# Advanced Sample Programs in CompuScope SDKs

There are three GaGe Software Development Kits (SDKs) for user programming of GaGe CompuScope cards: one for C/C#, MATLAB, and LabVIEW. Each of these SDKs includes advanced sample programs that are provided in a separate sub-folder and that are not described within the standard CompuScope SDK documentation. These Advanced sample programs are described within this document. This document describes the advanced sample programs in a generic fashion that is not specific to any of the three SDKs.

The table below lists which sample programs are supported under which programming language. Some sample programs for VisualBasic.NET and Delphi are provided within the C/C# SDK. The user may use the C Sample programs as a guide to construct sample programs that are unavailable for VB.NET and Delphi. Usage of some advanced sample programs requires installation of optional eXpert firmware. These programs are indicated by an asterisk (*).

| | C | C# | VB .Net | LabVIEW | MATLAB | CVI | Delphi |
|---|---|---|---|---|---|---|---|
| **GageAcquire** | X | X | X | X | X | X | X |
| **GageCoerce** | X | X | | X | X | X | |
| **GageComplexTrigger** | X | X | | X | X | X | |
| **GageDeepAcquisition** | X | X | | X | X | X | |
| **GageMultipleRecords** | X | X | | X | X | X | |
| **GageMultipleSystems** | X | X | | X | X | X | |
| | | | | | | | |
| **Streaming Sample Programs*** | X | | | | | | |
| **GageMulRecAveraging*** | X | X | | X | X | X | |
| **GageHistogram*** | X | | | X | | | |
| **CsPrf** | X | X | | | | X | |
| **GageFastAcquire2Disk** | X | X | | X | X | | |
| **GageCallback** | X | | | | | X | |
| **GageEvents** | X | X | | | | X | |
| **GageAsTransfer** | X | X | | | | X | |
| **GageAdvMulRecEx** | X | X | | X | X | | |

To determine which CompuScope models are supported by which advanced sample programs, please check the online Advanced Functionality Matrix at:

http://www.gage-applied.com/products/eXpert_FPGA_technology/eXpert_compatibility_matrix.htm

# Optional Firmware images

Some CompuScope models have the ability to be reconfigured with optional alternative firmware allowing on-board processing of waveform data before they are transferred to PC RAM. Currently, these firmware options include eXpert Multiple Record Signal Averaging, and eXpert Data Streaming (PCI Express CompuScopes only). When a CompuScope is updated with an optional firmware image, the image information is stored in CompuScope non-volatile memory. This memory has space for two or three firmware images: the standard CompuScope operating image and up to two optional images. The contents of non-volatile memory may be queried by an application using the CsGet(hSystem, CS_PARAMS, CS_EXTENDED_OPTIONS, &i64ExOptions) call, where hSystem is the CompuScope system handle. CS_PARAMS and CS_EXTENDED_OPTIONS are constants defined in CsDefines.h and CsExpert.h respectively. I64ExOption is a 64-bit integer type variable that is filled with the results of the query. The lower 32 bits of the i64ExOption will contain information about first alternative image and the higher 32 bits will contain information about the second one. There are corresponding calls to query available firmware in MATLAB (CsMl_GetExtendedOptions.m) and LabVIEW (CsLv_GetExtendedOptions.vi).

Based on the information about available firmware images, the user application can decide which image to load. Firmware images are loaded from any SDK by bitwise ORing the CompuScope mode (1 for "Single", 2 for "Dual" , 4 for "Quad" and 8 for "Octal") with a specific constant that indicates the image number. For instance, to specify an alternative image from C, either the constants CS_MODE_USER1 or CS_MODE_USER2 (for images #1 or #2) should be bitwise ORed with the u32Mode member of CSACQUSITIONCONFIG structure before it is used in the CsSet() call. (Note that the firmware image is not actually loaded until a call is made to CsCommit()).

Each of the CompuScope SDKs (C/C#, MATLAB and LabVIEW) provides a programming example for each optional firmware image (currently FIR filtering and signal averaging). The programming sequence for the loading each image, which is described above for C, is illustrated within each programming example

## *Streaming Sample Programs*

Introduction

Gage PCI Express CompuScopes now support direct streaming of waveform data to PC RAM for display analysis or storage. Streaming is an alternative to conventional Memory Mode operation. In Memory Mode, waveform data are first acquired into on-board CompuScope memory and are later downloaded to PC RAM in a separate operation. Accordingly, at any given time in Memory Mode, on-board memory is either written-to or read-from – never both. By contrast, Gage PCI Express CompuScopes are equipped with dual-port memory that allows on-board memory to be simultaneously written-to and read-from. Using this architecture, a data stream may be established that flows continuously though the on-board memory, which is used as a large FIFO(*First In First Out*).

The CompuScope C/C# SDK includes several sample programs that illustrate data streaming using PCI Express CompuScopes. In order for these sample programs to operate, the PCI Express CompuScope hardware must be equipped with the *eXpert Data Streaming* option. If your PCI Express CompuScope is not equipped with this eXpert Data Streaming, please contact your Gage Sales Agent about procuring this option, which may be deployed remotely with no factory upgrade required.

This document provides an overview and description of the operation of the streaming sample programs, as well as a description of the four new CompuScope API methods that are used within these sample programs to implement streaming.

CompuScope Streaming Operation

The CompuScope C/C# SDK includes three separate sample programs that illustrate streaming usage. These sample programs are listed in the table below:

| **Sample Program** | **Description** | **Comments** |
|---|---|---|
| *GageStream2Analysis_Simple* | Simple, single-threaded streaming acquisition from a single CompuScope | Best starting point for single CompuScope streaming software application. Analysis performed on raw waveform data |
| *GageStream2Analysis* | Multi- threaded streaming acquisition from a single CompuScope or Master/Slave CompuScope System | Best starting point for multi-CompuScope streaming software application. Analysis performed on time-stamp data |
| *GageStream2Disk* | Multi- threaded streaming acquisition and storage from a single CompuScope or Master/Slave CompuScope System | Streams waveform data to hard drive. Used to verify integrity of streamed data. |

The first two sample programs are designed to serve as starting points for user software applications and include the word “analysis”, which is used in the most general sense and could refer to any software process that consumes the data stream. This process could be actual numerical data analysis or other processes, such as data display or data transfer to a separate device. The analysis algorithm is clearly separated from the main body of the source code and so may easily be replaced with your own data analysis algorithm.

The simplest streaming sample program, *GageStream2Analysis_Simple*, operates only a single CompuScope card. It is simple because all programming is done within a single software thread. The example analysis algorithm within *GageStream2Analysis_Simple* simply sums up the contents of the waveform data buffer for all channels and stores the resultant summed value to an ASCII data. While, by itself, it is not useful in any real-world application, this analysis algorithm was chosen for its simplicity and fast execution speed.

