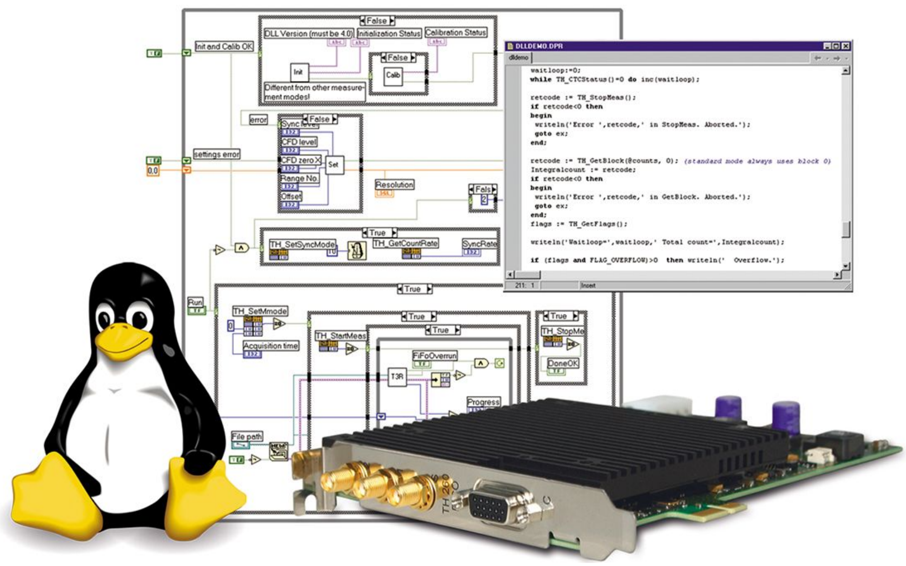
TimeHarp 260

TCSPC and MCS Board with
PCIe Interface

PicoQuant GmbH
Unternehmen für optoelektronische
Forschung und Entwicklung

TH260Lib – Programming Library for Custom Software Development under Linux



User's Manual

Version 3.1.0.1

Table of Contents

1. Introduction

The TimeHarp 260 is a cutting edge TCSPC system with PCIe (Peripheral Component Interconnect express) interface. Its new integrated design provides a flexible number of input channels at reasonable cost and allows innovative measurement approaches. The timing circuits allow high measurement rates up to 40 million counts per second (Mcps) and provide a very high time resolution. There are two versions of the TimeHarp 260. The PICO version (TimeHarp 260 P) has a resolution of 25 ps and a deadtime of 25 ns whereas the NANO version (TimeHarp 260 N) provides a time resolution of 250 ps*) with a deadtime of less than 2 ns. The modern PCIe interface provides very high throughput as well as 'plug and play' installation. The input triggers are programmable for a wide range of input signals. In case of the PICO version they have a programmable Constant Fraction Discriminator (CFD) for negative going signals while the NANO version provides level triggers for both negative and positive going signals. These specifications qualify the TimeHarp 260 for use with most common single photon detectors such as Single Photon Avalanche Diodes (SPADs) and Photomultiplier Tube (PMT) modules (via preamplifier). The best time resolution is obtained by using Micro Channel Plate PMTs (MCP–PMT) or modern SPAD detectors together with the PICO version. The width of the overall Instrument Response Function (IRF) can then be as short as 40 ps FWHM. Both models of the TimeHarp 260 can be purchased with 2 or 3 timing inputs. The use of these inputs is very flexible. In fluorescence lifetime applications the first channel is typically used as a synchronization input from a laser. The other input(s) are then used for photon detectors. In coincidence correlation applications all inputs can be used for photon detectors. The versions with two detector inputs + sync are called DUAL and the versions with one detector input + sync are called SINGLE.

The TimeHarp 260 can operate in various modes to adapt to different measurement needs. The standard histogram mode performs real–time histogramming in computer memory. Two different Time–Tagged–Time–Resolved (TTTR) modes allow recording of each photon event on separate, independent channels, thereby providing unlimited flexibility in off–line data analysis such as burst detection and time–gated or lifetime weighted Fluorescence Correlation Spectroscopy (FCS) as well as picosecond coincidence correlation, using the individual photon arrival times.

TTTR mode also allows capturing so called marker signals on four TTL inputs along with the regular photon events. This can be used for imaging applications or other synchronization purposes. Note that only the DUAL versions support markers.

The TimeHarp 260 standard software provides functions such as the setting of measurement parameters, display of results, loading and saving of measurement parameters and histogram curves. Important measurement characteristics such as count rate, count maximum and position, histogram width (FWHM) are displayed continuously. While these features will meet many of the routine demands, advanced users may want to include the TimeHarp's functionality in their own automated measurement systems with their own software. In particular where the measurement must be interlinked or synchronized with other processes or instruments this approach may be of interest. For this purpose a programming library is provided as a Dynamic Link Library (DLL) for Windows and as a shared library for Linux. The library supports custom programming on the x86 platform in virtually all major programming languages/environments for, notably C, C++, C#, Pascal (Delphi / Lazarus), MATLAB and LabVIEW. The Windows and Linux versions are fully API compatible, so that application programs can easily be ported between the Windows and Linux. This manual describes the installation and use of the TimeHarp 260 programming library TH260lib.so for Linux and explains the associated demo programs. Please read both this manual and the TimeHarp manual before beginning your own software developement with the library. The TimeHarp 260 is a sophisticated real-time measurement system. In order to work with the board using the TimeHarp library, sound knowledge in your chosen programming language is required. For details on the method of Time–Correlated Single Photon Counting, please refer to our TechNote on TCSPC.

*) TimeHarp 260 N manufactured before 2016 have a resolution of 1 ns but can be returned for an upgrade to 250 ps at moderate cost.

2. Release Notes

2.1. What's new in this version

The new version 3.1.0.1 of `TH260Lib` for Linux is primarily a functionality upgrade release to support new hardware versions manufactured after February 2017 (firmware 2.x). Apart from some histogramming throughput improvements the new release also fixes an issue where debug information was not properly retrieved via `TH260_GetHardwareDebugInfo` after initialization failures. The interface and data structures remain unchanged so that programs written for version 3.0 will continue to run without change except the adaption of version checking.

The last major release 3.0 provided these new features:

- Support of the latest hardware improvement of the TimeHarp 260 N - now running at 250 ps resolution*)
- Some minor bugfixes
- Updated demos
- API and data formats remain unchanged

Version 2.0 provided these new features:

- A new library routine `SetInputDeadTime` for suppression of some detector artefacts.
  (Note that this works only for TimeHarp 260 P purchased after April 2015. Old boards can be updated but must be returned to PicoQuant for this purpose.)
- A bugfix in the shutdown code called upon unloading the library
- Some small demo code improvements
- Some documentation fixes in section 7.2.

The changes are also marked in red in section 7.2 listing the individual library routines. See the notes there for synopsis.

2.2. General Notes

It is recommended to start your work with your TimeHarp 260 board by using the standard interactive TimeHarp 260 data acquisition software under Windows. This should give you a better understanding of the board's operation before attempting your own programming efforts. It also ensures that your optical/electrical setup is working.

This version of the TimeHarp 260 programming library requires at least version 3.x Linux kernels. It has been tested in applications built with gcc, Mono, Free Pascal and LabVIEW.

The following Linux distributions have been tested:

Ubuntu 14.04	kernel 3.13
OpenSUSE 13.2	kernel 3.16
Kubuntu 16.04.1	kernel 4.4
Linux Mint 18	kernel 4.4

In addition to an appropriate Linux version you need to have gcc and the kernel source headers for the running kernel installed. This is required to compile the TimeHarp 260 kernel driver.

This manual assumes that you have read the TimeHarp 260 manual and that you have experience with Linux and the chosen programming language. References to the TimeHarp manual will be made where necessary.

The library supports histogramming mode and both TTTR modes but your TimeHarp 260 board must have the library option enabled. If you have not initially purchased the library option (license) you can upgrade it any time later.

*) TimeHarp 260 N manufactured before 2016 have a resolution of 1 ns but can be returned for an upgrade to 250 ps at moderate cost.

