PQTimeHarp200 Class(es) Description

# I. Introduction

What follows is the description of the PQTimeHarp200 class(es) - this is a class(es) that facilitates the interaction with Time Harp 200 counting board (from Picoquant Inc.) from a software point of view, i.e. developing custom software from the provided DLL libraries. The guidelines explains the following things:

* Sketches the basic structure and organization of the class.
* Explains the functionality of important functions.
* Gives few examples how to use the class by a test program that implements and makes few basic measurements and/or processing of data.
* Includes short overview of an emulation class that allows to emulate the THLib DLL library so that one can test and debug the custom software that uses the class on any PC (it is not necessary to have Time Harp 200 counting board installed).

# II. Structure and organization of PQTimeHarp200 class(es)

## II.1. Folder Location of the Class(es)

An important thing to note is that the C# class to interact with Time Harp 200 device is actually split into two classes:

1. **TimeHarp200DLL.cs**
2. **PQTimeHarp200.cs**

The first one (i.e. **TimeHarp200DLL.cs**) contains the definition of the imported DLL functions from the THLib.dll so that one can use it within C# to implement other more complex functionality. The class file is located in:

**...\PQTimeHarp200App\KUL.MDS.Hardware\TimeHarp200**

The second class (i.e. **PQTimeHarp200.cs**) contains functions that intend to further extend the functionality of **TimeHarp200DLL.cs class** and facilitates development efforts to interact with Time Harp 200. The class file is located in:

**...\PQTimeHarp200App\KUL.MDS.Hardware**

**The main reason to have this functionality in two classes is to keep different types of logic separate and thus clean, which should make the code easier to maintain.**

**Note that these classes are in the folder KUL.MDS.Hardware and have the following locations relative to it:**

* **KUL.MDS.Hardware**
  + **PQTimeHarp200.cs**
  + **TimeHarp200**
    - **TimeHarp200DLL.cs**

The folder **PQTimeHarp200App, one level above KUL.MDS.Hardware, contains the files of the implemented test Console Application that uses the classes in KUL.MDS.Hardware to show how to integrate them in a custom software. It also helps to test the various algorithms and methods implemented in those classes.**

## II.2. Structure and organization of TimeHarp200DLL.cs ****class****

In this class we keep the definitions of the imported DLL functions from **THLib.dll** library as well as number of constants that come from either the header files provided with THLib or were defined additionally but kept in this place for semantic reasons. Also the structure(s) that represent the Time Harp TTTR binary file header are defined and kept here.

In short, if you do not intend to program for **PQTimeHarp200.cs** class, you do not need to bother with this file - most likely you are not going to use it directly for the custom software you want to develop. Thus all necessary functions you will ever need are wrapped and exposed through **PQTimeHarp200.cs** class. So for now forget for the existence of this class (the only thing you need to do is to include it in the files of your C# project).

## II.3. Structure and organization of PQTimeHarp200.cs ****class****

We can divide this class into two major parts:

1. Functions that are wrappers of the same functions from **TimeHarp200DLL.cs** class (respectively imported from **THLib.dll** library). An wrapper function adds few additional things - error checking and in some cases renames the original function's name to a easier to use name. Sometimes also we add a little more functionally related to some of the functions so that we can extend a little the functionality of the imported DLL functions and make easier the further programming efforts.
2. The rest of the class has the routines and algorithms necessary to interpret and process the data generated from Time Harp 200 device.

Therefore if you want to use some DLL function it is recommended to call it through **PQTimeHarp200D.cs, which uses internally the TimeHarp200DLL.cs** class and **takes care for error handling.**

**In the next section we will discuss the more important functions, along with their functionality, that one needs in order to integrate successfully the Time Harp 200 device in whatever higher level C# software.**

# III. Description of important functions in PQTimeHarp200.cs class

The class is split to few sections depending on the functionality that different methods implements.

So below we will discuss more in details the relevant parts of this sections.

## III.1. Initialization and common functions of PQTimeHarp200.cs ****class****

In this section of the class are the properties and methods that are (as the name states) used to initially set some parameters of Time Harp 200 device. Most of the methods are wrappers of the functions imported from the **THLib.dll** (through the **TimeHarp200DLL.cs** class).

Short description of most important methods and properties follows:

1. **public int ErrorCode - a property, gets the error code as an integer number. After each execution of a function from the THLib.dll the error code status it returns goes here.**
2. **public StringBuilder ErrorString - a property, gets the error string (as a StringBuilder) of the error code ErrorCode.**
3. **public long TotalSamplesAcquired - a property, gets the total number of photons read from Time Harp 200 and contained in the images in each scan session.**
4. **public string LibraryVersion - a property, gets the library version of THLib.dll.**
5. **public string HardwareVersion - a property, gets the hardware version of Time Harp board.**
6. **public string SerialNumber - a property, gets the serial number of Time Harp board.**
7. **public int MeasurementMode - a property, gets the current measurement mode. Two modes available - 0 = standard histogramming, 1 = TTTR mode.**
8. **private void Initialize(int \_\_iMeasurementMode)** - a method, initializes Time Harp device with the given measurement mode. The method is used only internally. Not very important method. If you want to set the measurement mode directly you can use either **SetMeasurementMode()** or **SetupAPDCountAndTiming()**. Basically I do it through the latter one, which also allows to set other important parameters in the same time. Moreover, **SetupAPDCountAndTiming()** calls internally the **SetMeasurementMode()** method.
9. **private void Calibrate()** - a method, calibrates Time Harp device. It is necessary to perform it prior to any measuring session. Very important function but we use it only internally and take care to perform it in the constructor of the class **PQTimeHarp200.cs**.
10. **public int RangeCode - a property, sets and gets the so call range code. It defines the time resolution of the channel (in multiples of 2 of the base time resolution of Time Harp) for the start-stop time of the event. RangeCode = 0 means 1x the base resolution; RangeCode = 1 means 2x the base resolution; RangeCode = 2 means 4x the base resolution, and thus up to RangeCode = 5, which means 32x the base resolution (resolution = (base\_resolution)RangeCode); Thus depending on the Sync rate you have to adjust the RangeCode such that the respective time resolution multiplied by the maximum channel number (Time Harp 200 has 4096 channels) can cover a full Sync period.**
11. **public float Resolution - a property, gets the current time resolution in [ns] as measured from the calibration of Time Harp. Basically the calibration happens when the constructor of the class is called.**
12. **public int GetCountRate(int \_\_iMeasurementMode, int \_\_iAcquisitionTime)** - a method, it returns the current count rate. The acquisition time **\_\_iAcquisitionTime** is in [ms]. The possible values for the measurement mode **\_\_iMeasurementMode are: 0 - one time histogramming or TTTR mode; 1 - continuous mode.**
13. **public int GetSyncRate()** - a method, reads and return the rate of laser pulses form the SYNC input of Time Harp.
14. **public void StartMeasurement()** - a method, instruct Time Harp to start a measurement. Most likely you do not want to call directly this method. Another method called **StartAPDAcquisition()** is recommended instead (it calls **StartMeasurement()** internally but handles also other things). However, you can use this method whenever you want to implement your own logic for interaction with Time Harp.
15. **public void StopMeasurement()** - a method, instruct Time Harp to stop a measurement. Most likely you do not want to call directly this method. Another method called **RequestMeasurementStop()** is recommended instead. However, you can use this method whenever you want to implement your own logic for interaction with Time Harp.
16. **public int GetElapsedMeasurementTime()** - a method, returns the elapsed time in [ms] since the beginning of a measurement. If you just want to check if a measurement is still running (without caring about elapsed time) it is recommended to read the property **IsMeasurementRunning, which keeps track if the measurement finished or not (and has a Boolean value, i.e. easier to check the running status).**
17. **private volatile bool m\_IsMeasurementToBeStop - a property (very important), which allows to stop (if set to true) the current measurement in a sense that it triggers a chain of events that: instructs Time Harp to stop measuring, Read() thread to stop reading the Time Harp buffer, Save() thread to stop saving TTTR data to file and BuildImage() thread to stop extracting images from TTTR global buffer. Since this is a private property you can access it and thus stop measurement/processing by calling RequestMeasurementStop() method. It** does exactly one thing - sets this property to true. The keyword volatile is used to make sure every thread will read the most up to date value of this property and act accordingly (either terminate, if it is true, or continue to work if it is false).
18. **public void RequestMeasurementStop() - a method (very important), which sets m\_IsMeasurementToBeStop to true and thus terminates Read(), Save() and BuildImage() threads, this causes also Time Harp to stop measurement. Use it whenever you want to stop completely acquisition and processing (e.g. to stop scanning).**
19. **public bool IsMeasurementRunning** - a property, gets (keeps track of) the acquisition status of Time Harp. I**f the measurement finished (or has not started) it reads false, otherwise true (i.e. measurement still running).**

These were the most important function within this subsection of the class. Now let's consider the next section.

