PicoHarp 330

High Resolution Time–Correlated
Single Photon Counting System
and High-Speed Time Tagger

PicoQuant GmbH
Unternehmen für optoelektronische
Forschung und Entwicklung

PH330Lib – Programming Library
for Custom Software Development
under Linux
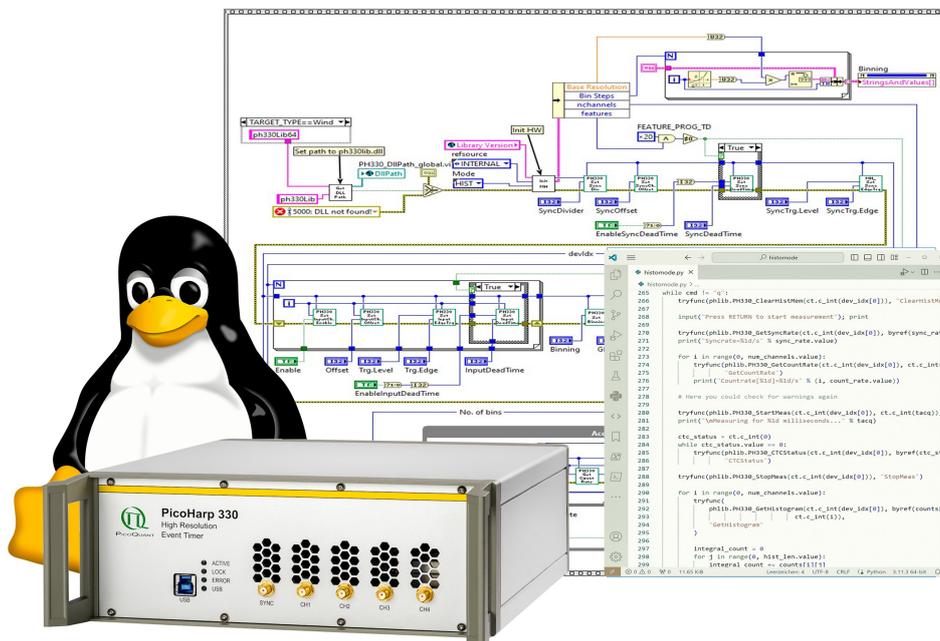
User's Manual

Version 2.0.0.0

Table of Contents

1. Introduction

The PicoHarp 330 is a cutting edge Time-Correlated Single Photon Counting (TCSPC) system and time tagger with USB 3.0 interface. Its new integrated design provides a flexible number of high performance input channels at very reasonable cost and enables innovative measurement approaches. The timing circuits allow high measurement rates up to over 80 million counts per second (Mcps) with an excellent time resolution of 1 ps, a single channel time jitter as small as 2 ps r.m.s. and a dead-time of only 680 ps. The latest Model PicoHarp 330 4P also features a very high precision internal clock source with a frequency accuracy of ±300 ppb and a frequency stability of ±10 ppb. The USB interface provides very high throughput as well as 'plug and play' installation.

The device's input triggers are adjustable for a wide range of input signals.They can be configured as Constant Fraction Discriminators (CFDs) or as programmable edge triggers, the latter even for both polarities. These specifications qualify the PicoHarp 330 for use with Superconducting Nanowire Single Photon Detectors (SNSPD), Single Photon Avalanche Diodes (SPADs), Hybrid Photodetectors (HPD), and Photomultiplier Tubes (PMT). Depending on detector and excitation source the FWHM of the overall Instrument Response Function (IRF) can be as small as 15 ps. The PicoHarp 330 can be purchased with one or two timing inputs and one synchronization (sync) input. The use of these inputs is very flexible. In fluorescence lifetime applications the sync channel is typically used as a synchronization input from a laser. The other inputs are then used for photon detectors. Alternatively, notably in quantum optics applications, all inputs including the sync input can be used for photon detectors.

The PicoHarp 330 can operate in various modes to adapt to different measurement needs. The standard histogram mode performs real–time histogramming in host memory. Two different Time–Tagged–Time–Resolved (TTTR) modes allow recording each photon event on separate, independent channels, thereby providing unlimited flexibility in off–line data analysis such as burst detection and time–gated or lifetime weighted Fluorescence Correlation Spectroscopy (FCS) as well as picosecond coincidence correlation, using the individual photon arrival times. The PicoHarp 330 is furthermore supported by a variety of accessories such as pre–amplifiers, signal adaptors and detector assemblies from PicoQuant.

For more information on the PicoHarp 330 hardware and software please consult the PicoHarp 330 manual. For details on the method of Time–Correlated Single Photon Counting, please refer to our TechNote on TCSPC.

The PicoHarp 330 standard software provides functions such as the setting of measurement parameters, display of results, loading and saving of measurement parameters and histogram curves. Important measurement characteristics such as count rate, count maximum and position, histogram width (FWHM) are displayed continuously. While these features will meet many of the routine requirements, advanced users may want to include the PicoHarp'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 (see separate manual) and as a shared library for Linux described here.

The library supports custom programming in virtually all major programming languages, notably C / C++, C#, Pascal, Python, Rust, LabVIEW and MATLAB. This manual describes the installation and use of the PicoHarp 330 programming library and explains the associated demo programs. Please read both this library manual and the PicoHarp 330 manual before beginning your own software development with the library. The PicoHarp 330 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

2.1. Scope and Compatibility

This manual solely covers the programming library PH330Lib for the product model PicoHarp 330. The hardware and software of the seminal predecessor product PicoHarp 300 is significantly different from that of the PicoHarp 330. There is no software compatibility across the two products. Please do not confuse the two lines. The PicoHarp 300 has its own manual and its own software. Also note that here in this document we use the common name of the library `PH330Lib` (as derived from the Windows version) while the actual shared library for Linux is named `libph330.so` in order to meet the Linux conventions.

The PicoHarp 330 programming library for Linux is suitable for the "x86-64" processor architecture only. We dropped support for 32-bit systems due to the fact that hardly any Linux distribution still offers it.

The library has been tested with gcc 11.4.0 and 13.2.0, Mono 6.8.0 and 6.12.0, Python 3.11.6 and 3.12.3, as well as with Lazarus 3.0 (FreePascal 3.2.2) and Rustc 1.79.0 (+cargo 1.79.0). The demos for LabVIEW and Matlab have only been tested under Windows using LabVIEW 2020 and MATLAB R2019a, due to our lack of the Linux versions. If you happen to test with Linux versions of LabVIEW 2020 or MATLAB please let us know the results.

This manual assumes that you have read the PicoHarp 330 manual. References to it will be made where necessary. It is also assumed that you have solid experience with the chosen programming language. Our support will not teach programming fundamentals.

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 that you can use to retrieve the version number (see section 7.2). Note that this call returns only the major two digits of the version string (e.g. presently 2.0). The library actually has two further sub–version digits, so that the complete version number has four digits (e.g. presently 2.0.0.0). The complete version number including the build date will be shown upon running the installtion script. If you need to check it later you can run the command:

```
strings /usr/local/lib64/ph330/libph330.so | grep "LIBPH330 VERSION"
```

The two sub–version digits are used to identify intermediate versions that may have been released for small updates or bug fixes. The interface of releases with identical major version will remain the same. The minor version is typically incremented when there are new features or functions added without breaking compatibilty in regard to the original interface of the corresponding major release. The rightmost digit of the complete version number usually increments to indicate bugfix releases of otherwise identical interface and functionality.

2.2. What's new in this Version

Version 2.0 is the second release of PH330Lib and primarily provided to support the new PicoHarp 330 4P with four input channels. The previous model with two input channels is of course also supported. A very useful new feature of PH330Lib v. 2.0 is the API call `PH330_SaveDebugDump`. It is provided to help debugging gateware issues by letting the user save a snapshot of the device's internal FPGA states to a file that then can be submitted for support.