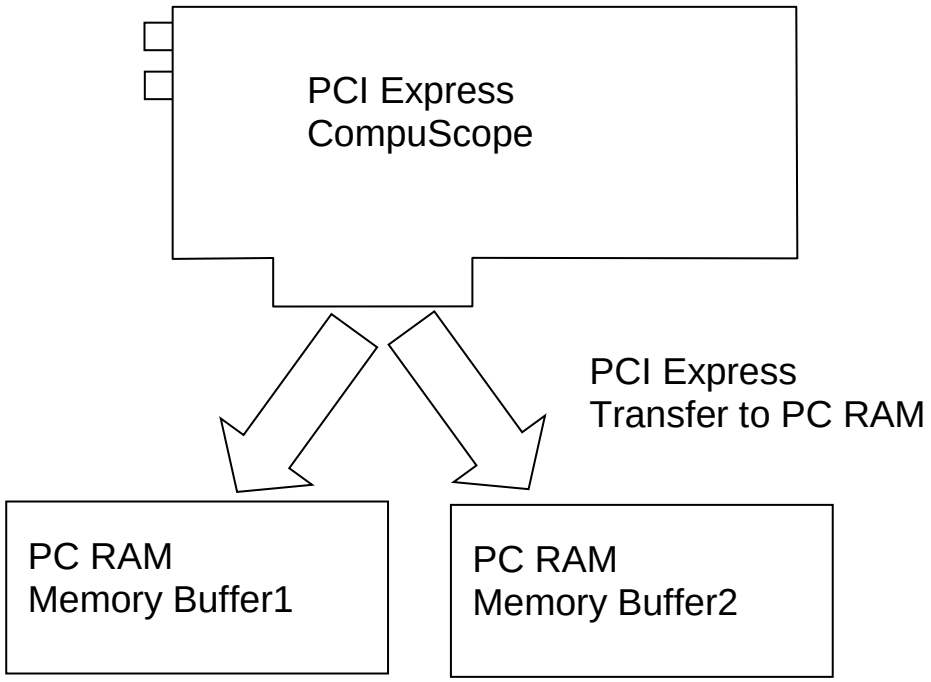
The more complex sample program, *GageStream2Analysis* operates a single CompuScope card or a Master/Slave CompuScope system. The data stream from each CompuScope card is

controlled from a separate software thread. Unlike *GageStream2Analysis_Simple*, which analyzes the raw waveform data, *GageStream2Analysis* analyzes the time-stamp values associated with each waveform. *GageStream2Analysis* is the best starting point for a streaming software application that operates multiple CompuScopes.

The third sample program, *GageStream2Disk* streams data from a single CompuScope card or from a Master/Slave CompuScope system to a target file on the hard drive. The data stream from each CompuScope card is controlled from a separate software thread. Data are stored in a header-less binary file, which consists of the waveform data in binary followed by the "tail" for each record, which contains the associated time-stamp value.

The general method of streaming waveform data from a PCI Express CompuScope to PC RAM is illustrated in the diagram below.

Using the *CsStmAllocateBuffer()* method, the user first allocates two PC RAM contiguous buffers of identical size, called *Buffer1* and *Buffer2*, that serve as targets for the data stream. If the operating system is unable to allocate the requested contiguous buffer, an allocation error will occur. The user toggles the two buffer targets as each one gets successively filled. While data are streaming to one buffer, the user analyzes (consumes) waveform data from the second buffer. As long as the user is able to sustainably analyze (or consume) all waveform data in the buffer under analysis before the streaming target buffer is filled up with new waveform data, then the streaming acquisition can proceed indefinitely with no data loss.



The size of the target memory buffer is internally manipulated so that it is simply related to the *SegmentSize*. Specifically, if the requested buffer size is less than the *SegmentSize*, then the buffer size is internally increased until the *SegmentSize* is an integral multiple of it. This way, the full segment may be transferred to the buffer in an integral number of data transfers and no partial transfers are required. On the other hand, if the requested buffer size is greater than the *SegmentSize*, then the buffer size is increased until it is an integral multiple of *SegmentSize*. This way, the buffer will fit an integral number of segments and no partial segments occur. These internal buffer size adjustments were made in order to keep the overall scheme simple.

Optimal selection of the buffer size depends upon the user application. Let us consider two examples of the acquisition of multiple short segments. In lightning monitoring, where lightning triggers occur randomly, the user would not want the buffer size to exceed *SegmentSize*. Otherwise, the consuming analysis process might have to wait hours or days before a lightning trigger occurred to fill up the buffer and allow consumption to begin. In lightning test, therefore, the user should make the buffer size equal to the *SegmentSize* so that lightning waveforms may be consumed as quickly as possible. Now consider the case of a scanning application where the triggers occur at a regular fast frequency. In this case, the user knows that multiple triggers will come and they can afford to make the buffers size much larger than the *SegmentSize*. If they had made the two sizes equal, then the software would waste a lot of time toggling between the two buffers, which would compromise overall throughput. If the user wanted to do only post-processing, then they should make the buffer size as large as possible in order to maximize throughput. If, on the other hand, they wanted to display some results during the scan, then they would have to choose an intermediate buffer size that allowed for decent throughput while still providing a reasonable result update rate.

Within the streaming sample programs after initial configuration, acquisition on CompuScope hardware is initiated by a call to *CsDo(hSystem, ACTION_START)*, as in normal Memory Mode. This call begins acquisition of waveform data into on-board CompuScope memory, which acts like a large FIFO. Next, the user initiates transfer of data from on-board CompuScope memory to one of the streaming target buffers using the *CsStmTransferToBuffer()* method. The waveform data within the other filled data buffer are then analyzed (using the *SumBufferData()* method, for example, for *GageStream2Analysis_Simple* ). After this analysis is complete, the user then awaits for the completion of data transfer to the first buffer using the *CsStmGetTransferStatus()*. The status returned by this method is checked to see if there was a FIFO overflow, which will terminate the sample program. If no overflow has occurred, then the sample program swaps the two buffers and continues the process.

Upon detection of any FIFO overflow errors (and the associated data loss), the Gage streaming sample programs will terminate after displaying the FIFO overflow error message. This need not be the case and the user can program his streaming application to continue acquiring after detection of a FIFO overflow error. We strongly recommend, however, that the user's software application keep track of any occurrences of the FIFO error and, therefore, any potential data loss. Please note that data loss may only occur after the two buffers have been filled at least once.

While the Gage sample programs use two toggling buffers, the user could modify the program to use more. Using a number of buffers that is equal to the number of cores on a multi-core CPU may allow a multi-threaded application to simultaneously analyze multiple buffers in a fashion that would improve the overall analysis throughput.

Used within the *GageStream2Analysis_Simple* sample program, the *SumBufferData()* method is an example analysis routine that sums all points within the data buffer. The resulting sum values are stored in an ASCII data file by *GageStream2Analysis_Simple*. The summing analysis is not useful by itself but was chosen for its simplicity and fast execution speed. If the user chooses *GageStream2Analysis_Simple* as a starting point for their own software application, the user should replace the contents of *SumBufferData()* with his own analysis subroutine. Again, this analysis routine could be data display, transfer, storage or any operation that consumes the data stream. Naturally, more complex analysis routines may require more processing time. Sufficiently complex data analysis will retard the analysis process to the point where the buffer under analysis cannot be emptied before the stream target buffer is filled, which will lead to a FIFO overflow and data loss. The complexity of the analysis that the streaming process can handle will depend upon system configuration (CPU core count and speed, PC RAM volume, etc.). For best performance, no other software applications should be executed while using eXpert data streaming.

The streaming sample programs operate in two distinct modes *Continuous Mode* and *Segmented Mode*. In Continuous Mode, the user does one long continuous acquisition that is initiated by a single trigger. For Continuous Mode operation, the user must specify a *SegmentCount* of 1 within the INI file. The *Depth* and *SegmentSize* values should correspond to the number of samples that the user wants to acquire for the one large waveform, as in the *GageAcquire* project.