## III.2. Special functions for continuous mode of PQTimeHarp200.cs ****class****

This subsection is dedicated of functions related to the continuous mode (as opposite to TTTR mode) of operation of Time Harp. Since it is not currently used I will not discuss them. We just directly go to the next subsection.

## III.3. Special functions for TTTR mode of PQTimeHarp200.cs ****class****

Short description of most important methods and properties follows:

1. **private int T3RDoDMA(uint[] \_\_uiDataBuffer, uint \_\_iRecordsCount)** - a method, initiates and performs DMA transfer in a single function call. The TTTR data from Time Harp buffer goes into the supplied **\_\_uiDataBuffer** array. **\_\_iRecordsCount is the number of records to be fetched (must not be bigger than the size of the supplied array).** These function is not used in the moment and can be accessed only within the class.
2. **private int T3RStartDMA(uint[] \_\_uiDataBuffer, uint \_\_iRecordsCount)** - a method, initiates and performs DMA transfer. It returns before completion. The TTTR data from Time Harp buffer goes into the supplied **\_\_uiDataBuffer** array. **\_\_iRecordsCount is the number of records to be fetched (must not be bigger than the size of the supplied array).** This function is used internally in the **Read()** thread to read the TTTR data from Time Harp buffer and copy it to the global TTTR buffer.
3. **private int T3RCompleteDMA()** - a method, waits for DMA transfer to complete (after calling **T3RStartDMA()** function). It returns the number of the fetched records. The TTTR buffer **\_\_uiDataBuffer** from T3RStartDMA() must not be accessed until this function returns. These function is used internally in the **Read()** thread to wait for the DMA transfer to complete.
4. **public int T3RSetMarkerEdges(int \_\_iMarkerEdge0, int \_\_iMarkerEdge1, int \_\_ iMarkerEdge2)** - a method, sets the active edges of the TTL signals that will be recognized as external markers. Every argument has one of the two possible values: 0 - sets the active edge to be the falling edge; 1 - sets the active edge to be the rising edge. I would recommend using 1 (active edge to be the rising edge) for every marker edge. The method is used in the class constructor to set the active edges - all markers get the same active edge. The method is made public in case one want to change the active edges directly for some reason.

These were the all functions within this subsection of the class. Now let's consider the next section.

## III.4. Special functions for routing of PQTimeHarp200.cs ****class****

This subsection is dedicated of functions related to the Routing (must have PRT/NRT 400 Router) operation of Time Harp. Since we do not have a router the functions are not currently used and I will not discuss them. We just directly go to the next subsection.

## III.5. Special functions for TTTR file handling of PQTimeHarp200.cs ****class****

This subsection is dedicated of functions related to reading and writing to a TTTR binary file as well as converting a TTTR file to a readable ASCII file or extracting and saving bitmap images from a TTTR file.

The following important structure and methods are available here:

1. **public struct** Files - a structure related to the handling of TTTR files (keeps track of important file parameters and operations). The two most important parameters to note here are:
   1. **public static** TimeHarp200DLL**.**StructTTTRFileHeader **TTTRFileHeader** - as the variable name indicate this is a structure (as defined in **TimeHarp200DLL.cs**) that reflects the file header of a TTTR binary file and is intended to facilitate the reading and writing of such headers. Note that this structure contains four other structures that logically represent the TTTR binary file header. These structures are: **TTTRFileHeader.TextHeader, TTTRFileHeader.BinaryHeader, TTTRFileHeader.BoardHeader, TTTRFileHeader.TTTRHeader. For more detail look of what parameters these particular sub-headers have, see their definition in TimeHarp200DLL.cs class. Another important thing to note is that within this TTTRFileHeader structure there are two fields (TTTRFileHeader.BinaryHeader.DisplayCurves and TTTRFileHeader.BinaryHeader.Params), which cannot be allocated by mere declaring of the structure. This is due to the fact that they are arrays of structures and C# does not allocate memory automatically for such compound fields. In order to solve the problem we allocate them in class constructor of PQTimeHarp200.cs. If you want to use this type of structure** TimeHarp200DLL**.**StructTTTRFileHeader **outside the scope of the class you need to take into account this (e.g. if you make your own routine to read/write such header based on this structure).**
   2. **public const int MAXIMUM\_TTTR\_RECORDS - this integer constant defines an upper limit for the maximum number of TTTR records allowed to have per TTTR binary file. This upper limit is defined by** **MAXIMUM\_TTTR\_RECORDS\_PER\_FILE** constant (which is defined in **TimeHarp200DLL.cs**). For the time being it equals **228 = 268 435 456 maximum records per TTTR file allowed. It has two purposes - one is to prevent the TTTR file to exceed the 2GB limit on 32-bit systems (which if not done will crash the program); the second is to limit the file size, which can in general grow indefinitely and thus make the processing a little easier. The value of 228 will result in roughly 1GB file size. In this way while acquiring TTTR data the class routine that writes TTTR records to file will automatically stop to write the records to the current file and continue to store them in a new TTTR file (and will make this for every consecutive file that exceeds this limit).**
2. **public void WriteTTTRDataToFile(**UInt32[] **\_\_TTTRBuffer**, int **\_\_indexLowerTTTRBuffer**, int **\_\_indexUpperTTTRBuffer**, string **\_\_sOutputFile) - this method takes the current available segment from the TTTR buffer and writes the records it contains to the TTTR file. If the number of records in the current TTTR file exceed the value defined by MAXIMUM\_TTTR\_RECORDS the method starts writing the next portion of records in a new TTTR file.**
3. **private void** **WriteTTTRBufferToFile(**BinaryWriter **\_\_bwOutputFile**, ref int **\_\_iRecordsCount**, int **\_\_indexLowerTTTRBuffer**, int **\_\_indexUpperTTTRBuffer**, UInt32[] **\_\_TTTRBuffer)** - the actual writing of TTTR records takes place here. It method takes the **\_\_TTTRBuffer and writes the segment between the indexes \_\_indexLowerTTTRBuffer and \_\_indexUpperTTTRBuffer. After the end of writing it updates the number of records \_\_iRecordsCount written to file so far, in order to keep tracking if we have not exceed the upper limit MAXIMUM\_TTTR\_RECORDS (so that on the next write it will continue writing the records to a new file). \_\_iRecordsCount is also later used to update the TTTRFileHeader.TTTRHeader.NumberOfRecords filed in the TTTR file header.**
4. **private char[] StringToCharArray(**string **\_\_InputString**, int **\_\_iCharLength**) - this function converts an input string **\_\_InputString** to a char array with length **\_\_iCharLength. If the length of the string is smaller than \_\_iCharLength the function fills it with '\0' character. If the length of the string is bigger than \_\_iCharLength the function cuts the string to the given \_\_iCharLength length. All this is necessary in order to write the strings to the TTTR file header correctly (with the correct size). See the method WriteTTTRHeaderToFile() to get an idea for the usage of this function.**
5. **public void** **WriteTTTRHeaderToFile(**BinaryWriter **\_\_bwOutputFile) - a method, the actual writing of the TTTR header to the TTTR binary file takes place here. One thing to note here is that there are five reserved fields in the TTTRHeader sub-structure (not used by Time Harp software) that we use to store information about few important parameters of the scan (in this way we can later extract the corresponding images from the TTTR file). The reserved fields and their corresponding parameters type are:**
   1. **TTTRHeader.Reserved1TimePPixel** - stores the time per pixel (actually at the moment not important and not used).
   2. **TTTRHeader.Reserved2XWidth** - stores the width of an image in pixels.
   3. **TTTRHeader.Reserved3YHeight** - stores the number of lines per image.
   4. **TTTRHeader.Reserved4GatingTimeCh** - stores the gating time in terms of channel number (note that the channel width is defined by the time resolution of Time Harp).
   5. **TTTRHeader.Reserved5BidirectionalScan** - stores the type of scan: uni-directional (= 0) or bi-directional (= 1).
6. **public void WriteTTTRNumberOfRecordsToFile(**BinaryWriter **\_\_bwOutputFile**, int **\_\_iFileOffsetForNumberRecords**, int **\_\_iRecordsCount) - a method that updates the number of records (i.e. NumberOfRecords filed in the TTTR file header) written in the TTTR binary file. The current number of records is given by \_\_iRecordsCount. \_\_iFileOffsetForNumberRecords is the argument that gives the file offset in bytes where the filed NumberOfRecords starts.**
7. **public static void ReadTTTRHeaderFromFile(**string **\_\_sInputFile**, ref TimeHarp200DLL.StructTTTRFileHeader **\_\_TTTRFileHeader) - a method that reads the TTTR header file. The result goes in the supplied \_\_TTTRFileHeader structure.**
8. **public static void** **ConvertTTTRFileToASCIIFile(**string **\_\_sInputFile**, string **\_\_sOutputFile) - a method, it converts a TTTR binary file to a respective readable ASCII file. \_\_sInputFile gives the file path to the TTTR binary file and \_\_sOutputFile gives the file name and path for ASCII file. Note that the respective ASCII file is much bigger (approx. 15x bigger) than the size of its binary counterpart.**
9. **public static void ConvertTTTRFileToBitmapImages(**string **\_\_sInputFile**, string **\_\_sOutputFile) - a method, it converts a binary TTTR file into image(s). It reads the supplied TTTR \_\_sInputFile and if it finds any image(s) it saves it to hard drive as a bitmap file(s). The output file name and path is given by \_\_sOutputFile. Note that to every extracted image the method adds a number to indicate the sequence in which it extracted the images (frame sequence) from the TTTR file. The method reads and extract images in chunks of** TimeHarp200DLL.**DMABLOCKSZ size. Note that the method extracts the images as the same settings used in the original scanning measurement. It means before the whole extraction process it reads TTTRHeader.Reserved2XWidth**, **TTTRHeader.Reserved3YHeight**, **TTTRHeader.Reserved4GatingTimeCh**, **TTTRHeader.Reserved5BidirectionalScan** parametersfrom the TTTR file header and use them to correctly extract the image(s).

