another cause is that a drive is currently executing the specified test sequence.

API: VCSCSISetDeviceTestSeqNumber

```
BOOL VCSCSISetDeviceTestSeqNumber(int nHA,int nTid,int nLun,
                                  int nTestSeqNumber)
```

This API is the API you use to tell the library which test-sequence you want the drive identified by nHA:nTid:nLun to run. The testing does not start until you call VCSCSIStartTestSequenceOnDrive (that is VCSCSISetDeviceTestSeqNumber simply tells the library out of the N-many test sequences that are defined, which test sequence is this drive suppose to run).

IMPORTANT: You can associate more than 1 drive to a particular test sequence!

The following code is valid:
VCSCSISetDeviceTestSeqNumber(4,4,0,2); //drive 4:4:0 is to run test-sequence #2
VCSCSISetDeviceTestSeqNumber(4,5,0,2); //drive 4:5:0 is to run test-sequence #2

Return Value:

TRUE – the library successfully associated test-sequence nTestSeqNumber to disk drive nHA:nTid:nLun

FALSE – the library was NOT successful in associating test-sequence nTestSeqNumber to disk drive nHA:nTid:nLun. Cause of this is either there is no drive nHA:nTid:nLun in the devices to be tested (possibly you forgot to call VCSCSIAddDiskDeviceToBeTested?); or there is no test-sequence nTestSeqNumber; or the device is currently executing a test sequence (you cannot change the test sequence number until the disk drive is done testing).

API: VCSCSIStartTestSequenceOnDrive

```
BOOL VCSCSIStartTestSequenceOnDrive(int nHA,int nTid,int nLun)
```

This API allows you to begin testing on a specific drive. That is, you do NOT need to start all the testing on all the drives at the same time. You can still call VCSCSIStartDiskTestSequence to start all testing on all drives. But if you wish to "stagger" the testing you need to call VCSCSIStartTestSequenceOnDrive.

Return Value:

TRUE – the library successfully started the testing on drive nHA:nTid:nLun

FALSE – the library was NOT successful in starting the testing on drive nHA:nTid:nLun. Cause is that there is NO drive nHA:nTid:nLun (did you forget to call VCSCSIAddDiskDeviceToBeTested?); or the device is already executing a test sequence (you must wait until it is done before starting a new test sequence on the drive).

API: VCSCSIAreAllTestsOnAllDrivesDone

```
BOOL VCSCSIAreAllTestsOnAllDrivesDone(
                            int nNumSleeps = 0,
                            int nNumMilliSecondBetweenSleep = 1000,
                            BOOL bCallCMQ = TRUE)
```

This API allows you to ask the library to tell you whether or not all the tests are done on all the drives (versus you having to poll every drive). You pass the API several parameters to tell the library how frequent you want the library to check the devices for whether or not they are complete. The library will check every nNumMilliSecondBetweenSleep milliseconds for a total number of nNumSleeps-many times.

IMPORTANT: If all testing on all drives is complete the library returns TRUE right away (that is it does not continue polling the total number of times).

Return Value:

TRUE – all testing on all drives has completed
FALSE – atleast one drive is still executing its test sequence

nNumSleeps:

This parameter is the number of times you want the library to check on the status of the drives. Passing in a value of 0 means we return immediately the status of all the testing (that is we do not poll the drives).

nNumMilliSecondBetweenSleep :

This parameter is how frequent you want the library to check the status of the drives (i.e. how much time to let elapse between checks)

bCallCMQ :

Set this parameter to TRUE to tell the library to clear the Window's message queue.

Example: VCSCSIAreAllTestsOnAllDrivesDone(10,1000,TRUE);
This basically asks the library to check the drives 10 times, at 1 second intervals on the status. At the first indication that all the testing is done, the library will return TRUE (and this could happen on any of 10 checks for status!!). But if after the full 10 seconds, if atleast one drive is still executing its test sequence, the library will return FALSE.

API: VCSCSIAreAllTestsOnThisDriveDone

```
BOOL VCSCSIAreAllTestsOnThisDriveDone(
                            int nHA,int nTid,int nLun,
                            int nNumSleeps = 0,
                            int nNumMilliSecondBetweenSleep = 1000,
                            BOOL bCallCMQ = TRUE);
```

This API is exactly like VCSCSIAreAllTestsOnAllDrivesDone except you pass in the device address and the library checks on the status of that specific drive.