Users of other PicoQuant TCSPC systems and their programming libraries will find PH330Lib very familiar. Compared to such earlier products, e.g. the MultiHarp family, the interface remains conceptually unchanged, except that support for new or extended features such as the programmable input configuration required the introduction of some new API calls.

2.3.   Warranty and Legal Terms

Disclaimer

PicoQuant GmbH disclaims all warranties with regard to the supplied software and documentation including all implied warranties of merchantability and fitness for a particular purpose. In no case shall PicoQuant GmbH be liable for any direct, indirect or consequential damages or any material or immaterial damages whatsoever resulting from loss of data, time or profits; arising from use, inability to use, or performance of this software and associated documentation.

License and Copyright Notice

With the PicoHarp 330 hardware product you have purchased a license to use the PicoHarp software. You have not purchased any other rights to the software itself. The software is protected by copyright and intellectual property laws. You may not distribute the software to third parties or reverse engineer, decompile or disassemble the software or part thereof. You may use and modify demo code to create your own software. Original or modified demo code may be re–distributed, provided that the original disclaimer and copyright notes are not removed from it. Copyright of this manual and on–line documentation belongs to PicoQuant GmbH. No parts of it may be reproduced, translated or transferred to third parties without written permission of PicoQuant GmbH.

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

Acknowledgements

The PicoHarp 330 programming library for Linux uses Libusb to access the PicoHarp 330 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 appliccable to Libusb is provided as part of the distribution archive. The LGPL does not apply to the PicoHarp 330 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. Installation of the Library

3.1. 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, 2 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 5.0 or higher. It has been tested with the following distributions:

Ubuntu 20.04.3 LTS (kernel 5.15.0)
Ubuntu 22.04.3 LTS (kernel 5.15.0)
Ubuntu 24.04.3 LTS (kernel 6.8.0)

Using the library requires libusb (https://libusb.info/). The formally required version is 1.0 or higher, tested versions were 1.0.23, 1.0.25 and 1.0.27. Libusb is typically installed by default on all major Linux distributions.

It is recommended to start your work by using the standard interactive PicoHarp 330 data acquisition software under Windows or Linux with Wine. This will 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.

3.2. Device Access Permissions

For device access through libusb suitable permissions for the device must be granted to the normal user, otherwise only the super-user `root` will have access. Recent Linux distributions use udev to handle this. For automated setting of the device access permissions with udev you can 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 rule files usually carry names starting with a number. Don't change these files as they could be overwritten when you upgrade your system. Instead, put your custom rule for the PicoHarp 330 in a separate file. The typical content of this file should be:

```
ATTR{idVendor}=="0d0e", ATTR{idProduct}=="0015", MODE="666"
```

A suitable rules file `PicoHarp330.rules` is provided in the folder `udev` under the unpacked distribution folder. You can simply copy it to the `/etc/udev/rules.d` folder. The install script in the same folder does just this. Note that this requires root permissions. As a normal user you must run it preceded with `sudo`. After that you need to disconnect and reconnect the device to get access.

If you have issues obtaining permissions recall 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 until a match is found. Different Linux distributions may use different rule file names for various categories. If there happen to be later rules that are more general (applying to a whole class of devices) they may override your custom rule and the desired access rights. It is therefore important that you use a rules file named such that it gets evaluated <u>after</u> the general case. The default naming `PicoHarp330.rules` most likely ensures this but if you see access problems you may want to check.

Note that the setting `MODE="666"` is quite permissive for all users. If you prefer tighter security regarding device access please study the documentation of udev and/or the recommendations of your distribution for handling USB device access, e.g. employing user classes with suitable access rights.

3.3. Installing the Library

---

[*] USB 3.0 was later renamed to USB 3.1 Gen 1 and is now called USB 3.2 Gen 1

Starting with version 2.0 the PicoHarp 330 software pack is no longer provided on physical media. Please download the software pack from the PicoQuant website at https://www.picoquant.com/downloads. On the download page scroll down to the category `TCSPC and Time Tagging Electronics` and select `Pico-Harp 330 - Operation Software`. The download will be packed in a ZIP-File. Please unzip this file to a temporary hard disk location of your choice. You will find a folder strucure with multiple items where you must navigate to the sub-folder `PH330Lib v.X.X.X.X\Linux\`. The library package for Linux is provided there as a gzipped tar file. Unpack this file to a location of your choice.

The shared library as such is provided as a binary file. By default it is assumed that it resides under `/usr/local/lib64/ph330`. 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 keep this location.

The shell script `install` in the unpacked directory `library` does the directory creation and installation in one step. Just run it preceded with `sudo` at the command prompt from within the `library` directory. The install script also takes care of the variations of different Linux distributions where the x64 library paths use either `usr/lib/` or `usr/lib64`. This is done by creating symbolic links rather than copying the library to multiple places.

After installation the library is ready to use and can be tested with the demos provided. On some distributions you may still 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`. This also circumvents the issue of varying default locations mentioned above.

Note for SELinux: If upon linking with `libph330.so` you get an error *"cannot restore segment prot after reloc"* you need to adjust the security settings for `mhlib.so`. As root you need to run:

```
chcon -t texrel_shlib_t /usr/local/lib/ph330/libph330.so
```

3.4.   Installing the Demo Programs

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

4.　The Demo Applications

4.1.　Functional Overview

Please note that all demo code provided is correct to the best of our 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 education 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. For the same reason, the measurement parameters are mostly hard–coded and thereby fixed at compile time. It is therefore necessary to change the source code and re-compile the demos in order to run them in a way that is matched to your individual measurement setup. Running them unmodified will probably result in useless data (or none at all) because of inappropriate sync divider, resolution, input level settings, etc. In order to understand these settings it is strongly recommended that you read the PicoHarp 330 manual and try them out using the regular PicoHarp 330 software for Windows.

For the reason of simplicity, the demos will always only use the first PicoHarp 330 device they find, although the library can support multiple devices. If you wish to use more than one PicoHarp 330 at the same time you need to modify the code accordingly. See section 5.6 on this topic.

There are demos for C / C++, C#, Pascal, Python, Rust, 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 real-time histogramming. These are the simplest demos and the best starting point for your own experiments. TCSPC histogramming is easy to use and useful in typical fluorescence decay measurements as well as in basic quantum optics experiments. The time differences between sync input and the channel inputs are calculated in real-time and put in histograms for each channel.

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 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 PicoHarp 330 actually supports two different Time–Tagging modes, T2 and T3 mode. When referring to both modes together we use the general term TTTR here. For details on the two modes, please refer to your PicoHarp manual. In TTTR mode it is also possible to record external TTL signal transitions as markers in the TTTR data stream (see the PicoHarp 330 manual) which is typically used e.g., for FLIM.

Because TTTR mode requires real–time processing and / or real–time storing of data, the TTTR demos are more demanding both in programming skills and computer performance. Also consider the speed performance of your programming language. Interpreted Python and Matlab, for example, are very slow. For more information on TTTR mode consult the corresponding section in your PicoHarp 330 manual.

Note that you must not call any of the `PH330_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 or temporal incoherence in the collected data.

Details on how to interpret and process the TTTR records can be studied in the advanced demos (see below). You may also consult the PTU file demo code installed together with the regular PicoHarp 330 software under Windows or Wine.

Advanced Demos

For several programming languages there are also advanced demos to show hardware triggered histogram measurements (see section 5.4) or instant processing (see section 5.2) and filtering of TTTR data streams (section 5.5). In case of LabVIEW there is an advanced demo allowing interactive input of most parameters on the fly.

4.2.   The Demo Applications by Programming Language

As outlined above, there are demos for C / C++, C#, Pascal, Python, Rust, LabVIEW and MATLAB. For each of these programming languages (except Rust) there are different demo versions for the measurement modes listed in the previous section. They are not 100% identical. For some programming languages (C, Python, Delphi, C#, LabVIEW) there are also some advanced demos, typically residing in a subfolder `advanced`. In this context please see section 5 on advanced techniques.

