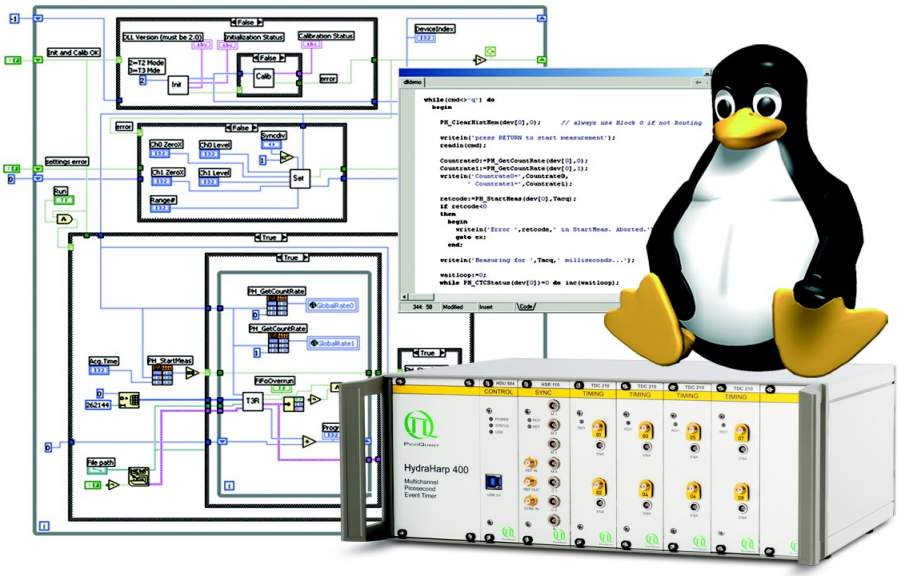
HydraHarp 400

Picosecond Histogram Accumulating
Real–time Processor

PicoQuant

HHLib Programming Library
for Custom Software Development
under Linux



User's Manual

Version 3.0.0.4 – April 2022

Table of Contents

1. Introduction

The HydraHarp 400 is a high-end but easy–to–use TCSPC system with USB interface. Its modular design makes it scalable and very flexible. The timing circuits allow high measurement rates up to 12.5 Mcounts/s per channel and provide a time resolution of 1 ps. The input channels are programmable for a wide range of input signals. They all have programmable Constant Fraction Discriminators (CFD) and a programmable input offset replacing cumbersome cable delays. These specifications qualify the HydraHarp 400 for use with all common single photon detectors such as Single Photon Avalanche Photodiodes (SPAD), Photo Multiplier Tubes (PMT), Micro-Channel Photo Multiplier Tubes (MCP–PMT), Hybrid Photodetectors (HPD) and Superconducting Nanowire Single Photon Detectors (SNSPD). The time resolution is well matched to these detectors and the overall Instrument Response Function (IRF) will typically not be limited by the HydraHarp electronics. Similarly inexpensive and easy–to–use diode lasers such as the PDL 800–B with interchangeable laser heads can be used as an excitation source perfectly matched to the time resolution offered by the detector and the electronics. Overall IRF widths down to 50 ps can be achieved with selected diode lasers and MCP–PMT detectors. Even 30 ps can be reached with femtosecond lasers. This permits lifetime measurements down to a few picoseconds with deconvolution e.g. via the FluoFit multi–exponential Fluorescence Decay Fit Software. For more information on the HydraHarp 400 hardware and software please consult the HydraHarp 400 manual. For details on the method of Time–Correlated Single Photon Counting, please refer to our Technical Note on TCSPC.

The HydraHarp 400 standard software for Windows 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 HydraHarp'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. As an alternative, a Linux version of the library is offered, which is subject of this manual. It is 100% API compatible with the Windows library, so that applications can be ported very easily.

The shared library '`hhlib.so`' for Linux supports custom programming in all major programming languages, notably C / C++, C#, Python, FreePascal/Lazarus, LabVIEW and MATLAB. This manual describes the installation and use of the HydraHarp programming library `hhlib.so` and explains the associated demo programs. Please read both this manual and the HydraHarp manual before beginning your own software development with the library. The HydraHarp 400 is a sophisticated real–time measurement system. In order to work with the system using the library, sound knowledge in your chosen programming language is required.

2. General Notes

This version of the HydraHarp 400 programming library is suitable for Linux kernel versions 4.0. and higher, running on the x64 platform. It uses Libusb so that no kernel driver is required. Note that the library is provided as a binary object only. The version for 32-bit platforms has been dropped because the major Linux distributions are no longer supporting it.

This manual assumes that you have read the HydraHarp 400 manual and that you have experience with the chosen programming language. References to the HydraHarp manual will be made where necessary.

The library supports histogramming mode and both TTTR modes.

Users who own a license for any older version of the library will receive free updates when they are available. Please check our website for the latest version before you begin programming.

Users upgrading from earlier versions of the HydraHarp 400 library or moving over from the Windows DLL may need to adapt their programs. Some changes are usually necessary to accommodate new measurement modes and improvements. However, the required changes are mostly minimal and will be explained in the manual (especially check section 7 and any notes marked red in section 8.2).

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 8.2). Note that this call returns only the major two digits of the version (e.g. 3.0). The DLL actually has two further sub–version digits, so that the complete version number has four digits (e.g. 3.0.0.4). They are shown only in the Windows file properties. These sub–digits help to identify intermediate versions that may have been released for minor updates or bug fixes. The interface of releases with identical major version will remain the same.

Please note that the development of Linux is highly dynamic and distributions may vary considerably. This manual can therefore only try and describe a snapshot valid at the time of writing.

2.1. What's new in this Version

The new version 3.0.0.4 of the HHLib package for Linux provides some more advanced demos in C, C#, Python and Delphi, most importantly some new demos for instant processing of TTTR data streams (see section 6.4).

2.2. 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 this product you have purchased a license to use the HydraHarp 400 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 online documentation belongs to PicoQuant GmbH. No parts of it may be reproduced, translated or transferred to third parties without written permission of PicoQuant GmbH.

Acknowledgements

The HydraHarp programming library for Linux uses Libusb to access the HydraHarp USB devices. Libusb is licensed under the LGPL which allows a fairly free use even in commercial projects. For details and precise terms please see http://libusb.info. In order to meet the license requirements a copy of the LGPL as applicable to Libusb is provided as part of the distribution archive. The LGPL does not apply to the HydraHarp programming library as a whole.

For this version of the library we also gratefully acknowledge the use of GNU/Linux as a development platform, as well as using the Tux logo (thanks to Larry Ewing, lewing@isc.tamu.edu and The GIMP) on the title page of this manual.

3. Firmware Update

This version of `HHLib` requires a firmware update of your HydraHarp hardware if the device was purchased before August 2012 and if it was not yet upgraded. The regular HydraHarp software for Windows can perform this update. When it is started it will check the firmware version and prompt for an upgrade if necessary.