Return Value:

TRUE - all testing on the drive nHA:nTid:nLun has completed
FALSE - the drive nHA:nTid:nLun is still executing its test sequence

nNumSleeps:

This parameter is the number of times you want the library to check on the status of the drive. Passing in a value of 0 means we return immediately the status of all the testing (that is we do not poll the drive).

nNumMilliSecondBetweenSleep :

This parameter is how frequent you want the library to check the status of the drive (i.e. how much time to let elapse between checks)

bCallCMQ:

Set this parameter to TRUE to tell the library to clear the Window's message queue.

API: VCSCSISetHWND

```
BOOL VCSCSISetHWND(HWND hWnd)
```

This API allows you to set which window the library is to clear the Window's messages from (when you instruct the library to do so)

The "Command Probability Sequencer" Test

The Command Probability Sequencer (CPS) Test is a test that allows you to define a set of commands and assign a numerical value to each command to indicate "how frequent you want the command to be issued". An example would be helpful – say you want to issue a bunch of Writes and Reads and every now-and-then "inject" an odd command like a "Log Sense" command or a "Mode Sense" command. And suppose you want basically the "Log Sense" and "Mode Sense" commands to be issued about 1 out of every 1000 commands. So, in every 1000 commands you want 1 "Log Sense", 1 "Mode Sense", 499 "Writes" and 499 "Reads". So what you do is you assign a "probability" to each command according to how frequent you want the command to appear. In our example, we want the "Log Sense" command to appear 1/1000 (or 0.001) of the time, the "Mode Sense" command to appear 1/1000 of the time, and the Write and Read commands to appear 499/1000 of the time. What the CPS Test guarantees is that in the long run, over millions and millions of I/Os, these 4 commands will appear 0.1%, 0.1%, 49.9% and 49.9% of the time. Let's now take a look at the actual API.

API: VCSCSIAddDiskComProbSeqTest

```
BOOL VCSCSIAddDiskComProbSeqTest(
                              DMM_UserDefinedCDB * pArrayUserDefinedCDB,
                              double * pArrayOfProbabilities,
                              int nLenOfArrays,
                              long lNumIOToIssue,
                              int nRandomOrSeq)
```

pArrayUserDefinedCDB:

This parameter is a pointer to an array of data structures of type DMM_UserDefinedCDB. This data structure has the following fields:

```
BOOL bValid;
eUSER_DEFINED_TYPES eUserDefinedType;
char cCDBBytes[16];
int nCDBLength;
int nDataDir;
int nTimeout;
int nTransferLength;
BYTE * pPayloadDataToDrive;
int nAmtDataToLogfile;
char cDataOutFile[MAX_PATH];
ePATTERN_TYPE eTestPattern;
char cPatternFile[MAX_PATH];
BOOL bOverlayLBA;
BOOL bCompare;
```

```
int nGap;
int nSeed;
int nNumKeysToIgnore;
int nListOfKeysToIgnore[5];
BOOL bOverlayKey;
int nOverlayKey;
__int64 i64HighBlockForTest;
```

bValid:
Set this to TRUE

eUserDefinedtype:
Set this to eScsiCDB

cCDBBytes:
Fill in the specific CDB in this field (this CDB is for 1 command, NOT all your commands)

nCDBLength:
Set this to the length of your CDB (should be 6, 10, 12, or 16)

nDataDir:
Set this to field to the below values:
0 if there is data going out to the drive
1 if there is data being retrieved from the drive

NOTE: If no data is being transferred, set this field to 0

nTimeout:
Set this to the desired timeout for the command

nTransferLength:
Set this to how much data is being transferred (if the command sends data to the drive then this field needs to be set to how much data is being sent to the drive; on the other hand if the command receives data from the drive then this data needs to be set to how much data is to be received; if no data is being transmitted set this field to 0)

pPayloadDataToDrive:
Many times you need to send data to the drive. An example would be a "Mode Select" command. Notice pPayloadDataToDrive is a pointer to an array of BYTE, so you need to fill in your buffer prior to calling this API.