This manual explains the special aspects of using the PicoHarp 330 programming library, it does NOT teach you how to program in the chosen programming language. We strongly recommend that you do not choose to develop a software project with the PicoHarp 330 library as your first attempt at programming. You will also need some knowledge about shared library concepts and related Linux conventions. 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 the library (`ph330lib.h` and `ph330defin.h`).

Be warned that wrong parameters and / or variables, invalid pointers and buffer sizes, inappropriate calling sequences etc. may crash your application or get the device locked up so that you need to restart it. Also note that the DLL is not re-entrant w.r.t. an individual device instance. This means, it cannot be accessed from multiple, concurrent processes or threads at the same time unless separate device instances are being used. All calls to one device instance must be made sequentially. The order of the calls is to some extent flexible, e.g. when parameters are set. Some other calls such as initialization, start and stop of measurements obviously must follow in a meaningful order. You may preferably want to stick to the order shown by 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 `ph330lib.h`, `ph330defin.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 "ph330defin.h"

  #include "ph330lib.h"

}
```

To test any of the demos, consult the PicoHarp 330 manual for setting up the device and establish a measurement setup that runs correctly and generates useable test data. This is best done with the regular PicoHarp 330 software under Windows or under Linux with Wine. Compare the settings (notably sync divider, binning and trigger levels) with those used in the demo and use the values that work in your setup when building and testing the demos. Observe the mode input variable going into `PH330_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.

The C demos are designed to run from the command line in a console or terminal window. They need no command line input parameters. The output files will be ASCII-readable only in case of the standard histogramming demos and most of the advanced demos. For the histogramming demo, the output files will contain multiple columns (one per channel) of integer numbers representing the counts in the histogram bins. You can use any editor or a data visualization program to inspect the histograms. In the simplest TTTR mode demo the output is stored in binary format for simplicity and 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 PicoHarp 330 software installation under Windows or Wine) for reading the PicoQuant 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 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 Mono.

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

With the C# demos you also need to check whether the hard-coded settings are suitable for your actual instrument setup. The demos are designed to run in a terminal window. They need no command line input parameters. They create their output files in their current working directory. The output files will be ASCII in case of the histogramming demo and some of the advanced demos. In the simplest TTTR mode demo the output is stored in binary format for simplicity and performance reasons. The ASCII files of the histogramming demos will contain single or multiple columns of integer numbers representing the counts from the histogram channels. You can use any editor or a data visualization program to inspect the ASCII histograms. The binary files must be read by dedicated programs according to the format they were written in. The file read demos provided for the PicoQuant TTTR data files (.PTU) and the advanced demo `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 PH330Lib demos focused on the key issues of using the library.

Observe the mode input variable going into `PH330_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.

The Pascal/ Lazarus Demos

Users of FreePascal / Lazarus please refer to the `Pascal` folder. The source code for Delphi (Windows) and Lazarus is identical. Everything for the respective Delphi demo is in the project file for that demo (`*.DPR`). Lazarus users can use the `*.LPI` files that refer to the same `*.DPR` files.

In order to make the exports of `mhlib.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.

The Delphi / Lazarus demos are also designed to run from the command line. They need no input parameters. They create output files in their current working directory. The output files of the will be ASCII in case of the histogramming demo and most of the advanced demos. In the simplest TTTR mode demo the output is stored in binary format for simplicity and performance reasons. You can use any data visualization program to inspect the ASCII histograms. The binary files must be read by dedicated programs according to the format they were written in. The file read demos for the regular PicoQuant TTTR data files (.PTU) and the advanced demo `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 PH330Lib demos focused on the key issues of using the library.

Observe the mode input variable going into `PH330_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.

The Python Demos

The Python demos are in the `Python` folder. Python users should start their work in histogramming mode from `histomode.py`. The code should be fairly self explanatory. If you update to a new library version please check the function parameters of your existing code against `ph330lib.h` in the PH330Lib installation directory. Note that special care must be taken where pointers to C–arrays are passed as function arguments.

The Python demos create output files in their current working directory. The output file will be readable text in case of the standard histogramming demo and most of the advanced demos. The histogramming demo output files will contain columns of integer numbers representing the counts from the histogram channels. You can use any data visualization program to inspect the histograms. In the simplest TTTR mode demo the output is stored in binary format for performance reasons. The binary files must be read by dedicated programs according to the format they were written in. The file read demos for the regular PicoQuant TTTR data files (.PTU) and the advanced demo `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 PH330Lib demos focused on the key issues of using the library. Note that even if it may be tempting to directly use the advanced demo `tttrmode_instant_processing` you should not do this routinely. It creates very large files and throughput with interpreted Python is very poor.

Observe the mode input variable going into `PH330_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.

The LabVIEW Demos

The LabVIEW demos for Linux are identical with the LabVIEW demos for Windows. They automatically detect the operating system and select the appropriate library name and path. Unfortunately we do not have Lab-VIEW for Linux, so this feature is untested under Linux. Pleas kindly report success or error if you happen to work with LabVIEW for Linux.

The first LabVIEW demo (`1_SimpleDemo_MHHisto.vi`) is very simple, demonstrating the basic usage and calling sequence of the provided SubVIs encapsulating the library functionality, which are assembled inside the LabVIEW library `mhlib_x86_x64_UIThread.llb`. 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 `Auto-Scale` for the 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_MHHisto.vi`) is a more sophisticated one allowing the user to control all hardware settings "on the fly", i.e. to change settings like acquisition 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 the 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 LabVIEW demo (`3_AdvancedDemo_MHT3.vi`) is the most advanced one and demonstrates the usage of T3 mode including real-time evaluation of the collected TTTR records. 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 the 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 Lab-VIEW type-definitions and two additional threads: a data processing thread (`MH_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 `MH_DataProcThread.vi` have been inlined for performance, so that no debugging is possible on these SubVIs.

Program specific SubVIs and type-definitions used by the demos are organized in corresponding sub-folders inside the demo folder (here relating to the installed MHLib package for Windows). General helper functions and type-definitions as well as encapsulating LabVIEW libraries (*.llb) can be found in the `_lib` folder (containing further sub-folders) inside the demo folder. In order to facilitate the use of all library functions, additional VIs called `MH_AllDllFunctions_xxx.vi` have been included. These VIs are not meant to be executed but should only give a structured overview of all available library functions and their required context.

Please note:

In addition to the library used by the demos (`mhlib_x86_x64_UIThread.llb`) a second LabVIEW library (llb) is included allowing the library calls to be executed in any thread of LabVIEWs threading engine (`mhlib_x86_x64_AnyThread.llb`). This llb 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 library functions (please refer to "Multitasking in LabVIEW": http://zone.ni.com/reference/en-XX/help/371361P-01/lvconcepts/multitasking_in_labview/). When using this llb you have to make sure that all library functions are called in a sequential order to avoid errors or even program crashes. Also be aware that library functions in `mhlib_x86_x64_AnyThread.llb` have the same names as in `mhlib_x86_x64_UIThread.llb` and opening both libraries at the same time would lead to name conflicts. Therefore, only experienced users should use `mhlib_x86_x64_AnyThread.llb`.

The MATLAB Demos

The MATLAB demos are provided in the `MATLAB` folder. They are contained in .m files. You need to have a MATLAB version that supports the `loadlibrary` and `calllib` commands. The earliest version we have tested in this regard is MATLAB 7.3 (under Windows) but any version from 6.5 on should work. For your specific version of MATLAB, please check the documentation of the MATLAB command `loadlibrary` as to whether and how it works. Be careful about the header file name specified in `loadlibrary`. The names are case sensitive and spelling errors 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 simplicity and performance reasons. You can use any data visualization program to inspect the ASCII histograms. The binary files from TTTR mode must be read by dedicated programs according to the format they were written in. The file read demos for the regular PicoQuant TTTR data files (.PTU) can be used as a starting point. They cannot be used directly on the binary demo output because they expect a file header the demos do not generate. This is intentional in order to keep the PH330Lib demos focused on the key issues of using the library. The file demo code can (with minor adaptions) in principle be used to process the TTTR records on the fly. However, MATLAB scripts are relatively slow compared to properly compiled code. This may impose throughput limits. You might want to consider compiling Mex files instead. In this case please study the advanced demos `tttrmode_instant_processing` (C, Python, Delphi, C#) which can be used as a starting point to learn this.

Observe the mode input variable going into `PH330_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.

The Rust Demo

For Rust there is currently only one demo for simple TTTR mode data recording. For ambitious programmers this should be sufficient as a starting point to also port the more advanced demos from C to Rust.

5. Advanced Techniques

5.1. 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 PicoHarp 330 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 PicoHarp 330 this permits data throughput as high as 9 Mcps (USB 2.0) or even up to 90 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 `PH330_ReadFiFo` that accepts a buffer address where the data is to be placed. The memory block size is fixed and must provide space for 1,048,576 event records. However, the actual transfer size will depend on how much data was available in the device's FIFO buffer. The call will typically return after about 10 ms but possibly quicker if no more data is available. The latency behavior at input rates close to zero is controlled by `PH330_SetOflCompression`. The actual time to return can also vary due to USB overhead and unpredictable Windows latencies, especially when the PC or the USB connection is slow.