These were the all functions within this subsection of the class. Now let's consider the next section, which is the most important one, the core of the class.

## III.6. Special functions for TTTR buffer handling and TTTR to pixels extraction of PQTimeHarp200.cs ****class****

This subsection is dedicated of functions related to reading TTTR data from Time Harp buffer, saving this data to file and eventually extracting images on-the-fly from the TTTR buffer data. Basically you will be using these functions to interact with Time Harp and do various high level operations.

The following important methods are available here:

1. **public void SetupAPDCountAndTiming(**bool **\_\_bBidirectionalScan**, double **\_\_dGatingTimeMillisec**, int **\_\_iXWidth**, int **\_\_iYHeight**, int **\_\_iMeasurementMode**, int **\_\_iAcquisitionTime**, bool **\_\_bSaveTTTRData**, string **\_\_sTTTRFile)** - a method (very important), by passing the indicated arguments we prepare for the acquisition process. You are suppose to call this method before starting any acquisition. The most important things that happen within the body of the method are:
   1. Set the scan type, i.e. if it is unidirectional (**\_\_bBidirectionalScan = false**) or bidirectional (**\_\_bBidirectionalScan = true**). This is important for the pixel extraction process so that it is done correctly.
   2. Get the latest value of the Sync rate and calculate the respective channel (ScanStatus.**GatingTimeChannel**) that corresponds to the gating time (gating time in terms of channel width is more convenient to work with). Note also that the gating time **\_\_dGatingTimeMillisec** must be passed to the method in milliseconds.
   3. Set the scan frame dimensions, i.e. set the width **\_\_iXWidth** and the **\_\_iYHeight** height of the scanning frame.
   4. Set measurement mode **\_\_iMeasurementMode**. It must be **\_\_iMeasurementMode = 0**, which will set the TTTR mode (necessary for imaging and if you want to have the whole data one can extract from the photons' arrival time).
   5. Set the acquisition time **\_\_iAcquisitionTime** - this is the time in milliseconds that the measurement will last. If you are not sure for how long the measurement must run or you want to control when the measurement will stop, set this parameter to either **\_\_iAcquisitionTime = 0** or **\_\_iAcquisitionTime =** TimeHarp200DLL.**ACQTMAX**. Both have the same effect, namely setting the acquisition time to its maximum value(= 36000000 ms = 10 hours). Use the one, which is more convenient to you. So if set the acquisition time to the maximum supported by Time Harp time, you can stop the measurement be calling **RequestMeasurementStop()** method.
   6. Indicate if you want to save or not the raw TTTR data to a TTTR binary file (**\_\_bSaveTTTRData = true** will save to hard drive the TTTR data, and **\_\_bSaveTTTRData = false**, will not save it). This gives you the flexibility to avoid unnecessary CPU load and avoid wasting hard drive space, especially when you do not need such data (e.g. when you align the system or just want to check the sample).
   7. The parameter **\_\_sTTTRFile** allows you to indicate the file name and path of the TTTR data file.
   8. Another important thing that happens here is the allocation of memory for the frame pixel buffer ScanStatus.**FramePixelBuffer**, i.e. the image in terms of one dimensional array of integers. Also we allocate here memory for the line pixel buffer ScanStatus.**LinePixelBuffer** - it is used in the extraction of pixels from the global TTTR buffer. This line pixel buffer, as the name indicates, represent the pixels of a line from the scanning frame.
2. **public void** **StartAPDAcquisition() - a method, by calling this method you will start the acquisition process. This method must be called only if the method SetupAPDCountAndTiming() The method create and run two threads, and returns before completion (it will not block the calling thread). Three important things happen within the body of the method:**
   1. Instructs Time Harp board to start acquisition by calling **StartMeasurement()** method.
   2. Spawn a separate reading thread (the **Read()** method) that reads the Time Harp TTTR buffer and copy its buffer data to the global TTTR buffer (for the time being 100x bigger than Time Harp's buffer).
   3. Spawn another thread (the **Save()** method) that regularly checks the global TTTR buffer for new TTTR data and if it finds such it saves it to the hard drive, to a binary TTTR file (compatible with Picoquant's format of this type of file). However, note that this thread will be started only if you chose to set as true the **\_\_bSaveTTTRData** argument of **SetupAPDCountAndTiming() method. This gives you the flexibility to control the CPU resources etc.**
3. **private void** **StopAPDAcquisition() - a method, it stops the Time Harp acquisition process. This method is only used internally within the class, cannot be called from outside the class. If you want to stop a current ongoing acquisition process call RequestMeasurmentStop().**
4. **public void** **RunAPDAcquisition() - a method, this method does the same as StartAPDAcquisition() but the only difference being that it waits for the Read() and Save() threads to complete before exiting. It means it will block the calling thread until those two treads are running. The main purpose of this method is to use it for pure acquisition and saving process, no other processing. So use this method only if you want to run a measurement that gathers TTTR data and saves it to file, nothing more than this (or if you want a blocking version of StartAPDAcquisition() method).**
5. **private void Read() - a method, this is one of the most important methods of the class. It is used only internally by the class. It reads the Time Harp TTTR buffer and copy its data to the global TTTR buffer (called ScanStatus.GlobalTTTRBuffer[]). Thus any processing of raw TTTR data must be done through accessing, i.e. reading this global buffer. The Read() method is the only method that writes to the global buffer, other threads are suppose only to read from it (otherwise some unpredictable result will occur). It is run in a separate thread so that it allows to run cooperatively/parallel with other threads. Few things may be noted for the internal logic that this Read() method implements:**
   1. **It bounds the Read() thread to a particular CPU core by setting the thread affinity (using the** Thread.BeginThreadAffinity() in the beginning of the **Read()** method body and Thread.EndThreadAffinity() at its end**) - the hope is that it may improve the DMA transfer and buffer handling (by not allowing the host to move the thread across the CPU) but so far I cannot prove it will.**
   2. **It uses two buffers (\_ui32TTTRBuffer1 and \_ui32TTTRBuffer2) that are as big as the Time Harp half FiFo buffer size in order to efficiently utilize the CPU resources. For example, while one buffer gets TTTR data from Time Harp the other one is processed - its contain is copied to the global TTTR buffer.**
   3. **The method runs until either the measurement time is over (acquisition time exceeded \_bMeasurementRunning = false) or a stop measurement request was issued (by RequestMeasurementStop() method, which sets** this.**m\_IsMesurementToBeStop = true).**