The firmware update has important consequences that you must observe:

1. After the update you will no longer be able to use any HydraHarp software prior to version 2.0.
2. Custom software you may have written for 1.x file import will require minor adaptions.
3. You will no longer be able to use custom software based on HHLib prior to version 2.0.
4. Custom HHLib-based software you have written for 1.x will require minor adaptions.
5. After the update you may need small CFD adjustments for any known good setup.
6. In case of a power failure or computer crash during the update the device may become in-operational.
7. Reverting to old firmware or repairing a disrupted update requires a return to factory and may incur costs.

Also note: When the firmware update has been performed the device must be switched off an on again in order to become operational with the new software.

4. Installation of the HHLib Software Package

4.1. Requirements

Supported hardware is at this time solely the “x86-64” CPU platform as found in the majority of recent PCs. Support for 32-bit platforms has been dropped, for the simple reason that all major Linux distributions are no longer supporting it. Required is a PC with USB 3.0, at least two CPU cores, 1.5 GHz CPU clock and 4 GB of memory. For optimal TTTR mode throughput to disk a fast solid state disk is recommended.

The library is designed to run on Linux kernel versions 4.0 or higher. It has been tested with the following distributions:

Linux Mint 19.3 (kernel 5.3.0)
Linux Mint 20.2 (kernel 5.4.0)
Ubuntu 20.04 (kernel 5.13.0)

Using the library requires libusb (http://libusb.info). The formally required version is 1.0 or higher, tested versions were 1.0.20, 1.0.21 and 1.0.23. Libusb is typically installed by default on all major Linux distributions.

It is recommended to start your work with the HydraHarp 400 by using the standard interactive HydraHarp data acquisition software under Windows. This should give you a better understanding of the instrument’s operation before attempting your own programming efforts. It also ensures that your optical/electrical setup is working.

4.2. Libusb Access Permissions

For device access through libusb, your kernel needs support for the USB filesystem (usbfs) and that filesystem must be mounted. This is done automatically, if /etc/fstab contains a line like this:

```
usbfs /proc/bus/usb usbfs defaults  0  0
```

This should routinely be the case if you installed any of the tested distributions. The permissions for the device files used by libusb must be adjusted for user access. Otherwise only root can use the device(s). The device files are located in `/proc/bus/usb/`. Any manual change would not be permanent, however. The permissions will be reset after reboot or re-plugging the device. A much more elegant way of finding the right files and setting the suitable permissions is by means of hotplugging scripts or udev. Which mechanism you can use depends on the Linux distribution you have. Most of the recent distributions use udev.

For automated setting of the device file permissions with udev you have to add an entry to the set of rules files that are contained in `/etc/udev/rules.d`. Udev processes these files in alphabetical order. The default file is usually called `50-udev.rules`. Don't change this file as it could be overwritten when you upgrade udev. Instead, put your custom rule for the HydraHarp in a separate file. The contents of this file for the handling of the current HydraHarp 400 models should be:

```
ATTR{idVendor}=="0e0d", ATTR{idProduct}=="0004", MODE="666"
ATTR{idVendor}=="0e0d", ATTR{idProduct}=="0009", MODE="666"
```

A suitable rules file `HydraHarp.rules` is provided in the folder `udev` on the distribution media. You can simply copy it to the `/etc/udev/rules.d` folder. The install script in the same distribution media folder does just this. Note that the name of the rules file is important. Each time a device is detected by the udev system, the files are read in alphabetical order, line by line, until a match is found. Note that different distributions may use different rule file names for various categories. For instance, Kubuntu organizes the rules into further files: `20-names.rules`, `40-permissions.rules`, and `60-symlinks.rules`. In Fedora they are not separated by those categories, as you can see by studying `50-udev.rules`. Instead of editing the existing files, it is therefore usually recommended to put all of your modifications in a separate file

like `10-udev.rules` or `10-local.rules`. The low number at the beginning of the file name ensures it will be processed before the default file. However, later rules that are more general (applying to a whole class of devices) may later override the desired acecss rights. This is the case for USB devices handled through Libusb. It is therefore important that you use a rules file for the HydraHarp that gets evaluated after the general case. The default naming `HydraHarp400.rules` most likely ensures this but if you see problems you may want to check.

Note that there are different udev implementations with different command sets. On some distributions you must reboot to activate changes, on others you can reload rule changes and restart udev with these commands:

```
# udevcontrol reload_rules
# udevstart
```

4.3. Installing the Library

The library package is distributed as a gzipped tar file. The shared library as such is provided as a binary file. By default it resides under `/usr/local/lib/hh400` or under `/usr/local/lib64/hh400` dependent on the Linux distribution. 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. You can create the directory `/usr/local/lib/hh400` and copy all files from the directory `library` *from* the tar archive to `/usr/local/lib/hh400` (`hhlib.so`, `hhlib.h`, `hhdefin.h` and `errorcodes.h`).

The shell script `install` in the library distribution directory does the directory creation and installation in one step. As root, just run it at the command prompt from within the `library` directory. The install script also takes care of the peculiarities of some programming tools that expect library names to begin with `lib` and the case of some Linux distributions where the x64 library paths use `usr/lib/` instead of `usr/lib64`. This is done by creating a symbolic link rather than copying the library to different places. Note that the related conventions seem to vary: lately on Ubuntu the library was not found by mono and it was necessary to use `usr/lib/` despite the existence of `usr/lib64`.

After installing, the library is ready to use and can be tested with the demos provided. On some distributions you may need to adjust the library path and/or access permissions. 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 `hhlib.so` you get an error *"cannot restore segment prot after reloc"* you need to adjust the security settings for `hhlib.so`. As root you need to run:

```
chcon -t texrel_shlib_t /usr/local/lib/hh400/hhlib.so
```

4.4. Installing the Demo Programs

The demos can be installed by simply copying the entire directory `demos` from the tar archve to a disk location of your choice. This need not be under the root account but you need to ensure proper file permissions. While the gcc compiler for the C demos is readily installed in most linux distributions, you will need to obtain and install Mono, Python, Lazarus, Matlab or LabVIEW for Linux separately if you wish to use these programming environments.

5. The Demo Applications

5.1. 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 explanatory purposes and a starting point for your own work.

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 must be run from the command line (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 may result in useless data (or none at all) because of inappropriate sync divider, resolution, input level settings, etc.

For the same reason of simplicity, the demos will always only use the first HydraHarp 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 (see section 6.1).

There are demos for C/C++, C#, Pascal/Lazarus, Python, 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 already fairly sophisticated and allows interactive input of most parameters. In some programming languages (C, C#, Python, Pascal) there are also advanced demos to show hardware triggered measurements.

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 extremely sophisticated 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 HydraHarp 400 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 the HydraHarp manual and our Technical Note on TTTR. Observe the mode input variable going into `HH_Initialize`. It makes a difference if you run T2 or T3 mode. For instance, T2 mode will not allow you to work with high sync rates. For meaningful measurements you will need to adjust the sync divider and the resolution (binning) dependent on your choice of mode.

In TTTR mode it is also possible to record external TTL signal transitions as markers in the TTTR data stream (see the HydraHarp manual) which is typically used for imaging applications.

