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.

These SDKs include 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.

	C	C#	VB .Net	LabVIEW	MatLAB	CVI	Delphi
Simple	X	X	X	X	X	X	X
Acquire	X	X	X	X	X	X	X
Coerce	X	X		X	X	X	
ComplexTrigger	X	X		X	X	X	
DeepAcquisition	X	X		X	X	X	
MultipleRecords	X	X		X	X	X	
MultipleSystems	X	X		X	X	X	
AdvMulRec	X					X	
AsTransfer	X	X				X	
Average	X	X		X	X	X	
Callback	X					X	
CsPrf	X	X				X	
Events	X	X				X	
FIR	X	X		X	X	X	
MinMaxDTC	X					X	

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.

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.

Advanced Multiple Record Sample program

Only available with CS14200, CS12400 and CS14105

CompuScope models such as the CS14200, CS12400 and CS14105 allow for the acquisition of pre-trigger data in Multiple Record mode. For these CompuScope models, the user is able to set up a Multiple Record Segment Size that is larger than the post-trigger depth so that pre-trigger data may be accumulated. In addition, trigger Time Stamp values are logged and may be retrieved for each Multiple Record.

The improved Multiple Record features complicate the storage of Multiple Record data in CompuScope on-board memory. For example, the trigger position may be located anywhere within the Multiple Record segment memory and a memory footer exists to store information such as Time-Stamp data. In order to manage these complications, standard GaGe sample programs in all CompuScope SDKs are designed to download Multiple Records one-at-a-time in a software loop. The CsTransfer() method internally manages on-board storage complications and extracts the data for a single Multiple Record segment. While this technique is easy to use, aggregate PCI data transfer rate is compromised, since software overhead of initiating separate transfers for each Multiple Record is introduced. In applications where rapid repetitive Multiple Record acquisitions are required, this software overhead can limit performance. In order to provide the fastest possible download of Multiple Record data, GaGe has provided the AdvMultipleRecord sample program for the C/C# SDK.

The AdvMultipleRecord program requires the same input files and provides the same output files as GageMultipleRecord – the standard Multiple Record C SDK sample program. That is, the CompuScope configuration settings are read from an INI file and the output files are individual Multiple Record .DAT files. The difference is that the AdvMultipleRecord internally downloads all Multiple Record data in one PCI data transfer, rather than using a separate PCI transfer for each Multiple Record segment. Only after all data transfer is complete are the records parsed by AdvMultipleRecord for storage in individual Multiple Record DAT files.

The coding functionality within AdvMultipleRecord and GageMultipleRecord is identical up until the subroutine call to SaveMulRecRawData() within AdvMultipleRecord. The SaveMulRecRawData() subroutine calls the CsExpertCall()subroutine using an ActionId that is equal to the constant EXFN_RAWMULREC_TRANSFER. This call to CsExpertCall() transfers all segment data for all active channels from a Multiple Record acquisition to the specified memory buffer in a single PCI data transfer operation. The buffer must have been previously allocated using the GetMulRecRawDataBufferSize() subroutine, which also calls CsExpertCall().

Once all the raw Multiple Record data have been transferred to an internal buffer, the records are parsed within a loop whose index cycles through the number of Multiple Records. Within the loop, the CompuScope API method or function called CsRetrieveChannelFromRawBuffer() extracts the data for a single record file, whose attributes are specified by variables within the InData structure. The data for the extracted record are returned within the buffer pointed to by pBuffer. Also returned is Time-Stamp data along with other information in the OutData structure. The Multiple Record waveform data are optionally converted into Volts and then are stored within a DAT file with a corresponding header.

The user may easily modify the AdvMultipleRecord program so that the complete raw data buffer data is stored to a binary file. This way, the user does not need to waste time parsing data during repetitive Multiple Record acquisitions. After the acquisitions are all finished, the user can reload the raw data files and extract the data of interest without sacrificing measurement time.

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 Finite Impulse Response (FIR) Filtering, Signal Averaging, and Peak Detection. 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 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” and 4 for “Quad”) 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.

GageAverage

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 signal averaging 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 by using the variable for the Number of Records, which is normally used to select the number of records to be acquired in a Multiple Record acquisition. With the signal averaging firmware loaded, the CompuScope hardware co-adds this number of consecutive waveforms, instead of stacking them in on-board memory as is usually done in Multiple Record Mode.

Co-added waveform data are stored within a 32-bit format buffer within the on-board firmware. The resulting averaged waveform must therefore be transferred as a 32-bit data buffer. With the averaging 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. Co-added waveform data must be divided by the number of waveform averages in order to obtain the averaged waveform. Updated Sample Resolution and Sample Offset values may then be used for waveform voltage conversion.