eTestPattern:
Most the time, the data going out to a drive is just a specific pattern (like eIncrementing or eDecrementing). For example in a Write

command, you just tell the library what pre-defined pattern you want to ship to the drive (that is you don't fill out pPayloadDataToDrive!)

cPatternFile:
If the pattern is not one of the pre-defined patterns, and you want a user-defined pattern to go to the drive, then set this field to the location (fully qualified path and filename) of the file. Note in this case you must set eTestPattern to eUserPatBinary

bOverlayLBA:
If you want the pattern to have the LBA overlayed on the first 4 bytes of each block, set this field to TRUE

bCompare:
On Read commands, if you want the library to compare the data read to a specific pattern, then set this field to TRUE

nGap:
If you want the library to add a specific value to the LBA in your command, set nGap to how much you want the library to add. The CDB you specify might have LBA = 0, but you don't necessarily want every time this command is selected for the library to issue the exact same command. For example, if you want to Write at LBAs 0, 16, 32, 48, 64, …. (notice the gap between each LBA is exactly 16) then you'd set nGap to 16.

nSeed:
Many operations requires the library to select a random number. The random-number generator is seeded to 0. To seed the random number generator to something other than 0, pass in your value in nSeed

nNumKeysToIgnore:
The CPS test has the ability to ignore certain sense keys. What this means is you can tell the library to ignore, for example, the sense key of 06 ("Unit Attention") and "if" in the middle of a test that sense key is received, the test will simply ignore the check-condition and let the test continue (the default behavior is to fail the test when a check-condition is received).

nListOfKeysToIgnore:
Fill in your sense keys in this field. For example, if you want to ignore sense key 06 ("Unit Attention") and 03 ("Media Error") then you set nNumKeysToIgnore to 2, and nListOfKeysToIgnore[0] = 0x06, nListOfKeysToIgnore[1] = 0x03

bOverlayKey:
For better identify whether a block of data is "your" block or not, the library can overlay a user-defined "key" on the block. This key is 4 bytes long and is overlayed on bytes 4-thru-7 of the block. A typically usage might be one where you have a unique key for each department in your company, and each department, when writing data to the drive, overlays their unique key. Then say 1 day later when they read the data from the drive, if their unique key is not in the block, then they know it is NOT the data they wrote to the drive. Overlaying the key is one way to answer the question "How do we know a block of zeroes is really the block of zeroes we wrote?"

nOverlayKey:
This is the value of the key to overlay. It can be any 32-bit value. By way of an example, suppose your key is 0x02001A7F and you overlay the LBA too. If you were writing the pattern eAllZeroes then here is what two blocks of data would look like:

```
Block 0x000003a4:
00 00 03 A4 02 00 1A 7F 00 00 00 00 00 00 00 00 …..

Block 0x721AB032:
72 1A B0 32 02 00 1A 7F 00 00 00 00 00 00 00 00 ……..
```

i64HighBlockForTest:
Set this to the high block for the test (that is whenever any command is chosen from your list of commands, and the nGap field is applied to the command (if applicable), if the LBA goes beyond i64HighBlockForTest then the library will wrap the LBA back to the Start Block (which is usually 0).

pArrayOfProbabilities:

This parameter is a pointer to an array of probabilities for each command. Note it is important that the length of this array must match exactly the length of pArrayUserDefinedCDB. Note also that the sum of these probabilities must add up to exactly 1.0

nLenOfArrays:

This parameter is how many commands are in your array pointed to by pArrayUserDefinedCDB. It is important that the pointer pArrayOfProbabilites has length nLenOfArrays.

lNumIOToIssue:

This parameter is how many blocks of data or commands from your list you want the library to issue. If you have a Write command that transfers 32 blocks of data, then when this command is issue it counts

as 32 towards the total value lNumIOToIssue. But, for example, issuing a "Log Sense" command counts as 1 IO issued.

nRandomOrSeq:

This parameter gives you the flexibility to issue the commands in sequence (i.e. issue command 0 from your list, then issue command 1 from your list, then issue command 2 from your list, and so forth) or to issue the commands randomly. Pass in the following values for nRandomOrSeq:

0 – Issue the commands randomly according to the probabilities assigned
1 – issue the commands sequentially

Example of using API VCSCSIAddDiskComProbSeqTest:

```
//In the example below, we will set up only 2 commands, a Write
//command and a Read command.  We will set the probability of
//the Write command to 12% and that of the Read command to
//88%

DMM_UserDefinedCDB arrOfCDB[2];
Double arrOfProb[2];

//Set up the Write command
memset(&arrOfCDB[0],0,sizeof(DMM_UserDefinedCDB));

arrOfCDB[0].bValid = TRUE;

arrOfCDB[0]. eUserDefinedType = eScsiCDB;

arrOfCDB[0].cCDBBytes[0] = 0x2A;
arrOfCDB[0].cCDBBytes[1] = 0x00;
arrOfCDB[0].cCDBBytes[2] = 0x00;
arrOfCDB[0].cCDBBytes[3] = 0x01;  //LBA=0x00010000
arrOfCDB[0].cCDBBytes[4] = 0x00;
arrOfCDB[0].cCDBBytes[5] = 0x00;
arrOfCDB[0].cCDBBytes[6] = 0x00;
arrOfCDB[0].cCDBBytes[7] = 0x00;
arrOfCDB[0].cCDBBytes[8] = 0x01;  //1 block transfer
arrOfCDB[0].cCDBBytes[9] = 0x00;

arrOfCDB[0].nCDBLength = 10;

arrOfCDB[0].nDataDir = 0;

arrOfCDB[0].nTimeout = 10;

arrOfCDB[0].nTransferLength = 512;

arrOfCDB[0].eTestPattern = eIncrementing;  //Write "incrementing" pat
```

```
arrOfCDB[0].nGap = 2;  Skip 2 blocks between each Write command


//Set up the Read command
memset(&arrOfCDB[1],0,sizeof(DMM_UserDefinedCDB));

arrOfCDB[1].bValid = TRUE;

arrOfCDB[1]. eUserDefinedType = eScsiCDB;

arrOfCDB[1].cCDBBytes[0] = 0x28;
arrOfCDB[1].cCDBBytes[1] = 0x00;
arrOfCDB[1].cCDBBytes[2] = 0x00;
arrOfCDB[1].cCDBBytes[3] = 0x01;  //LBA=0x00010000
arrOfCDB[1].cCDBBytes[4] = 0x00;
arrOfCDB[1].cCDBBytes[5] = 0x00;
arrOfCDB[1].cCDBBytes[6] = 0x00;
arrOfCDB[1].cCDBBytes[7] = 0x00;
arrOfCDB[1].cCDBBytes[8] = 0x01;  //1 block transfer
arrOfCDB[1].cCDBBytes[9] = 0x00;

arrOfCDB[1].nCDBLength = 10;

arrOfCDB[1].nDataDir = 1;  //1 = "receiving data"

arrOfCDB[1].nTimeout = 10;

arrOfCDB[1].nTransferLength = 512;

arrOfCDB[1].eTestPattern = eIncrementing;  //Write "incrementing" pat

arrOfCDB[1].nGap = 2;  Skip 2 blocks between each Write command

//Set up the probabilities.  Note .12 + .88 = 1.0
arrOfProb[0] = .12;  //12% probability for the Write
arrOfProb[1] = .88;  //88% probability for the Read

//Now add the CPS Test to the test sequence.  Note that the CPS Test
//itself is a single test, so it is just 1 test in your test
//sequence
VCSCSIAddDiskComProbSeqTest(&arrOfCDB[0],
                            &arrOfProb[0],
                            2,        //2 commands in our list
                            1000000,  //issue 1000000 commands
                            0 //issue commands randomly
                           );
```

Issuing a threaded CDB to a drive

Typically one would issue a CDB in its own thread so that the main window can continue doing other tasks while the drive processes the CDB. One really does this only if the CDB is going to take a long time – an example of this is issuing a format command where the format may take over an hour. The VCPSSL has 4 APIs that allow you to issue a CDB in its own thread – these are listed below.