Because TTTR mode requires real–time processing and / or real–time storing of data, the TTTR demos are fairly demanding both in programming skills and computer performance. See the section about TTTR mode in your HydraHarp manual.

Note that you must not call any of the HH_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 inconsistency.

Details on how to interpret and process the TTTR records can be studied in the advanced LabVIEW demos and in the advanced demos `tttrmode_instant_processing` (C, Python, Delphi, C#). You may also consult the file demo code installed together with the regular HydraHarp software.

Continuous Mode Demos

This measurement mode allows continuous and gapless streaming of short term histograms to the PC. Since it is an advanced real-time technique, beginners are advised better not to use it for their first try. For the same reason, demos exist only in some languages. For details please see the Advanced Techniques section 6.7.

5.2. The Demo Applications by Programming Language

As outlined above, there are demos for C / C++, Pascal/Lazarus, Python, 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 HydraHarp 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 HydraHarp programming library as your first attempt at programming. You will require knowledge on how to call and link shared libraries from your programming language. The ultimate reference for details about how to use the library is in any case the source code of the demos and the header files of HHLib (`hhlib.h` and `hhdefin.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. The latter is quite unlikely, nevertheless, 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 re-entrant. This means, it cannot be accessed from multiple, concurrent processes or threads at the same time. All calls must be made sequentially and in meaningful order, as 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 `hhlib.h, hhdefin.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 "hhdefin.h"
  #include "hhlib.h"
}
```

To test any of the demos, consult the HydraHarp manual for setting up your HydraHarp 400 and establish a measurement setup that runs correctly and generates useable test data. This is best done with the regular HydraHarp software for Windows. 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 multiple columns of integer numbers representing the counts from the 65,536 histogram channels. 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 demos (provided by way of the regular HydraHarp software installation) for reading the HydraHarp TTTR data files (.PTU) and the advanced demos `tttrmode_instant_processing` can be used as a starting point to learn this. The file read demos 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 HHLib demos focused on the key issues of using the library.

The C# Demos

The C# demos are provided in the `Csharp` subfolder. They have been tested with MS Visual Studio (Windows) as well as with Mono under Windows and Linux. The only difference is the library name, which in principle could also be unified, e.g. by means of a symbolic link.

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.

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 mode demos. In 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 4096 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 file demos (provided by way of the regular HydraHarp software installation) for reading the HydraHarp TTTR data files (.PTU) and the advanced demos `tttrmode_instant_processing` can be used as a starting point to learn this. The file read demos 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 HHLib demos focused on the key issues of using the library.

The Python Demos

The Python demos are provided in the `Python` subfolder. They have been developed with Python 3 and only this is officially supported. Python 2 works at the time of this writing but will not be actively maintained.

With the Python 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 mode 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 4096 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/Lazarus users refer to the 'Pascal' directory. The source code for Delphi (under Windows) and Lazarus (Windows or Linux) is the same, differences regarding the library name are handled automatically. The main code for the Lazarus demo is in the Delphi project file for that demo (`*.DPR`). Lazarus users use the `*.LPI` files that refer to the corresponding `*.DPR` file.

In order to make the exports of `hhlib.so` 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 in `hhlib.pas`. Please check the function parameters of your code against `hhlib.h` in the demo directory whenever you update to a new library version.

Lazarus is somewhat picky as to what it accepts as a library name. It expects a name starting with `lib`. It also does not easily allow linking with a library that is not in the default path. In order to solve both problems, the install script for HHLib creates a link making it also appear under the alias `/usr/lib/libhh400.so` or `/usr/lib64/libhh400.so` respectively. This is the path the Lazarus demos use for accessing `hhlib.so`. Another complication arising with Lazarus is that it does not always by default know how to link in the C runtime library needed by `hhlib.so`. This is why you must explicitly enter it in the Compiler Options. Under Linking in the field Options you should enter something like `/lib/libgcc_s.so.1` (or `/lib64/libgcc_s.so.1` for x64) and tick the box "Pass options to linker...". The loction of `libgcc_s.so.1` may be different in some recent Linux distribution so you may need to search for `libgcc_s.so.1` and set the path accordingly.

The 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 demos (provided by way of the regular HydraHarp software installation) for reading the HydraHarp TTTR data files (.PTU) and the advanced demos `tttrmode_instant_processing` can be used as a starting point to learn this. The file read demos 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 HHLib demos focused on the key issues of using the library.

The LabVIEW Demos

The LabVIEW demo VIs are provided in the *src* sub-folder inside the `LabVIEW20xx` folders. The correct

library path is selected automatically, provided that the library is is installed in the designated folders (i.e. `usr/local/lib` or `usr/local/lib64` respectively). The demo code was created with LabVIEW 2020, for backward compatibility the source code was also converted to LabVIEW 2010.

Program specific SubVIs and type-definitions used by the demos are organized in corresponding sub-folders inside the demo folder (*SubVIs*, *Types*). General helper functions and type-definitions as well as encapsulating libraries (`*.llb`) can be found in the `_lib` folder (containing further sub-folders) inside the demo folder. In order to facilitate the usage of the DLL functions additional VIs called `HH_AllDllFunctions_xxx.vi` have been included. These VIs are not meant to be executed but should only give a structured overview of all available DLL functions and their functional context.

In addition to the LabVIEW library used by the demos (`hhlib_x86_x64_UIThread.llb`) a second library is included allowing the DLL calls to be executed in any thread of LabVIEWs threading engine (`hhlib_x86_x64_AnyThread.llb`). This library is intended for time critical applications where user actions on the Front Panel (like e.g. resizing or moving) must not affect the execution of a data acquisition thread containing these DLL functions (please refer to "Multitasking in LabVIEW": `http://zone.ni.com/reference/en-XX/help/371361P-01/lvconcepts/multitasking_in_labview/`). When using this library you have to make sure that all DLL functions are called in a sequential order to avoid errors or even program crashes. Also be aware that DLL functions in `hhlib_x86_x64_AnyThread.llb` have the same names like in `hhlib_x86_x64_UIThread.llb` and opening both libraries at the same time would lead to name conflicts. Therefore only experienced users should use `hhlib_x86_x64_AnyThread.llb`.

The first demo (`1_SimpleDemo_HHHisto.vi`) is a very simple one demonstrating the basic usage and calling sequence of the provided SubVIs encapsulating the DLL functionality, which are assembled inside the LabVIEW library `hhlib_x86_x64_UIThread.llb` (in the folder `_lib/PQ/HydraHarp`). The demo starts by calling some of these library functions to setup the hardware in a defined state and continues with a measurement in histogramming mode by calling the corresponding library functions inside a while-loop. Histograms and count rates for all available hardware channels are displayed on the Front Panel in a Waveform Graph (you might have to select *AutoScale* for axes) and numeric indicators, respectively. The measurement is stopped if either the acquisition time has expired, if an error occurs (which is reported in the *error out* cluster), if an overflow occurs or if the user hits the *STOP* button.