Users who own a license for any older version of the library will receive free updates when they are available. For this purpose, please check the PicoQuant web site or write an email to *info@picoquant.com*.

Users upgrading from earlier versions of `TH260Lib` may need to adapt their programs. This is the price for technical progress. However, the required changes are usually minimal and will be explained in the manual (especially check the notes marked in red in section 7.2).

Note that despite of our efforts to keep changes minimal, data structures, program flow and function calls may still change in future versions without advance notice. Users must maintain appropriate version checking in order to avoid incompatibilities. There is a function call tat you can use to retrieve the version number (see section 7.2). The interface of releases with identical major version will usually remain the same.

2.3. Warranty and Legal Terms

Disclaimer

PicoQuant GmbH disclaims all warranties with regard to the supplied software and documentation including all implied warranties of merchantability and fitness for a particular purpose. In no case shall PicoQuant GmbH be liable for any direct, indirect or consequential damages or any material or immaterial damages whatsoever resulting from loss of data, time or profits; arising from use, inability to use, or performance of this software and associated documentation. Demo code is provided 'as is' without any warranties as to fitness for any purpose.

License and Copyright

With the TimeHarp 260 DLL option you have purchased a license to use the TimeHarp 260 programming library. You have not purchased any other rights to the software itself. The software is protected by copyright and intellectual property laws. You may not distribute the software to third parties or reverse engineer, decompile or disassemble the software or part thereof. You may use and modify demo code to create your own software. Original or modified demo code may be re–distributed, provided that the original disclaimer and copyright notes are not removed from it. Copyright of this manual and on–line documentation belongs to PicoQuant GmbH. No parts of it may be reproduced, translated or transferred to third parties without written permission of PicoQuant GmbH.

The kernel driver module for the PCIe interface is licensed independently from the TimeHarp 260 programming library TH260Lib.so. As opposed to TH260Lib.so it is distributed as source code under the GNU Public License (GPL). Please see the folder *driver* in the distribution archive for details.

TimeHarp is a registered trademark of PicoQuant GmbH.

Other products and corporate names appearing in this manual may or may not be registered trademarks or subject to copyrights of their respective owners. PicoQuant GmbH claims no rights to any such trademarks. They are used here only for identification or explanation and to the owner's benefit, without intent to infringe.

3. Installation of the TH260Lib Software Package

The TH260Lib package for Linux is provided for 32-bit and 64-bit versions of Linux. Unpack the the distribution archive somewhere in your home directory and see the corresponding folders in the distribution archive. Please use consistently only the items that fit your Linux version.

3.1. Installing the Driver

The programming library will access the TimeHarp 260 board(s) through a dedicated kernel driver. As opposed to TH260Lib.so this module is distributed as source code under the GNU Public License (GPL). Please see the subdirectory `driver` for details.

The kernel driver must be compiled as a module for the specific kernel it is intended to run with. The driver module `th260pcie.ko` can be built by issuing the `make` command in the `driver` source directory, that you should copy as a whole from the distribution archive to a temporary disk directory. In that disk location run `make`.

If the compiler and kernel headers are installed correctly you should get no errors running `make`. It is important to have the right kernel header files included. You need to ensure they are installed and located (possibly via symlink) under `/lib/modules/`uname -r`/build/include` where ``uname -r`` is retrieving the kernel version via shell command. Most current distributions do this properly, provided you instructed them to install the kernel headers.

As user 'root' you can then use the command `insmod th260pcie.ko` to load the driver.

After that you can run `tail /var/log/messages` or `dmesg` to see if the board was found and if the driver was loaded correctly. Recent Linux kernels demand loadable modules to be signed. If dmesg shows a warning on missing signature you can still work but the kernel will be marked “tainted”. To avoid this you may want to sign the th260pcie kernel module. See instructions on the Web for how to do this, e.g. at kernel.org.

If the driver does not find any boards you may want to use `lspci` to check if the board has been detected as a PCI device at all.

Routinely the driver should be loaded at boot time through a suitable startup script or the distribution's specific module loading scheme. This is distribution dependent and cannot be explained in more detail here.

In order to let users without root privileges use the board you will need to set up a udev rule that sets appropriate permissions. Such a rule script can be found in the folder udev of the TH260Lib archive. On typical Linux distributions it just needs to be copied to `/etc/udev/rules.d` or the equivalent folder of your Linux distribution. You will need to reload the udev rules or restart the computer in order to activate the new rule.

3.2. Installing the Library

The library is distributed as a binary file. By default it resides under `/usr/local/lib/th260` on 32-bit systems and in under `/usr/local/lib64/th260` on 64-bit systems. This is not a strict requirement but it is where the demo programs will look for the library files and therefore it is recommended to use this location. If your linux distribution does not follow this convention you may have to fix the paths manually.

The shell script `install` in the `lib32` or `lib64` distribution directory does the installation in one step. Just start it (as root) at the command prompt from within the library directory. After installing, the library is ready to use and can be tested with the demos provided.

If you want to install the library in a different place and/or if you want to simplify access to the library you can add the chosen path to `/etc/ld.so.conf` and/or to the path list in the environment variable `LD_LIBRARY_PATH`.

Note for SELinux: If upon linking with th260lib.so you get an error `"cannot restore segment prot after reloc"` you need to adjust the security settings for th260lib.so. As root you need to run:
`chcon -t texrel_shlib_t /usr/local/lib/th260/th260lib.so`

3.3. Installing the Demo Programs

The demos can be installed by simply copying the entire directory `demos32` or `demos64` from the distribution archive to a disk location of your choice. This need not be under the root account but you may need to adjust the file permissions. Use `demos32` if your Linux is a 32-bit version or `demos64` if your Linux is a 64-bit version.

4. The Demo Applications

4.1. Functional Overview

Please note that all demo code provided is correct to our best knowledge, however, we must disclaim all warranties as to fitness for a particular purpose of this code. It is provided 'as is' for no more than educational purposes and as a starting point for your own work.

Because the TCSPC data acquisition requires real–time processing and / or real–time storing of data, the work with the library is demanding both in programming skills and computer performance.

The demos are kept as simple as possible to maintain focus on the key issues of accessing the library. This is why most of the demos have a minimalistic user interface and / or run from the simple command box (console). For the same reason, the measurement parameters are mostly hard–coded and thereby fixed at compile time. It is therefore necessary to change the source code and re-compile the demos in order to run them in a way that is matched to your individual measurement setup. Running them unmodified will probably result in useless data (or none at all) because of inappropriate settings of sync divider, resolution, input levels, etc.

For the same reason of simplicity, the demos will always only use the first TimeHarp 260 device they find, although the library can support multiple devices. If you have multiple devices that you want to use simultaneously you need to change the code to match your configuration.

There are demos for C / C++, C#, Pascal / Lazarus, LabVIEW and MATLAB. For each of these programming languages / systems there are different demo versions for various measurement modes:

Histogramming Mode Demos

These demos show how to use the standard measurement mode for on–board histogramming. These are the simplest demos and the best starting point for your own experiments. In case of LabVIEW the standard mode demo is more sophisticated and allows interactive input of most parameters.

TTTR Mode Demos

These demos show how to use TTTR mode, i.e. recording individual photon events instead of forming histograms on board. This permits advanced data analysis methods, such as single molecule burst detection, the combination of fluorescence lifetime measurement with FCS and picosecond coincidence correlation or even Fluorescence Lifetime Imaging (FLIM).

The TimeHarp 260 actually supports two different Time–Tagging modes, T2 and T3 mode. When referring to both modes together we use the general term TTTR here. For details on the two modes, please refer to your TimeHarp manual. In TTTR mode it is also possible to record external TTL signal transitions as markers in the TTTR data stream, which is typically used for FLIM. For more information see the section about TTTR mode in your TimeHarp manual.

Note that you must not call any of the `TH260_Setxxx` routines while a TTTR measurement is running. The result would potentially be loss of events in the TTTR data stream. Changing settings during a measurement makes no sense anyway, since it would introduce data inconsistency.

