Accessing PhotoniQ Events Using Windows Messaging

Application Note

Overview

The example code provided can be used as a template for custom application development. The sample application is a basic dialog type that simply initializes the PhotoniQ support DLL, configures the PhotoniQ hardware, and executes data collection requests via the PhotoniQ DLL interface. Used directly, the DLL DataInterface() call is useful for requesting only one event per call. By configuring the PhotoniQ hardware for *windows messaging* mode, the user may request multiple (100's of thousands if desired) events. This example provides a *wrapper* around the documented DataInterface() call to allow these block requests to be serviced. The user should still refer to the PhotoniQ user manual for a description of the low-level DLL calls, as well as the format and definition of all the control parameters within the PhotoniQ hardware.

The example takes a GUI button hit as a stimulus to request a block of events from the PhotoniQ. In the button servicing routine, a GetEvents() call is made. This get events call, creates a new processing thread within which the DataInterface() call is made. The new thread runs in parallel with the main window. The thread will signal the main window via custom window messages. Within the message handler, the partial blocks of events are parsed and neatly stacked in the local EventBuffer[]. There may be multiple messages to create the desired block of events. For instance, a request of 100K events using an internal sample rate of 10 KHz may result in 20 to 30 windows messages each containing close to 5,000 events. The data message handler shows how the events are conditioned and *stacked* contiguously in the user-defined buffer. After all events are gathered, or the specified timeout(s) have expired, the data collection (PhotoniQ DLL) thread posts a "Xfer Complete" message back to the main window. Note, examine the example button service code where the GetEvents() call is made. It must simply return and let the main window idle. Processing of the data will be blocked if the user attempts to spin in a code loop waiting for the "Xfer Complete". The main windows message servicing must be allowed to execute or data will get overwritten in the A and B ping-pong buffers (see below for more detail).

Initializing the PhotoniQ DLL

During application start-up, the PhotoniQ initialization takes place. First, the application makes sure that the PhotoniQ.dll file is successfully located and loaded into memory. The code then calls an initialize defaults function. Within this function one can set any or all of the configuration parameters supported (see user manual). For our example, internal triggering is selected, sample rate, integration time, delay, etc. The code also examines the maximum number of channels the system supports and enables all of them. The user will most likely change these to meet their particular needs. These parameter all reside in a large table called the PhotoniQConfigTable. Via DLL calls, the user can read the table, and write the table back to the hardware as described in the section below. This initialization process also sets up local variables that will get passed to the DLL during a DataInterface() call. These variables select the *messaging* mode for data collection processing.

Configuring the PhotoniQ

The PhotoniQ keeps a flash copy and a ram copy of the configuration table. Via API calls, one can read or write either version of the table. Note, when the flash copy of the configuration table is modified, the ram copy is updated to mirror it. It is strongly recommended to not modify the flash version to prevent an unwanted permanent configuration. The example code will read a copy of the ram-based configuration from the PhotoniQ during initialization. The application is then holding a true mirror of the table locally. The API calls provided will modify the local configuration table only. Once the user has made all the configuration changes locally, use the API call that writes the configuration table back to the PhotoniQ hardware. The write-back process, if successful, forces the new desired configuration and the PhotoniQ is now programmed as desired.

In the file photoniq.h, a structure of type: PHOTONIQ_CONFIG_TABLE is defined. This structure represents the user, custom, and factory configuration entries specified in the user manual. All three configuration spaces are placed contiguously in the structure. The full structure is written over on a `bool PhotoniqReadConfigFromRam(unsigned int Length)` call, at that point it represents the PhotoniQ hardware table. During a `bool PhotoniqWriteConfigToRam(unsigned int Length),` the local structure is copied to the PhotoniQ hardware. The DSP in the Photoniq hardware handles any processing of the table required to implement the desired configuration. The hardware is then ready to capture data.