The second demo for histogramming mode (`2_AdvancedDemo_HHHisto.vi`) is a more sophisticated one allowing the user to control all hardware settings "on the fly", i.e. to change settings like acqisition time (*Acqu. ms*), resolution (*Resol. ms*), offset (*Offset ns* in *Histogram* frame), number of histogram bins (*Num Bins*), etc. before, after or while running a measurement. In contrast to the first demo settings for each available channel (including the Sync channel) can be changed individually (*Settings* button) and consecutive measurements can be carried out without leaving the program (*Run* button; changes to *Stop* after pressing). Additionally, measurements can be done either as "single shot" or in a continuous manner (*Conti*. Checkbox). Various information are provided on the Front Panel like histograms and count rates for each available (and enabled) channel as Waveform Graphs (you might have to select *AutoScale* for axes), Sync rate, readout rate, total counts and status information in the status bar, etc. In case an error occurs a popup window informs the user about that error and the program is stopped.
The program structure of this demo is based upon the National Instruments recommendation for queued message and event handlers for single thread applications. Some comments inside the source code should help the user to get an overview of the program and to facilitate the development of customized extensions.

The third demo (`3_AdvancedDemo_HHT3.vi`) is the most advanced one and demonstrates the usage of T3 mode. The Front Panel resembles the second demo but in addition to the histogram display a second Waveform Graph (you might have to select *AutoScale* for axes) also displays a time chart of the incoming photons for each available (and enabled) channel with a time resolution depending on the Sync rate and the entry in the *Resol. ms* control inside the *Time Trace* frame (which can be set in multiples of two). In contrast to the second demo there is no control to set an overflow level or the number of histogram bins, which is fixed to 32.768 in T3 mode. Also in addition to the acquisition time (called *T3Acq. ms* in this demo; set to 360.000.000 ms = 100 h by default) a second time (*Int.Time ms* in *Histogram* frame) can be set which controls the integration time for accumulating a histogram.
The program structure of this demo extends that of the second demo by extensive use of LabVIEW type-definitions and two additional threads: a data processing thread (`HH_DataProcThread.vi`) and a visualization thread. The communication between these threads is accomplished by LabVIEW queues. Thereby the FiFo read function (case *ReadFiFo* in *UIThread*) can be called as fast as possible without any

additional latencies from data processing workload.
Some comments inside the source code should help the user to get an overview of the program and to facilitate the development of customized extensions. Please note that due to performance reasons some of the SubVIs inside `HH_DataProcThread.vi` have been inlined so that no debugging is possible on these SubVIs.
This demo can also be used as a starting point to calculate the absolute arrival times of photons (e.g. for correlation analysis). To achieve this one has to multiply the sync event 'numsync' with the known sync period and add 'dtime' multiplied with the channel resolution (refer to SubVIs *`HH_ProcData.vi`* and *`ProcessTTRecHHT3.vi`* within the SubVI *`HH_DataProcThread.vi`*).

For example:

Sync frequency: e.g. 10 MHz
=> sync period: 100 ns
Number of sync event (i.e. value of 'numsync'): e.g. 1000
=> time of sync event: 1000 x 100 ns = 100000 ns = 100000000 ps
Resolution of channels: e.g. 4 ps (in case of binning = 4)
=> time difference between sync event and detection of photon: 'dtime' x 4 ps
Arrival time of detected photon
=> 100000000 ps + ('dtime' x 4 ps)

The MATLAB Demos

The MATLAB demos are provided in the `MATLAB` directory. They are contained in `.m` files. You need to have a MATLAB version that supports the `calllib` function. The earliest version we have tested in this regard is MATLAB 7.3 (under Windows) but any version from 6.5 should work. Be very careful about the header file name specified in 'loadlibrary'. This name is 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 demos (provided by way of the regular HydraHarp software installation) for reading the HydraHarp TTTR data files (.PTU) and the advanced demos `tttrmode_instant_processing` can be used as a starting point to learn this. The file read demos 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 HHLib demos focused on the key issues of using the library.

6. Advanced Techniques

6.1. Using Multiple Devices

The library is designed to work with multiple HydraHarp devices (up to 8). The demos always use the first device found. If you have more than one HydraHarp and you want to use them together you need to modify the code accordingly. At the API level of HHLib the devices are distinguished by a device index (0 .. 7). The device order corresponds to the order Libusb enumerates the devices. This can be somewhat unpredictable. 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 `HH_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 the library routine `HH_GetSerialNumber` any time later on a device you have successfully opened and initialized. The serial number of a physical HydraHarp device can be found at the back of the housing. It is a 8 digit number starting with 0100. The leading zero will not be shown in the serial number strings retrieved through `HH_OpenDevice` or `HH_GetSerialNumber`.

It is important to note that the list of devices may have gaps. If you have e.g. two HydraHarps you cannot assume to always find device 0 and 1. They may as well appear e.g. at device index 2 and 4 or any other index. Such gaps can be due to other PicoQuant devices (e.g. a Sepia II laser device) occupying some of the indices, as well as due to repeated unplugging / replugging of devices. The only thing you can rely on is that a device you hold open remains at the same index until you close or unplug it.

Note that an attempt at opening a device that is currently used by another process will result in the error code `ERROR_DEVICE_BUSY` being returned from `HH_OpenDevice`.

As outlined above, if you have more than one HydraHarp 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 the 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.

6.2. 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 HydraHarp 400 uses USB bulk transfers. This is supported by the PC hardware that can transfer data to the host memory without much help of the CPU. For the HydraHarp this permits data throughput as high as 9 Mcps (USB 2.0) or even 40 Mcps (USB 3.0) 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 `HH_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 circumsances be ideal to use the maximum size, e.g. with respect to cache usage.

As noted above, the transfer is implemented efficiently without using the CPU excessively. Nevertheless, assuming large block sizes, the transfer takes some time. The operating system 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 best way of doing this is to use multithreading. In this case you

design your program with two threads, one for collecting data (i.e. working with `HH_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 it cannot be demonstrated in detail here. Greatest care must be taken not to access HHLib from different threads without strict control of mutual exclusion and maintaining the right sequence of function calls. However, the technique allows throughput improvements of 50..100% 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 HydraHarp software for Windows, where sustained count rates over 9 millions of counts/sec (to disk) can be achieved with the USB 2.0 HydraHarp and even 40 Mcps with the USB 3.0 version.

When working with multiple HydraHarp devices, the overall USB throughput is limited by the host controller or any hub the devices must share. You can increase overall throughput if you connect the individual devices to separate host controllers without using hubs. If you install additional USB controller cards you should prefer PCI-express models. Traditional PCI can become a bottleneck in itself. However, modern mainboards often have multiple USB host controllers, so you may not even need extra controller cards. 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.

6.3. 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. It would therefore seem reasonable to always use the largest possible size. However, it may not under all circumstances be ideal to use the maximum size. A large block size takes longer to fill. If you use an early HydraHarp model with USB 2.0 and 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. The HydraHarp model with USB 2.0 has a “watchdog” timer that terminates large transfer requests prematurely so that they do not wait forever if new data is coming in very slowly or not at all. The timeout period is approximately 10 ms. `HH_ReadFiFo` may therefore return 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 experiment with smaller block sizes.