4.2. The Demo Applications by Programming Language

As outlined above, there are demos for C / C++, Pascal / Lazarus, C#, LabVIEW and MATLAB. For each of these programming languages / systems there are different demo versions for the measurement modes listed in the previous section. They are not 100% identical.

This manual explains the special aspects of using the TimeHarp programming library, it does NOT teach you how to program in the chosen programming language. We strongly recommend that you do not choose a development with the TimeHarp programming library as your first attempt at programming. You will also need some knowledge about Linux and dynamic linking. The ultimate reference for details about how to use the libary is in any case the source code of the demos and the header files of TH260Lib (`th260lib.h` and `th260defin.h`).

Be warned that wrong parameters and / or variables, invalid pointers and buffer sizes, inappropriate calling sequences etc. may crash your application and / or your complete computer. This may even be the case for relatively safe operating systems because you are accessing a kernel mode driver through `TH260Lib`. This driver has high privileges at kernel level, that provide all power to do damage if used inappropriately. Make sure to backup your data and / or perform your development work on a dedicated machine that does not contain valuable data. Note that the library is not fully re-entrant. This means, it cannot be accessed arbitrarily from multiple, concurrent processes or threads at the same time. Only calls accessing different boards can be made concurrently. All calls to one individual board must be made sequentially in the order shown in the demos.

The C / C++ Demos

These demos are provided in the `C` subfolder. The code is actually plain C to provide the smallest common denominator for C and C++. Consult `th260lib.h`, `th260defin.h` and this manual for reference on the library calls. The library functions must be declared as `extern "C"` when used from C++. This is achieved most elegantly by wrapping the entire include statements for the library headers:

```
extern "C"
{
  #include "th260defin.h"
  #include "th260lib.h"
}
```

To test any of the demos, consult the TimeHarp manual for setting up your TimeHarp 260 and establish a measurement setup that runs correctly and generates useable test data. Compare the settings (notably sync divider, binning and CFD levels) with those used in the demo and use the values that work in your setup when building and testing the demos.

The C demos are designed to run in a console (terminal window). They need no command line input parameters. They create their output files in their current working directory (`*.out`). The output files will be ASCII-readable in case of the standard histogramming demos. For this demo, the ASCII files will contain one or multiple columns of integer numbers representing the counts in the histogram bins. You can use any editor or a data visualization program to inspect the ASCII histograms. For the TTTR modes the output is stored in binary format for performance reasons. The binary files must be read by dedicated programs according to the format they were written in. The file read demos provided for the TimeHarp TTTR data files can be used as a starting point. They cannot be used directly on the demo output because they expect a file header the demos do not generate. This is intentional in order to keep the TH260Lib demos focused on the key issues of using the library.

By default, the TTTR mode demo is configured for T2 mode. You need to change the mode input variable going into TH260_Initialize to a value of 3 if you want T3 mode. Note that you probably also need to adjust the sync divider and the resolution in this case.

The C# Demos

The C# demos are provided in the `Csharp` subfolder. They have been tested with MS Visual Studio 2010 under Windows as well as with Mono under Linux. The only difference is the library name, which in principle could also be unified.

Calling a native DLL (unmanaged code) from C# requires the `DllImport` attribute and correct type specification of the parameters. Not all types are easily portable. Especially C strings require special handling. The demos show how to do this.

With the C# demos you also need to check wether the hardcoded settings are suitable for your actual instrument setup. The demos are designed to run in a console (terminal window). They need no command line input parameters. They create their output files in their current working directory (*.out). The output files will be ASCII in case of the histogramming demos. For TTTR mode the output is stored in binary format for performance reasons. The ASCII files will contain single or multiple columns of integer numbers representing the counts from the histogram channels. You can use any editor or a data visualization program to inspect the ASCII histograms. The binary files must be read by dedicated programs according to the format they were written in.

The Pascal / Lazarus Demos

Pascal or Lazarus users refer to the `Pascal` folder. Lazarus users can use the `*.LPI` files to load the projects.

In order to make the exports of `TH260Lib` known to your application you have to declare each function in your Pascal code as 'external'. This is already prepared in the demo source code. Please check the function parameters of your code against th260lib.h in the demo directory whenever you update to a new library version.

The Pascal / Lazarus demos are also designed to run in a console (terminal window). They need no command line input parameters. They create output files in their current working directory. The output files will be ASCII in case of the histogramming demo. In TTTR mode the output is stored in binary format for performance reasons. You can use any data visualization program to inspect the ASCII histograms. The binary files must be read by dedicated programs according to the format they were written in. The file read demos provided for the TimeHarp 260 TTTR data files can be used as a starting point. They cannot be used directly on the demo output because they expect a file header the demos do not generate. This is intentional in order to keep the demos focused on the key issues of using the library.

By default, the TTTR mode demo is configured for T2 mode. This will not allow you to work with high sync rates. You need to change the mode input variable going into TH260_Initialize to a value of 3 if you want T3 mode. At the same time you need to modify your program for an appropriate sync divider and a suitable range (resolution).

The LabVIEW Demos

The LabVIEW demo VIs are provided in the `LABVIEW` folder. They are contained in LabVIEW libraries (`*.llb`). The top–level VIs are `HISTOmain.vi` in `HISTOmode.vi` and `TTTRmain.vi` in `TTTRmode.vi`. Note that the toplevel VIs share some identical sub-VIs in `common.llb`. You need to have LabVIEW 8.0 or higher.

The LabVIEW demos are the most sophisticated demos here. The standard mode demo resembles the standard TimeHarp software with input fields for all settable parameters. Run the top–level VI named `HISTOmain.vi`. It will first initialize the hardware. The status of initialization will be shown in the top left display area. Make sure you have a functional setup with signals correctly connected. You can then adjust the sync level until you see the expected sync rate in the meter below. Then you can click the Run button below the histogram display area. The demo implements a simple Oscilloscope mode. Make sure to set an acquisition time of not much more than e.g. a second, otherwise you will see nothing for a long time. If the input discriminator settings are correct you should see a histogram. You can stop the measurement with the same (*Run*) button.

The TTTR mode demo for LabVIEW is a little simpler. It provides the same panel elements for setting parameters etc. but there is no graphic display of results. Instead, all data is stored directly to disk. By default, the TTTR mode demo is configured for T2 mode. This will not allow you to work with high sync rates. You need to change the mode input variable going into to the Initialization VI to a value of 3 if you want T3 mode. You also need to use an appropriate sync divider and a suitable range (resolution).

To run the TTTR mode demo you start `TTTRmain.vi`. First set up the Sync and CFD levels. You can watch the sync rate in a graphic rate meter. Then you can select a measurement time and a file name. When you click the Run button a measurement will be performed, with the data going directly to disk. There is a status indicator showing the current number of counts recorded. There is also a status LED indicating any FIFO overrun.

Internally the TTTR mode demo also deserves a special note: each TTTR record as returned in the buffer of TH260_ReadFiFo actually is a DWORD (32bit). However, LabVIEW stores DWORD data (U32) always in big endian format. On the Intel platform (little endian) this results in reversed bytes compared to C programs. For consistency with the demo programs for reading TTTR data this byte reversing of the data going to disk is avoided in the demo by declaring the buffer for `TH260_ReadFiFo` as a byte array (hence 4 times larger than the DWORD array). You may instead want to work with a U32 array if your goal is not storing data to disk but doing some on–line analysis of the TTTR records. In this case you must initialize the array with U32 and change the type of buffer in the library calls of `TH260_ReadFiFo` to U32.

The LabVIEW demos access the library routines via the 'Call Library Function' of LabVIEW. For details refer to the LabVIEW documentation. Consult `th260lib.h` or the manual section further down for the parameter types etc. Make sure to specify the correct calling convention (stdcall).

Strictly observe that the `TH260_xxxx` library calls relating to the same device are not re–entrant. They must be made sequentially and in the right order. They cannot be called in parallel as is the default in LabVIEW if you place them side by side in a diagram. Although you can configure each library call to avoid parallel execution, this still gives no precise control over the order of execution. For some of the calls this order is very important. Sequential execution must therefore be enforced by sequence structures or data dependency. In the demos this is typically done by chained and/or nested case structures. This applies to all VI hierarchy levels, so sub–VIs containing library calls must also be executed in correct sequence.