As noted above, the transfer is implemented efficiently without excessive CPU load. Nevertheless, assuming large block sizes, the transfer takes some time. Linux therefore gives the unused CPU time to other processes or threads i.e. it waits for completion of the transfer without burning CPU time. This wait time is what can also be used for doing 'useful things' in terms of any desired data processing or storing within your own application. The proper way of doing this is to use multi-threading. In this case you design your program with two threads, one for collecting the data (i.e. working with `PH330_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 an advanced technique and it cannot be demonstrated in all detail here. Currently only the most advanced LabVIEW demo uses this technique. Greatest care must be taken not to access the PH330Lib routines from different threads without strict control of mutual exclusion and maintaining the right sequence of function calls, unless the threads act on different devices. However, the technique allows significant throughput improvements and advanced programmers may want to use it. It might be interesting to note that this is how TTTR mode is implemented in the regular PicoHarp 330 software, where sustained count rates up to 90 Mcps can be handled.

When working with multiple devices, the overall USB throughput is usually 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 sharing hubs. If you install additional USB controller cards you should prefer fast PCI–express models. 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.

5.2. 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 programming 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 `PH330_ReadFiFo`) and another (or more) for processing, displaying, or storing the data (see also section Fehler: Verweis nicht gefunden). 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 `PH330_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 time tag 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.

5.3. Working with Warnings

The library provides routines for obtaining and interpreting warnings about critical measurement conditions. The mechanism and warning criteria are the same as those used in the regular PicoHarp 330 software (see the manual). In order to obtain and use these warnings also in your custom software you may want to use the library routine `PH330_GetWarnings`. This may help inexperienced users to notice possible mistakes before starting 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 `PH330_GetWarnings` does not obtain the count rates on its own, because the corresponding calls take some time and might waste USB bandwidth and processing time. It is therefore necessary that the library routines for count rate retrieval (on all channels) have been called before `PH330_GetWarnings` is called. Since most interactive measurement software periodically retrieves the rates anyhow, this is not a serious complication. Note that there are library calls for retrieval of individual count rates (`PH330_GetSyncRate` and `PH330_GetCountRate`) but in case of performance critical applications it is more efficient to use `PH330_GetAllCountRates` retrieving all rates in one call.

The routine `PH330_GetWarnings` delivers the cumulated warnings in the form of a bit field. In order to translate this into readable information you can use `PH330_GetWarningsText`. Before passing the bit field into `PH330_GetWarningsText` you can mask out individual warnings by means of the bit masks defined in `mhdefin.h`. See the appendix section 7.3 for a description of the individual warnings.

5.4. Hardware Triggered Measurements

This measurement scheme allows to start and stop the acquisition by means of external TTL signals rather than software comands. Since it is an advanced real-time technique, beginners are advised to not try their first steps with it. For the same reason, demos exist only in some programming languages (C, C#, Pascal, Python).

Before using this scheme, consider when it is useful to do so. For instance, it may be tempting to use the hardware triggering to implement very short histogramming durations. However, remember that TTTR mode is usually the most efficient way of retrieving the maximum information on photon dynamics. By means of marker inputs the photon events can be precisely assigned to complex external event scenarios.

The PicoHarp's data acquisition can be controlled in various ways. Default is the internal CTC (counter timer circuit). In that case the measurement will take the duration set by the `tacq` parameter passed to `PH330_StartMeas`. The other way of controlling the histogram boundaries (in time) is by external TTL signals fed to the control connector pins C1 and C2 (see appendix section *Connectors* of the MultiHarp manual).

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 of this scheme is controlled by the parameters supplied to the call of `MH_SetMeasControl`. Dependent on the parameter `meascontrol` the following modes of operation can be obtained:

Symbolic Name	Value	Function
MEASCTRL_SINGLESHOT_CTC	0	Default value. Acquisition starts by software command and runs until CTC expires. The duration is set by the `tacq` parameter passed to `PH330_StartMeas`.
MEASCTRL_C1_GATE	1	Data is 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 `PH330_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`.
MEASCTRL_SW_START_SW_STOP	6	Data collection is started and stopped by software using `PH330_StartMeas` and `PH330_StopMeas`. This permits overcoming the limit of 100 h imposed by the hardware CTC. This is not a hardware triggered measurement scheme but it needed to be listed here for completeness.

The symbolic constants shown above are defined in `ph330defin.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.

5.5. Working with Event Filtering

Filtering TTTR data streams in hardware helps to reduce USB bus load by eliminating photon events that carry no information of interest as typically found in many coincidence correlation experiments. Please read the PicoHarp 330 manual for more details.

The filter has several programmable parameters. The parameter `timerange` determines the time window the filter is acting on. The parameter `matchcnt` specifies how many other events must fall into the chosen time window for the filter condition to act on the event at hand. The parameter `inverse` inverts the filter action, i.e. when the filter would regularly have eliminated an event it will then keep it and vice versa. For the typical case, let it be not inverted. Then, if `matchcnt` is 1 we will obtain a simple 'singles filter'. This is the most straight forward and most useful filter in typical quantum optics experiments. It will suppress all events that do not have at least one coincident event within the chosen time range, be this in the same or any other channel.

In addition to the filter parameters explained so far it is possible to mark individual channels for use. Used channels will take part in the filtering process. Unused channels will be suppressed altogether. Furthermore, it is possible to indicate if a channel is to be passed through the filter unconditionally, whether it is marked as 'use' or not. The events on a channel that is marked neither as 'use' nor as 'pass' will not pass the filter, provided the filter is enabled. The parameters `usechannels` and `passchannels` are actually bitmasks where channels to be used or passed are indicated by their corresponding bits set to one.

The filter can also be switched into a test mode where the data is not transferred to USB. Instead one will then use `PH330_GetFilterInputRates` and `PH330_GetFilterOutputRates` in order to check its effect of data rate reduction. This helps to initially try out and optimize the filter parameters without running into FIFO overrun issues.

5.6. Using Multiple Devices

The library is designed to work with multiple PicoHarp 330 devices (up to 8). For simplicity the demos use only the first device found. If you wish to use more than one PicoHarp 330 at the same time you need to modify the code accordingly. At the API level of PH330Lib the devices are distinguished by a device index (0 .. 7). The device order corresponds to the order in which Linux enumerates the devices. If the devices were plugged in or switched on sequentially when Linux was already up and running, the order is given by that sequence. Otherwise it 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 `PH330_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 `PH330_GetSerialNumber` any time later on a device you have successfully opened. The serial number of a physical PicoHarp 330 device can be found at the back of the housing. It is an 8 digit number starting with 010. The leading zero will not be shown in the serial number strings retrieved through `PH330_OpenDevice` or `PH330_GetSerialNumber`.

As outlined above, if you have more than one PicoHarp 330 and you want to use them together you need to modify the demo code accordingly. This requires 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 such as the regular PicoHarp 330 software.

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.

Note that combining multiple devices by software does not make a proper replacement for a hardware device with more channels. This is due to multiple reasons. First, the clocks of the devices are not infinitely accurate and may therefore drift apart. Second, the software-combined devices cannot start or stop measurements at exactly the same time. Software timing is not accurate enough and will cause unpredictable delays of some milliseconds. Third, the data of the devices arrives in separate data streams and cannot easily be merged together. Even though the first and second issue can partially be solved by means of external clock signals, hardware controlled measurements and/or markers, the approach is somewhat cumbersome.

6. Problems, Tips & Tricks

6.1. PC Performance Requirements

Performance requirements for the library are the same as with the standard PicoHarp 330 software for Windows. The PicoHarp 330 device and its software interface are a complex real–time measurement system demanding appropriate performance both from the host PC and the operating system. This is why a reasonably modern CPU and sufficient memory are required. At least a quad core, 2 GHz processor, 4 GB of memory and a fast hard disk are recommended.

6.2. USB Interface

In order to deliver maximum throughput, the PicoHarp 330 uses USB 3.0[*] bulk transfers. This is why the PicoHarp 330 must rely on having a USB host interface providing USB 3.x speed. USB 3.x host controllers of modern PCs are usually integrated on the mainboard. For older PCs they may be upgraded as plug-in cards. Throughput is then usually limited by the host controller and operating system, not the PicoHarp 330. Do not run other bandwidth demanding devices on the same USB interface when working with the PicoHarp 330. USB cables must be qualified for USB 3.x speed. Old 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 worn out plugs and sockets or subtle damages to USB cables, caused, e.g., by sharply bending or crushing them. Observe the USB LED on the front panel: It should light up green to indicate a USB 3.0 connection. If it shows yellow the device is connected only at USB 2.0 speed and will deliver very poor throughput. If the USB LED does not light up at all there may be a driver issue, check the Windows device manager then.

6.3. Troubleshooting

Troubleshooting should begin by testing your hardware and driver setup. This is best accomplished by the standard PicoHarp 330 software for Windows. Under Linux it can also be used with Wine. Only if this software is working properly you should start working with the library. If there are problems even with the standard software, please consult the PicoHarp 330 manual for detailed troubleshooting advice.

Under Linux the PicoHarp 330 programming library will access the PicoHarp 330 device through Libusb. You need to make sure Libusb has been installed correctly. Normally this is readily provided by all recent Linux distributions. You can use `lsusb` to check if the device has been detected and is accessible. Please consult the PicoHarp 330 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 `PH330_OpenDevice`. Opening the device may also fail due to insufficient access rights (permissions). This may appear as if the device is not present at all. In this case look at the output of `lsusb`. The PicoHarp should appear with its vendor ID `0D0E` and the device ID `0013`. If the device is actually listed there and you still cannot open it then you probably have not set the right access permissions. See section 3.2 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.

6.4. 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 DLL. In any case your software should issue a warning if it detects versions other than those it was tested with. Note that the call of `PH330_GetLibraryVersion` returns only the major two digits of the version (e.g. 2.0). The library actually has two further sub–version digits,

---

[*] USB 3.0 was later renamed to USB 3.1 Gen 1 and is now called USB 3.2 Gen 1

so that the complete version number has four digits (e.g. 2.0.0.0). These sub–digits help to identify intermediate versions that may have been released for minor updates or bug fixes. The complete version number including the build date will be shown upon running the library installtion script. If you need to check it later you can run the command:

```
strings /usr/local/lib64/ph330/libph330.so | grep "LIBPH330 VERSION"
```

The interface of releases with identical major version will remain the same. The minor version is typically incremented when there are new features or functions added without breaking compatibilty in regard to the original interface of the corresponding major release. The very last digit is typically incremented upon bugfixes without functional changes.

6.5.   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.

6.6.   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 for software updates before investing time into a larger programming effort.

6.7.   Bug Reports and Support

The PicoHarp 330 TCSPC system has gone through extensive testing. It builds on over 25 years of experience with several predecessor models and the feedback of hundreds of users. Nevertheless, it is a fairly new product and some bugs may still be found. In any case we would like to offer you our support if you experience problems with the system. Do not hesitate to contact PicoQuant in case of difficulties with your PicoHarp.

If you observe errors or bugs caused by the PicoHarp 330 system please try to find a reproducible error situation. Then prepare a detailed description of the problem and all relevant circumstances, especially the versions of the software you were using, the version of Linux and the serial number of your PicoHarp 330. Then use our support page at www.picoquant.com/contact/support to create a ticket. Alternatively you can also write an email to support@picoquant.com. Your qualified feedback will help us to improve the product and documentation.

A very useful new feature of PH330Lib v. 2.0 is the API call `PH330_SaveDebugDump`. It is provided to help debugging gateware issues by letting the user save a snapshot of the device's internal FPGA states to a file that then can be submitted for support. Please implement this feature in your custom code whenever feasible.

Of course we also appreciate good news: If you have obtained exciting results with one of our instruments, please let us know, and where appropriate, please mention the instrument in your publications.

At our 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 kindly submit your publications for addition to this list.

7. Appendix

7.1. Data Types

The PicoHarp 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 also that on platforms other than the x86-64 architecture byte swapping may occur when binary Pico-Harp 330 data files are read there for further processing. We recommend using the native x86-64 architecture environment consistently.

7.2. Functions Exported by `PH330Lib.so`

See `ph330defin.h` for predefined constants given in capital letters in the following subsections here.
Note that these indeed are constants fixed at compile time of the library and that you cannot change them, even if in some of the demo programs it might look like it.

Return values < 0 denote errors. See `errorcodes.h` for the possible error codes.

Note that PH330Lib is a multi-device library with the capability to control more than one PicoHarp 330 simultaneously. For this reason all device specific functions (i.e. the functions from section 7.2.2 on) take a device index as their first argument.

Note also that functions taking a channel number as an argument expect the channels enumerated `0..N-1` while the regular interactive PicoHarp 330 software enumerates the channels `1..N` as shown on the physical front panel. This is for the efficiency of internal data structures and for consistency with earlier products.

7.2.1.   General Functions

These functions work independent from any device.

```
int PH330_GetLibraryVersion (char* vers);
```

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

Note: Use this call to ensure compatibility of the library with your own application.

```
int PH330_GetErrorString (char* errstring, int errcode);
```

arguments:	errstring:	pointer to a buffer for at least 40 characters
	errcode:	error code returned from a PH330_xxx library 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.

7.2.2.   Device Related Functions

All functions here and further below are device related and require a device index.

```
int PH330_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

Note: Once a device is opened by your software it will not be available for use by other programs until you close it.

```
int PH330_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 PH330_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
	refsource:	reference clock to use
		0 = use internal clock
		1 = use 10 MHz external clock
		2 = use 100 MHz external clock
		3 = use 500 MHz external clock

```
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 PicoHarp 330 manual for more information on the measurement modes.

7.2.3.   Functions for Use on Initialized Devices

All functions below can only be used after PH330_Initialize was successfully called.

```
int PH330_GetHardwareInfo (int devidx, char* model, char* partno, char* version);
```

```
arguments:           devidx:           device index 0..7
                     model:            pointer to a buffer for at least 24 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 PH330_GetFeatures (int devidx, int* features);
```

```
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 in a standardized way what capabilities the device has. If you want it anyway, use the bit masks from mhdefin.h to evaluate individual bits in the pattern.

```
int PH330_GetSerialNumber (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 PH330_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 number of allowed binning steps
return value:        =0                success
                     <0                error
```

Note: The base resolution of a device is its best possible resolution as determinded by the hardware. It also corresponds to the timing resolution in T2 mode. In T3 and Histogramming mode it is possible to "bin down" the resolution (see PH330_Set-Binning) The value returned in binsteps is the number of permitted binning steps. The range of values you can pass to PH330_SetBinning is then 0..binsteps-1.