Data Collection via Window Messaging

The example data collection code is called upon reception of a GUI button hit. Within the *button hit* message service, the code sets up and executes a call to GetEvents(). Prior to the call, local variables, such as various timeouts are setup. These are the same variables shown in the documentation when one calls the DataInterface() function directly. The DataInterface call is encapsulated within GetEvents(). GetEvents() ultimately spawns a new thread from within which the DataInterface() call is made.

The DataInterface code resides in the PhotoniQ.dll. When requesting *blocks* of events, the DataInterface() sends portions of blocks to the service code in the main window processor. There are custom WM_xx messages defined in Photoniq.h for passing valid events back and also for signaling the completion of the block event transfer. Completion occurs when either all events have been sent or if a programmed timeout period is reached. It is important to note that the GetEvents() function spawns the DataInterface thread then simply returns and allows the main window to idle. Idle means that the windows messaging now guides what the main application is doing. WM_TIMER messages can go off, WM_PAINT, etc. If the user attempts to spin in a code loop waiting for the DataInterface thread to finish, data will be corrupted and lost. The DataInterface() posts the custom messages along with buffer pointers when the next sub-chunk of events is ready. The number of events in a sub-chunk is variable and depends upon sample rate, integration period, etc. The DataInterface() gathers as many events as it can over a 30 msec window. After each 30 msec window, the events gathered are placed in a buffer and a message is posted to the main window. The DataInterface() uses a buffer ping-pong technique so it may continue to take events and save them in the *other* buffer while the windows message service code offloads the previous block of events.

Vertilon Corporation, 66 Tadmuck Road, Westford, MA 01886 / Tel: (978) 692-7070 / Fax: (978) 692-7010 / www.vertilon.com

The windows message service `LRESULT ProcessDataPacket(WPARAM wParam, LPARAM lParam)` runs during each DataInterface() dump of an event buffer. In this code you will see how the number of events varies and how they are off-loaded and placed contiguously in the local EventBuffer[]. When the DataInterface() call is complete, the message service `LRESULT ProcessDataXferDone(WPARAM wParam, LPARAM lParam)` runs. This function is where one would typically splice their own custom code into. It is important to check for timeout and requested data events. Sometimes the DataInterface() call returns more than the desired number of events so it is truncated to the requested amount here. If the user decides to perform some processing on the 30 msec event dump boundary, then custom code can be inserted in the ProcessDataPacket() service.

For information about the format of the events, see the documentation regarding messaging mode and the example code since the example enables TriggerStamps to be appended to each event. The messaging mode automatically adds four words (16-bits) to the beginning of each ping-pong buffer. These are stripped off and thrown away within the ProcessDataPacket() service because they are essentially duplications. The important point is that the format for each of the events in the example is as follows. All data is in 16-bit words and the TriggerStamp is 32-bits wide, split into low and high words.

Header (1)

Data (NumChannels Enabled)

TriggerStamp Low (1)

TriggerStamp High (1)

For the example code we enable all channels that the system supports. Assuming it is a 64 channel system, each event we will receive 67 words of information. These groups of 67 words will be packed within the message data ping-pong buffer sent by the DataInterface() function. There will be one word of a header, 64 words of data (one word for each channel) and two words of trigger stamp (low word, high word). Approximately every 30 msec of event capture, the DataInterface() will post a data ready message back to the main window. As described above, this message gets serviced in the function `LRESULT ProcessDataPacket(WPARAM wParam, LPARAM lParam)`. The wParam is either ‘1’ or ’0’. The value specifies which of the two ping-pong buffers holds the valid event data. The lParam contains the number of valid 16-bit words in the data buffers. These data buffers are called `DataBufferA[]` and `DataBufferB[]`. As mentioned above, the first four words of the data buffer are ignored. Thus the number of words in the data buffer = lParam – 4. lParam should be an exact multiple of the size of one event packet. In the code example there is 67 words per event. Note that each ProcessDataPacket() service needs to dynamically figure out the number events packed in the data buffers by examining the passed wParam and lParam.