Each SDK contains an advanced sample program that uploads the signal averaging image, performs a signal averaging acquisitions with an adjustable number of averages, and then displays or stores the resulting averaged waveform.

GageFIR

Finite Impulse Response (FIR) filtering of waveform signals is a powerful method for removing unwanted signal features (like noise) and emphasizing signal features of interest. Unlike signal averaging, multiple repetitive waveforms need not be acquired and an FIR filtering algorithm may be applied to a single waveform data set.

The general form of the FIR filter is:

$$Y_i = \sum_{j=0}^{N} A_j X_{i-j}$$

where:

$\{X_i\}$ is the *input data set*,
$\{Y_i\}$ is the *output data set*,
$\{A_j\}$ is the *set of FIR filter coefficients* $(0 \le j < N)$
and N is the *number of taps* and is equal to the number of coefficients.

The CompuScope FIR firmware image allows up to 20 distinct tap coefficients to be used. FIR filtering is implemented as a numerical *convolution* algorithm. Since both waveform data and tap coefficients are real values with no imaginary components, the FIR filtering algorithm may be used to implement a numerical *correlation* algorithm, simply by reversing the order of the tap coefficients.

During data transfer to the PCI bus, the FIR filtering algorithm is applied to waveform data that have already been acquired into CompuScope acquisition memory. Since the waveform data in CompuScope memory remain unfiltered, different filters may be applied to this same raw waveform data. This is done simply by modifying the tap coefficients and downloading the waveform data again.

Many sets of standard FIR filter coefficients or *filter core*s are symmetric, meaning that *Aj = A-j*. The FIR filtering firmware image allows up to 39 symmetric tap coefficients to be loaded. In this case, the FIR filter calculation is modified to be:

$$Y_i = \sum_{j=0}^{N} \left(A_j X_{i-j} + A_j X_{i+j} \right)$$

As for the signal averaging firmware, FIR filtered data are returned in a 32-bit data format. With the firmware loaded, the Sample Size, Sample Resolution and Sample Offset values are changed accordingly.

In addition, a coefficient scaling factor that scales all tap coefficients is provided. The idealized coefficients *Aj* that are listed above are related to the coefficients, *i16CoefFactor[j],* that are loaded to the FIR filtering image as follows:

$$A_j = \frac{i16CoefFactor[j]}{u32Factor}$$

where *u32Factor* must be a power of 2, with limits listed below. The default value of *u32Factor* used within the FIR filtering sample programs is 32768.

Since the idealized coefficients, *Aj*, are floating point values and generally have absolute values less than 1, greater numerical precision on these coefficients may be obtained by increasing the value of *u32Factor*. However, using a larger value increases the risk that the FIR filtered data values will exceed the available 32-bit width of the output data buffer. Optimal selection of *u32Factor* also requires knowledge of the amplitude of the acquired signal, since larger signal amplitudes will lead to earlier overload of the 32-bit output data buffer.

As an example, consider a symmetric moving average filter core with 39 constant coefficients. Let us assume further that the data may cover the whole 14-bit ADC range of a CompuScope 14200. In this case, summing full scale data points 39 times requires an extra 6 bits, since $2^6 = 64$. This leaves only 32 – (14 + 6) = 12 bits. Consequently, in order to guarantee no output data overload, *u32Factor* should be no larger than $2^{12} = 4096$.

From C, the FIR operation is configured by a call to CsSet(hSystem, CS_FIR_CONFIG, &FirConfig), where hSystem is the CompuScope system handle, CS_FIR_CONFIG is a constant defined in CsDefines.h and FirConfig is a variable of the type

CS_FIR_CONFIG_PARAMS that is defined in CsExpert.h. This call does not require a commit action and takes effect immediately.

The parameters for configuration of the FIR filtering algorithm are specified within a variable of type CS_FIR_CONFIG_PARAMS:

Field name	**Type**	**Description**
u32Size	uIn32	Total size, in Bytes, of the structure
bEnable	BOOL	Enable FIR. If Disabled, a unity filter is used
bSymmetrical39th	BOOL	If true, assume that the coefficients are part of a 39-tap symmetrical filter core
u32Factor	uInt32	Scaling factor used for all coefficients. Allowed values are 2^(2*n+1) 1 <= n <= 10
i16CoefFactor	int16 [16]	Core coefficients are represented in a fixed point format. This array contains numerators, while the denominator is stored in the u32Factor field

Each SDK contains an advanced sample program that uploads the FIR filtering image, performs an acquisition with FIR filtering, and then displays or stores the resulting waveform.