```
int PH330_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
```

Note: The value returned in `nchannels` is the number of channels. The range of values you can pass to the library calls accept-
         ing a channel number is then `0..nchannels-1`.

int PH330_GetModuleInfo (int devidx, int* modelcode, int* versioncode);

```
arguments:           devidx:            device index 0..7
                     modelcode:         pointer to an integer,
                                            returns the model code of the module
                     versioncode:       pointer to an integer,
                                            returns the version code of the module

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

Note: This routine is for retrieval of hardware version details. You only need this information for support enquiries.

int PH330_GetDebugInfo(int devidx, char *debuginfo);

```
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. You can call it immediately after receiving an error code <0 from any library call. It
         is of particular value after detecting a `FLAG_SYSERROR` from `PH330_GetFlags`. In case of `FLAG_SYSERROR` please
         provide this information for support.

int PH330_SaveDebugDump(int devidx, char* filepath);   **// new since v. 2.0**

```
arguments:           devidx:            device index 0..7
                     filepath:          pointer to a string holding the destination path including a
                                        trailing path delimiter

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

Note: Use this call to obtain and save hardware debug information. You can call it immediately after receiving an error code <0
         from any library call. It is of particular value after detecting a `FLAG_SYSERROR` from `PH330_GetFlags` and in case of er-
         rors in `PH330_Initialize`. Please provide the saved file(s) for support.

int PH330_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 < 81 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 tim-
         ing jitter. The readings obtained with `PH330_GetCountRate` and `PH330_GetAllCountRates` 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 PH330_SetSyncTrgMode (int devidx, int mode);

```
arguments:              devidx:             device index 0..7
                        mode:               0 = TRGMODE_ETR = set edge trigger mode
                                            1 = TRGMODE_CFD = set constant fraction discriminator mode
return value:           =0                  success
                        <0                  error
```

Note: This call selects the sync channel's trigger mode. Edge trigger mode is useful for pulses with repeatable shape, CFD mode is useful for pulses with fluctuating amplitude but has a longer dead time. After the trigger mode has been changed it must be (re-)configured via PH330_SetSyncEdgeTrg or PH330_SetSyncCFD, respectively.

int PH330_SetSyncEdgeTrg(int devidx, int level, int edge);

```
arguments:              devidx:             device index 0..7
                        level:              trigger level in mV  TRGLVLMIN..TRGLVLMAX
                        edge:               0 = falling, 1 = rising
return value:           =0                  success
                        <0                  error
```

Note: This call is meaningful and permitted only when the sync channel is in TRGMODE_ETR (see PH330_SetSyncTrgMode).

int PH330_SetSyncCFD(int devidx, int level, int zerocross);

```
arguments:              devidx:             device index 0..7
                        level:              trigger level in mV  CFDLVLMIN..CFDLVLMAX
                        zerocross:          zero cross level in mV  CFDZCMIN..CFDZCMAX
return value:           =0                  success
                        <0                  error
```

Note: This call is meaningful and permitted only when the sync channel is in TRGMODE_CFD (see PH330_SetSyncTrgMode).

int PH330_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
```

Note: This is equivalent to changing the cable delay on the sync input. Actual resolution is the device's base resolution.

int PH330_SetSyncChannelEnable (int devidx, int enable);

```
arguments:              devidx:             device index 0..7
                        enable:             desired enable state of the sync channel
                                                0 = disabled
                                                1 = enabled
return value:           =0                  success
                        <0                  error
```

Note: This makes sense only in T2 mode. Histogramming and T3 mode need an active sync signal.

`int PH330_SetSyncDeadTime (int devidx, int on, int deadtime);`

arguments:	devidx:	device index 0..7
	on:	0 = set minimal dead-time, 1 = activate extended dead-time
	deadtime:	extended dead-time in ps
		minimum = EXTDEADMIN
		maximum = EXTDEADMAX
return value:	=0	success
	<0	error

Note: This call is primarily intended for the suppression of afterpulsing artefacts of some detectors. Note that an extended dead-time does not prevent the TDC from measuring the next event and hence enter a new dead-time. It only suppresses events occuring within the extended dead-time from further processing. When an extended dead-time is set then it will also affect the count rate meter readings. The actual extended dead-time is only approximated to the nearest step of the device's base resolution.

`int PH330_SetInputTrgMode (int devidx, int channel, int mode);`

arguments:	devidx:	device index 0..7
	channel:	input channel index 0..nchannels-1
	mode:	0 = TRGMODE_ETR = set edge trigger mode
		1 = TRGMODE_CFD = set constant fraction discriminator mode
return value:	=0	success
	<0	error

Note: This call selects an input channel's trigger mode. Edge trigger mode is useful for pulses with repeatable shape, CFD mode is useful for pulses with fluctuating amplitude. After the trigger mode has been changed it must be configured via `PH330_SetInputEdgeTrg` or `PH330_SetInputCFD`, respectively. The maximum input channel index must correspond to `nchannels-1` with `nchannels` obtained through `PH330_GetNumOfInputChannels`.

`int PH330_SetInputEdgeTrg(int devidx, int channel, int level, int edge);`

arguments:	devidx:	device index 0..7
	channel:	input channel index 0..nchannels-1
	level:	trigger level in mV TRGLVLMIN..TRGLVLMAX
	edge:	0 = falling, 1 = rising
return value:	=0	success
	<0	error

Note: This call is meaningful and permitted only in `TRGMODE_ETR` (see `PH330_SetInputTrgMode`). The maximum input channel index must correspond to `nchannels-1` with `nchannels` obtained through `PH330_GetNumOfInputChannels`.

`int PH330_SetInputCFD(int devidx, int channel, int level, int zerocross);`

arguments:	devidx:	device index 0..7
	channel:	input channel index 0..nchannels-1
	level:	trigger level in mV CFDLVLMIN..CFDLVLMAX
	zerocross:	zero cross level in mV CFDZCMIN..CFDZCMAX
return value:	=0	success
	<0	error

Note: This call is meaningful and permitted only in `TRGMODE_ETR` (see `PH330_SetInputTrgMode`). The maximum input channel index must correspond to `nchannels-1` with `nchannels` obtained through `PH330_GetNumOfInputChannels`.

`int PH330_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: This is equivalent to changing the cable delay on the chosen input. Actual offset resolution is the device's base resolution. The maximum input channel index must correspond to `nchannels-1` where `nchannels` must be obtained through `PH330_GetNumOfInputChannels`.

int PH330_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` with `nchannels` obtained through `PH330_GetNumOfInputChannels`.

int PH330_SetInputDeadTime (int devidx, int channel, int on, int deadtime);

```
arguments:           devidx:             device index 0..7
                     channel:            input channel index 0..nchannels-1
                     on:                 0 = set minimal dead-time, 1 = activate extended dead-time
                     deadtime:           extended dead-time in ps
                                             minimum = EXTDEADMIN
                                             maximum = EXTDEADMAX

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

Note: This call is primarily intended for the suppression of afterpulsing artefacts of some detectors. Note that an extended dead-time does not prevent the TDC from measuring the next event and hence enter a new dead-time. It only suppresses events occuring within the extended dead-time from further processing. When an extended dead-time is set for a channel then it will also affect the corresponding count rate meter readings. Also note that the actual extended dead-time is only approximated to the nearest step of the device's base resolution.

int PH330_SetInputHysteresis (int devidx, int hystcode);

```
arguments:           devidx:             device index 0..7
                     deadtime:           code for the hysteresis
                                             0 = 3mV approx. (default)
                                             1 = 35mV approx.

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

Note: This call is intended for the suppression of noise or pulse shape artefacts of some detectors by setting a higher input hysteresis for the input edge triggers. The setting acts on all input channels simultaneously but it is without effect when an input is in CFD mode. It is only available if the present hardware supports it and will return `PH330_ERROR_INVALID_OPTION` otherwise.

int PH330_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 is meaningful only in Histogramming Mode. It determines if a measurement run will stop when 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 PH330_SetBinning (int devidx, int binning);
```

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

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

Note: Binning only applies in Histogramming and T3 Mode. 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 PH330_SetOffset (int devidx, int offset);
```

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

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

Note: This offset only applies in histogramming and T3 mode. It affects only the difference between stop and start before it is put into the T3 record or is used to increment the corresponding histogram bin. It is intended for situations where the range of the histogram is not long enough to look at "late" data. By means of the offset the "window of view" is shifted to a later range. This is not the same as changing or compensating cable delays. If the latter is desired please use `PH330_SetSyncChannelOffset` and/or `PH330_SetInputChannelOffset`.