The USB 3.0 model of the HydraHarp works slightly differently. It transfers data in chunks of 128 records and `HH_ReadFiFo` will return immediately after the number of complete chunks of 128 records that were available in the FiFo and can fit in the buffer have been transferred. Remainders smaller than the chunk size are only transferred when no complete chunks are in the FiFo. Due to this different concept of operation of the USB 3.0 model it is therefore not necessary to worry about the transfer block size and using the largest is always fine unless you care about subtle cache optimizations.

Regardless of the hardware model the requested block size must be a multiple of 128 records. The smallest is therefore 128.

6.4. Instant TTTR Data Processing

As outlined earlier, collecting TTTR mode streams is time critical when data rates are high. This is why such streams are often just written to disk and then only subsequently post-processed. Nevertheless there are circumstances where it is desirable to process the data instantly “on the fly” as it arrives. This may be for the purpose of an instant preview or for data reduction. The advanced LabVIEW demo nicely demonstrates how to obtain an instant preview. This requires interpreting and bitwise dissecting the TTTR data records as well as correcting for overflows. In order to demonstrate this also for other programming languages there are advanced demos in the subfolders `tttrmode_instant_processing` (C, Python, Delphi, C#). These demos do not write binary output but instead perform an instant processing and write the results out in ASCII. Please note well that this is done purely for educational purposes. Instant processing and writing the results out in ASCII is time consuming and dramatically reduces the achievable troughput. Furthermore, the resulting files are many times larger than the original binary data. Any meaningful application derived from these demos should therefore not write out individual data records but perform some sort of application specific data analysis for preview and/or data reduction. Typical and meaningful examples are histogramming (see subfolders `t3rmode_instant_histogramming` in C, Python, Delphi and C#) or intensity over time traces as shown in the LabVIEW demo. Please note also that such real-time processing requires a suitable choice of program-

ming language. For instance, interpreted Python and Matlab scripts are many times slower than natively compiled code. Ultimate performance is obtained only with a proper compiled language such as C or Pascal. Furthermore, true efficiency (and maximum throughput) can in such a scenario only be achieved by making use of parallel processing on multiple CPUs. This requires programming with multiple threads. In this case you should design your program with at least two threads, one for collecting the data (i.e. working with `HH_ReadFiFo`) and another (or more) for processing, displaying, or storing the data (see also section 6.2). This is not trivial and requires some programming experience.

If you need quick results and your throughput requirements are moderate you may still try and work with the code from the demos in the subfolders `tttrmode_instant_processing`. For understanding the mechanisms they are worth studying anyhow. Looking at the code you will see that after retrieving a block of TTTR records via `HH_ReadFiFo` there is a loop over that block with code to dissect each individual record. Dependent on what kind of record it is, there will be different actions taken. A “special record” carries information on overflows and markers while a regular event record carries photon timing data. While overflows will typically not be of further interest (except correcting for them as shown) the pieces of interest are markers and photons. When they occur you notice the calls into the subroutines GotMarker and GotPhoton (with variants for T2 and T3 mode). These are the points where you may want to accommodate you application specific code for whatever you may want to do with a photon or a marker. In your derived code you may soon want to throw out the ASCII output for each an every record. It is only there for demonstration purposes.

6.5. Working with Warnings

The HydraHarp programming 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 HydraHarp software. In order to obtain and use these warnings also in your custom software you may want to use the library routine `HH_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 `HH_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 `HH_GetSyncRate` as well as `HH_GetCountRate` (for all channels) have been called before `HH_GetWarnings` is called. Since most interactive measurement software periodically retrieves the rates anyhow, this is not a serious complication.

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

6.6. Hardware Triggered Histogram Measurements

This measurement scheme requires a HydraHarp 2.0, i.e. Gateware 2.x. It 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 better not to use it for their first exercises. For the same reason, demos exist only in some of the programming languages. Before using this scheme, also consider when it is useful to do so. 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. Also consider using continuous mode (see section 6.7) if you care about repeated and gapless histogram recoding.

The HydraHarp's data acquisition can be controlled in various ways. Default is the HydraHarp's internal CTC (counter timer circuit). In that case the histograms will take the duration set by the `tacq` parameter passed to `HH_StartMeas`. The other way of controlling the histogram boundaries (in time) is by external TTL signals fed to the control input 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. The exact behaviour is controlled by the parameters supplied to the call of `HH_SetMeasControl`. Dependent on the parameter `meascontrol` the following modes of operation can be selected:

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 `HH_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 `HH_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 `hhdefin.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. Be aware that the speed of you computer and the delays introduced by the operating system's task switching impose some limits on how fast you can run this scheme.

6.7. Working in Continuous Mode

This measurement mode works essentially like regular histogramming mode but it allows continuous and gapless streaming of short term histograms to the PC. Since it is an advanced real-time technique, beginners are advised better not to use it for their first experiments. for the same reason, the corresponding demos exist only in some of the programming languages.

Before using this mode, 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. Only when the expected count rates become very high and when the individual photon timing relations are not of interest it may be advisable to switch to continuous mode.

The temporal boundaries of the individual histograms in a continuous mode stream can be controlled in two different ways. One is by the HydraHarp's internal CTC (counter timer circuit). In that case the histograms will take the duration set by the `tacq` parameter passed to `HH_StartMeas` and they will line up seamlessly in time. The other way of controlling the individual histogram boundaries (in time) is by external TTL signals fed to the connectors C1 and C2. In that case it is possible to have new histograms 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 `HH_SetMeasControl`. Dependent on the parameter measconrol the folowing modes of operation can be selected:

Symbolic Name	Value	Function
MEASCTRL_CONT_C1_GATED	4	Histograms are collected for each period where C1 is active. This can be the logical high or low periods dependent on the value supplied to the parameter `startedge`.

MEASCTRL_CONT_C1_START_CTC_STOP	5	Histogram 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`. Histogram duration is set by the `tacq` parameter passed to `HH_StartMeas`. The current histogram ends if a new trigger occurs before the CTC has expired.
MEASCTRL_CONT_CTC_RESTART	6	Histogram collection is started and stopped exclusively by the internal CTC. Consecutive histograms will line up without gaps. Histogram duration is set by the `tacq` parameter passed to `HH_StartMeas`.