   4. **The method handles also the FiFo overrun cases by restarting the measurement as soon as it happens. The restarting itself takes not more than 30-100 µs. In this way the acquisition can continue without much disruption. FiFo overrun may happen at very high count rates when some other program makes an intensive use of the CPU (this may slow down Read() thread and delay enough the buffer acquisition to experience FiFo overrun). Basically at maximum count rate of 3Mcps the FiFo gets half full for 20 ms and the FiFo full 40 ms. So if the DMA transfer is not fast enough (for the time being it takes 16-24 ms to transfer the half buffer) to keep in pace with the count rate an overrun will ocur.**
   5. **The method sets a limit to the time FiFo gets half full - the time out for the buffer to get half full is set by the local variable \_iFIFO\_TIME\_OUT = 2000** (in milliseconds). After this time is reached a DMA will be forced but in this way the transfer takes 100 ms (on the current old PC). However, this allows to get and process the buffer at low count rates.
   6. In order to efficiently utilize the CPU resources, at every loop cycle we sleep the thread for 1 ms. Thus the method checks if the FiFo buffer is half full or full or time out on every 1 ms. This is done due to the fact that the time to get FiFo half full is 20ms it is not necessary to check the FiFo status very often.
   7. **One important thing that Read() method performs before writing the FiFo TTTR data to the global TTTR buffer is to check (by function CheckGlobalTTTROverwrite()) whether or not the global buffer where the current data must be placed is still in a processing by either Save() thread or BuildImage() thread. If this is the case an exception will be thrown (this will stop the further execution of the program). Never experience so far such event, which means the old PC still can cope with the processing. If it happens increasing the global buffer size may help.**
6. **private void** **Save()** - a method, **this is one of the most important methods of the class. It is used only internally by the class. It checks the global TTTR buffer ScanStatus.GlobalTTTRBuffer[] for new TTTR records (pushed into this buffer from Read() thread) and if present reads and saves them to the hard drive. Few things should be noted about the work and the implementation logic of this method:**
   1. **The method runs until either the measurement time is over (acquisition time exceeded \_bMeasurementRunning = false) or a stop measurement request was issued (by RequestMeasurementStop() method, which sets** this.**m\_IsMesurementToBeStop = true).**
   2. In order to efficiently utilize the CPU resources, at every loop cycle we sleep the thread for 5 ms (controlled by the local variable **\_iTHREAD\_SLEEP\_TIME = 5**). Thus the method checks if the TTTR records are available in the global TTTR buffer every 5 ms. This is done due to the fact that the time to get FiFo half full is 20ms it is not necessary to check for records very often. This reduces the CPU load a lot and helps to decrease the rate of FiFo overrun in **Read()** thread (by giving it more CPU time). One note here: the performance analysis showed that saving to file is fast (~ few milliseconds per chunk data) but due to CPU loads etc there are slow downs (on the order of hundreds of milliseconds). However, due to the big global TTTR buffer size this slowdowns can be compensated easily, thus providing smooth work of the underlying software.
   3. **The saving itself is carried out by WriteTTTRDataToFile()** method, which **Save()** thread calls for this purpose. It saves the detected available chunk of TTTR data from the global TTTR buffer. It takes care of the entire saving procedure, to properly save the data in the respective binary format (according the Picoquant's specifications). Note also that it handles the case when the binary TTTR file becomes too big (bigger than Files.**MAXIMUM\_TTTR\_RECORDS** implies) it create a new TTTR file and continue to save the data there (and so on for every such case).
   4. When the measurement is done or stopped the **Save()** and **Read()** threads will terminate. However, because In order not to lose the last chunk of records the **Save()** thread is implemented such that it waits first for the **Read()** thread to terminate, then checks if there is still data in the global TTTR buffer, if yes, saves and then terminates. Thus all data that we could fetch from Time Harp's FiFo buffer will be stored in the binary TTTR file.
7. **public int[][]** **BuildImage()** - a method, **this is one of the most important methods of the class. It must be called from outside the class as soon as you start the acquisition process. It checks regularly the global TTTR buffer ScanStatus.GlobalTTTRBuffer[] for new TTTR records (pushed into this buffer from Read() thread) and if present extracts pixels from the photon events. The extracted pixels go into the global frame pixel buffer** ScanStatus.**FramePixelBuffer[]** **(one dimensional array of integers with size defined by the number of pixels in the image). Few things should be noted about the work and the implementation logic of this method:**
   1. **The returned value of this method is an int[][] array that points to the frame pixel buffer** ScanStatus.**FramePixelBuffer[]**. One can also access directly the latter buffer (if needed for some reason) but in general you do not need to do it. Just **read the returned buffer and it will give you the same result.**
   2. **Note that the returned by this method value \_i32FramePixelBuffer[][] is of type int[][], i.e. array of arrays (the idea was to return array of images found in the TTTR buffer). However, at certain point it turned out that it is enough to use a single one dimensional array (which represent just one frame) to do the same job. Therefore the returned frame pixel buffer is located in the first cell of int[][], i.e. in \_i32FramePixelBuffer[0][...]. I keep the buffer as the indicated type of array of arrays in case in the future one uses it.**
   3. **The method runs until:**
      1. **The measurement time is over (acquisition time exceeded \_bMeasurementRunning = false).**
      2. **Stop measurement request was issued (by RequestMeasurementStop() method, which sets** this.**m\_IsMesurementToBeStop = true).**
      3. **There is frame ready in the frame pixel buffer (\_bFrameReady = true).**
      4. **The frame time out exceeded (\_bFrameTimeOut = true). Then it returns whatever it has as pixels into the frame pixel buffer. The value of the time out is controlled by** ScanStatus.**FrameTimeOut** parameter (currently 500 ms).
   4. Just note: because of the functionality in 7.3.3 and 7.3.4 you must use this method in a loop. Thus at every loop cycle you will either get the ready image (if **\_bFrameReady = true**, i.e. all pixels for the current frame processed) or if the frame time out get exceeded **(i.e. if \_bFrameTimeOut = true)** you will get the current state of the scanning progress in terms of extracted so far pixels in the frame pixel buffer. If the frame was not ready on the next loops **BuildImage()** will continue the pixel extraction process until it complete a frame. Thus if frame get completed on the next execution of **BuildImage()** the frame buffer will be cleared. So if you want to save image you need to copy the content of the frame pixel buffer before it gets cleared.