API: VCSCSIIssueThreadedCDB

```
int VCSCSIIssueThreadedCDB(int nHA,int nTid,int nLun,
                           BYTE * pCDBBytes,int nCDBLen,
                           BYTE * pInOutBuf,int nInOutBufLen,
                           int nDirection,
                           int nTimeout);
```

Return Value:

The return value is a 32-bit integer that identifies the thread that is executing the CDB you passed in. As such, it is very important to save this value off so that the library will know which thread you are asking for status of, which thread you are done with.

pCDBBytes:

Pass in the CDB bytes in this parameter

nCDBLen:

Pass in the length of your CDB (should be 6, 10, 12, or 16)

pInOutBuf:

If you have data that needs to be shipped to the drive, the data must be stored in pInOutBuf. Similarly, if the CDB is expecting data from the drive, the data will be stored in this buffer. The buffer must exists after the call to VCSCSIIssueThreadedCDB and all the way until you release the resources for the thread via a call to VCSCSIReleaseThreadID

nInOutBufLen:

This is the length of the buffer pointed to by pInOutBuf

nDirection:

Pass in 0 if data is being shipped to the drive
Pass in 1 if data is being received from the drive
If no data is transfer, pass in 0

nTimeout:

Pass in the timeout value for the CDB

API: VCSCSIGetThreadedCDBStatus

```
int VCSCSIGetThreadedCDBStatus(int nThrdID)
```

Return Value:

The possible return values are:

0 (or eTestInProgress)
1 (or eCompleteOnSuccess)
2 (or eCompleteOnFailure)
3 (or eTestNotStartedYet)
10 (or eUnknownStatus)

`eTestInProgress: This means that when you made the call to VCSCSIGetThreadedCDBStatus` the library was currently executing that CDB on your drive.

eCompleteOnSuccess: The test has completed and it completed successfully. If you were expecting data from the drive, that data has already been stored into the buffer `pInOutBuf` (i.e. the buffer you passed to VCSCSIIssueThreadedCDB).

eCompleteOnFailure: The test has completed and it failed. You can get more information on what happened, for example sense data, by calling `VCSCSIGetThreadedCDBStatusWData.`

eTestNotStartedYet: This just means that the test has not started yet. This is not an error condition!

nThrdID:

This value MUST be exactly the return value from VCSCSIIssueThreadedCDB

API: VCSCSIGetThreadedCDBStatusWData

```
int VCSCSIGetThreadedCDBStatusWData(
                                    int nThrdID,
                                    BYTE * pSenseBuf,int nSenseBufLen,
                                    double * pTimeToDoCmd,
                                    BYTE * pInOutBuf,int nInOutBufLen)
```

Return Values:

The return values are identical to VCSCSIGetThreadedCDBStatus

nThrdID:

This value MUST be exactly the return value from VCSCSIIssueThreadedCDB

pSenseBuf:

This parameter can be NULL

nSenseBufLen:

This parameter is the length of the buffer pointed at by pSenseBuf. It can be 0.

pTimeToDoCmd:

This parameter can be NULL. The library records how long it took from issuing the CDB to the driver to the completion of the I/O

pInOutBuf:

This parameter can be NULL. This parameter is only used if the CDB was expecting data from the drive.

nInOutBufLen:

Length of the buffer pointed at by pInOutBuf

API: VCSCSIReleaseThreadID

```
BOOL VCSCSIReleaseThreadID(int nThrdID)
```

Calling this API is very important – you must let the library know that you are done with all the resources the library has allocated to issue the threaded CDB.

nThrdID:

This is the value returned from VCSCSIIssueThreadedCDB

Miscellaneous API

API: VCSCSIAddDiskExternalProgramTest

```
BOOL VCSCSIAddDiskExternalProgramTest(char * pProgramName)
```

The above API adds the external program test to your test sequence. What this means is that you can have the library call an application that you have written to perform any tasks that you want. As an example, suppose you want to read a certain mode page and store that information into the logfile. Once you have your application written, you can call it as a test step.