The symbolic constants shown above are defined in `hhdefin.h`. You will find that there are more symbolic constants with values 0..3 not shown in the table above. These are reserved for use with regular histograming mode. There are also symbolic constants for the parameters controlling the active edges (rising/falling). Regarding `HH_StartMeas` only the parameter `startedge` is meaningful in continuous mode. The parameter `stopedge` is only used in regular histograming mode.

In continuous mode each histogram is retrieved as a structured data block via `HH_GetContModeBlock`. This data block has a small header that provides a number of the histogram, starting time and duration of that histogram in nanoseconds, First occurence time of each marker signal, and occurence count of each marker signal. Please see the HydraHarp manual for more information on markers (normally used in TTTR mode) and study the source code of the continuous mode demos for details on the continuous mode data block structure. After the header there is the actual histogram data and after that follows a sum of all counts in that histogram. The latter is useful in applications where mere intensity dynamics are of interest.

One complication in using these data structures is that the size of the histograms depends both on the chosen number of time bins and on the number of active input channels. The size of the data blocks is therefore not fixed. The demo code shows how to deal with this. Note that the structure of the continuous mode block header has slightly changed between HHLib versions 2.x and 3.x.

7. Problems, Tips & Tricks

7.1. PC Performance Issues

Performance issues with HHLib are the same as with the standard HydraHarp software. The HydraHarp 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 fairly modern CPU and sufficient memory are required. At least a 1.5 GHz dual core processor, 4 GB of memory and a fast hard disk are recommended. However, as long as you do not use TTTR or continuous mode, these issues should not be of severe impact. If you do intend to use TTTR or continuous mode with streaming to disk you should also have a fast modern hard disk, ideally a solid state disk.

7.2. USB Interface

In order to deliver maximum throughput, the HydraHarp 400 uses state–of–the–art USB bulk transfers. This is why the HydraHarp must rely on having a USB host interface matched to the device speed (USB 2.0 or USB 3.0). USB host controllers of modern PCs are usually integrated on the mainboard. For older PCs they may be upgraded as plugin cards.

Do not run other bandwidth demanding devices on the same USB interface when working with the HydraHarp. USB cables must be qualified for the USB speed of your HydraHarp model (at least USB 2.0). Older and cheap cables often do not meed this requirement and can lead to errors and malfunction. Similarly, many PCs have poor internal USB cabling, so that USB sockets at the front of the PC are often unreliable. Obscure USB errors may also result from subtle damages to USB cables, caused e.g. by sharply bending or crushing them. Worn out and dirty cable connectors are another frequent source of trouble.

7.3. Troubleshooting

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

Under Linux the library will access the HydraHarp device through Libusb 1.0 (see http://libusb.info). You need to make sure that libusb has been installed correctly. Normally this is readily provided by recent Linux distributions. You can use `lsusb` to check if the device has been detected and is accessible. Please consult the HydraHarp manual for hardware related problem solutions. Note that an attempt at opening a device that is currently used by another process will result in the error code `ERROR_DEVICE_BUSY` being returned from `HH_OpenDevice`. `HH_OpenDevice` may also fail to open the device due to insufficient access rights (permissions). This appears as if the device is not present at all. In this case look at the output of `lsusb`. The HydraHarp model with USB 2.0 interface should appear with its vendor ID `0E0D` and the device ID `0004`. The USB 3.0 model should appear with vendor ID `0E0D` and device ID `0009`. If the device is actually listed there and you still cannot open it then you probably have not set the right permissions. See section 4.2 to fix this.

As a next step, try the readily compiled demos supplied with the library. 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 advanced LabVIEW demos allow to enter most of the settings interactively.

7.4. 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.

7.5. New Linux Versions

The library has good chances to remain compatible with upcoming Linux versions. This is because the interface of libusb is likely to remain unchanged, even if libusb changes internally. You can even revert to an earlier version if necessary. Of course we will also try to catch up with new developments that might break compatibility, so that we will provide upgrades when necessary. However, note that this is work carried out voluntarily and implies no warranties for future support.

7.6. Software Updates

We 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 product website for the latest version before investing time in programming.

7.7. Bug Reports and Support

The HydraHarp 400 TCSPC system has gone through several iterations of hardware and software improvement as well as extensive testing. Nevertheless, it is a fairly complex prooduct 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 HydraHarp or the programming library. However, please note that the Linux library is a free supplement to the Windows version and its development is work carried out voluntarily by individual Linux enthusiasts. It therefore implies no warranties for other than voluntary support.

If you should observe errors or bugs caused by the HydraHarp 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. Also include 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. The simplest way of doing this in case of the HydraHarp 400 is citing the original publication about the instrument[1]. At our Website we also maintain a large bibliography of publications referring 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.

1 Wahl M., Rahn H.-J., Röhlicke T., Kell G., Nettels D., Hillger F., Schuler B., Erdmann R.: Scalable time-correlated photon counting system with multiple independent input channels. *Review of Scientific Instruments*, Vol.79, 123113 (2008)

8. Appendix

8.1. Data Types

The HydraHarp programming library is written in C and its data types correspond to C / C++ data types with bit-widths 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 Intel x86 type of machines byte swapping may occur when the HydraHarp data files are read there for further processing. We recommend using the native x86 environment consistently.

The HydraHarp software distribution pack includes a set of demo programs (source code) for various programming languages to show how access to HydraHarp data files can be implemented. These demos also show how to process TTTR records and the related code fragments can be used for real-time processing of freshly collected data as well. They will be installed in the subfolder `Filedemo` under the chosen installation folder of the HydraHarp software (not that of the programming library).

8.2. Functions Exported by `hhlib.so`

See `hhdefin.h` for predefined constants given in capital letters here. Return values `< 0` denote errors. See `errorcodes.h` for the error codes. Note, that `HHLib` can control more than one HydraHarp simultaneously. For that reason all device specific functions (i.e. the functions from section 8.2.2 on) take a device index as the first argument. Note that functions taking a channel number as an argument expect the channels enumerated 0..N-1 while the graphical HydraHarp software as well as the front panel enumerates the channels 1..N. This is due to internal data structures and consistency with earlier products.

8.2.1. General Functions

These functions work independent from any device.

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

arguments:	errstring:	pointer to a buffer for at least 40 characters
	errcode:	error code returned from a HH_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 HH_GetLibraryVersion (char* vers);`

arguments:	vers:	pointer to a buffer for at least 8 characters
return value:	=0	success
	<0	error

Note: This is the only function you may call before HH_Initialize. Use it to ensure compatibility of the library with your own application.

8.2.2. Device Specific Functions

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

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

arguments:	devidx:	device index 0..7
	serial:	pointer to a buffer for at least 8 characters
return value:	=0	success
	<0	error

`int HH_CloseDevice (int devidx);`

arguments:	devidx:	device index 0..7
return value:	=0	success
	<0	error

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

`int HH_Initialize (int devidx, int mode, int refsource);`

arguments:	devidx:	device index 0..7
	mode:	measurement mode 0 = histogramming mode 2 = T2 mode 3 = T3 mode 8 = continuous mode