   5. In order to efficiently utilize the CPU resources, at every loop cycle we sleep the thread for 20 ms (controlled by the local variable **\_iTHREAD\_SLEEP\_TIME = 20**). Thus the method checks if the TTTR records are available in the global TTTR buffer every 20 ms. This is done due to the fact that the time to get FiFo half full is 20ms it is not necessary to check for records very often. This reduces the CPU load a lot and helps to decrease the rate of FiFo overrun in **Read()** thread (by giving it more CPU time). One note here: the performance analysis showed that extracting of pixels from raw TTTR records is extremely fast process (not more than several milliseconds at worst).
8. **public static void** **ExtractPhotonEventsFromTTTRBuffer**(UInt32[] **\_\_ui32TTTRBuffer**, ref int **\_\_indexTTTRBufferLowerBound**, int **\_\_indexTTTRBufferUpperBound**, ref bool **\_bFrameReady**, int[] **\_\_i32FramePixelBuffer**) - a method, takes the available chunk of TTTR records from the global TTTR buffer (here passed as **\_\_ui32TTTRBuffer** argument) and extract pixels from the photon events. The extracted pixels go into the frame pixel buffer (here passed as **\_\_i32FramePixelBuffer** argument). The method is implemented following the following logic:
   1. **Get from the passed arguments of the method the chunk of TTTR data to be processed - the raw TTTR records comes from \_\_ui32TTTRBuffer[]**. The indexes **\_\_indexTTTRBufferLowerBound** and **\_\_indexTTTRBufferUpperBound** define the range of array cells to be processed (searched for photon events). **\_bFrameReady** will return true to indicate that all pixels within the current frame have been processed. The binned photons go as pixels in the frame pixel buffer **\_\_i32FramePixelBuffer[]**.
   2. Get the current state of the scanning parameters into the local variables:
      1. int \_iXWidth = ScanStatus.XWidth - the width of the frame in pixels.
      2. int \_iYHeight = ScanStatus.YHeight - the number of lines per frame.
      3. int \_iPixelCount = ScanStatus.PixelCount - the number of pixels by frame.
      4. int \_iPixelCounter = ScanStatus.PixelsCounter - the number of currently processed pixels (keeps track of the processed pixels so far).
      5. int \_iOverflow = ScanStatus.Overflow - keeps track of the overflow markers fromm the TTTR records.
      6. int[] \_i32LinePixelBuffer = ScanStatus.LinePixelBuffer - line pixel buffer, used in the pixel extraction process (we extract every frame on line by line bases). For proper extraction we need to preserve the state of this buffer, therefore its defined as a global variable.
   3. **Start the extraction process by the while-loop, which analysis the TTTR records for photon events.**
   4. **Before analyzing each TTTR record we convert it to a higher level representation (which is much easier to work with). This is done by the function ConvertSingleTTTRtoTrueTimeTagPTTTR(). This representation is a structure called \_PTTTRecord, abbreviation of Processed TTTR Record (more about this in the next section).**
   5. **The first thing that we must do in order to extract a frame from TTTR data is to find a reference frame marker.**
   6. **Second if we have found a reference frame marker we must found the next line marker. This gives us enough information to calculate the pixel time. Then the binning of photons into pixels can start. While searching for a next line marker we add the found photon events to a global Line Processed TTTR buffer (see ScanStatus.LinePTTTRBuffer array). Thus when we have found two reference line markers we can easily bin there photons using this global line PTTTR buffer. We need such a global buffer because we cannot predict when the second line marker will come (it may be not in the current data chunk). So we need to preserve state in this way. Note that in order not to miss photon events we dedicate a big enough size for ScanStatus.LinePTTTRBuffer[] (currently 100x the TimeHarp200DLL.DMABLOCKSZ**, **which corresponds roughly to a capacity of 6 million photons per frame line). However, this gives a considerable memory footprint of approx. 84MB (in each array cell we need to store a structure as big as 14 bytes).**
   7. **Once we have found two consecutive line markers and have all photon events between them into the global ScanStatus.LinePTTTRBuffer[] we can calculate the pixel time for the current frame line and then bin the photons corresponding to this line. The binning of photons and their transfer to the frame pixel buffer is done by PhotonEventsToPixelBuffer()** method. This method also takes care to properly process uni- or bi-directional scans.
   8. One thing to note is that always when a frame has been found (**\_bFrameReady = true**) the analysis of the TTTR records stops, then we preserve the current state of processing, we exit the while-loop and return the frame that is in the pixel frame buffer. In this case we know the frame pixel buffer will reach the software, be further stored or analyzed and on the next call of **BuildImage()** function we clear the frame pixel buffer and continue extraction from the last known state.
   9. The method also handle cases when there were missing line markers, due to for example FiFo overrun. This means that if we are still processing a frame and we suddenly find the next reference frame maker the method completes the unfinished frame (indicating **\_FrameReady = true**). The frame pixel buffer is then returned and afterwards we start to process the new frame. So if you see dark lines during scan it means such event occurred (it happens when there was buffer overrun, which may cause loss of reference line markers).
   10. One thing to note is that always when a frame has been found (**\_bFrameReady = true**) the analysis of the TTTR records stops, then we preserve the current state of processing, we exit the while-loop and return the frame that is in the pixel frame buffer. In this case we know the frame pixel buffer will reach the software, be further stored or analyzed and on the next call of **BuildImage()** function we clear the frame pixel buffer and continue extraction from the last known state.
   11. **Note also that you have to options to define a frame in terms of frame and line markers. The beginning of frame is always required to be a frame marker (it is also counted as line maker). Then you can complete a frame by:**
       1. **Another line marker - e.g. if you have 512x512 pixels image, you have to insert 513 line markers to complete a frame (the starting frame marker is counted as line marker). In this way the next starting reference frame marker will be the 514 marker.**
       2. **A frame marker - e.g. if you have 512x512 pixels image, you have to insert 513 marker as a frame marker to complete the frame (the starting frame marker is counted as line marker). In this way the next starting reference frame marker will be the 513 marker, i.e. the frame marker will play two roles - to complete the previous frame and to start the new frame.**
       3. **Note for the upper two options: I found particularly easier (from the point of view of writing a YanusIV protocol for the scan) and more flexible to use the first option. Then one can start the next frame with a frame marker at later times if needed.**
9. **public static void** **SaveFramePixelBufferAsBitmapImage**(string **\_\_sOutputFile**, int **\_\_iXWidth**, int **\_\_iYHeight** , int[] **\_\_i32FramePixelBuffer**, int **\_\_iFrameCounter**) - a method, it saves to disk as a bitmap image the given frame pixel buffer **\_\_i32FramePixelBuffer[]**. This method is used in the method **ConvertTTTRFileToBitmapImages()** to extract images form a TTTR binary file and save them to file.