In Segmented Mode, which is the streaming version of Multiple Record Mode, the user specifies a *SegmentCount* greater than 1 and must ensure that a corresponding number triggers occur. Otherwise, the program will appear to freeze as it awaits trigger events that never occur (assuming that the Trigger TimeOut is infinite). Each waveform is transferred to PC RAM with a *Tail* that contains Trigger TimeStamp information.

As with other C/C# SDK sample programs, configuration parameters for the streaming sample programs are entered in an INI text file. Both *GageStream2Analysis_Simple* and *GageStream2Analysis* obtain their configurations from the same INI text file, while *GageStream2Disk* has its own INI file. Most parameters have the same meaning as in other CompuScope C/C# SDK sample projects.

There are maximum size limits on the *SegmentSize* and *SegmentCount*. For *SegmentSize*, the maximum is:

Max *SegmentSize* = $\frac{32\ GigaSamples - 16\ Samples}{Mode\ Number}$ for 12-, 14- and 16-bit CompuScopes

Max *SegmentSize* = $\frac{64\ GigaSamples - 32\ Samples}{Mode\ Number}$ for 8-bit CompuScopes

Where the *ModeNumber* is 1 for Single Channel Mode, 2 for Dual Channel Mode, 4 for Quad Channel Mode, 8 for Octal Channel Mode. If the user sets the *SegmentSize* to -1, the acquisition will be endless and the user must terminate it by aborting the acquisition.

The maximum allowed *SegmentCount* value is 4 Billion (actually 4,294,967,295) records. It is possible to use higher *SegmentCount* values because the API allows for construction of a 64-bit *SegmentCount* variable. The user would have to write their own software to exploit this, however.

By necessity, waveform data transferred during a streaming acquisition must include data from all active channels. The number of active channels is determined by the CompuScope Mode. There is currently no provision for streaming only data from a subset of the active channels.

The list below indicates how the data from different channels are packed for the different CompuScope modes. For example, "*1212121212*" indicates that the data will be transferred as *Channel #1/Sample #1, Channel #2/Sample #1, Channel #1/Sample #2, Channel #2/Sample #2, Channel #1/Sample #3, Channel #2/Sample #3…….*

| CompuScope Mode | Channel Data Format |
|---|---|
| SINGLE | 11111111 |
| DUAL | 12121212 |
| QUAD | 12341234 |
| OCTAL | 12345678 |

CompuScope API Methods used for Streaming

**int32 CsStmAllocateBuffer( IN CSHANDLE CsHandle,**
**IN uInt16 nCardIndex,**
**IN uInt32 u32BufferSize,**
**OUT PVOID *pVa )**

This function allocates a memory buffer that can be used as a target for data streaming. The buffer allocated is physically contiguous, as required for a streaming target. This buffer must be freed via the CsStmFreeBuffer() function when it is no longer needed.

Parameters:

CsHandle : Handle of a CompuScope system

nCardIndex : Index of cards in a Master/Slave CompuScope system
1 = Master, 2 = Slave1, 3 = Slave2 ...
This parameter is ignored for a single card CompuScope system

u32BufferSize : The buffer size to allocate in bytes

pVa : Pointer to the allocated buffer if the function returns CS_SUCCESS

**int32 CsStmFreeBuffer( IN CSHANDLE CsHandle,**
**IN uInt16 nCardIndex,**
**IN PVOID pVa )**

This function frees a buffer previously allocated by CsStmAllocateBuffer() function.

Parameters:

CsHandle : Handle of a CompuScope system

nCardIndex : Index of cards in a Master/Slave CompuScope system.
1 = Master, 2 = Slave1, 3 = Slave2 ...
This parameter will be ignored for a single card CompuScope system

pVa : Pointer to the buffer to be freed.

**int32 CsStmTransferToBuffer( IN CSHANDLE CsHandle,**
**IN uInt16 nCardIndex,**
**IN PVOID pBuffer,**
**IN uInt32 u32TransferSize )**

This function starts DMA transfer of waveform data from on-board memory to the buffer allocated by the CsStmAllocateBuffer() function.

Parameters:

CsHandle : Handle of a CompuScope system

nCardIndex : Index of cards in a Master/Slave CompuScope system.
1 = Master, 2 = Slave1, 3 = Slave2 ...
This parameter will be ignored in a single card CompuScope system

pBuffer : Pointer to the buffer allocated by CsStmAllocateBuffer()

u32TransferSize : The size of DMA transfer in samples.

**int32 CsStmGetTransferStatus( IN CSHANDLE CsHandle,**
**IN uInt16 nCardIndex,**
**IN uInt32 u32WaitTimeoutMs,**
**OUT uInt32 *pu32ErrorFlag,**
**OUT uInt32 *pu32ActualSize,**
**OUT uInt8 *pu8EndOfData )**

This function returns the status of the current DMA transfer started by the CsStmTransferToBuffer() function.

Parameters:

CsHandle : Handle of a CompuScope system
nCardIndex : Index of cards in a Master/Slave CompuScope system.
1 = Master, 2 = Slave1, 3 = Slave2 ...
This parameter will be ignored in a single card CompuScope system

u32WaitTimeoutMs : If u32WaitTimeoutMs = 0, this function returns immediately with the status of the current DMA transfer.
If u32WaitTimeoutMs != 0, this function will wait until the current DMA transfer is completed or the WaitTimeoutMs value in milliseconds has expired, whichever comes first.

| | |
|---|---|
| pu32ErrorFlag : | Returns errors may occur during streaming. |
| pu32ActualSize : | Returns the number of valid samples in the buffer once the DMA has completed. Usually this should be equal to the requested size of transfer. |
| pu8EndOfData : | If this return value is 1, all data from the current acquisition have been transferred to application. No more data are available to transfer. In this case pu32ActualSize may differ from the requested size of the transfer. In infinite streaming mode (infinite segment size or infinite segment count) this value is always 0 and pu32ActualSize = the requested size of transfer. |

## *GageMulRecAveraging*

Usage of the signal averaging optional firmware image allows repetitive waveform acquisitions to be rapidly averaged, in order to reduce random noise. In the past, signal averaging required waveforms to be downloaded for averaging within the host PC's CPU so that averaging was limited by the data transfer speed. With the MulRecAveraging firmware, repetitive waveforms are averaged within the firmware with no data transfer required until up to 1024 averages have been performed. Consequently, much higher repetition rates may be achieved.

Once the signal averaging firmware has been loaded, the user adjusts the Number of Averages to be acquired for each waveform to be averaged. In addition, the user may set the SegmentCount to be greater than 1 so that averaged waveforms are stacked in on-board acquisition memory, as in Multiple Record mode. In fact, the firmware does not actually average waveforms but *co-adds* them, which means that the waveforms are summed together but he resultant summed waveform is not divided by the number of averaged waveforms in order to calculated the true signal average. The firmware performs co-adding rather than true averaging because division is a complex operation within an FPGA, is often unnecessary and may easily be done is software, if necessary. With the MulRecAveraging firmware loaded, the CompuScope hardware co-adds a preset number of consecutive waveforms, and then stacks the resultant co-added waveforms in on-board memory, as is usually done in Multiple Record Mode.