The MATLAB Demos

The MATLAB demos are provided in the `MATLAB` folder. They are contained in .m files. You need to have a MATLAB version that supports the `loadlibrary` and `calllib` commands. The earliest version we have tested is MATLAB 7.3 but any version from 6.5 should work. Note that recent versions of MATLAB require a compiler to be set up for work with DLLs. For your specific version of MATLAB, please check the documentation of the MATLAB command `loadlibrary` as to what it requires. Be careful about the header file name specified in `loadlibrary`. The names are case sensitive and a wrong spelling will lead to an apparently successful load - but later no library calls will work.

The MATLAB demos are designed to run inside the MATLAB console. They need no command line input parameters. They create output files in their current working directory. The output file will be ASCII in case of the histogramming demo. In TTTR mode the output is stored in binary format for performance reasons. You can use any data visualization program to inspect the ASCII histograms. The binary files must be read by dedicated programs according to the format they were written in. The file read demos provided for the TimeHarp 260 TTTR data files can be used as a starting point. They cannot be used directly on the demo output because they expect a file header the demos do not generate. This is intentional in order to keep the demos focused on the key issues of using the library.

By default, the TTTR mode demo is configured for T2 mode. This will not allow you to work with high sync rates. You need to change the mode input variable going into `TH260_Initialize` to a value of 3 if you want T3 mode. At the same time you need to modify your program for an appropriate sync divider and a suitable range (resolution).

5. Advanced Techniques

5.1. Using Multiple Devices

The library is designed to work with multiple TimeHarp 260 devices (up to 4). The demos always use the first device found. If you have more than one TimeHarp 260 and you want to use them together you need to modify the code accordingly. At the API level of TH260Lib the devices are distinguished by a device index (0 .. 3). The device order corresponds to the order Linux enumerates the devices. It may therefore be difficult to know which physical device corresponds to the given device index. In order to solve this problem, the library routine `TH260_OpenDevice` provides a second argument through which you can retrieve the serial number of the physical device at the given device index. Similarly you can use `TH260_GetSerialNumber` any time later on a device you have successfully opened. The serial number of a physical TimeHarp device can be found on a label at the back of the PCB. It is a 8 digit number starting with 010. The leading zero will not be shown in the serial number strings retrieved through `TH260_OpenDevice` or `TH260_GetSerialNumber`. If you install multiple devices in one PC it is a good idea to write down the serial nubers and their respective installation slots before you close the PC.

As outlined above, if you have more than one TimeHarp 260 and you want to use them together you need to modify the demo code accordingly. This requires briefly the following steps: Take a look at the demo code where the loop for opening the device(s) is. In most of the demos all available devices are opened. You may want to extend this so that you

1. filter out devices with a specific serial number and
2. do not hold open devices you don't actually need.

The latter is recommended because a device you hold open cannot be used by other programs.

By means of the device indices you picked out, you can then extend the rest of the program, so that every action taken on the single device is also done on all devices of interest, i.e. initialization, setting of parameters, starting a measurement etc. At the end the demos close all devices. It is recommended to keep this approach. It does no harm if you close a device that you haven't opened.

5.2. Efficient Data Transfer

The TTTR modes are designed for fast real–time data acquisition. TTTR mode is most efficient in collecting data with a maximum of information. It is therefore most likely to be used in sophisticated on–line data processing scenarios, where it may be worth optimizing data throughput.

In order to achieve the highest throughput, the TimeHarp 260 uses busmaster DMA transfers. This is supported by the PC hardware that can transfer data to the host memory without much help of the CPU. For the TimeHarp 260 this permits data throughput as high as 40 Mcps and leaves time for the host to perform other useful things, such as on–line data analysis or storing data to disk.

In TTTR mode the data transfer process is exposed to the library user in a single function `TH260_ReadFiFo` that accepts a buffer address where the data is to be placed, and a transfer block size. This block size is critical for efficient transfers. The larger the block size, the better the transfer efficiency. This is because setting up a transfer costs some fixed amount of time, independent of the block size. The maximum transfer block size is 131,072 (128k event records). However, it may not under all circumstances be ideal to use the maximum size. The minimum size is 128.

As noted above, the transfer is implemented efficiently without using the CPU excessively. Nevertheless, assuming large block sizes, the transfer takes some time. Linux therefore gives the unused CPU time to other processes or threads, i.e., it waits for completion of the transfer without burning CPU time. This wait time is what can also be used for doing ‘useful things’ in terms of any desired data processing or storing within your own application. The way of doing this is to use multi-threading. In this case you design your program with two threads, one for collecting the data (i.e. working with `TH260_ReadFiFo`) and another for processing or storing the data. Multiprocessor systems can benefit from this technique even more. Of course you need to provide an appropriate data queue between the two threads and the means of thread synchronization. Thread priorities are another issue to be considered. Finally, if your program has a graphic user interface you may need a third thread to respond to user actions reasonably fast. Again, this is an advanced technique and cannot be demonstrated in detail here. Greatest care must be taken not to access the library from different threads without strict control of mutual exclusion and maintaining the right sequence of function calls.

However, the technique allows significant throughput improvements and advanced programmers may want to use it. It might be interesting to note that this is how TTTR mode is implemented in the regular Time-Harp 260 software for Windows, where sustained count rates as high as 40 Mcps (to disk) can be achieved.

In case of using multiple devices it is also beneficial for overall throughput if you use multi-threading in order to fetch and store data from the individual devices in parallel. Again, re-entrance issues must be observed carefully in this case, at least for all calls accessing the same device.

5.3. Working with Very Low Count Rates

As noted above, the transfer block size is critical for efficient transfers. The larger the block size, the better the transfer efficiency. This is because setting up a transfer costs some fixed amount of time, independent of the block size. However, it may not under all circumstances be ideal to use the maximum size. A large block size takes longer to fill. If the count rates in your experiment are very low, it may be better to use a smaller block size. This ensures that the transfer function returns more promptly. It should be noted that the Time-Harp has a “watchdog” timer that terminates large transfer requests prematurely so that they do not wait forever if new data is coming very slowly. This results in `TH260_ReadFiFo` returning less than requested (possibly even zero). This helps to avoid complete stalls even if the maximum transfer size is used with low or zero count rates. However, for fine tuning of your application may still be of interest to use a smaller block size. The block size must be a multiple of 128 records. The smallest permitted size is 128.

Also note that with very low count rates (and sync rates) the hardware meters read via `TH260_GetSyncRate` as well as `TH260_GetCountRate` are of limited precision. The hardware meters are using a counter time window of 100 ms. Consequently, their resolution at the lower rate end is limited. If you must determine very slow sync rates you may want to use `TH260_GetSyncPeriod`. Note, however, that this routine does not average over multiple periods and may therefore deliver slightly more fluctuating results. If you need to determine very low count rates, the only solution is to perform a measurement and count the results.

5.4. Working with Warnings

The library provides routines for obtaining and interpreting warnings about critical measurement conditions. The mechanism and warning criteria are the same as those used in the regular TimeHarp software for Windows. In order to obtain and use these warnings also in your custom software you may want to use the library routine `TH260_GetWarnings`. This may help inexperienced users to notice possible mistakes before stating a measurement or even during the measurement.

It is important to note that the generation of warnings is dependent on the current count rates and the current measurement settings. It was decided that `TH260_GetWarnings` does not obtain the count rates on its own, because the corresponding calls take some time and might waste too much processing time. It is therefore necessary that `TH260_GetSyncRate` as well as `TH260_GetCountRate` (for all channels) have been called before `TH260_GetWarnings` is called. Since most interactive measurement software periodically retrieves the rates anyhow, this is not a serious complication.

The routine `TH260_GetWarnings` delivers the cumulated warnings in the form of a bit field. In order to translate this into readable information you can use `TH260_GetWarningsText`. Before passing the bit field into `TH260_GetWarningsText` you can mask out individual warnings by means of the bit masks defined in `hhdefin.h`.