These were the all functions within this subsection of the class. Now let's consider the next section, which is also an important one.

## III.7. Special functions for Scanning Status and various data processing routines of PQTimeHarp200.cs ****class****

This subsection contains number of different functions and structures that facilitate the processing of TTTR data from Time Harp buffer. Many of this parameters get initialized and assigned in the **SetupAPDCountAndTiming()** method.

Some of the most important structures and methods are the following:

1. **public struct ScanStatus** - a structure that contains various parameters related to the scanning process. It also contains few of the important global buffers. Here is the short list of the important ones:
   1. **public static byte** **FrameMarker** - the decimal value for the frame marker. If the inserted in the data stream of Time Harp markers have this value they will be recognized as frame makers from the methods that rely on this information. Currently the **FrameMarker = 2**. If you want you can change it at runtime because this structure is publically accessible but then you must change it before any processing strats.
   2. **public static byte** **LineMarker** - the decimal value for the line marker. If the inserted in the data stream of Time Harp markers have this value they will be recognized as line makers from the methods that rely on this information. Currently the **LineMarker = 4**. If you want you can change it at runtime because this structure is publically accessible but then you must change it before any processing strats.
   3. **public static int TimePPixelChannel** - the time per pixel in terms of channel number (the width of the channel is always the same and equals 100 ns).
   4. **public static int PixelCount** - the number of pixels within a frame.
   5. **public static int PixelCounter** - the number of currently processed pixels per current frame.
   6. **public static int XWidth** - the number of pixels per line.
   7. **public static int YHeight** - the number of lines per frame.
   8. **public static int SyncTimeChannel** - the period between two sync pulses in terms of channel number (the channel number is defined in the same way as the time tag for the start-stop time, i.e. it is a the ratio of the time of the sync period over the used time resolution of Time Harp). It is parameter that's needed for the gating algorithm.
   9. **public static int GatingTimeChannel** - the gating time in terms of channel number (the channel number is defined in the same way as the time tag for the start-stop time, i.e. it is a the ratio of the gating time over the used time resolution of Time Harp). It is parameter that's needed for the gating algorithm.
   10. **public static ushort GatingTimeReverseChannel -** the gating time in terms of reverse channel number, i.e. **SyncTimeChannel -** **GatingTimeChannel**. It is parameter that's needed for the gating algorithm. This is parameter that's basically used to filter photons according the given gating time (I am not sure if I need to compare the start-stop time tag with the reverse gating channel or just with the gating channel, this needs to be tested).
   11. **public static bool IsBidirectionalScan - bidirectional or unidirectional scans. By default IsBidirectionalScan = false (i.e. unidirectional scan).**
   12. **public static int Overflow** - the overflow flag. It keeps track of the number of times an overflow flag occurred in the Time Harp 200 data stream.
   13. **public static int** **FrameTimeOut** - default frame time out in [ms] after which all pixels in a frame will be processed and shown. Used in the **BuildImage()** method. This will cause that **BuildImage()** will return the frame pixel buffer before completion of a frame so that the scanning progress can be seen. The default value is **FrameTimeOut = 500** [ms].
   14. **public static int**[] **FramePixelBuffer** - the global frame pixel buffer that store the processed pixels per given frame.
   15. **public static int**[] **LinePixelBuffer** - the global line pixel buffer that store the processed pixels per given line (which is part from a frame).
   16. **public static volatile** UInt32[] **GlobalTTTRBuffer** - the global TTTR buffer, which is a pool for the Time Harp 200 FiFo buffers. The size is 100x the TimeHarp200DLL.**DMABLOCKSZ**. Note that it needs to be volatile so that every thread gets the most up to date value.
   17. **public static volatile int** **SizeGlobalTTTRBuffer** - this is not exactly the size of **GlobalTTTRBuffer[]** (in terms of the length of the buffer) but the current size up to which the buffer is filled with meaningful TTTR values. Note that it needs to be volatile so that every thread gets the most up to date value.
   18. **public static** ProcessedTTTRrecord[] **LinePTTTRBuffer** - the line Processed TTTR buffer. The size is 100x the TimeHarp200DLL.**DMABLOCKSZ** (100x the size of the Time Harp Half FiFo buffer). This is the global PTTTR buffer (processed TTTR records) in which we store the found photons until a line is completed. Then the photons can be easily binned into pixels.
   19. **public static int LinePTTTRBufferIndex** - the index of the last available processed TTTR record in the given buffer (it must not exceed the length of the buffer, if it happens it wraps to the beginning of the buffer).
   20. **public static long** **LinePTTTRBufferTimeTag1** - the time tag of the line marker 1, it marks the beginning of a line.
   21. **public static long** **LinePTTTRBufferTimeTag2** - the time tag of the line marker 2, it marks the end of a line. Thus **LinePTTTRBufferTimeTag2 - LinePTTTRBufferTimeTag1** will give the duration of single line from the current frame. Then knowing the number of pixels per line ScanStatus.**XWidth**, one can calculate the time per pixel ScanStatus.**TimePPixelChannel**. Then the binning of photons within the current frame line can be done. Note that such calculation of pixel time is done for every line.
2. **public struct** **ThreadsStatus** - a structure that keeps track of some of the threads' related data. It is used to interact with the created threads. There are three sub-structures here that are related to three different threads:
   1. **public struct** **APDReadThread** - this structure keeps track of the thread that spawns **Read()** method (that fetches the Time Harp 200 FiFo buffer).
   2. **public struct** **APDSaveThread** - this structure keeps track of the thread that spawns **Save()** method (that saves chunks of TTTR data from the global TTTR buffer to a binary TTTR file).
   3. **public struct** **BuildImageThread** - this structure keeps track of the main thread that executes **BuildImage()** method (which extracts pixels from the global TTTR buffer).
3. **private struct** **ExternalMarkers** - a structure that helps to work with the external markers. The information is later use to extract pixels from the TTTR buffer. There are two sub-structures here :
   1. **public struct** **FrameMarker** - this structure is dedicated to the frame maker. It stores the index and time tag of a frame in the currently processed TTTR buffer. The parameter that is basically in use from this structure is
      1. **public static bool** **Found** - indicates if we have found (and currently processing) a frame (i.e. frame marker).
   2. **public struct** **LineMarker** - this structure is dedicated to the line maker. It stores the index and time tag of a line in the currently processed TTTR buffer. The parameter that is basically in use from this structure is
      1. **public static bool** **Found** - indicates if we have found a line marker.
4. **private struct** **TTTRMasks** - structure that uses hexadecimal masks to help conversion of a TTTR record to a higher level of representation (e.g. PTTTR record with clearly defined data fields).
5. **private struct** **ProcessedTTTRecord** - structure that is a higher level of representation of a TTTR record (a TTTR record is just unsigned integer). It is more convenient to work with it. The structure's data fields are:
   1. **public long TimeTag** - the time tag of photon arrival in terms of channel number. Then the true time tag is TrueTimeTag[s] = TimeTag \* 100[ns] \* 1e-9.
   2. **public ushort** **Channel** - channel (reversed start-stop time).
   3. **public byte** **Route** - router channel (max. 4 channels).
   4. **public byte Valid** - valid flag, Valid = 0 (means overflow or external trigger event), Valid = 1 (means photon arrival event).
   5. **public byte DataOverflow** - overflow flag, Overflow = 0 (means no overflow), Overflow = 1 (means overflow).
   6. **public byte DataMarker** - markers flag, DataMarker[3bits]. Each combination of bits represents the markers' state, e.g. **DataMarker = 101** -> DataMarker1 = 1, DataMarker2 = 0, DataMarker3 = 1.
6. **public static void ConvertSingleTTTRtoPTTTR**(UInt32 **\_\_ui32TTTRValue**, ref ProcessedTTTRecord **\_\_PTTTRRecord**) - a method, converts a single raw TTTR record from the buffer to a Processed TTTR (PTTTR), as it populates the necessary data fields - this information can be then used in any kind of data processing.
7. **public static void** **ConvertSingleTTTRtoTrueTimeTagPTTTR**(UInt32 **\_\_ui32TTTRValue**, ref int **\_\_iOverflow**, ref ProcessedTTTRecord **\_\_PTTTRRecord**) - a method, converts a single raw TTTR record from the buffer to a Processed TTTR (PTTTR), as it populates the necessary data fields - this information can be then used in any kind of data processing. The difference compare the precious method is that it takes into account the overflow markers so that it actually calculate the true time tag of the event. Note that this method calls internally **ConvertSingleTTTRtoPTTTR()**.
8. **public static void PhotonEventsToPixelBuffer**(int **\_\_iTimePPixel**, int **\_\_iXWidth**, int **\_\_iPixelCount**, ref int **\_\_iPixelCounter**, ref bool **\_\_bFrameReady**, int[] **\_\_i32LinePixelBuffer**, int[] **\_\_i32FramePixelBuffer**) - a method, the actual binning of photons happens here. It stores the pixels with the binned photons in the frame pixel buffer **\_\_i32FramePixelBuffer[]** (which refer the global frame pixel buffer ScanStatus.**FramePixelBuffer[]**). We keep track of the curently processed pixels due to **\_\_iPixelCounter** parameter (every modification of this parameter will be reflected back to the calling function). Three major things happen within this method:
   1. Binning of photons - the raw photon events from ScanStatus.**LinePTTTRBuffer[]** buffer, which encloses photon events between two line markers, get binned into pixel. Those pixels are stored temporary in the line pixel buffer **\_\_i32LinePixelBuffer[]** (which refer the global line pixel buffer ScanStatus.**LinePixelBuffer[]**). While binning we also perform gating of the photons.
   2. Transfer the pixels from the line pixel buffer **\_\_i32LinePixelBuffer[]** to the frame pixel buffer **\_\_i32FramePixelBuffer[]**. Depending whether you indicated the scan as bidirectional or not it chooses the appropriate algorithm to transfer the pixels the frame pixel buffer. Note that if the scan was bidirectional we flip every second line (which have odd index) to get the pixels in the right order.
   3. It checks if the frame is ready, and if so it resets few parameters in order to prepare for the next frame.