As an example, consider the acquisition of 10,000 sample waveforms using MulRecAveraging in which the Number of Averages was set to 1000 and the SegmentCount was set to 2000. In this example, a total of 2000 X 1000 = 2 million waveforms would be acquired by the CompuScope hardware. However, each consecutive 1000 waveforms would be co-added together and the resultant 2000 co-added waveforms would be stacked in on-board memory. Since the hardware re-arm between successive groups of 1000 averaged acquisition is very fast, the user could use this functionality to follow noisy signals that rapidly evolve in time. Alternatively, the user could co-add the 2000 waveforms in software in order to create a "Super-averaged" waveform that is the result of 2 million averages.

Co-added waveform data are stored within a 32-bit format buffer within the on-board memory. The resulting averaged waveform is therefore transferred as a 32-bit data buffer. With the MulRecAveraging firmware loaded, the Sample Size, Sample Resolution and Sample Offset values are changed to reflect the 32-bit data format. Querying these parameters after the firmware is loaded will return the updated values. The updated Sample Resolution includes the number of Number of Averages factor by which the co-added waveforms must be divided in order to obtain the averaged waveform. Consequently, the updated Sample Resolution and Sample Offset values may be used with the Voltage Conversion equation in order to directly convert the co-added waveforms into for the correct voltage waveforms..

Each SDK contains an advanced sample program that uploads the MulRecAveraging image, performs multiple averaging acquisitions and then displays or stores the resulting averaged waveforms.

## *GageHistogram*

Gage Histogram firmware is managed and configured like other eXpert firmware – through CompuScopes Manager. The Gage Histogram firmware may be operated from CsTest or from a CompuScope SDK sample program called GageHistogram, which is available for C and LABVIEW.

When activated, the Histogram firmware computes a histogram of the incoming raw waveform data. As an example, consider an 8-bit Gage CompuScope whose waveform data can take on values between 0 and 255. When activated, the Histogram firmware establishes 256 counters or bins, each corresponding to data values 0 through 255. In Histogram Mode, when a waveform sample with a data value of N is acquired, the counter that corresponds to this value N is incremented by 1.

The counter values are stored in 32-bit variables so that the maximum count that can be accommodated within each bin in Single Channel Mode is 2^32 counts, or slightly more than 4 Billion counts. For architectural reasons, in Dual Channel Mode, the maximum count is 2^31 counts. Internally, the eXpert Histogram firmware actually computes multiple histograms, which are then combined to determine the final histogram. Accordingly, overflow may occur on any of the multiple histograms. As a result, the user may find that a histogram overflow may occur slightly below the maximum count values listed above.

If any of the histogram counter bins overflows, the data transfer subroutine CsTransfer() will return the error code CS_HISTOGRAM_FULL. In addition, when an overflow occurs all histogram count values will be returned with half their actual values so that the overflow is artificially suppressed.

In Histogram Mode, the user may select to have all of the histogram counts reset to zero before each acquisition. This is done by the call to CsDo(ACTION_RESET_HISTOGRAM). Otherwise, counts from sequential waveform acquisitions will be added to existing counts. The histogram will also be reset at the call CsDo(ACTION_COMMIT).

## GageCsPrf

GageCsPrf is an advanced sample program that may be used to evaluate the repetitive capture performance of CompuScope hardware. The program makes a series of repetitive acquisitions using the specified settings and provides subsequent timing measurements. All timing measurements are done using the QueryPerformanceCounter Windows API timing function. The total time required to complete a single acquisition is provided along with its inverse, the Pulse Repeat Frequency (PRF). In addition, the timings of the separate operations that occur within one acquisition are provided. Repetitive acquisition sequences are repeated using different acquisition depths and timing results are provided for acquisition depths.

GageCsPrf is based upon the GageAcquire sample program and uses similar controls. Only controls that are different from those of GageAcquire are described here. See the GageAcquire documentation for the common controls.

Since GageCsPrf does not store acquired data, the *SaveFileName* key within the INI file is ignored. The timing results file name is specified by the *ResultsFile* key in the *PrfConfig* branch in the INI file. Also, the *TransferLength* key in the *Application* branch, along with the *SegmentSize* and *Depth* keys in the *Acquisition* branch, are ignored since the *Depth* is internally adjusted by *GageCsPrf*. Internally, the *Depth* is always selected as a power of two. The range of internally selected *Depth* values is bound by the smallest power of two that is greater than or equal to the value of the *StartDepth* key in the *PrfConfig* branch and the highest power of two that does not exceed the value of the *FinishDepth* key in the *PrfConfig* branch.

Results are stored in a tab delimitated text file. The first 5 lines describe the measurement configuration, such as the CompuScope model number and memory size, acquisition configuration and number of acquisitions in one repetitive capture sequence (loop count). These lines are followed by 7 columns of data. The first column (*depth*) specifies the size of the acquisition. The remaining columns are the result of the timing measurements. *Total time* is the time required by the complete acquisition sequence and *PRF* is its inverse, the repeat frequency of acquisitions. *Start time* is the time required for execution of the CsDo (ACTION_START) method. In this time the CompuScope system will perform all necessary operations to start an acquisition. *Busy time* is the time taken by the data acquisition itself. Please note that this time includes waiting for the trigger event. Consequently, if the trigger event is infrequent, the measured *Busy time* will be long. For measurement of the maximum PRF, trigger from a fast signal source, such as a sine wave with a frequency of 1 MHz or more. *Transfer time* and *transfer rate* describe data transfer from one channel. Please note that the data *transfer time* includes two components: a fixed transfer set-up overhead time and the actual data transfer duration, which is proportional to the data volume. As data volume increases, the importance of the overhead time diminishes. Consequently, the calculated aggregated *transfer rate* improves with the *Depth*.

Within GageCsPrf, the power-saving mode is enabled. This is fine for all CompuScope models except the CS82G and CS8500. For these models, power-saving mode will severely reduce repetitive capture performance (PRF). In order to get the best PRF from these models, you must disable power-saving mode. This is done by adding 128 to the Mode value within the PRF.ini file. For single-channel mode, use "Mode=129" and for dual-channel mode, use "Mode=130".

By following the coding illustrated in GageCsPrf, a user can achieve the fastest possible repetitive capture performance from CompuScope hardware.

## GageFastAcquire2Disk

Several Gage customers require their CompuScopes to acquire fast repetitive waveforms and store them as quickly as possible to hard drive. CompuScope hardware allows for fast PCI data transfer, extremely fast hard drive arrays are commercially available and MS Windows provides powerful multi-threading capability. Optimally combining these elements, however, to fully exploit their performance is not straightforward. In order to allow Gage customers to realize the best possible performance, Gage has created *GageFastAcquire2Disk.* This sample program uses a flexible, CompuScope driver-level engine and that allows the fastest possible repetitive CompuScope capture to hard drive.