Important: The library will pass your application the following information so that you will know which device your application is being called by. The information passed in the command-line parameters is "ha=nn,tid=mm,lun=xx". An example would be "ha=4,tid=5,lun=0". Notice there are no spaces in the information we pass on the command line.

pProgramName:

This parameter should contain the fully qualified path and application name. The path should have no spaces in it. For example, avoid names like "c:\Program Files\MyApp\MyApplication.exe". Rather, store the application in a folder like "c:\Workdir\MyApp\MyApplication.exe" (notice there are no spaces)

API: VCSCSIAddDiskUserDefined

```
BOOL VCSCSIAddDiskUserDefined(DMM_UserDefinedCDB * pUserDefinedCDB)
```

pUserDefinedCDB:

This parameter is a pointer to a data structure of type DMM_UserDefinedCDB. This data structure has the following fields (we list only the fields that are applicable to this API):

```
BOOL bValid;  //Set this to TRUE
eUSER_DEFINED_TYPES eUserDefinedType;  //Set this to eScsiCDB
char cCDBBytes[16];  //The CDB Bytes that forms 1 command
int nCDBLength;      //Length of your CDB Bytes (6, 10, or 12)
int nDataDir;        //Set this to 0 if data out to the drive,
                     //1 if data is being retrieved from the drive,
                     //0 if no data is being transferred
int nTimeout;
int nTransferLength;  //How many bytes is being transferred
BYTE * pPayloadDataToDrive;  //Buffer containing the bytes to send to
                             //drive
```

API: VCSCSICanThisTestRun

```
BOOL VCSCSICanThisTestRun(int nQD,
                          int nDrvCnt,
                          int nTransSzInKB,
                          int nGiveRecommendationToParamNo,
                          int * pRecommendedValue);
```

The above API allows you to see if a particular set of testing can run reliably on your test system. A common problem is you want to run 64 drives with a queue-depth of 32 with large transfers. Your system may run out of resources. You call this API to "gauge" whether or not you may have problems running nDrvCnt-many drives, with a queue-depth of nQD, where each transfer has max size nTransSzInKB.

Return Value:

TRUE – your system should be able to run nDrvCnt-many drives with a queue-depth of nQD where each transfer may be up to nTransSzInKB

FALSE – your system may NOT be able to run nDrvCnt-many drives with a queue-depth of nQD where each transfer may be up to nTransSzInKB. You most likely will need to reduce either the number of drives under test, or your queue-depth, or how large your transfers are (OR a combination of all 3!!).

nQD:

This parameter is the desired Queue-Depth you wish to run at

nDrvCnt:

This parameter is the desired number of drives you wish to test

nTransSzInKB:

This parameter is the largest transfer size in KiloBytes you wish to transfer

nGiveRecommendationToParamNo :

This value should be one of the following 3 values :

```
0 – you want a new recommendation for the Queue-Depth
1 – you want a new recommendation for the Drive-Count
2 – you want a new recommendation for the Transfer-Size
```

pRecommendedValue :

In the event that your request for testing nQD,nDrvCnt,nTransSzInKB may cause problems, the library will tell you by how much you need to reduce a particular parameter. For example, if you pass in nGiveRecommendationToParamNo = 0, then you want the library to tell you by how much you need to reduce the Queue-Depth by.

Example:

```
int nNewRec;
BOOL bSuccess = VCSCSICanThisTestRun(32,64,64,0,&nNewRec);
if (bSuccess == FALSE)
{
  //nNewRec has the new value you need to use for Queue-Depth.
  //In this example, you need to reduce Queue-Depth from 32 down
  //to nNewRec
  //The following call should succeed
  bSuccess = VCSCSICanThisTestRun(nNewRec,64,64,0,&nNewRec);
}
```

Example:

```
int nNewRec;
BOOL bSuccess = VCSCSICanThisTestRun(32,64,64,1,&nNewRec);
if (bSuccess == FALSE)
{
  //nNewRec has the new value you need to use for Drive-Count.
  //In this example, you need to reduce Drive-Count from 64 down
  //to nNewRec
  //The following call should succeed
  bSuccess = VCSCSICanThisTestRun(32,nNewRec,64,1,&nNewRec);
}
```

API: VCSCSIGetDefaultDriver

```
int VCSCSIGetDefaultDriver()
```

This API returns the default driver which the library is using.

Return Value:

```
2 - NTPort
5 - StsClass
10 - STBTrace
```

API: VCSCSISetDefaultDriver

```
BOOL VCSCSISetDefaultDriver(int nDefaultDriver)
```

This API sets the default driver to the value you specify.