```
int PH330_SetHistoLen (int devidx, int lencode, int* actuallen);
```

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

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

Note: This call is only meaningful in histogramming mode. It sets the number of bins of the collected histograms. The histogram length obtained with `MAXLENCODE` is `MAXHISTLEN` while `DFLTLENCODE` results in `DFLTHISTLEN` (65536), which is the default after initialization if `PH330_SetHistoLen` is not called.

```
int PH330_ClearHistMem (int devidx);
```

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

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

Note: This clears the histogram memory of all channels. Only meaningful in histogramming mode.

```
int PH330_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
                                        6 = MEASCTRL_SW_START_SW_STOP
                    startedge:      edge selection code
                                        0 = falling
                                        1 = rising
                    stopedge:       edge selection code
                                        0 = falling
                                        1 = rising

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

Note: This sets the measurement control mode and must be called before starting a measurement. The default after initialization (if this function is not called) is 0, i.e. CTC controlled acquisition time. The modes 1..3 allow hardware triggered measurements through TTL signals at the control port. Note that this needs custom software. For a guideline please see the advanced demos `histomode_extcontrol`. The mode `MEASCTRL_SW_START_SW_STOP` permits controlling the duration of measurements purely by software and thereby overcoming the limit of 100h imposed by the hardware CTC. Note that in this case the results of `PH330_GetElapsedMeasTime` will be less accurate.

```
int PH330_SetTriggerOutput(int devidx, int period);
```

```
arguments:          devidx:         device index 0..7
                    period:         in units of 100ns, TRIGOUTMIN..TRIGOUTMAX, 0 = off

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

Note: This can be used to set the period of the programmable trigger output. The period 0 switches it off. Observe laser safety when using this feature for triggering a laser.

```
int PH330_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
```

Note: If beforehand MEASCTRL_SW_START_SW_STOP is set via `PH330_SetMeasControl`, the parameter tacq will be ignored and the measurement will run until `PH330_StopMeas` is called. This can be used to overcome the limit of 100 h imposed by the hardware CTC. However, the results of `PH330_GetElapsedMeasTime` will in this case be less accurate as it can only use the timers of the operating system.

```
int PH330_StopMeas (int devidx);
```

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

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

Note: This call can be used to force a stop before the acquisition time expires. For clean-up purposes it must in any case be called after a measurement, also if the measurement has expired on its own.

int PH330_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

Note: This call can be used to check if a measurement has expired or is still running.

int PH330_GetHistogram (int devidx, unsigned int *chcount, int channel);

arguments: devidx: device index 0..7
                    chcount   pointer to an array of at least *actuallen* dwords (32bit)
                                        where the histogram data can be stored
                    channel: input channel index 0..nchannels-1

return value: =0   success
                    <0   error

Note: The histogram buffer size must correspond to the value actuallen obtained through PH330_SetHistoLen.
            The maximum input channel index must correspond to nchannels-1 with nchannels obtained through PH330_GetNu-
            mOfInputChannels.

int PH330_GetAllHistograms(int devidx, unsigned int *chcount);

arguments: devidx: device index 0..7
                    chcount: buffer for a multidimensional array of the form
                                        unsigned int histograms[nchannels][actuallen]

return value: =0   success
                    <0   error

Note: This can be used as a replacement for multiple calls to PH330_GetHistogram when all histograms are to be retrieved in
            the most time-efficient way. The multidimensional array receiving the data must be dimensioned according to the number of
            input channels of the device and the chosen histogram length. The corresponding value actuallen can be obtained
            through PH330_SetHistoLen and nchannels can be obtained through PH330_GetNumOfInputChannels.

int PH330_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

Note: This is not meaningful in T2 mode.

int PH330_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

Note: Allow at least 100 ms after PH330_Initialize or PH330_SetSyncDivider or any of the input configuration calls in or-
            der 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
            counter.

int PH330_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 PH330_Initialize 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 with nchannels obtained through PH330_GetNumOfInputChannels.

int PH330_GetAllCountRates(int devidx, int* syncrate, int* cntrates);

arguments: devidx: device index 0..7
                    syncrate: pointer to an integer variable receiving the sync rate
                    cntrates: pointer to an array of integer variables of the form
                                        int cntrates[nchannels] receiving the input rates

return value: =0   success
                    <0   error

Note: This can be used as replacement of PH330_GetSyncRate and PH330_GetCountRate when all rates need to be retrieved in an efficient manner. Make sure that the array cntrates is large enough for the number of input channels your device has. The safest approach is to dimension it for MAXINPCHAN.

int PH330_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 ph330defin.h (e.g. FLAG_OVERFLOW) to extract individual bits through a bitwise AND.

int PH330_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 to obtain the elapsed measurement time of a measurement. This relates to the current measurement when still running or to the previous measurement when already finished. Note that when MEASCTRL_SW_START_SW_STOP is used (controlling the duration of meaurements purely by software) the results of PH330_GetElapsedMeasTime will be less accurate.

int PH330_GetWarnings (int devidx, int* warnings);

arguments: devidx: device index 0..7
                    warnings   pointer to an integer
                                        returns warnings, bitwise encoded (see ph330defin.h)

return value: =0   success
                    <0   error

Note: Prior to this call you must call either PH330_GetAllCountRates or call PH330_GetSyncRate and PH330_GetCout-Rate for all channels. Otherwise the received warnings will at least partially be incorrect or incomplete.

```
int PH330_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 PH330_GetWarnings
return value:	=0	success
	<0	error

Note: This can be used to translate warnings obtained by PH330_GetWarnings to a human-readable text.

```
int PH330_GetSyncPeriod (int devidx, double* period);
```

arguments:	devidx:	device index 0..7
	period:	pointer to a double precision float (64 bit)
		returning the sync period in seconds
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. Resolution (unit) is that of the device's base resolution. Accuracy is determined by single shot jitter and clock stability.

7.2.4.   Special Functions for TTTR Mode

```
int PH330_ReadFiFo (int devidx, unsigned int* buffer, int* nactual);
```

arguments:	devidx:	device index 0..7
	buffer:	pointer to an array of TTREADMAX dwords (32bit)
		where the retrieved TTTR data will be stored
	nactual:	pointer to an integer
		returns the number of TTTR records received
return value:	=0	success
	<0	error

Note: The call will return typically after 10 ms and possibly less if no more data could be fetched. The latency behavior at input rates close to zero is controlled by PH330_SetOflCompression. The actual time to return can also vary due to USB overhead and unpredictable Windows latencies, especially when the PC or the USB connection is slow. The buffer must not be accessed until the call returns.

```
int PH330_SetMarkerEdges (int devidx, int en1, int en2, int en3, int en4);
```

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

Note: This can be used to change the active edge on which the external TTL signals connected to the marker inputs are triggering. Only meaningful in TTTR mode.

```
int PH330_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

Note: This can be used to enable or disable the external TTL marker inputs. Only meaningful in TTTR mode.

```
int PH330_SetMarkerHoldoffTime (int devidx, int holdofftime);
```

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 ±20 ns.