A GageFastAcquire2Disk sample is now available as an Advanced Sample Program within CompuScope SDKs for C/C#, LabVIEW and MATLAB. The program will support waveform acquisition and storage sequence from any single CompuScope system (single card or Master/Slave system) constructed using any CompuScope model. User may even launch concurrent instances of Gage FastAcquire2Disk in order to simultaneously acquire data from independent CompuScope systems. The sample programs are all relatively thin shells that call the GageFastAcquire2Disk engine, which operates from within the CompuScope drivers. Performance is therefore virtually identical from all three SDKs. The engine incorporates the following features to maximize overall throughput:

- Multi-threading for concurrent, parallel operations where possible – for example, storing previously acquired waveforms to hard drive while new waveforms are being acquired
- Pre-opening of target waveform files to avoid costly file open operations during acquisition sequence
- Block transfer size from CompuScope hardware optimized for most efficient usage of PC RAM under Windows
- Optimized storage file transfer size for fast storage rate and easy management under Windows
- Hardware interrupt usage to avoid costly repetitive polling operations

The GageFastAcquire2Disk engine uses a lot of system resources. While CPU processing power per se is not important, the host system should be equipped a fast PCI bus, a fast memory bus, a fast hard drive and as much PC RAM as possible. Specifically, the PC systems should be equipped at least an amount of PC RAM that is equal to the number of active CompuScope channels multiplied by the 128 MB. Further, before running any GageFastAcquire2Disk sample program, the user should close all unnecessary Windows applications, since these will divert resources from GageFastAcquire2Disk.

From the GageFastAcquire2Disk sample program all SDKs, the Gage FastAcquire2Disk operation method is the same. Before activating the GageFastAcquire2Disk engine, the sample program must first initialize the drivers and commit CompuScope configuration settings, as usual. Next, the program activates the GageFastAcquire2Disk engine and initializes it with acquisition settings (e.g. number of waveforms to be acquired, data of interest that is to be transferred). The initialization also validates input parameter and allocates buffers required for GageFastAcquire2Disk acquisitions. Finally, initialization creates all Windows folders and pre-opens all SIG files that will be required for the acquisitions. Next, the program starts the GageFastAcquire2Disk acquisition sequence, so that the CompuScope hardware begins acquiring waveforms and storing them to files. Normally, the process continues until the requested number of waveforms has been acquired and stored to hard drive. From C, all control of the GageFastAcquire2Disk engine is invoked using the CsExpertCall() API call with different u32ActionId constants. The user may issue the following commands and queries:

1. COMMAND INITIALIZE (implemented by the CsExpertCall() with u32ActionId = EXFN_DISK_STREAM_INITIALIZE): Activate and initialize the GageFastAcquire2Disk engine. Allocate required resources and validates input GageFastAcquire2Disk parameters. Create target folders and pre-open target SIG files.

2. COMMAND START (implemented by the CsExpertCall() with u32ActionId = EXFN_DISK_STREAM_START): Start the GageFastAcquire2Disk acquisition sequence, which will normally terminate only once all waveforms are acquired and stored to hard drive files.

3. COMMAND STOP (implemented by the CsExpertCall() with u32ActionId = EXFN_DISK_STREAM_STOP): Stop the GageFastAcquire2Disk acquisition sequence after completing any acquisition that is in progress and storing all resultant waveform files to hard drive.

4. COMMAND CLOSE (implemented by the CsExpertCall() with u32ActionId = EXFN_DISK_STREAM_CLOSE): De-activate the GageFastAcquire2Disk engine. Free all allocated system resources, including active CompuScope handles.

5. QUERY STATUS (implemented by the CsExpertCall() with u32ActionId = EXFN_DISK_STREAM_STATUS): Returns status that indicates if GageFastAcquire2Disk acquisition sequence is finished.

6. QUERY # ACQUISITIONS (implemented by the CsExpertCall() with u32ActionId = EXFN_DISK_STREAM_ACQ_COUNT): Return the number of acquisitions that

have been completed since the GageFastAcquire2Disk acquisition sequence has started.

7. QUERY # FILES (implemented by the CsExpertCall() with u32ActionId = EXFN_DISK_STREAM_WRITE_COUNT): Return the number of files that have been stored since the GageFastAcquire2Disk acquisition sequence has started.

As discussed, adjustment of configuration settings (e.g. sampling rate, input range) for GageFastAcquire2Disk is done in a similar fashion as for the standard SDK sample programs. However, there are some additional setup variables that have been added for GageFastAcquire2Disk. These variables are described below only for FastAcquire2Disk.INI configuration file for the C Sample FastAcquire2Disk program but corresponding variables exist for the other SDKs.

Within FastAcquire2Disk.INI, the *AcqCount* variable determines the total number of acquisitions that will be performed by GageFastAcquire2Disk. Only completion of this number of acquisition will terminate the GageFastAcquire2Disk engine, unless it encounters an error or is intentionally stopped or aborted. The *Channels* variable selects the channels whose data will be stored to hard drive files. For instance, if a user is only interested in the even channel on an 8-channel CompuScope card, he can set *Channels=2,4,6,8*.

As in the standard SDK samples, GageFastAcquire2Disk allows adjustment of the *StartPosition* and *TransferLength*, which determine how much waveform data are to be downloaded from which point in the waveforms. In addition, GageFastAcquire2Disk allows the user to select how many records to download and from which starting record for Multiple Record acquisitions. For instance, if the user does Multiple Record acquisitions acquiring 120 Records, he can choose to download and store only 10 records starting from record #40 by setting *RecordStart=40* and *RecordCount=10*.

Finally, there are three separate Trigger TimeOut Settings in GageFastAcquire2Disk, all of which are expressed in milliseconds and all of which are disabled by a value of -1. The *TriggerTimeOut* in the *Acquisition* key within the INI file, which is standard for all C SDK programs, is a hardware Trigger TimeOut. There is a high–speed counter on the CompuScope hardware that forces a trigger event if the counter value corresponding to the Trigger TimeOut is exceeded. The advantage of this hardware Trigger TimeOut is that has sub-microsecond precision and is also very accurate and deterministic, as compared to a software Trigger TimeOut. The disadvantage, however, is that the hardware counter will eventually roll over, since it has finite width.

The *TriggerTimeOut* in the *AppConfig* key is a software Trigger TimeOut. Its use is necessary when the required Trigger TimeOut exceeds the rollover limit of the hardware Trigger TimeOut (typically over an hour to several hours, depending upon the CompuScope model and sampling rate). When using this software Trigger TimeOut, the user should disable the hardware Trigger TimeOut by setting it to -1.

The final timeout, *StatusTimeout*, is an intentional software latency introduced so that the QUERY operation listed above (QUERY STATUS, QUERY # ACQUISITIONS and QUERY # FILES) are not called in a loop at a rate sufficient to compromise the speed of the

GageFastAcquire2Disk data transfer and storage operations. The default setting of 500 milliseconds ensures that the driver will not poll the status of GageFastAcquire2Disk more than twice a second, which will not significantly affect operation and is usually a sufficient status update rate for the user.

Gage FastAcquire2Disk stores waveforms acquired by CompuScope hardware in GageScope SIG files, which are binary data files with a 512 byte information header. For easy management under Windows, Gage has limited the maximum SIG files size created by GageFastAcquire2Disk to a maximum of 256 Megabytes. (This number may be increased in future, but the principles discussed below will still apply). In order to optimize system memory usage, GageFastAcquire2Disk internally limits the size of a single block data transfer from the CompuScope hardware to a maximum of 128 Megabytes per active channel. Consequently, the user should have a bare minimum of 128 Megabytes of PC RAM per active channel. Of course, this bare minimum does not leave any room for the O/S or for applications, so that significantly more than this minimum is highly recommended for optional operation.