GageMinMaxDtc (Peak Detection)

The eXpert Peak Detection firmware option allows on-board detection of the minimum and maximum amplitudes that occur within a waveform, along with their positions within the waveform. Calculated Peak Information Sets for each waveform are accumulated within the CompuScope FPGA for PCI download. The data reduction associated with transforming the raw waveform data into the compact Peak Information Set correspondingly reduces the PCI data traffic so that a faster repetitive capture rate may be accommodated. This section describes operation of the eXpert Peak detection firmware from the C programming environment. Please note: the terms Peak detection and MinMax detection are used interchangeably. The terms Peak Information Set and MinMax Segment Info structure are also used interchangeably.

Usage of Peak Detection from C

In the “Advanced” folder within the “C Samples” folder of the CompuScope C/C# SDK is a Visual C sample project called GageMinMaxDtc that operates CompuScope hardware using the eXpert Peak Detection firmware option. This project configures the CompuScope hardware and does an acquisition using the eXpert Peak Detection firmware until a pre-set number of waveforms have been acquired. The project stores a selectable number Peak Information Sets in an ASCII file.

GageMinMaxDtc receives input configuration settings from an INI file, as do standard C

SDK programs. Standard CompuScope input parameters are listed within the INI files and are the same as those documented in the C SDK. The following parameters are added for control of Peak Detection acquisitions.

SegmentCount - Determines the number of waveforms to be acquired by the program.

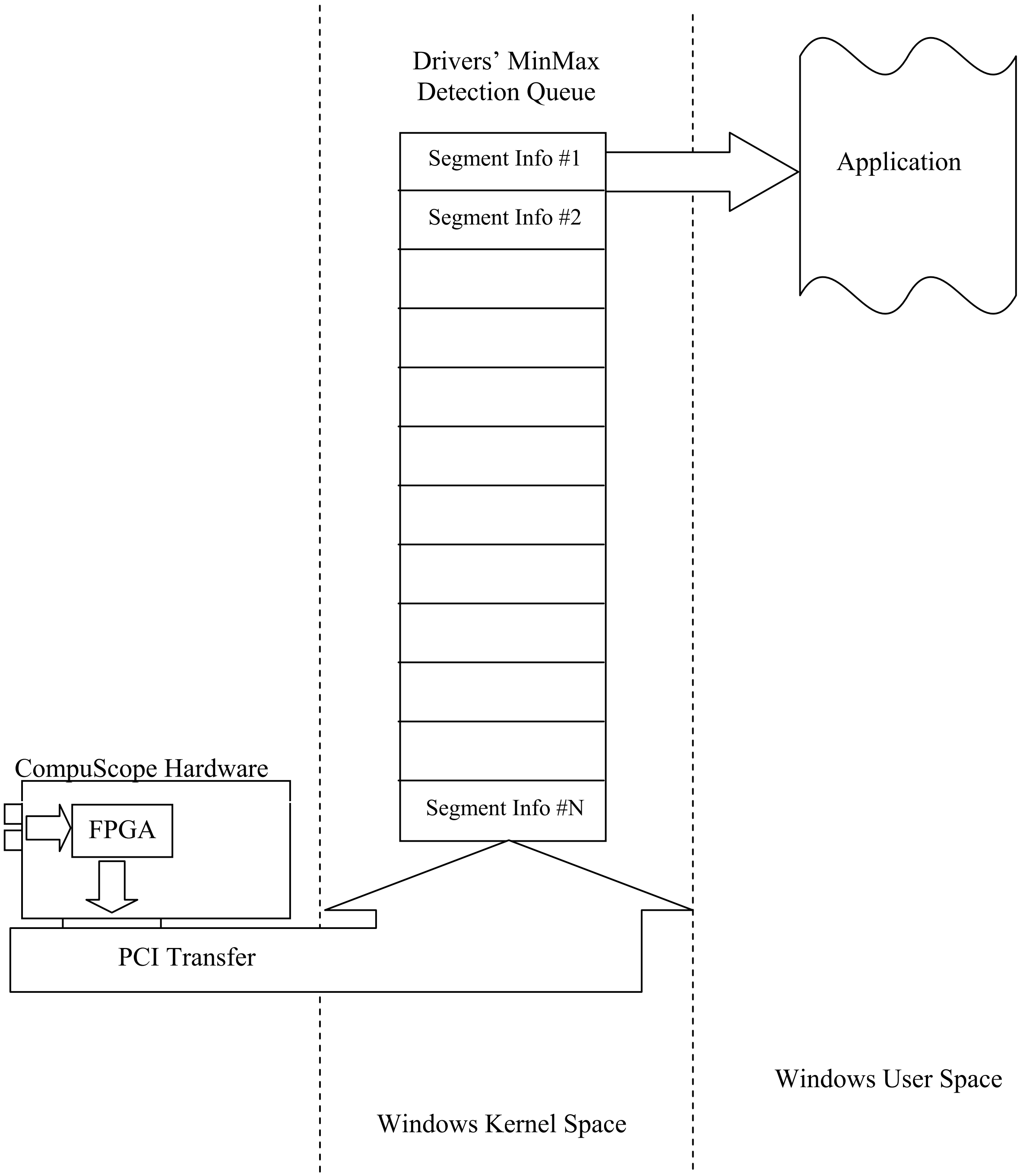
QueueSize - Sets the number of entries within MinMax Detection Queue that is described below.