5.5. Hardware Triggered Measurements

This measurement scheme works essentially like regular histogramming mode but it allows to start and stop the acquisition by means of external TTL signals. Since it is an advanced real-time technique, beginners are advised not to use it for first experiments. For the same reason, the correspondin demos exist only in C.

Before using this scheme, consider when it is useful to do so. Remember that TTTR mode is usually the most efficient way of retrieving the maximum information on photon dynamics. By means of marker inputs the photon events can be precisely assigned to complex external event scenarios.

The TimeHarp's data acquisition can be controlled in various ways. Default is the TimeHarp's internal CTC (counter timer circuit). In that case the histograms will take the duration set by the `tacq` parameter passed to `TH260_StartMeas`. The other way of controlling the histogram boundaries (in time) is by external TTL signals fed to the control connector pins C1 and C2. In that case it is possible to have the acquisition started

and stopped when specific signals occur. It is also possible to combine external starting with stopping through the internal CTC. Details are cotrolled by the parameters supplied to .
Dependent on the parameter `meascontrol` the following modes of operation can be obtained:

Symbolic Name	Value	Function
MEASCTRL_SINGLESHOT_CTC	0	Default value. Acquisition starts by software command and runs until CTC expires. The duration is set by the `tacq` parameter passed to `TH260_StartMeas`.
MEASCTRL_C1_GATE	1	Histograms are collected for the period where C1 is active. This can be the logical high or low period dependent on the value supplied to the parameter `startedge`.
MEASCTRL_C1_START_CTC_STOP	2	Data collection is started by a transition on C1 and stopped by expiration of the internal CTC. Which transition actually triggers the start is given by the value supplied to the parameter `startedge`. The duration is set by the `tacq` parameter passed to `TH260_StartMeas`.
MEASCTRL_C1_START_C2_STOP	3	Data collection is started by a transition on C1 and stopped by by a transition on C2. Which transitions actually trigger start and stop is given by the values supplied to the parameters `startedge` and `stopedge`.

The symbolic constants shown above are defined in `th260defin.h`. There are also symbolic constants for the parameters controlling the active edges (rising/falling).

Please study the demo code for external hardware triggering and observe the polling loops required to detect the beginning and end of a measurement.

6. Problems, Tips & Tricks

6.1. PC Performance Issues

The TimeHarp device and its software interface are a complex real–time measurement system demanding appropriate performance both from the host PC and the operating system. This is why a reasonably modern CPU and sufficient memory are required. At least a dual core, 2 GHz processor, 2 GB of memory and a fast hard disk are recommended.

6.2. PCIe Interface

In order to deliver maximum throughput, the TimeHarp 260 uses state–of–the–art busmastering DMA transfers. For this purpose it requires an interrupt line. Dependent on the design of the PC's mainboard there may be limited interrupt ressources so that slot cards and/or onboard devices need to share interrupt lines. This may lead to conflicts and/or performance degradation. Interrupt sharing can sometimes be avoided by using another slot. In some cases it is also possible to change interrupt assignments in the BIOS setup. Contact PicoQuant for assistance if you are in doubt which PC or mainboard to buy.

6.3. Power Saving

If your computer is configured to allow power saving (suspend/sleep) then (dependent on the BIOS configuration) the TimeHarp device may be powered down more or less unexpectedly. In order to avoid loss of data you may need to design your software so that it detetcs the corresponding power events (signals) sent by the operating system, stop the current measurement and save the data. Upon wakeup you will need to repeat the initialization sequence of library calls to allow new measurements.

Another form of power saving that can cause complications is PCIe link power management (ASPM). There are Mainboards where ASPM is not implemented properly so that the TimeHarp 260 and other high speed PCIe cards will not work. Some mainboard manufacturers such as ASUS provide BIOS updates that fix the issues. In cases where such updates are not abvailable the workaround is to disable PCIe link power saving in the BIOS (if available) or at operating system level (if available).

6.4. Troubleshooting

Troubleshooting should begin by testing your hardware setup. This is best accomplished by the standard TimeHarp software for Windows (supplied by PicoQuant). Only if this software is working properly you should start work with the library under Linux. If there are problems even with the standard software, please consult the TimeHarp manual for detailed troubleshooting advice.

The library will access the TimeHarp device through a dedicated kernel driver. You need to make sure the driver has been installed and loaded correctly. You also need to make sure access permissions for users other than root are set via udev. See section 7. Please consult the TimeHarp manual for hardware related problem solutions.

The next step, if hardware and driver are working, is to make sure you have the right library version installed. See section 7. You should also make sure your board has the right firmware with license to use the DLL.

To get started, try the readily compiled demos supplied with the DLL. For first tests take the standard histogramming demos. If this is working, your own programs should work as well. Note that the hard coded settings may not be compatible with your experimental setup. Then the pre–compiled demo may not work as expected. Only the LabVIEW demo allows to enter the settings interactively.

6.5. Version Tracking

While PicoQuant will always try to maintain a maximum of continuity in further hardware and software development, changes for the benefit of technical progress cannot always be avoided. It may therefore happen, that data structures, calling conventions or program flow will change. In order to design programs that will recognize such changes with a minimum of trouble we strongly recommend that you make use of the functions provided for version retrieval of hardware and library. In any case your software should issue a warning if it detects versions other than those it was tested with.

6.6. Software Updates

We work hard to constantly improve and update the software for our instruments. This includes updates of the configurable hardware (FPGA). Such updates are important as they may affect reliability and interoperability with other products. The software updates are free of charge, unless major new functionality is added. It is strongly recommended that you check the PicoQuant website for library updates before investing time and effort into a new software development.

6.7. Bug Reports and Support

The TimeHarp 260 TCSPC system has gone through extensive testing. Nevertheless, it is a fairly new product and some glitches may still occur under the myriads of possible PC configurations and application circumstances. We therefore would like to offer you our support in any case of problems with the system. Do not hesitate to contact your sales representative or PicoQuant in case of difficulties with your TimeHarp or the programming library.

If you should observe errors or bugs caused by the TimeHarp system please try to find a reproducible error situation. Email a detailed description of the problem and all relevant circumstances, especially other hardware installed in your PC, to support@picoquant.com. Please provide a listing of your PC configuration and attach it to your error report. Your feedback will help us to improve the product and documentation.

Of course we also appreciate good news: If you have obtained exciting results with one of our instruments, please let us know, and where appropriate, please mention the instrument in your publications. At our Web–site we maintain a large bibliography of publications related to our instruments. It may serve as a reference for you and other potential users. See http://www.picoquant.com/scientific/references/. Please submit your publications for addition to this list.

7. Appendix

7.1. Data Types

The TimeHarp programming library `TH260Lib` is written in C and its data types correspond to standard C / C++ data types as follows:

`char`	8 bit, byte (or characters in ASCII)
`short int`	16 bit signed integer
`unsigned short int`	16 bit unsigned integer
`int` `long int`	32 bit signed integer
`unsigned int` `unsigned long int`	32 bit unsigned integer
`__int64` `long long int`	64 bit signed integer
`unsigned int64` `unsigned long long int`	64 bit unsigned integer
`float`	32 bit floating point number
`double`	64 bit floating point number

Note that on platforms other than the Intel x86 architecture byte swapping may occur when the TimeHarp data files are read there for further processing. We recommend using the native Intel architecture environment consistently.

7.2. Functions Exported by `TH260Lib`

See **`th260defin.h`** for predefined constants given in capital letters here. Return values `< 0` denote errors. See **`errcodes.h`** for the error codes. Note, that `TH260Lib` is a multi device library with the capability to control more than one TimeHarp 260 simultaneously. For that reason all device specific functions (i.e. the functions from section 7.2.2 on) take a device index as first argument. The TimeHarp 260 may have one or two input channels. Note that functions taking a channel number as an argument expect the channels enumerated 0..N-1 while the graphical TimeHarp 260 software (Windows) as well as the connector labelling enumerates the channels 1..N. This is due to internal data structures and consistency with earlier products.

7.2.1. General Functions

These functions work independent from any device.