Vertilon Corporation, 66 Tadmuck Road, Westford, MA 01886 / Tel: (978) 692-7070 / Fax: (978) 692-7010 / www.vertilon.com

Description and Flow of Example Code

The following describes the specifics of the included example code. It places the system into a typical mode of operation and shows examples of initializing, configuring and making event requests via the GetEvent() call.

The code begins by following the typical program initialization steps. The OnInitDialog() will force a fake *event trigger* (front panel GUI) event. The OnBnClickedTrigger() code is smart enough to know that this is the initial start-up. The PhotoniQ initialization code insures that the PhotoniQ.dll is loaded and the PhotoniQ hardware is connected to the USB port. It also sets up local variables that will ultimately be used by the GetEvent() call (passed thru to the DataInterface() call), which enables the windows messaging mode of data collection. One can see where the window handle and custom WM_xx message is loaded into variables that get used in the DataInterface() thread. Upon initialization, the code also starts a 60 second WM_TIMER just for example purposes. One may want to use a timer as a stimulus to call GetEvents() rather than a user button click.

Once the PhotoniQ is initialized, the example code calls the configuration setting routine PhotoniqSetDefaults(). This routine has examples of calling API functions that will set the number of channels, sample rate, integration period and delay, trigger source, etc. The configuration chosen is merely for example purposes. This function can be customized per the user's needs. After calling the setup, the code initializes local and GUI variables and updates the screen.

Data collection is initiated by the user clicking on a GUI button. The OnBnClickedTrigger() function contains a call to GetEvents(). In the example code the call is requesting 1000 events. Timeouts for event to event are specified as well as total collection timeout. Lastly, a pointer to the local EventBuffer[] is passed. This buffer is where the ProcessDataPacket() code assembles the fragmented event data into one contiguous block of events. Note that this buffer is different than the DataBufferA[] and DataBufferB[].

There are provisions to dump individual event packets (channel data) to the screen with channel text. There are also some basic data processing routines which handle only one event at a time. Some of the data event and raw data processing buffers are only one event in depth. Here the raw scaling to picocoulombs can take place.

Miscellaneous

The DataBufferA[] and DataBufferB[] are fixed at 1M words. The EventBuffer[] is fixed at 1M times the maximum channels (128) words. This may be too large or inefficient for most applications. Size the buffers appropriately and/or use dynamic memory allocation. Note that the PhotoniQ.dll does assume that the two data buffers are 1M in size.

See documentation regarding *low speed* and *high speed* PhotoniQ sampling modes. This example codes uses *high speed* mode exclusively. Do not change this. Note that the example code forces the PhotoniQ to *Standby* mode before making a GetEvents() request. The DataInterface() function has code that changes the PhotoniQ to *Acquire* mode, then collects the desired number of events. There is no reason for the user to make an API call forcing *Acquire* mode.

In summary, the DataInterface() DLL function is spawned and run in a thread independent of the main window. The main window must be allowed to service all messages in real time while the DataInterface() thread runs. In the provided example, the user must exit the GetEvents() call, which spawns the data collection thread, and allow the data collection to take place via the two windows messaging services ProcessDataPacket() and ProcessDataXferDone(). The user may not spin in a wait loop after creating the data collection thread and wait for it to finish. If a wait loop is entered, the main window will be blocked from servicing the data ready messages posted by the DataInterface() function. These messages will get queued up and serviced after the loop exits. If the user can insure that the requested event data will only require two posted messages from the DLL, then it is possible all event data will be valid. Once the ping-pong buffers are used more than twice, oldest data is overwritten by newest data. In that case corruption will occur even though all posted messages that were delayed, will make it through to the message processor.

AN3323.1.0 Jan 2011

Vertilon Corporation, 66 Tadmuck Road, Westford, MA 01886 / Tel: (978) 692-7070 / Fax: (978) 692-7010 / www.vertilon.com