GageFastAcquire2Disk stores all acquired waveform data within GageScope SIG files that reside within a folder structure designed for easy management. The folder structure is created upon initialization of the GageFastAcquire2Disk engine. First, all files are placed within a Master folder local to the GageFastAcquire2Disk application, whose name is specified by the *FolderName* variable (Default= "Signal Files"). Next, folders within the Master folder are created for data from different active CompuScope channels. These folders are given names in the from YYYY-MM-DD_HH-MM-SS_CHANXX, where YYYY-MM-DD and HH-MM-SS are the system date and time when the folders are created upon initialization and XX is the CompuScope channel number. Within these folders, a series of sequential folders called *Folder.001, Folder.002, Folder.003, etc.* are created. The contents of each of these folders is limited to a maximum of 16,000 SIG files and initialization creates a sufficient number of folders to accommodate the entire GageFastAcquire2Disk acquisition sequence. Within these folders, SIG files called *File-00000.SIG, File-00001.SIG, etc*. are created and opened so that they are ready to accept data from the subsequent GageFastAcquire2Disk acquisition sequence. We describe below the three most common GageFastAcquire2Disk use cases and describe how the resultant waveform data are stored in each case.

The most common use case is acquisition of many relatively short waveforms (a few hundred Samples ~ 100 MegaSamples) by CompuScope hardware in Single Record Mode. In this case, the GageFastAcquire2Disk engine will generally pack many single waveforms into a single Gage SIG file. All but the last SIG file will contain the same number of records. The last SIG file will most likely contain a smaller number of left-over records. As an example, consider the user who configures GageFastAcquire2Disk to acquire twenty thousand waveforms of 65536 Samples each in Single Record Mode from a 14-bit CompuScope, where each Sample will occupy 2 data bytes. In this case, GageFastAcquire2Disk will fill each 256 MB SIG File with 2048 records, since ($256 \times 1024 \times 1024$ MB)/(65536 Samples/waveform)/(2 Bytes/Sample) = 2048 waveforms. GageFastAcquire2Disk would create nine such SIG files and the final tenth SIG file would contain only the remaining $20{,}000 - 9 \times 2048 = 1568$ records.

The second case is acquisition of relatively long waveforms (>256 Megabytes) by CompuScope hardware in Single Record Mode. Since the waveform size exceeds the maximum file size, the GageFastAcquire2Disk engine will divide the waveform into multiple contiguous SIG files. For example, a user might want to perform sixteen acquisitions of 1,100,000,000 Samples each using a 12-bit CompuScope. Because each 12-bit Sample occupies 2 data bytes, GageFastAcquire2Disk will only allow storage of 128 MegaSamples per SIG file. In this case, the 1. 1 Billion Samples of data from a single acquisition will be divided across nine SIG files. The first eight files will contain 128 MegaSamples, and the ninth file will contain the remaining $1,100,000,000 - 8 \times 128 \times 1024 \times 1024 = 26,258,176$ Samples.

These nine SIG files will contain completely contiguous data so that, for example, the first Sample in the third SIG file was acquired directly after the last Sample in the second SIG file. The relationship amongst the nine contiguous SIG files is indicated by the Start Address contained within the SIG files. As is often the case, let us assume that the data requested for storage by the user began at the Trigger Address, which is always presented by the software as Trigger Address 0. In this case, the first SIG file will contain a Start Address of 0, which indicates that the first Sample within the file is the Trigger Point. The second SIG file, however, will contain a Start Address of $-128 \times 1024 \times 1024 = -134,217,728$, which correctly indicates that the Start Address occurred 134,217,728 before the first Sample in the second file. The Start Address will decrease by 134,217,728 Samples for each successive SIG file in the nine file set.

Since the user requested sixteen acquisitions of 1. 1 Billion Samples each, each of the 12 acquisitions will create its own set of nine contiguous files. While each set of nine files represents a continuous acquisition within no breaks between files, between each file set there will be a time-gap as acquisition ceases while data are downloaded from the CompuScope hardware and stored to files.

In Multiple Record Mode, GageFastAcquire2Disk stores files in a similar fashion as in Single Record Mode. GageFastAcquire2Disk always tries to store all records from a single Multiple Record acquisition within a single SIG file. However, GageFastAcquire2Disk never mixes records from different Multiple Record acquisitions within the same SIG file. Division of data from a single Multiple Record is avoided, if possible. If the total Multiple Record data volume exceeds 256 MB, GageFastAcquire2Disk will distribute records across multiple SIG files, as in the first case above, while not splitting records if possible. The rare extreme Multiple Record case occurs when the user acquires very long Multiple Records. In this case, GageFastAcquire2Disk is forced to split the records across multiple files. For example, if a user chooses to acquire ten Multiple Records of 600,000,000 Samples each from an 8-bit CompuScope, GageFastAcquire2Disk will create ten contiguous SIG files sets – one set for each record. Each set will consist of three SIG Files, the first two of size 256 MB and the last one containing the roughly 100 MB of remaining left-over data.

The GageScope SIG files created by Gage FastAcquire2Disk contain an informative 512 Byte header, followed by the CompuScope waveform data in binary. Gage provides the C CsConvertFromSigHeader() method, which decodes the header (from a buffer into which a SIG file header has transferred) and fills a structure with relevant header variables. These variables may then be used to select and interpret subsequent SIG file waveform data. Users

can also use the CsConvertToSigHeader() method to create their own SIG file headers. Corresponding methods exist in the other SDKs.

Key elements that retard GageFastAcquire2Disk's maximum achievable repetitive acquisition rate are the time required to download data from the CompuScope hardware to PC RAM and the time to transfer data from PC RAM to the target hard drive. In almost all cases, it is the latter element that is the primary bottleneck to data throughput. Maximizing the storage rate to drive requires that certain optimization steps.

First of all, users should be wary of high storage rate claims by hard drive manufacturers. Usually, these claims refer to the data storage rate that may be maintained by the drive during burst storage. Most drive devices are equipped with a limited amount of cache memory. Usually, manufacturers' throughput claims refer only to burst storage volumes that do not exceed this cache size, which is typically a few MB While very high storage rates may be realized into this disk cache memory, as soon as the cache is full, the rate drops dramatically, since the cache contents must be more slowly transferred to the actual drive media. What the Gage user really requires is this slower *sustained* storage rate, which is usually not provided by the hard drive manufacturer.

For best data throughput, the users should dedicate a specific drive as a storage target and should not store other files on this drive. Most importantly, the user should make sure not to store GageFastAcquire2Disk data on the same drive upon which the Windows Operating System or Page File is installed. The constant read/write housekeeping operations that Windows performs in normal course will seriously compromise the storage rate of Gage FastAcquire2Disk.

For best Disk performance, the user should obtain hard disk device drivers from the hard drive manufacturer, rather than using the generic drivers included in Windows, which typically will not provide the fastest throughput for your manufacturer's drive. Finally, disk throughput may be improved dramatically by configuring multiple hard drives in a RAID disk array. A RAID disk array may realize sustained data throughput rates of 100 MB/s and more.