LastSegmentSave - Determines the number of peak data sets to be stored in the output TXT file. If this value exceeds SegmentCount, then the most recently acquired segments are those that are stored.

TsResetMode - This setting determines when the Time Stamping counter is reset. When it is set to 0, the Time Stamping Counter is reset only once at the start of the acquisition sequence. When it is set to 1, the Time Stamping Counter is reset at the beginning of each segment acquisition. For peak detect operation, this setting should always be 0.

DetectorResetMode - This parameter determines where the Peak detection algorithm should begin looking for peaks within the acquired waveform. When DetectorResetMode is set to 0, the Peak detection (also called MinMax detection) algorithm is reset at the trigger position within an acquired segment. This way, all peaks will be detected within post-trigger waveform data only. When DetectorResetMode is set to 1, the MinMax detection algorithm is reset immediately at the start of segment acquisition so that MinMax peaks may be detected within pre- or post-trigger data.

In order to understand operation of the eXpert Peak Detection firmware option it is important to understand the hardware/firmware/software architecture, which is illustrated in the Figure below. First, waveform acquisition by the CompuScope is triggered, as usual. Instead of storing the raw data in CompuScope on-board memory, the raw waveform data are analysed within the on-board FPGA to determine the peak parameters. These parameters are assembled into the Peak Information Set (also called the "MinMax Segment Info structure" in the API function descriptions). Peak Information Sets are accumulated within the FPGA and are periodically PCI transferred by the driver to the "MinMax Detection Cue", which is a Windows Kernel Level buffer that may accommodate a number of Peak Information Sets that is specified by the QueueSize value in the INI file.


Drivers' MinMax
Detection Queue
Segment Info #1
Segment Info #2
Segment Info #N
Application
CompuScope Hardware
FPGA
PCI Transfer
Windows User Space
Windows Kernel Space

Peak Information Sets are constantly added to the MinMax Detection Cue as they become available. The user application polls the MinMax Detection Cue to check if new sets are available. When available, new sets are transferred from the Kernel Level MinMax Detection Cue to an Application Level buffer, where they may be accessed by the application for analysis, display or storage.

The contents of the Peak Information Set (or MinMax Segment Info structure) and the size of each entry are shown in the table below.

PEAK INFORMATION SET (8 BYTES + 16 BYTES + 24 BYTES × NUMBER OF CHANNELS)					
NAME	**DESCRIPTION**	**TYPE**	**SIZE**	**GROUP NAME**	**GROUP SIZE**
Structure Size	The size of the base structure for compatibility purposes	uInt32	4 Bytes	Header	8 Bytes
Number of channels	The number of channel information sets	uInt32	4 Bytes		
Trigger Number	The trigger count, which may be used to account for missed triggers, if any	uInt32	4 Bytes	Trigger information set	16 Bytes
Reserved		uInt32	4 Bytes		
Trigger Time-Stamp	The Time-Stamp counter output that marks the occurrence time of the trigger event.	int64	8 Bytes		
Max Amplitude	The maximum value that occurs within the waveform data set	int16	2 Bytes	Channel information set	24 Bytes
Min Amplitude	The minimum value that occurs within the waveform data set	int16	2 Bytes		
Reserved		uInt32	4 Bytes		
Max Time-Stamp	The Time-Stamp counter output that marks the occurrence time of the maximum	int64	8 Bytes		
Min Time-Stamp	The TimeStamp counter output that marks the occurrence time of the minimum	int64	8 Bytes		

The Peak Information Set contains the Min and Max Amplitude information in units of raw ADC samples. Also included are the Time-Stamp time values for the trigger event, and the MinMax positions. These values come from the on-board Time-Stamping counter and may be converted to absolute time values with knowledge of the Time-Stamping counter's clock source frequency, which may be obtained by the driver, as usual.