`int TH260_GetErrorString (char* errstring, int errcode);`

```
arguments:          errstring:       pointer to a buffer for at least 40 characters
                    errcode:         error code returned from a TH260_xxx function call

return value:       >0               success
                    <0               error
```

Note: This function is provided to obtain readable error strings that explain the cause of the error better than the numerical error code. Use these in error handling message boxes, support enquiries etc.

`int TH260_GetLibraryVersion (char* vers);`

```
arguments:          vers:            pointer to a buffer for at least 8 characters

return value:       =0               success
                    <0               error
```

Note: Use the version information to ensure compatibility of the library with your own application.

7.2.2. Device Specific Functions

All functions below are device specific and require a device index.

`int TH260_OpenDevice (int devidx, char* serial);`

```
arguments:          devidx:          device index 0..3
                    serial:          pointer to a buffer for at least 8 characters

return value:       =0               success
                    <0               error
```

Note: Opens the device for use. Must be called before any of the other functions below can be used.

`int TH260_CloseDevice (int devidx);`

```
arguments:          devidx:          device index 0..3

return value:       =0               success
                    <0               error
```

Note: Closes and releases the device for use by other programs.

`int TH260_Initialize (int devidx, int mode);`

```
arguments:          devidx:          device index 0..3
                    mode:            measurement mode
                                        0 = histogramming mode
                                        2 = T2 mode
                                        3 = T3 mode

return value:       =0               success
                    <0               error
```

Note: This routine must be called before any of the other routines below can be used. Note that some of them depend on the measurement mode you select here. See the TimeHarp manual for more information on the measurement modes.

7.2.3. Functions for Use on Initialized Devices

All functions below can only be used after `TH260_Initialize` was successfully called.

int TH260_GetHardwareInfo (int devidx, char* model, char* partno, char* version);

```
arguments:          devidx:         device index 0..3
                    model:          pointer to a buffer for at least 16 characters
                    partno:         pointer to a buffer for at least  8 characters
                    version:        pointer to a buffer for at least 16 characters

return value:       =0              success
                    <0              error
```

int TH260_GetSerialNumber (int devidx, char* serial);

```
arguments:          devidx:         device index 0..3
                    vers:           pointer to a buffer for at least 8 characters

return value:       =0              success
                    <0              error
```

int TH260_GetFeatures (int devidx, int* features);

```
arguments:          devidx:         device index 0..3
                    flags:          pointer to an integer
                                       returns features of this board (a bit pattern)

return value:       =0              success
                    <0              error
```

Note: Use the predefined bit feature values in `th260defin.h` (`FEATURE_xxx`) to extract individual bits through a bitwise AND. Typically this is only for information, or to check if your board has a specific (optional) capability.

int TH260_GetBaseResolution (int devidx, double* resolution, int* binsteps);

```
arguments:          devidx:         device index 0..3
                    resolution:     pointer to a double precision float (32 bit)
                                       returns the base resolution in ps
                    binsteps:       pointer to an integer,
                                       returns the maximally allowed binning steps

return value:       =0              success
                    <0              error
```

Note: The value returned in `binsteps` is the maximum value allowed for the TH260_SetBinning function.

int TH260_GetNumOfInputChannels (int devidx, int* nchannels);

```
arguments:          devidx:         device index 0..3
                    nchannels:      pointer to an integer,
                                       returns the number of installed input channels

return value:       =0              success
                    <0              error
```

Note: The number of input channels is counting only the regular detector channels. It does not count the sync channel. Neverthe-less, it is possible to connect a detector also to the sync channel, e.g. in histogramming mode for antibunching or in T2 mode.

int TH260_SetTimingMode(int devidx, int mode); // TimeHarp 260 P only

```
arguments:          devidx:         device index 0..3
                    mode:           0 = Hires (25ps), 1 = Lowres (2.5 ns, a.k.a. "Long range")
                                        will change the base resolution of the board

return value:       =0              success
                    <0              error
```

```
int TH260_SetSyncDiv (int devidx, int div);

arguments:          devidx:          device index 0..3
                    div:             sync rate divider
                                        (1, 2, 4, .., SYNCDIVMAX)

return value:       =0               success
                    <0               error
```

Note: The sync divider must be used to keep the effective sync rate at values < 40 MHz. It should only be used with sync sources of stable period. The readings obtained with TH260_GetCountRate are corrected for the divider setting and deliver the external (undivided) rate. When the sync input is used for a detector signal the divider should be set to 1.

```
int TH260_SetSyncCFD (int devidx, int level, int zerox);   // TimeHarp 260 P only

arguments:          devidx:          device index 0..3
                    level:           CFD discriminator level in millivolts
                                        minimum = CFDLVLMIN
                                        maximum = CFDLVLMAX
                    zerox:           CFD zero cross level in millivolts
                                        minimum = CFDZCMIN
                                        maximum = CFDZCMIN

return value:       =0               success
                    <0               error
```

```
int TH260_SetSyncEdgeTrg (int devidx, int level, int edge);   // TimeHarp 260 N only

arguments:          devidx:          device index 0..3
                    level:           Trigger level in millivolts
                                        minimum = CFDLVLMIN
                                        maximum = CFDLVLMAX
                    edge:            Trigger edge
                                        0 = falling
                                        1 = rising

return value:       =0               success
                    <0               error
```

```
int TH260_SetSyncChannelOffset (int devidx, int value);

arguments:          devidx:          device index 0..3
                    value:           sync timing offset in ps
                                        minimum = CHANOFFSMIN
                                        maximum = CHANOFFSMAX

return value:       =0               success
                    <0               error
```

```
int TH260_SetInputCFD (int devidx, int channel, int level, int zerox);   // TimeHarp 260 P only

arguments:          devidx:          device index 0..3
                    channel:         input channel index 0..1
                    level:           CFD discriminator level in millivolts
                                        minimum = CFDLVLMIN
                                        maximum = CFDLVLMAX
                    zerox:           CFD zero cross level in millivolts
                                        minimum = CFDZCMIN
                                        maximum = CFDZCMAX

return value:       =0               success
                    <0               error
```

Note: The maximum channel index must correspond to `nchannels-1` as obtained through TH260_GetNumOfInputChannels().

```
int TH260_SetInputEdgeTrg (int devidx, int channel, int level, int edge);   // TimeHarp 260 N only

arguments:          devidx:          device index 0..3
                    channel:         input channel index 0..1
                    level:           CFD discriminator level in millivolts
                                        minimum = DISCRMIN
```

```
                                          maximum = DISCRMAX
                    edge:                 Trigger edge
                                             0 = falling
                                             1 = rising

return value:       =0                    success
                    <0                    error
```

Note: The maximum channel index must correspond to `nchannels-1` as obtained through TH260_GetNumOfInputChannels().

`int TH260_SetInputChannelOffset (int devidx, int channel, int value);`

```
arguments:          devidx:               device index 0..3
                    channel:              input channel index 0..nchannels-1
                    value:                channel timing offset in ps
                                             minimum = CHANOFFSMIN
                                             maximum = CHANOFFSMAX

return value:       =0                    success
                    <0                    error
```

Note: The maximum channel index must correspond to `nchannels-1` as obtained through TH260_GetNumOfInputChannels().

`int TH260_SetInputChannelEnable (int devidx, int channel, int enable);`

```
arguments:          devidx:               device index 0..3
                    channel:              input channel index 0..nchannels-1
                    enable:               desired enable state of the input channel
                                             0 = disabled
                                             1 = enabled

return value:       =0                    success
                    <0                    error
```

Note: The maximum channel index must correspond to `nchannels-1` as obtained through TH260_GetNumOfInputChannels().