## GageASTransfer, GageEvents, GageCallback

These sample programs illustrate advanced synchronization techniques for multi-threaded applications. These techniques are essential to creating a complex application for real time data analysis or for operating multiple inter-related instruments. This is because these techniques allow a multi-threaded application to perform other tasks while CompuScope hardware is busy acquiring or transferring data without the usual need to poll its status. Without the need for polling, data acquisition and transfer do not tax the CPU, leaving it free to perform other operations. Nevertheless, these techniques add significant complexity to the overall application design, making it prone to errors such as thread deadlock. Consequently, usage of these techniques should not be considered unless they are truly required.

As the name suggests, GageASTransfer illustrates asynchronous data transfer. When called, the standard CsTransfer method does not exit and return until the requested data transfer operation is complete and all data have been transferred into the target buffer. By contrast, when CsTransferAS is called, it returns immediately after initiating the data transfer, which is then left to finish in the background. While data are being transferred, the controlling application may do something else, even though the data transfer is not yet complete. Completion of the data transfer is signalled by the "end of the transfer" event. Progress of the data transfer may be checked by CsTransferASResult method. GageASTransfer is a non-multi-threaded C application that polls the "end-of-transfer" event while checking the transfer status every 100 milliseconds.

GageEvents is a multi-threaded sample program that illustrates usage of the notification events that can be assigned to specific operation of the CompuScope, allowing synchronization between different threads of execution. GageEvents uses the "end-of-busy" and "end-of-transfer" event notifications to trigger appropriate operations. In parallel, GageEvents processes older waveform data to determine the minimum and maximum points within the waveform. The handling of events that is illustrated within GageEvents is the recommended method of synchronization in a multi-threaded C application.

Some environments, such as Visual Basic and LabWindows/CVI, do not allow the programmer to create multi-threaded programs directly. These environments, however, do provide a functionality called "Callbacks", which allows the programmer to associate a callback function with notification of an event. The thread associated with the callback is then launched within the CompuScope driver. Use of callbacks has limitations. For instance, only one callback function may be executed at a time. Synchronization using callbacks is illustrated by the GageCallback sample program.

## GageAdvMulRecEx

Historically, Gage CompuScopes have allowed for the collection of a large and non-predetermined amount of pre-trigger data in Multiple Record Mode. In order to achieve this capability, CompuScopes have configured their acquisition memory as multiple circular buffers of size *SegmentSize* in Multiple Record mode. The disadvantage of configuring each Segment memory as a circular buffer is that this complexity practically required Multiple Records to be downloaded to PC RAM one-at-a time with a separate transfer for each Multiple Record, rather than downloading all Multiple Records in one data transfer. This one-at-a-time record download does not cause a significant problem for long Records (more than 10,000 Samples or so). However, for shorter record lengths, the overhead of setting up multiple PCI transfers (one per record) can seriously reduce the aggregate data download rate.

In order to significantly improve the download rate for Multiple Record acquisitions with short records, Gage has implemented a separate Multiple Record operation mode on some CompuScope models called *Rectangular Multiple Record Mode.* Instead of being configured as multiple circular buffers, the memory is configured in a rectangular fashion. Memory contents consist of a rectangular array in which each row contains a single waveform record and each column corresponds to a specific Sample within each waveform record. (In fact, the architecture is a little more complex since Trigger Timestamp Data are stored a the end of each waveform.) With this architecture, all records from a Multiple Record Mode may be

downloaded in one transfer and the resultant data buffer may be easily parsed to extract single records or may be stored directly for future analysis. A diagram of the on-board memory architecture in rectangular mode is shown below for the acquisition of M records of N Samples each.

| | | | | | | |
|---|---|---|---|---|---|---|
| Record #1<br>Sample #1 | Record #1<br>Sample #2 | Record #1<br>Sample #3 | .. | Record #1<br>Sample #N-1 | Record #1<br>Sample #N | TimeStamp #1 |
| Record #2<br>Sample #1 | Record #2<br>Sample #2 | Record #2<br>Sample #3 | .. | Record #2<br>Sample #N-1 | Record #2<br>Sample #N | TimeStamp #2 |
| Record #3<br>Sample #1 | Record #3<br>Sample #2 | Record #3<br>Sample #3 | .. | Record #3<br>Sample #N-1 | Record #3<br>Sample #N | TimeStamp #3 |
| … | … | … | .. | … | … | … |
| Record # M-1<br>Sample #1 | Record # M-1<br>Sample #2 | Record # M-1<br>Sample #3 | .. | Record # M-1<br>Sample #N-1 | Record # M-1<br>Sample #N | TimeStamp # M-1 |
| Record #N<br>Sample #1 | Record #M<br>Sample #2 | Record #N<br>Sample #3 | .. | Record #M<br>Sample #N-1 | Record #M<br>Sample #N | TimeStamp #M |

The maximum number of records that may be acquired in Multiple Record Mode is:

Max Number of Records ≈ Max Acquisition Memory per channel / (#Samples per Record)

The equation is not exactly correct because space is required for some inter-record information, such as Timestamp values. The amount of inter-record information is dependent upon the CompuScope model.

Acquisition using Rectangular Multiple Record Mode is illustrated within the AdvMulRecEx sample program within each CompuScope SDK. The configuration of Rectangular Multiple Record Mode is the same as for Circular Multiple Record Mode with some difference in the interpretation of the input parameters. As usual, the Post Trigger *Depth* specifies how many Samples will be acquired after each trigger. The *Segment Size*, however, determines the amount of waveform data that will always be acquired for each Record. Consequently, a pre-trigger data volume of (*SegmentSize – Depth*) will be acquired for each record – even if the CompuScope must ignore triggers in order to acquire this pre-trigger data volume. Another important element is that there is a maximum allowed amount of pre-trigger data, which is equal to 128 kiloSamples/(Number of Active Channels).

Once a Rectangular Multiple Record Mode acquisition is completed, resulting data are downloaded using the CsTransferEx() API method (or the corresponding method in other SDKs). The most important difference between CsTransferEx() and the older CsTransfer() method is that CsTransferEx() is able to download many Multiple Records in a single data transfer. A second important different is that, unlike CsTransfer(), CsTransferEx() is able to download only select portions of waveform data within each segment, which may reduce the volume of data to be transferred.

The input parameters to the CsTransferEx function, whose actual variable names are slightly different for each SDK, are as follows:

*Channel number* – The channel from which record data are to be downloaded  A Flag exists that allows download from all active channels.
*Start Record* – The first record to be downloaded
*Number of Records* – The number of records to be downloaded, starting from *Start Record*
*Start Sample* – The first Sample within each record to be downloaded
*Number of Samples* – The number of Samples to be downloaded, starting from *Start Sample*, from within each record

While CsTransferEx() allows download of huge volumes of data in a single transfer, the user should be careful not download too much data at one time.  Doing this may cause the operating system to start using part of the hard drive as PC RAM, which may seriously retard data throughput.  Generally, the user should avoid downloading more than 100 Megabytes of data in a single data transfer.  Smaller data volumes may be downloaded in sequential transfers, between which each volume is dispatched to hard drive storage or data analysis algorithms.

# Special CompuScope API function for usage with eXpert firmware

## CsExpertCall

The **CsExpertCall** function is required for control of certain eXpert firmware features in CompuScope cards.

int32 CsExpertCall( CSHANDLE hCsHandle, VOID *pFunctionParams )

**Parameters**

| | |
|---|---|
| *hCsHandle* | hHandle of the CompuScope system |
| *pFunctionParams* | The pointer to Function Params structure |

**Return values**

CS_SUCCESS indicates success.