## III.8. Constructor of PQTimeHarp200.cs ****class****

This is a description of the class constructor of PQTimeHarp200 class, and shows how to use it:

1. **public PQTimeHarp200**(int **\_\_iMeasurementMode**, int **\_\_iCFDDiscrMin**, int **\_\_iCFDZeroCross**, int **\_\_iSyncLevel**, int **\_\_iRangeCode**, int **\_\_iOffset**, int **\_\_iMarkerEdge**) - this is the class constructor. Needs to be called in the beginning of the session when we want to interact with Time Harp. It sets up few hardware parameters necessary for the proper operation of Time Harp. In order to understand what it makes let's see what happens within the body of the method:
   1. First it gets THLib library version (by calling **GetLibraryVersion()**) and checks if this is the supported version (if not it rises an exception that will stop the program).
   2. It makes a calibration of Time Harp (whatever it means, ask Picoquant for what exactly it makes) by calling **Calibrate()** method. This is a mandatory thing before any acquisition, so we do it as well.
   3. It gets the serial number and the hardware version.
   4. Next it sets the CFD voltage level in [mV] (by calling **SetCFDDiscrMin(\_\_iCFDDiscrMin)**) according **\_\_iCFDDiscrMin** value that you passed as argument to the constructor. A good value according current tests is 50 mV.
   5. Next it sets the CFD zero cross level voltage in [mV] (by calling **SetCFDZeroCross(\_\_iCFDZeroCross)**) according **\_\_iCFDZeroCross** value that you passed as argument to the constructor. A good value according current tests is 20 mV.
   6. Next it sets the Sync voltage level in [mV] (by calling **SetSyncLevel(\_\_iSyncLevel)**) according **\_\_ iSyncLevel** value that you passed as argument to the constructor. A good value according current tests is -50 mV.
   7. Next it sets the so called Range code (possible values from 0..5), which determines the time resolution of Time Harp. It you set it to 0, it will use the basic resolution of Time Harp. Basically the resulting time resolution is defined by **basic\_resolution x 2\_\_iRangeCode**. The value for range code can be passed as argument to the constructor. The value must be such that when multiplied by the maximum channels, i.e. 4096 (the time resolution is the channel width), you can cover a full sync period. If you do not do this you may lose photons that would have arriving time, which could not be mapped from the current channel width and maximum channel number. Setting Range code is done by calling **SetRange(\_\_iRangeCode)** method. For Sync rates that are bigger or equal to 10MHz you can safely set **\_\_iRangeCode = 0**.
   8. Next it sets the Offset value **\_\_iOffset**, defines the start-stop time offset for attributing a channel for the arriving photon (used in order to compensate for delays in the cables). The value is approximation of the desired offset (in steps of 2.5ns). So far just set it to **\_\_iOffset = 0**.
   9. Next it sets the active TTL edge for external markers by calling **T3RSetMarkerEdges()**. Depending of the value you passed to the constructor method it will set either **\_\_iMarkerEdge = 0** (the active TTL edge is the falling edge) or **\_\_iMarkerEdge = 1** (the active TTL edge is the rising edge). I would recommend to set **\_\_iMarkerEdge = 1**, i.e. using the rising edge to create a trigger for the external markers (so that Time Harp registers the external marker as soon as it appears). In general one can set different edges for each of the three possible markers, but I do not see any advantage of this and just keep in the implementation only one type of edge at a time.
   10. Next it gets the time resolution **m\_fResolution = GetReolution()** of Time Harp (so that this will be the channel width of the start-stop photon arriving time). This is necessary for the gating or whatever processing that relays on the precise arrival time of the photons.
   11. Next we initialize (by calling **Initialize(\_\_iMeasurementMode)**) Time Harp with the given measurement mode **\_\_iMeasurementMode** (again it must be passed as argument to the constructor). Use value of **\_\_iMeasurementMode = 0** for the TTTR mode (basically this is what we need).
   12. The last thing that happens is allocate memory for some of the TTTR file headers fields (which are part of a structure that represent the TTTR file header). Necessary because C# does not allocate memory for such compound marshaling data types. Note that we allocate the following fields:
       1. Files.**TTTRFileHeader**.**BinaryHeader**.**DisplayCurves** - an array of structures.
       2. Files.**TTTRFileHeader**.**BinaryHeader**.**Params** - an array of structures.
       3. Note: the above two are not used by our software but I keep it in order to be to Time Harp TTTR specs as close as possible. You can safely ignore these parameters because the methods take care to work properly with them.

These was the description of the class constructor. Now let's see example of how to use the class for imaging.

# IV. Example how to use PQTimeHarp200.cs class for scanning

In order to show how to use the class in scanning applications (extract images from the TTTR binary data on-the-fly) we will use the small console test app - see **PQTimeHarp200App.cs** class in the root folder **PQTimeHarp200App**.

So let's just describe the most important points present in this example:

1. First we declare and allocate a variable **m\_apdAPD1** of type the a class of **PQTimeHarp200**:

// Initialize Time Harp 200 - set measurement mode and other hardware settings

PQTimeHarp200 m\_apdAPD1 = new PQTimeHarp200(

\_iMeausurementMode,

\_iCFDDiscrMin,

\_iCFDZeroCross,

\_iSyncLevel,

\_iRangeCode,

\_iOffset,

\_iMarkerEdge);

Assign the appropriate arguments' values as described in section **III.8.** (see **PQTimeHarp200()** constructor in there).

After initializing the class if you need you can get some information as serial number of the Time Harp board, time resolution, Sync rate, Count rate etc. (see the source code in the test app for further details).

1. Go to the method called **RunTimeHarpMode3()** - this is the method where we make the image acquisition. You will see many local variables that are needed to make the scan within the test app, but they are not necessarily needed for any other software that uses the **PQTimeHarp200** class.
2. So next important step before starting the acquisition process is to set up the acquisition and scanning parameters (by passing them to the **SetupAPDCountAndTiming()** method):

// Prepare Time Harp 200 for measurement

\_\_apdTimeHarp200\_1.SetupAPDCountAndTiming(

\_\_IsBidirectionalScan,

\_\_dGatingTimeMillisec,

\_\_iXWidth,

\_\_iYHeight,

\_\_iMeausurementMode,

\_\_iAcquisitionTime,

\_\_bSaveTTTRData,

\_\_sTTTRFile);

The method needs to know if the scan is bidirectional or not (**\_\_IsBidrectionalScan = true** indicates a bidirectional scan, otherwise unidirectional). The gating time **\_\_dGatingTimeMillisec** in milliseconds, the frame dimensions (width **\_\_iXWidth** and height **\_\_iYHeight**). The measurement mode **\_\_iMeasurementMode** (set it to 0 for the TTTR mode). The acquisition time **\_\_iAcquisitionTime** in milliseconds. Note that if you are not sure for the duration of the acquisition, set up the acquisition time to zero - this will set up the acquisition time to its maximum value, approx. 10h (then you can stop the measurement and any processing by calling **RequestMeasurementStop()** method, more for this below). Indicate if you want or not to save the binary TTTR data to a file(s), **\_\_bSaveTTTRData = true** will save the TTTR data to file, while **\_\_bSaveTTTRData = false** will not save it (useful if you want to just check something without wasting CPU time and hard drive space). The last argument, **\_\_sTTTRFile** is a string that indicates the file name and path where the binary TTTR file will be saved. For more details on this method see section **III.6.** and the description of **SetupAPDCountAndTiming()** in there.