```
                    refsource:      reference clock to use
                                       0 = internal
                                       1 = external

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 HydraHarp manual for more information on the measurement modes.

8.2.3. Functions for Use on Initialized Devices

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

```
int HH_GetHardwareInfo (int devidx, char* model, char* partno, char* version);     // CHANGED IN V3.0

arguments:          devidx:         device index 0..7
                    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  8 characters

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

```
int HH_GetFeatures (int devidx, int* features);                                    // NEW SINCE V3.0

arguments:          devidx:         device index 0..7
                    features:       pointer to a buffer for an integer (actually a bit pattern)

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

Note: You do not really need this function. It is mainly for integration in PicoQuant system software such as SymPhoTime in order to figure out what capabilities the device has. If you want it anyway, use the bit masks from hhdefin.h to evaluate individual bits in the pattern.

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

arguments:          devidx:         device index 0..7
                    vers:           pointer to a buffer for at least 8 characters


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

```
int HH_GetBaseResolution (int devidx, double* resolution, int* binsteps);

arguments:          devidx:         device index 0..7
                    resolution:     pointer to a double precision float (64 bit)
                                       returns the base resolution in ps
                    binsteps:       pointer to an integer,
                                       returns the maximum allowed binning steps

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

Note: Use the value returned in `binsteps` as maximum value for the HH_SetBinning function.

```
int HH_GetNumOfInputChannels (int devidx, int* nchannels);

arguments:          devidx:         device index 0..7
                    nchannels:      pointer to an integer,
                                       returns the number of installed input channels

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

int HH_GetNumOfModules (int devidx, int* nummod);

arguments:	devidx:	device index 0..7
	nummod:	pointer to an integer, returns the number of installed modules
return value:	=0	success
	<0	error

Note: This routine is primarily for maintenance and service purposes. It will typically not be needed by end user applications.

int HH_GetModuleInfo (int devidx, int modidx, int* modelcode, int* versioncode);

arguments:	devidx:	device index 0..7
	modidx:	module index 0..5
	modelcode:	pointer to an integer, returns the model of the module identified by modidx
	versioncode:	pointer to an integer, returns the versioncode of the module identified by modidx
return value:	=0	success
	<0	error

Note: This routine is primarily for maintenance and service purposes. It will typically not be needed by end user applications.

int HH_GetModuleIndex (int devidx, int channel, int* modidx);

arguments:	devidx:	device index 0..7
	channel:	index of the identifying input channel 0..nchannels-1
	modidx:	pointer to an integer, returns the index of the module where the input channel given by channel resides.
return value:	=0	success
	<0	error

Note: This routine is primarily for maintenance and service purposes. It will typically not be needed by end user applications. The maximum input channel index must correspond to nchannels-1 as obtained through HH_GetNumOfInputChannels().

int HH_GetHardwareDebugInfo(int devidx, char *debuginfo); **// NEW SINCE V3.0**

arguments:	devidx:	device index 0..7
	debuginfo:	pointer to a buffer for at least 65536 characters
return value:	=0	success
	<0	error

Note: Use this call to obtain debug information for support enquiries if you detect FLAG_SYSERROR or HH_ERROR_STATUS_FAIL.

int HH_Calibrate (int devidx);

arguments:	devidx:	device index 0..7
return value:	=0	success
	<0	error

int HH_SetSyncDiv (int devidx, int div);

arguments:	devidx:	device index 0..7
	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 ≤ 12.5 MHz. It should only be used with sync sources of stable period. Using a larger divider than strictly necessary does not do great harm but it may result in slightly larger timing jitter. The readings obtained with HH_GetCountRate are internally corrected for the divider setting and deliver the external (undivided) rate. The sync divider should not be changed while a measurement is running.

int HH_SetSyncCFD (int devidx, int level, int zerox);

```
arguments:          devidx:          device index 0..7
                    level:           CFD discriminator level in millivolts
                                        minimum = DISCRMIN
                                        maximum = DISCRMAX
                    zerox:           CFD zero cross level in millivolts
                                        minimum = ZCMIN
                                        maximum = ZCMAX

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

Note: The values are given as a positive numbers although the electrical signals are actually negative. After a firmware update to version 2.0 the CFD settings may require slight adjustments.

int HH_SetSyncChannelOffset (int devidx, int value);

```
arguments:          devidx:          device index 0..7
                    value:           sync timing offset in ps
                                        minimum = CHANOFFSMIN
                                        maximum = CHANOFFSMAX

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

int HH_SetInputCFD (int devidx, int channel, int level, int zerox);

```
arguments:          devidx:          device index 0..7
                    channel:         input channel index 0..7
                    level:           CFD discriminator level in millivolts
                                        minimum = DISCRMIN
                                        maximum = DISCRMAX
                    zerox:           CFD zero cross level in millivolts
                                        minimum = ZCMIN
                                        maximum = ZCMAX

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

Note: The values are given as a positive numbers although the electrical signals are actually negative. The maximum input channel index must correspond to `nchannels-1` as obtained through HH_GetNumOfInputChannels(). After a firmware update to version 2.0 the CFD settings may require slight adjustments.

int HH_SetInputChannelOffset (int devidx, int channel, int value);

```
arguments:          devidx:          device index 0..7
                    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 input channel index must correspond to `nchannels-1` as obtained through HH_GetNumOfInputChannels().

int HH_SetInputChannelEnable (int devidx, int channel, int enable);

```
arguments:          devidx:          device index 0..7
                    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 HH_GetNumOfInputChannels(). Upon initialization all channels are enabled. You only need to call this routine if you wish to disable some channels.

int HH_SetStopOverflow (int devidx, int stop_ovfl, unsigned int stopcount);

```
arguments:          devidx:          device index 0..7
                    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 HH_SetBinning (int devidx, int binning);

```
arguments:          devidx:          device index 0..7
                    binning:         measurement binning code
                                        minimum = 0   (default = smallest, i.e. base resolution)
                                        maximum = (MAXBINSTEPS-1)         (largest)

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

Note: the binning code corresponds to repeated doubling, i.e.

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

int HH_SetOffset (int devidx, int offset);

```
arguments:          devidx:          device index 0..7
                    offset:          histogram time offset in ns
                                        minimum = OFFSETMIN  (0, default)
                                        maximum = OFFSETMAX

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

Note: This offset must not be confused with the input offsets in each channel that act like a cable delay. In contrast, the offset here is subtracted from each start–stop measurement before it is used to either address the histogram channel to be incremented (in histogramming mode) or to be stored in a T3 mode record. The offset therefore has no effect in T2 mode and it has no effect on the relative timing of laser pulses and photon events. It merely shifts the region of interest where time difference data is to be collected. This can be useful e.g. in time-of-flight measurements where only a small time span at the far end of the range is of interest.

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

```
arguments:          devidx:          device index 0..7
                    lencode:         histogram length code
                                        minimum = 0
                                        maximum = MAXLENCODE    (default)
                    actuallen:       pointer to an integer,
                                        returns the current length (time bin count) of histograms
                                        calculated as 1024 * (2^lencode)

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

Note: Upon initialization the maximum length is set. You only need to call this routine if you want a shorter length.

int HH_ClearHistMem (int devidx);

```
arguments:          devidx:          device index 0..7

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

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

```
arguments:          devidx:          device index 0..7
                    meascontrol:     measurement control code
                                        0 = MEASCTRL_SINGLESHOT_CTC
                                        1 = MEASCTRL_C1_GATED
                                        2 = MEASCTRL_C1_START_CTC_STOP
                                        3 = MEASCTRL_C1_START_C2_STOP
                                        4 = MEASCTRL_CONT_C1_GATED
                                        5 = MEASCTRL_CONT_C1_START_CTC_STOP
                                        6 = MEASCTRL_CONT_CTC_RESTART
                    startedge:       edge selection code
                                        0 = falling
                                        1 = rising
                    stopedge:        edge selection code
                                        0 = falling
                                        1 = rising

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

int HH_StartMeas (int devidx, int tacq);