**Remarks**

The Function Params structure will be different depending on the action to be performed, but they all have the same form:

```
typedef struct
{
	struct
	{
		uInt32		u32Size;
		uInt32		u32ActionId;
		...
	} in;

	struct
	{
		.....
	} out;
}
```

# FUNCTION PARAM STRUCTURES AND FUNCTION ID TO BE USED WITH CsExpertCall()

## EXFN_CREATEMINMAXQUEUE

Call CsExpertCall() with the function Id EXFN_CREATEMINMAXQUEUE to create a MinMax Detection Queue within the Windows Kernel for usage by the driver.

```
typedef struct _CSCREATEMINMAXQUEUE
{
      struct
      {
            uInt32        u32Size;
            uInt32        u32ActionId;
            uInt32        u32QueueSize;
            uInt16        u16DetectorResetMode;

            uInt16        u16TsResetMode;
      } in;

      struct
      {
            HANDLE        *hQueueEvent;
            HANDLE        *hErrorEvent;
            HANDLE        *hSwFifoFullEvent;
      } out;

} CSCREATEMINMAXQUEUE, *PCSCREATEMINMAXQUEUE;
```

**Parameters**

| | |
|---|---|
| *u32Size* | Size of this structure |
| *u32ActionId* | The function Id. Must be EXFN_CREATEMINMAXQUEUE |
| *u32QueueSize* | Number of SegmentInfo in the Driver MinMax Detection Queue. (e.g. a value of 50 indicates that the driver MinMax Detection Queue can hold up to 50 MinMax Segment Info structure entries before the *hSwFifoFullEvent* event gets signalled). |
| *u16DetectorResetMode* | MinMax detector reset mode.<br>0: Reset on Trigger. Peaks detected only in post-trigger data<br>1: Reset on Start of Segment |
| *u16TsResetMode* | MinMax Time stamp reset mode<br>0: Reset on Start Acquisition<br>1: Reset on Start of Segment |
| *hDataAvailableEvent* | Event for data available in MinMax Detection Queue |
| *hErrorEvent*; | Error event |
| *hSwFifoFullEvent*; | Event for MinMax Detection Queue full |

**Remarks**

The queue must be created before the Peak Detection image is loaded onto the CompuScope FPGA.

Upon return from this function, the application may receive one or more of the following 3 events:

*hDataAvailableEvent*
This event will be signalled whenever there is a SegmentInfo item in the MinMax Detection Queue.
As soon as the event is signalled, the application should call CsExpertCall with the function Id EXFN_GETSEGMENTINFO to retrieve the SegmentInfo from the driver.

*hErrorEvent*
This event will be signalled when a fatal error occurs in the current acquisition.
Once the error is signalled, the current acquisition will be automatically aborted.

*hSwFifoFullEvent*
The event will be signalled when the MinMax Detection Queue is full.
When the driver MinMaxQueue FIFO is full, any MinMax Segment Info structure entries that come from hardware will be discarded. The current acquisition will continue.
The event remains in a signalled state, unless the application resets it via a call to CsExpertCall with the function Id EXFN_CLEARERRORMINMAXQUEUE.

## EXFN_DESTROYMINMAXQUEUE

Call CsExpertCall() with the function Id EXFN_DESTROYMINMAXQUEUE to destroy the driver's MinMax Detection Queue created by EXFN_CREATEMINMAXQUEUE.

```
typedef struct _CSDESTROYMINMAXQUEUE
{
      struct
      {
            uInt32        u32Size;
            uInt32        u32ActionId;
      } in;

} CSDESTROYMINMAXQUEUE, *PCSDESTROYMINMAXQUEUE;
```

**Parameters**

| | |
|---|---|
| *u32Size* | Size of this structure |
| *u32ActionId* | The function Id. Must be `EXFN_DESTROYMINMAXQUEUE` |

**Remarks**
The function fails if it is called when the Peak Detection image is loaded onto the CompuScope FPGA. The standard image CS_MODE_USER0 must be loaded onto the CompuScope FPGA before destroying the queue.

## EXFN_GETSEGMENTINFO

Call CsExpertCall() with the function Id EXFN_GETSEGMENTINFO to retrieve a MinMax Segment Info structure from driver's MinMax Detection Queue, which was created by EXFN_CREATEMINMAXQUEUE.

```
typedef struct _CSPARAMS_GETSEGMENTINFO
{
      struct
      {
            uInt32        u32Size;
            uInt32        u32ActionId;
            uInt32        u32BufferSize;
      } in;

      struct
      {
            MINMAXSEGMENT_INFO      *pBuffer;
      } out;

} CSPARAMS_GESEGMENTINFO, *PCSPARAMS_GESEGMENTINFO;


typedef struct _MINMAXSEGMENT_INFO
{
      uInt32              u32Size;             //size of this structure
      uInt32              u32NumberOfChannels; //Number of channels
      TRIGGERTIMEINFO     TrigTimeInfo;        //The Triggering information
      MINMAXCHANNEL_INFO  MinMaxChanInfo[1];   //The MinMaxInfo for each
                                               channel

}MINMAXSEGMENT_INFO, *PMINMAXSEGMENT_INFO;


typedef struct _TRIGGERTIMEINFO
{
      int64       i64TriggerTimeStamp; //The Time Stamp counter value for
                                       the Trigger Event
      uInt32      u32TriggerNumber;    //The Trigger Number

}TRIGGERTIMEINFO, *PTRIGGERTIMEINFO;


typedef struct _MINMAXCHANNEL_INFO
{
      int16       i16MaxVal;           //Max ADC code within the waveform
      int16       i16MinVal;           //Min ADC code within the waveform
      int64       i64MaxPosition;      //Time Stamp counter value for the
                                       max position
      int64       i64MinPosition;      //Time Stamp counter value for the
                                       min position

} MINMAXCHANNEL_INFO, *PMINMAXCHANNEL_INFO;
```

**Parameters**

| | |
|---|---|
| *u32Size* | Size of this structure |
| *u32ActionId* | The function Id. Must be `EXFN_GETSEGMENTINFO` |
| *u32BufferSize* | Size of the buffer in bytes. |
| pBuffer | pointer to the buffer receiving MinMaxSegment info. |

**Remarks**
The MinMaxSegmentInfo size will be different depending on the mode (Dual or Single channel) and the number of cards in Master/Slave system. The *u32BufferSize* must be equal to at least the MinMaxSegmentInfo size.

## EXFN_CLEARERRORMINMAXQUEUE

Call CsExpertCall() with the function Id EXFN_CLEARERRORMINMAXQUEUE to reset the event *hSwFifoFullEvent.*

```
typedef struct _CSPARAMS_CLEARERRORMINMAXQUEUE
{
      struct
      {
            uInt32        u32Size;
            uInt32        u32ActionId;
      } in;

} CLEARERRORMINMAXQUEUE, *PCLEARERRORMINMAXQUEUE;
```

**Parameters**

| | |
|---|---|
| *u32Size* | Size of this structure |
| *u32ActionId* | The function Id. Must be `EXFN_GETSEGMENTINFO` |

**Remarks**
This function will reset the *hSwFifoFullEvent.* If the driver's MinMax Detection Queue remains full, however, this event will get signalled again.