`int TH260_SetInputDeadTime (int devidx, int channel, int tdcode); // new since v2.0, TH260 P only`

```
arguments:          devidx:               device index 0..3
                    channel:              input channel index 0..nchannels-1
                    tdcode:               code for desired deadtime of the input channel
                                             minimum = TDCODEMIN
                                             maximum = TDCODEMAX

return value:       =0                    success
                    <0                    error
```

Note: The maximum channel index must correspond to `nchannels-1` as obtained through TH260_GetNumOfInputChannels(). The codes 0..7 correspond to approximate deadtimes of 24, 44, 66, 88 112, 135, 160 and 180 ns. Exact values are subject to production tolerances on the order of 10%. This feature is not available in boards produced before April 2015 but can be upgraded on request. The main purpose is that of suppressing artefacts (afterpulsing) produced by some types of detectors. Whether or not a given board supports this feature can be checked via `TH260_GetFeatures` and the bit mask `FEATURE_PROG_TD` as defined in `thdefin.h`. Note that the programmable deadtime is not available for the sync input.

`int TH260_SetStopOverflow (int devidx, int stop_ovfl, unsigned int stopcount);`

```
arguments:          devidx:               device index 0..3
                    stop_ofl:             0 = do not stop,
                                          1 = do stop on overflow
                    stopcount:            count level at which should be stopped
                                             minimum = STOPCNTMIN
                                             maximum = STOPCNTMAX

return value:       =0                    success
                    <0                    error
```

Note: This setting determines if a measurement run will stop if any channel reaches the maximum set by `stopcount`. If `stop_ofl` is 0 the measurement will continue but counts above `STOPCNTMAX` in any bin will be clipped.

int TH260_SetBinning (int devidx, int binning);

```
arguments:          devidx:          device index 0..3
                    binning:         measurement binning code
                                        minimum = 0      (smallest, i.e. base resolution)
                                        maximum = (MAXBINSTEPS-1)               (largest)

return value:       =0               success
                    <0               error
```

Note: binning corresponds to repeated multiplication of the base resolution by 2 as follows:

0 = 1x base resolution,
1 = 2x base resolution,
2 = 4x base resolution,
3 = 8x base resolution, and so on.

int TH260_SetOffset (int devidx, int offset);

```
arguments:          devidx:          device index 0..3
                    offset:          histogram time offset in ns
                                        minimum = OFFSETMIN
                                        maximum = OFFSETMAX

return value:       =0               success
                    <0               error
```

Note: The offset programmed here is fundamentally different from the input offsets. It applies only **after** the time difference of input channel and sync has been calculated. It can be used to move arge stop-start differences into the histogram range that would normally not be recorded. It is only meaningful in histogramming and T3 mode.

int TH260_SetHistoLen (int devidx, int lencode, int* actuallen);

```
arguments:          devidx:          device index 0..3
                    lencode:         histogram length code
                                        minimum = 0
                                        maximum = MAXLENCODE    (default)
                    actuallen:       pointer to an integer,
                                        returns the current length (time bin count) of histograms
                                        calculated according to:  actuallen = 1024*(2^lencode)

return value:       =0               success
                    <0               error
```

Note: This sets the number of time bins in histogramming and T3 mode. It is not meaningful in T2 mode.

int TH260_ClearHistMem (int devidx);

```
arguments:          devidx:          device index 0..3

return value:       =0               success
                    <0               error
```

Note: This clears the histogram memory. It is not meaningful in T2 and T3 mode.

int TH260_SetTriggerOutput (int devidx, int period);

```
arguments:          devidx:          device index 0..3
                    period:          trigger period in units of 100ns  (0=off)
                                        minimum = TRIGOUTMIN
                                        maximum = TRIGOUTMAX

return value:       =0               success
                    <0               error
```

Note: This can be used to trigger external light sources. Use with caution when triggering lasers: Software can fail.

int TH260_SetMeasControl (int devidx, int meascontrol, int startedge, int stopedge);

```
arguments:          devidx:          device index 0..3
                    meascontrol:     measurement control code
                                        0 = MEASCTRL_SINGLESHOT_CTC
                                        1 = MEASCTRL_C1_GATED
                                        2 = MEASCTRL_C1_START_CTC_STOP
                                        3 = MEASCTRL_C1_START_C2_STOP
                    startedge:       edge selection code
                                        0 = falling
                                        1 = rising
                    stopedge:        edge selection code
                                        0 = falling
                                        1 = rising

return value:       =0               success
                    <0               error
```

Note: This is a very specialized routine for externally (hardware) controlled measurements. Normally it is not needed. See section Fehler: Referenz nicht gefunden for details.

int TH260_StartMeas (int devidx, int tacq);

```
arguments:          devidx:          device index 0..3
                    tacq:            acquisition time in milliseconds
                                        minimum = ACQTMIN
                                        maximum = ACQTMAX

return value:       =0               success
                    <0               error
```

Note: This starts a measurement in the current measurement mode. Should be called after all settings are done. Previous measurements should be stopped before calling this routine again.

int TH260_StopMeas (int devidx);

```
arguments:          devidx:          device index 0..3

return value:       =0               success
                    <0               error
```

Note: This **must** be called after the acquisition time is expired. Can also be used to force stop before the acquisition time expires.

int TH260_CTCStatus (int devidx, int* ctcstatus);

```
arguments:          devidx:          device index 0..3
                    ctcstatus        pointer to an integer,
                                        returns the acquisition time state
                                        0 = acquisition running
                                        1 = acquisition has ended

return value:       =0               success
                    <0               error
```

Note: This routine should be called to determine if the acuisition time has expired.

int TH260_GetHistogram (int devidx, unsigned int *chcount, int channel, int clear);

```
arguments:          devidx:          device index 0..3
                    chcount          pointer to an array of at least actuallen double words (32bit)
                                        where the histogram data can be stored
                    channel:         input channel index 0..nchannels-1
                    clear            denotes the action upon completing the reading process
                                        0 = keeps the histogram in the acquisition buffer
                                        1 = clears the acquisition buffer

return value:       =0               success
                    <0               error
```

Note: The histogram buffer size `actuallen` must correspond to the value obtained through TH260_SetHistoLen(). The maximum input channel index must correspond to `nchannels-1` as obtained through TH260_GetNumOfInputChannels().

int TH260_GetResolution (int devidx, double* resolution);

```
arguments:          devidx:          device index 0..3
                    resolution:      pointer to a double precision float (64 bit)
                                        returns the resolution at the current binning
                                        (histogram bin width) in ps

return value:       =0               success
                    <0               error
```

Note: This is meaningful only in histogramming and T3 mode. T2 mode always runs at the boards's base resolution.

int TH260_GetSyncRate (int devidx, int* syncrate);

```
arguments:          devidx:          device index 0..3
                    syncrate:        pointer to an integer
                                        returns the current sync rate

return value:       =0               success
                    <0               error
```

Note: This is used to get the pulse rate at the sync input. The result is internally corrected for the current sync divider setting. Allow at least 100 ms after TH260_Initialize or TH260_SetSyncDivider to get a stable rate reading. Similarly, wait at least 100 ms to get a new reading. This is the gate time of the hardware counters.

int TH260_GetCountRate (int devidx, int channel, int* cntrate);

```
arguments:          devidx:          device index 0..3
                    channel:         number of the input channel 0..nchannels-1
                    cntrate:         pointer to an integer
                                        returns the current count rate of this input channel

return value:       =0               success
                    <0               error
```

Note: Allow at least 100 ms after TH260_Initialize to get a stable rate reading. Similarly, wait at least 100 ms to get a new reading. This is the gate time of the hardware counters. The maximum channel index must correspond to `nchannels-1` as obtained through TH260_GetNumOfInputChannels().

int TH260_GetFlags (int devidx, int* flags);

```
arguments:          devidx:          device index 0..3
                    flags:           pointer to an integer
                                        returns current status flags (a bit pattern)

return value:       =0               success
                    <0               error
```

Note: Use the predefined bit mask values in th260defin.h (e.g. FLAG_OVERFLOW) to extract individual bits through a bitwise AND.

int TH260_GetElapsedMeasTime (int devidx, double* elapsed);

```
arguments:          devidx:          device index 0..3
                    elapsed:         pointer to a double precision float (64 bit)
                                        returns the elapsed measurement time in ms

return value:       =0               success
                    <0               error
```

Note: During a measurememt this can be called to obtain the measurement time that has elapsed so far. After a measurement it will return the time that actually elapsed before the measurement was stopped (e.g. due to histogram overflow or forced stop).

int TH260_GetWarnings (int devidx, int* warnings);

```
arguments:          devidx:          device index 0..3
                    *warnings:       pointer to integer bitfield receiving the warnings

return value:       =0               success
                    <0               error
```

Note: You must call TH260_GetCoutRate and TH260_GetCoutRate for all channels prior to this call.