1. Now we have setup everything so we can start the acquisition by calling **StartAPDAcquisition()** method:

// Start the APD acquisition

\_\_apdTimeHarp200\_1.StartAPDAcquisition();

This will create two threads - one will read Time Harp 200 buffer and the other will save the acquired TTTR data to file (if **\_\_bSaveTTTRData = true**). For more details on this method see section **III.6.** andthe description of **StartAPDAcquisition()** method in there.

1. Now somewhere here you must instruct the scanner (YanusIV or any other) to start scanning. Be aware to make the protocol such that it inserts the proper frame and line markers (beginning of frame and beginning of line). For discussion on the markers and how the class handles them see section **III.7.**
2. So now we have started acquisition, we have started scanning, then it is time to start the pixel extraction process. For this purposes we need a while-loop that will periodically call **BuildImage()** method of the class. It is responsible for the processing of the TTTR data and the extraction of pixels from the raw photon events:

Here I just give the most important things that the while-loop must do (in the test app it also shows the captured frame on the screen).

while (!\_bStop) //acquire images and process them

{

\_i32FrameBuffer = \_\_apdTimeHarp200\_1.BuildImage(); //start BuildImage() method and get its result

\_bMeasurementRunning = \_\_apdTimeHarp200\_1.IsMeasurementRunning; //check if measurement is running,

!!!DO SOME PROCESSING OF THE PIXEL BUFFER HERE!!!

// Check if we continue acquisition or not

if (\_bMeasurementRunning)

{

\_bStop = false; //this value will cause that we continue acquisition

}

else

{

\_bStop = true; //this value will cause that we stop acquisition

!!!YOU CAN ALSO CALL HERE RequestMeasurementStop() METHOD TO STOP ACQUISITION AND PROCESSING!!!

}

} //END while (!\_bStop) loop

Note few things here:

* We need to call the **BuildImage()** method in order to extract the pixels from the TTTR data stream.
* Since the **BuildImage()** is implemented to return when a frame is ready or frame time out is exceeded, you must call it in a loop to continuously obtain the scanned frames (or parts of scanned frames if the frame is longer than the frame time out, currently 500 ms and can be controlled by ScanStatus.**FrameTimeOut** parameter of the class).
* The result from **BuildImage()** is returned in **\_i32FrameBuffer** array. In the current implementation it must be of type int[][] and the frame (always one frame is returned in this array) is in the first cell (index 0), i.e. **\_i32FrameBuffer[0][...]** (through the second index dimension you can access the pixels themselves).
* For more details on the **BuildImage()** method look at in section **III.6.** and the description of **BuildImage()**.
* At the end of the loop check if the measurement is still running or alternatively decide if you want to continue acquisition and pixel extraction. Note that in order to completely stop the acquisition and pixels extraction you must call **RequestMeasurementStop()** method - it will stop fetching Time Harp FiFo, it will stop saving data, and it will stop extracting pixels. For more details see section **III.1.** with the description of **RequestMeasurementStop()** method.
* Note that you also can stop acquisition and processing if you stop the while-loop and exit it. Then you can call **RequestMeasurementStop()** method and it will stop any acquisition/processing if still ongoing (this how it is done in the test app).

Well, that's all you need for scanning with Time Harp. Next section is for emulation class that simulates to some extend the acquisition process of Time Harp.

# V. Emulation of THLib DLL by a special C# class

By emulating the THLib DLL functions one can simulate the Time Harp acquisition process. This allows to run and test the software relaying on Time Harp device on any PC. The advantage is obvious - you can run test easier and fix bugs faster. The simulation is done by using a already acquired binary TTTR file with known settings (image dimensions, bidirectional or not scan etc.). For example, if you know the image pattern in the TTTR file you can check if you can reproduce it (then you know your algorithms work correctly).

**There are there class files in the folder KUL.MDS.Hardware\TimeHarp200 and have the following locations relative to it:**

* **KUL.MDS.Hardware**
  + **PQTimeHarp200.cs**
  + **TimeHarp200**
    - **TimeHarp200DLL.cs**
    - **TimeHarp200DLL\_dll\_class.cs**
    - **TimeHarp200DLL\_emulation\_dll\_class.cs**

**TimeHarp200DLL.cs** is the current class - may be either the emulation class or the THLib DLL class.

**TimeHarp200DLL\_dll\_class.cs** is the THLib DLL class (to interact with Time Harp hardware).

**TimeHarp200DLL\_emulation\_dll\_class.cs** is the emulation class of THLib DLL library (you can view it as the software Time Harp).

Depending what you want to do you copy the content of **TimeHarp200DLL\_dll\_class.cs** or **TimeHarp200DLL\_emulation\_dll\_class.cs** into **TimeHarp200DLL.cs** and work with the latter (thus if you make any modifications they will not affect the original dll or emulation class). Always recompile after changing the content of **TimeHarp200DLL.cs** (so that the change can take effect).

Now let's see the code section in the emulation class where you define how the emulation will work (e.g. which TTTR file will be used):

private const string TIME\_HARP\_EMULATION\_TTTR\_FILE = "tttr\_binary\_file.t3r"; //emulation file private const int THREAD\_SLEEP\_TIME = 5; // thread sleep time in [ms] (used in GetDataIntoFiFo()) private const int FIFO\_SIZE\_RECORDS\_BLOCK = 4096; //block of records to read at once

private const bool STOP\_ON\_NUMBEROFRECORDS = true; //stop or not if we reach the end of TTTR file

These for parameters defines how the emulation will work (you do not need to change anything else). The impact of each of the parameters is the following:

* **TIME\_HARP\_EMULATION\_TTTR\_FILE** - a string, which point to the TTTR file to be used in the emulation of the Time Harp acquisition. By default if you do not specify a directory but just a file name, you must put the TTTR file in the same directory as the compiled executable.
* **THREAD\_SLEEP\_TIME** - emulates or DMA transfer (just wait the given milliseconds before starting the next read of TTTR data from the emulation file). I would recommend not to change it. It is not the most crucial thing for the emulation process.
* **FIFO\_SIZE\_RECORDS\_BLOCK** - this defines the number of records to read at once (before putting the reading thread **GetDataIntoFiFo()** to sleep and assigning status flag for FiFo Half buffer, FiFo Full buffer or FiFo Empty buffer). I would recommend not to change it. It is not the most crucial thing for the emulation process.
* **STOP\_ON\_NUMBER\_OF\_RECORDS** - defines whether or not the reading of TTTR file will continue from the beginning if we reach the end of the file. If the value is set to **true**, the reading will stop and acquisition will finish. If value is set to **false** the file position will move to the beginning of the TTTR section of the TTTR file and will continue to read records - in this way with a single TTTR file we can simulate either endless acquisition time or acquisition time bigger than the original acquisition time.

That is all you need to know about the emulation class in order to use it. Do not forget to setup properly some of the parameters when calling **SetupAPDCountAndTiming()** - the following parameters must be the same in order to reproduce the result from the original TTTR file:

* Measurement mode must be the same as was the original measurement.
* Range code must be the same as was the original measurement.
* Acquisition time must be the same as was the original measurement (well, you can play with different time , it is not strict condition, but then you will produce a new TTTR file).
* The width of the image must be the same as was the original measurement.
* The height of the image must be the same as was the original measurement.
* Gating time must be the same as was the original measurement (not strict but you will produce a new TTTR file and image, respectively).
* The type of scan (Bidirectional or unidirectional) must be the same as was the original measurement.

END OF THE STORY :)