IMPORTANT: You must re-start your application if you change the default driver! You do NOT need to reboot your system!

nDefaultDriver:

This parameter must be set to one of the following values:

2 – NTPort
5 – StsClass
10 – STBTrace

API: VCSCSISetSenseKeysToAllow

```
int VCSCSISetSenseKeysToAllow(int * pListOfKeyAsqAscq,
                              int nNumTripletsInList);
```

The above API gives you the capability to prevent the test from stopping due to a particular combination of sense keys. For example, may be you want the library to ignore a "Unit Attention" (06/29/xx). So, for example, if a Write Test is in progress and for whatever reason a check-condition occurs and the sense data is 06/29/00 then the library can "ignore" it and let the test proceed.

pListOfKeyAsqAscq:

Note that pListOfKeyAsqAscq is a pointer to an array of integers. So for each triplet of sense keys you want the library to ignore, you will be passing in 3 integers.

You can pass in -1 to mean "ignore all combinations of this value" (see the Example below).

nNumTripletsInList:

This parameter is the number of triplets you pass in.

Return Value:

The number of triplets the library stored. The library will not accept any more than 5 triplets.

Example: Ignore the 06/29 combination

```
int arr[6];
//Ignore 06/29/xx
arr[0] = 0x06;
arr[1] = 0x29;
```

```
arr[2] = -1;  //The -1 means the library is to ignore all combinations 06/29/xx
//Ignore 03/21/00
arr[3] = 0x03;
arr[4] = 0x21;
arr[5] = 0x00;
VCSCSISetSenseKeysToAllow(&arr[0],2);
```

API: VCSCSIGetSenseKeysToAllow

```
int VCSCSIGetSenseKeysToAllow(int * pListOfKeyAsqAscq,
                              int nHowManyTripletsCanListHold);
```

The above API fills in the array of integers pointed at by pListOfKeyAsqAscq with the current triplets of sense data to ignore.

Return Value:

Returns the number of triplets the library stored in your array of integers

pListOfKeyAsqAscq:

This parameter must be a pointer to an array of integers. It will be filled up with the current triplets the library will ignore. It must have 3 integers for each triplet!

nHowManyTripletsCanListHold:

This parameter is the number of triplets your array can hold. The library will NOT fill in any more than you specify in this parameter (so if the library has 5 triplets and you specify 2 in this parameter, the library will fill in 2 triplets, NOT 5).

API: VCSCSIGetStatusArray

```
int VCSCSIGetStatusArray(int nHA,int nTid,int nLun,
                         int nTestNumber,
                         STATUS_ELEMENT * parrStatusElement,
                         int nNumStatusEltInYourArray);
```

The above API allows you to receive the last N-many I/Os that were issued by the library. The usual use of this API is that when an I/O fails, you can receive the exact I/Os that led up to the failure.

Return Value:

The return value is the number of Status Elements stored in your array. It will never exceed your passed in parameter nNumStatusEltInYourArray.

nTestNumber:

Remember the test numbers start at 0.

parrStatusElement:

This parameter must be a pointer to an array of STATUS_ELEMENT. This data structure has the following fields:

```
BYTE cCDB[16];
int nLenOfCDB;
int nStatusOfIO;
BYTE cSenseKey;
BYTE cSenseASQ;
BYTE cSenseASCQ;
```

NNumStatusEltInYourArray:

This parameter is the size of the array pointed at by parrStatusElement.

Example:

```
//Get the Status Elements for test number 2 for drive 4:5:0
STATUS_ELEMENT arrSE[100];
int nNumSEAvailable = VCSCSIGetNumEltsInStatusArray(4,5,0,2);
int nNumSEReturned = VCSCSIGetStatusArray(4,5,0,2,&arrSE[0],100);

//NOTE: nNumSEReturned will be guaranteed to be less than or equal to
//100 (since your array has size 100)
```

API: VCSCSIGetNumEltsInStatusArray

```
int VCSCSIGetNumEltsInStatusArray(int nHA,int nTid,int nLun,
                                  int nTestNumber);
```

Return Value:

The return value is the number of Status Elements the library has stored for drive nHA:nTid:nLun test number nTestNumber.