A distinctive parameter in the Peak Information Set is the Trigger Number parameter. It is possible that, while the Peak detection algorithm is processing a waveform caused by a given trigger, another trigger event occurs. This trigger event and its associated peak information will thus be missed by the CompuScope hardware. In this event, however, the Trigger Number value will be incremented by the missed trigger. Consequently, the user will find that the Trigger number between consecutive Peak Information Sets increased by 2 instead of by 1 and so will know that a trigger was missed and can correctly account for it.

From the above chart, we may calculate the size of a Peak Information Set as:

24 Bytes + 24 Bytes × Number of active channels

Since waveform acquisitions typically consist of thousands of Bytes of data or more, the Peak Detection firmware clearly leads to a significant reduction in data volume and the associated PCI transfer traffic. This reduction allows much higher repetitive waveform capture rates to be achieved without trigger losses.

In order to provide further clarification, important steps within the GageMinMaxDtc sample project are described below. Within GageMinMaxDtc, configuration of standard CompuScope input parameter settings is done as usual.

1) Create the MinMax Detection Queue

   This is done by making a call to CsExpertCall() with a function Id of EXFN_CREATEMINMAXQUEUE within the FunctionParams structure.
   Once this function has been executed, the MinMax Detection Cue is set up and the application will be able to receive the handle of *hDataAvailableEvent* event.

2) Set the acquisition mode to CS_MODE_USER1 (or CS_MODE_USER2)

   The step loads the Peak Detection image from the CompuScope on-board flash memory to the CompuScope FGPA. The user must first determine whether the Peak Detection image resides in the USER1 or USER2 flash location. This may be determined from CompuScope Manager or from the driver.

   This example code loads the USER1 image to the FPGA:

   ```
   CsAcqConfig.u32Mode = CS_MODE_USER1 | CS_MODE_DUAL;
   CsSet(hSystem, CS_ACQUISITION, &CsAcqCfg);
   CsDo(hSystem, ACTION_COMMIT);
   ```

3) Start acquisition.

   Example:

   ```
   CsDo(hSystem, ACTION_START);
   ```

4) Wait for the *hDataAvailableEvent* event.

5) Once the event is signalled, call CsExpertCall() with the function Id EXFN_GETSEGMENTINFO to retrieve a single MinMax Segment Info structure entry (Peak Information Set) from the driver.

6) Repeat calling CsExpertCall() with the function Id EXFN_GETSEGMENTINFO until receipt of error CS_SEGMENTINFO_EMPTY. This error indicates that all available Peak Information Sets have been transferred to the application.

7) Process, display and/or store the data, if necessary

8) Return to step 4

9) The MinMax Detection application will continue until the number of Peak Information Sets acquired is equal to the pre-set number specified by SegmentCount in the INI file. This criterion may be easily changed according to the requirement.

In a MinMax Detection acquisition, the MinMax Segment Info structure entries that come from CompuScope Hardware are in RAW format. The GageMinMaxDtc sample project will convert these MinMax Segment Info structure entries to a readable format and save them in the MinMax Detection Queue.

The MinMax Detection Queue acts as a FIFO buffer. The driver saves new MinMax Segment Info structure entries at the bottom of the queue and the user application reads old MinMax Segment Info structure entries at the top of queue. Whenever the application retrieves a MinMax Segment Info structure entry, it removes that MinMax Segment Info structure entry from the queue, leaving space for the driver to save new incoming MinMax Segment Info structure entries.

If the application is not fast enough to remove MinMax Segment Info structure entries from the driver's MinMax Detection Queue, the queue will become full. When the queue is full, there is no more space for the driver to save new MinMax Segment Info structure entries, thus any new RAW MinMax Segment Info structure entries coming from CompuScope hardware will be discarded. When this happens, the driver will notify the application via the *hSWFifoFull* event.

There are actually three potential ways of missing waveform triggers and the associated MinMax Segment Info structure entries (Peak Information Sets). First, a trigger may be missed if it occurs while a previous waveform is being acquired. Second, if the Peak Information Sets within the FPGA are not transferred quickly enough, they may accumulate and exceed the capacity of the FPGA and some sets will be lost. Finally, as discussed in the previous paragraph, sets will be lost if the MinMax Detection Queue is not purged frequently enough by the controlling application. Any and all of these loss situations may be detected simply by checking the Trigger Number within the Peak Information Set. If the difference between the Trigger Numbers from successive Peak Information Sets is greater that 1, then the excess is exactly equal to the number of missed sets, regardless of the loss mechanism.

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. 0: Reset on Trigger. Peaks detected only in post-trigger data 1: Reset on Start of Segment
u16TsResetMode	MinMax Time stamp reset mode 0: Reset on Start Acquisition 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.