```
arguments:          devidx:          device index 0..7
                    tacq:            acquisition time in milliseconds
                                        minimum = ACQTMIN
                                        maximum = ACQTMAX

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

int HH_StopMeas (int devidx);

```
arguments:          devidx:          device index 0..7

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

Note: Can also be used before the acquisition time expires.

int HH_CTCStatus (int devidx, int* ctcstatus);

```
arguments:          devidx:          device index 0..7
                    ctcstatus        pointer to an integer,
                                        returns the acquisition time state
                                        0 = acquisition time still running
                                        1 = acquisition time has ended

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

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

```
arguments:          devidx:          device index 0..7
                    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 HH_SetHistoLen().
The maximum input channel index must correspond to `nchannels-1` as obtained through HH_GetNumOfInputChannels().

int HH_GetResolution (int devidx, double* resolution);

```
arguments:          devidx:          device index 0..7
                    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
```

int HH_GetSyncRate (int devidx, int* syncrate);

```
arguments:          devidx:          device index 0..7
                    syncrate:        pointer to an integer
                                        returns the current sync rate
return value:       =0               success
                    <0               error
```

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

```
arguments:          devidx:          device index 0..7
                    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 HH_Initialize or HH_SetSyncDivider to get a stable rate meter reading.
Similarly, wait at least 100 ms to get a new reading. This is the gate time of the counters.
The maximum input channel index must correspond to `nchannels-1` as obtained through HH_GetNumOfInputChannels().

int HH_GetFlags (int devidx, int* flags);

```
arguments:          devidx:          device index 0..7
                    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 hhdefin.h (e.g. FLAG_OVERFLOW) to extract individual bits through a bitwise AND.

int HH_GetElapsedMeasTime (int devidx, double* elapsed);

```
arguments:          devidx:          device index 0..7
                    elapsed:         pointer to a double precision float (64 bit)
                                        returns the elapsed measurement time in ms
return value:       =0               success
                    <0               error
```

Note: This can be used while a measurement is running but also after it has stopped.

int HH_GetWarnings (int devidx, int* warnings);

```
arguments:          devidx:          device index 0..7
                    warnings         pointer to an integer
                                        returns warnings, bitwise encoded (see phdefin.h)
return value:       =0               success
                    <0               error
```

note: You must call HH_GetCoutRate and HH_GetCoutRate for all channels prior to this call.

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

arguments:	devidx:	device index 0..7
	text:	pointer to a buffer for at least 16384 characters
	warnings:	integer bitfield obtained from HH_GetWarnings
return value:		=0 success
		<0 error

```
int HH_GetSyncPeriod (int devidx, double* period);                    // NEW SINCE V3.0
```

arguments:	devidx:	device index 0..7
	period:	pointer to a double precision float (64 bit) returning the sync period in ps
return value:		=0 success
		<0 error

note: This call only gives meaningful results while a measurement is running and after two sync periods have elapsed. The return value is undefined in all other cases. Accuracy is determined by single shot jitter and crystal tolerances.

8.2.4. Special Functions for TTTR Mode

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

arguments:	devidx:	device index 0..7
	buffer:	pointer to an array of count double words (32bit) where the TTTR data can be stored must provide space for at least 128 records
	count:	number of TTTR records to be fetched must be a multiple of 128, max = size of buffer, absolute 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.
Buffer must not be accessed until the function returns.
USB 2.0 devices: Call will return after a timeout period of ~10 ms, if not all data could be fetched.
USB 3.0 devices: The transfer operates in chunks of 128 records. The call will return after the number of complete chunks of 128 records that were available in the FiFo and can fit in the buffer have been transferred. Remainders smaller than the chunk size are only transferred when no complete chunks are in the FiFo.

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

arguments:	devidx:	device index 0..7
	me<n>:	active edge of marker signal <n>, 0 = falling, 1 = rising
return value:	=0	success
	<0	error

```
int HH_SetMarkerEnable (int devidx, int en0, int en1, int en2, int en3);
```

arguments:	devidx:	device index 0..7
	en<n>:	desired enable state of marker signal <n>, 0 = disabled, 1 = enabled
return value:	=0	success
	<0	error

```
int HH_SetMarkerHoldoffTime (int devidx, int holdofftime);                    // NEW SINCE V3.0

arguments:          devidx:          device index 0..7
                    holdofftime      hold-off time in ns (0..HOLDOFFMAX)
return value:       =0               success
                    <0               error
```

Note: This setting is not normally required but it can be used to deal with glitches on the marker lines. Markers following a previous marker within the hold-off time will be suppressed. Note that the actual hold-off time is only approximated to about ±8ns.

8.2.5. Special Functions for Continuous Mode

```
int HH_GetContModeBlock (int devidx, void* buffer, int* nbytesreceived);      // CHANGED SINCE V3.0

arguments:          devidx:          device index 0..7
                    buffer:          pointer to a buffer where the data will be stored
                    nbytesreceived:  pointer to an integer,
                                        returns the number of bytes received
return value:       =0               success
                    <0               error
```

Note: Required buffer size and data structure depends on the number of active input channels and histogram bins. Allocate MAXCONTMODEBUFLEN bytes to be on the safe side. The data structure changed slightly in v.3.0 to provide information on the number of active input channels and histogram bins. This simplifies accessing the data. See the C demo code.

8.3. Warnings

The following is related to the warnings (possibly) generated by the library routine `HH_GetWarnings`. The mechanism and warning criteria are the same as those used in the regular HydraHarp software 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 HH_GetCoutrate has been called for all channels before `HH_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 counts 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_TOO_LOW this warning should not occur with current software and firmware. It is kept in this list for backward compatibility only. Note, however, that there actually is a minimum sync rate of about 0.25 Hz in histogramming and T3 mode. Due to the lower limit of the rate meters this condition cannot be detected, hence no warning is issued.			
WARNING_SYNC_RATE_TOO_HIGH In histogramming or T3 mode: The divided pulse rate at the sync input is higher than 12.5 MHz (deadtime limit). Use a larger divider if possible. In T2 mode: The pulse rate at the sync input is higher than 9 MHz (bus limit if running at USB 2.0 speed) or higher than 12.5 MHz after the divider (deadtime limit if running at USB 3.0 speed). The most common reason for this error is that a laser sync signal is still connected. T2 mode is normally intended to be used without a sync signal. 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_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 You have selected T2 or T3 mode and the overall pulse rate at the input channels is higher than 9 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_DIVIDER_GREATER_ONE You have selected T2 mode and 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.		√	
WARNING_TIME_SPAN_TOO_SMALL This warning is issued in histogramming and T3 mode when the sync period (1/SyncRate) is longer that 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: Histogramming mode: Span = Resolution * 65536 T3 mode: 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.	√		√

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

PicoQuant GmbH
Rudower Chaussee 29 (IGZ)
12489 Berlin
Germany

P +49-(0)30-1208820-0
F +49-(0)30-1208820-90
info@picoquant.com
http://www.picoquant.com