`int TH260_GetWarningsText (int devidx, char* text, int warnings);`

```
arguments:          devidx:           device index 0..3
                    text:             pointer to a buffer for at least 16384 characters
                    warnings:         integer bitfield obtained from TH260_GetWarnings

return value:       =0                success
                    <0                error
```

Note: This helps to identify suspicious measurement conditions that may be due to inappropriate settings.

`int TH260_GetHardwareDebugInfo (int devidx, char* text);`

```
arguments:          devidx:           device index 0..3
                    text:             pointer to a buffer for at least 16384 characters

return value:       =0                success
                    <0                error
```

note: Call this routine if you receive the error code TH260_ERROR_STATUS_FAIL or the flag FLAG_SYSERROR. See th260defin.h and errorcodes.h for the numerical values of these codes. Provide the result for support.

`int TH260_GetSyncPeriod(int devidx, double* period);`

```
arguments:          devidx:           device index 0..3
                    period:           pointer to a double precision float (64 bit)
                                         returns the sync period in seconds

return value:       =0                success
                    <0                error
```

note: As opposed to GetSyncRate this does not integrate over multiple periods. The period value is only useful in applications with periodic sync signals. In case of very long periods it takes a correspondingly long time to get a meaningful result. This time is increased to n-fold if a sync divider n is set.

7.2.4. Special Functions for TTTR Mode

`int TH260_ReadFiFo (int devidx, unsigned int* buffer, int count, int* nactual);`

```
arguments:          devidx:           device index 0..3
                    buffer:           pointer to an array of count double words (32bit)
                                         where the TTTR data can be stored
                    count:            number of TTTR records the buffer can hold
                                         (min = TTREADMIN, max = TTREADMAX)
                    nactual:          pointer to an integer
                                         returns the number of TTTR records received

return value:       =0                success
                    <0                error
```

Note: CPU time during wait for completion will be yielded to other processes / threads. The call will return after a timeout period of a few ms if no more data could be fetched. The buffer must not be accessed until the call returns.

`int TH260_SetMarkerEdges (int devidx, int me0, int me1, int me2, int me3);`

```
arguments:          devidx:           device index 0..3
                    me<n>:            active edge of marker signal <n>,
                                         0 = falling,
                                         1 = rising

return value:       =0                success
                    <0                error
```

int TH260_SetMarkerEnable (int devidx, int en0, int en1, int en2, int en3);

```
arguments:          devidx:          device index 0..3
                    en<n>:           desired enable state of marker signal <n>,
                                        0 = disabled,
                                        1 = enabled

return value:       =0               success
                    <0               error
```

int TH260_SetMarkerHoldoffTime (int devidx, int holdofftime);

```
arguments:          devidx:          device index 0..3
                    en<n>:           desired holdoff time for marker signals in nanoseconds
                                        min = 0,
                                        max = 25500

return value:       =0               success
                    <0               error
```

Note: Afer receiving a marker the system will suppress subsequent markers for the duration of `holdofftime` (ns). This can be used to suppress glitches on the marker signals. This is only a workaround for poor signals. Try to solve the problem at its root, i.e. the quality of marker source and cabling.

7.3. Warnings

The following is related to the warnings (possibly) generated by the library routine `TH260_GetWarnings`. The mechanism and warning criteria are the same as those used in the regular TimeHarp 260 software for Windows and depend on the current count rates and the current measurement settings.

Note that the software can detect only a subset of all possible error conditions. It is therefore not safe to assume "all is right" just by obtaining no warning. It is also necessary that `TH260_GetCoutrate` has been called for all channels before `TH260_GetWarnings` is called.

The warnings are to some extent dependent on the current measurement mode. Not all warnings will occur in all measurement modes. Also, count rate limits for a specific warning may be different in different modes. The following table lists the possible warnings in the three measurement modes and gives some explanation as to their possible cause and consequences.

Warning	Histo Mode	T2 Mode	T3 Mode
WARNING_SYNC_RATE_ZERO No pulses are detected at the sync input. In histogramming and T3 mode this is crucial and the measurement will not work without this signal.	√		√
WARNING_SYNC_RATE_VERY_LOW The detected pulse rate at the sync input is below 100 Hz and cannot be determined accurately. Other warnings may not be reliable under this condition.	√		√
WARNING_SYNC_RATE_TOO_HIGH The pulse rate at the sync input (after the divider) is higher than 40 MHz. Sync events will be lost in dead time. T2 mode is normally intended to be used without a fast sync signal and without a divider. If you see this warning in T2 mode you may accidentally have connected a fast laser sync.	√	√	√
WARNING_INPT_RATE_ZERO No counts are detected at any of the input channels. In histogramming and T3 mode these are the photon event channels and the measurement will yield nothing. You might sporadically see this warning if your detector has a very low dark count rate and is blocked by a shutter. In that case you may want to disable this warning.	√	√	√
WARNING_INPT_RATE_TOO_HIGH The overall pulse rate at the input channels is higher than 40 MHz. The measurement will inevitably lead to a FiFo overrun. There are some rare measurement scenarios where this condition is expected and the warning can be disabled. Examples are measurements where the FiFo can absorb all data of interest before it overflows.	√	√	√
WARNING_INPT_RATE_RATIO This warning is issued in histogramming and T3 mode when the rate at any input channel is higher than 5% of the sync rate. This is the classical pile-up criterion. It will lead to noticeable dead-time artefacts. There are rare measurement scenarios where this condition is expected and the warning can be disabled. Examples are antibunching measurements.	√		√

Warning	Histo Mode	T2 Mode	T3 Mode
WARNING_DIVIDER_GREATER_ONE In T2 mode: The sync divider is set larger than 1. This is probably not intended. The sync divider is designed primarily for high sync rates from lasers and requires a fixed pulse rate at the sync input. In that case you should use T3 mode. If the signal at the sync input is from a photon detector (coincidence correlation etc.) a divider > 1 will lead to unexpected results. There are rare measurement scenarios where this condition is intentional and the warning can be disabled. In histogramming and T3 mode: The pulse rate at the sync input is below 40 MHz and the Sync-Divider >1 is not needed. The measurement may yield unnecessary jitter if the sync source is not very stable.	√	√	√
WARNING_DIVIDER_TOO_SMALL The pulse rate at the sync input (after the divider) is higher than 40 MHz and Sync events will be lost in dead time. Increase the sync divider.	√		√
WARNING_TIME_SPAN_TOO_SMALL This warning is issued in histogramming and T3 mode when the sync period (1/SyncRate) is longer than the start to stop time span that can be covered by the histogram or by the T3 mode records. You can calculate this time span as follows: Span = Resolution * 32768 Events outside this span will not be recorded. There are some measurement scenarios where this condition is intentional and the warning can be disabled.	√		√
WARNING_OFFSET_UNNECESSARY This warning is issued in histogramming and T3 mode when an offset >0 is set even though the sync period (1/SyncRate) can be covered by the measurement time span without using an offset. The offset may lead to events getting discarded. There are some measurement scenarios where this condition is intentional and the warning can be disabled.	√		√
WARNING_COUNTS_DROPPED This warning is issued when the front end of the data procesing pipeline was not able to process all events that came in. This will occur typically only at very high count rates during intense bursts of events.	√	√	√

If any of the warnings you receive indicate wrong pulse rates, the cause may be inappropriate input settings, wrong pulse polarities, slow rise times, poor pulse shapes or bad connections. If in doubt, check all signals with an oscilloscope of sufficient bandwidth.

PicoQuant GmbH
Unternehmen für optoelektronische Forschung und Entwicklung
Rudower Chaussee 29 (IGZ), 12489 Berlin, Germany
Telephone: +49 / (0)30 / 6392 6929
Fax: +49 / (0)30 / 6392 6561
e-mail: info@picoquant.com
WWW: http://www.picoquant.com