```
int PH330_SetOflCompression (int devidx, int holdtime);
```

arguments: devidx: device index 0..7
                    holdtime: hold time in ms (0..HOLDTIMEMAX)

return value: =0   success
                    <0   error

Note: This setting is not normally required but it can be useful when data rates are very low and there are more overflows than photons. The hardware will then count overflows and only transfer them to the FiFo when holdtime has elapsed. The default value is 2 ms. A value of zero means no compression. If you are implementing a real-time preview and data rates are very low you may observe "stutter" when holdtime is chosen too large because then there is nothing coming out of the FiFo for longer times. Whenever there is a true event record arriving (photons or markers) the previously accumulated overflows will instantly be transferred. This may be the case merely due to dark counts, so the "stutter" would rarely occur. In any case you can switch overflow compression off by setting holdtime 0. Have a look on the file demos to see how overflow records are to be decoded. When compression is off the number of overflows in such a record is always 1. Otherwise it may grow to larger numbers.

7.2.5.   Special Functions for TTTR Mode with Event Filtering

The library supports event filtering in hardware (see section Fehler: Verweis nicht gefunden). This helps to reduce USB bus load in TTTR mode by eliminating photon events that carry no information of interest as typically found in many coincidence correlation experiments. Please read the PicoHarp 330 manual for details.

```
int PH330_SetEventFilterParams(int devidx, int timerange, int matchcnt, int inverse);
```

arguments: devidx: device index 0..7
                    timerange: time distance in ps to other events to meet filter condition
                                     (TIMERANGEMIN..TIMERANGEMAX)
                    matchcnt: number of other events needed to meet filter condition
                                     (MATCHCNTMIN..MATCHCNTMAX)
                    inverse: set regular or inverse filter logic
                                        0 = regular,
                                        1 = inverse

return value: =0   success
                    <0   error

Note: This sets the parameters for the Event Filter implemented in the FPGA hardware. The value `timerange` determines the time window the filter is acting on. The parameter `matchcnt` specifies how many other events must fall into the chosen time window for the filter condition to act on the event at hand. The parameter `inverse` inverts the filter action, i.e. when the filter would regularly have eliminated an event it will then keep it and vice versa. For the typical case, let it be not inverted. Then, if `matchcnt` is 1 we obtain a simple 'singles filter'. This is the most straight forward and most useful filter in typical quantum optics experiments. It will suppress all events that do not have at least one coincident event within the chosen time range, be this in the same or any other channel. In order to mark individual channel as 'use' and/or 'pass' please use `PH330_SetEventFilterChannels`.The parameter settings are irrelevant as long as the filter is not enabled.

```
int PH330_SetEventFilterChannels(int devidx, int usechannels, int passchannels);
```

arguments: devidx: device index 0..7
                    usechannels: integer bitfield with bit0 = leftmost input channel,..
                                    bit7 = rightmost input channel,
                                    bit8 = sync channel,
                                    bit9 and higher must be 0
                                        bit value 1 = use this channel,
                                        bit value 0 = ignore this channel
                    passchannels: integer bitfield with bit0 = leftmost input channel,..
                                    bit7 = rightmost input channel,
                                    bit8 = sync channel
                                    bit9 and higher must be 0
                                        bit value 1 = unconditionally pass this channel,
                                        bit value 0 = pass this channel subject to filter condition

return value: =0   success
                    <0   error

Note: This selects the filter channels. The bitfield usechannels is used to indicate if a channel is to be used by the filter. The bit-field passchannels is used to indicate if a channel is to be passed through the filter unconditionally, whether it is marked as 'use' or not. The events on a channel that is marked neither as 'use' nor as 'pass' will not pass the filter, provided the filter is enabled. The settings for the sync channel are meaningful only in T2 mode and will be ignored in T3 mode. The channel settings are irrelevant as long as the filter is not enabled. .

```
int PH330_EnableEventFilter(int devidx, int enable);
```

arguments: devidx: device index 0..7
                    enable: desired enable state of the filter
                                        0 = disabled
                                        1 = enabled

return value: =0   success
                    <0   error

Note: When the filter is disabled all events will pass. This is the default after initialization. When it is enabled, events may be filtered out according to the parameters set with PH330_SetEventFilterParams and PH330_SetEventFilterChannels.

```
int PH330_SetFilterTestMode(int devidx, int testmode);
```

arguments: devidx: device index 0..7
                    testmode: desired mode of the filter
                                        0 = regular operation
                                        1 = testmode

return value: =0   success
                    <0   error

Note: One important purpose of the event filters is to reduce USB load. When the input data rates are higher than the USB band-with, there will at some point be a FiFo overrun. It may under such conditions be difficult to empirically optimize the filter set-tings. Setting filter test mode disables all data transfers into the FiFo so that a test measurement can be run without interrup-tion by a FiFo overrun. The library routines PH330_GetFilterInputRates and PH330_GeFilterOutputRates can then be used to monitor the count rates before and after the filter. When the filtering effect is satisfactory the test mode can be switched off again to perform the regular measurement.

```
int PH330_GetFilterInputRates(int devidx, int* syncrate, int* cntrates);
```

arguments: devidx: device index 0..7
                    syncrate: pointer to an integer variable receiving the sync rate
                    cntrates: pointer to an array of integer variables of the form
                                        int cntrates[num_channels] receiving the count rates

return value: =0   success
                    <0   error

Note: This call retrieves the count rates before entering the filter. A measurement must be running to obtain valid results. Allow at least 100 ms to get a new reading. This is the gate time of the rate counters. Make sure that the array cntrates is large enough for the number of input channels your device has. The safest approach is to dimension it for MAXINPCHAN.

```
int PH330_GetFilterOutputRates(int devidx, int* syncrate, int* cntrates);
```

```
arguments:              devidx:             device index 0..7
                        syncrate:           pointer to an integer variable receiving the sync rate
                        cntrates:           pointer to an array of integer variables of the form
                                               int cntrates[num_channels]  receiving the count rates

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

Note: This call retrieves the count rates after the filter before entering the FiFo. A measurement must be running to obtain valid results. Allow at least 100 ms to get a new reading. This is the gate time of the rate counters. Make sure that the array cntrates is large enough for the number of input channels your device has. The safest approach is to dimension it for MAXINPCHAN.

7.3. Warnings

The following is related to the warnings (possibly) generated by the library routine `PH330_GetWarnings`. The mechanism and warning criteria are the same as those used in the regular PicoHarp 330 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 `PH330_GetSyncrate` and `PH330_GetCoutrate` have been called (the latter for all channels) before `PH330_GetWarnings` is called. For speed you can use `PH330_GetAllCoutrates` instead.

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_VERY_LOW The detected pulse rate at the sync input is below 100 Hz and cannot be determined accurately. Other warnings may not be reliable under this condition.	√		√
WARNING_SYNC_RATE_TOO_HIGH The pulse rate at the sync input (after the divider) is higher than 81 MHz. This is close to the TDC limit. Sync events will be lost above 82 MHz. T2 mode is normally intended to be used without a fast sync signal and without a divider. If you see this warning in T2 mode you may accidentally have connected a fast laser sync.	√	√	√
WARNING_INPT_RATE_ZERO No counts are detected at any of the input channels. In histogramming and T3 mode these are the photon event channels and the measurement will yield nothing. You might sporadically see this warning if your detector has a very low dark count rate and is blocked by a shutter. In that case you may want to ignore or disable this warning.	√	√	√
WARNING_INPT_RATE_TOO_HIGH The overall pulse rate at the input channels is higher than 80 MHz (USB 3.0 connection) or higher than 9 MHz (USB 2.0 connection). This is close to the throughput limit of the present USB connection. The measurement will likely lead to a FIFO overrun. There are some rare measurement scenarios where this condition is expected and the warning can be ignored or 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 ignored or disabled. Examples are antibunching measurements or rapidFLIM where pile-up is either tolerated or corrected for during data analysis. One can usually also ignore this warning when the current time bin width is larger than the dead-time.	√		√
WARNING_DIVIDER_GREATER_ONE In T2 mode: The sync divider is set larger than 1. This is probably not intended. The sync divider is designed primarily for high sync rates from lasers and requires a fixed pulse rate at the sync input. In that case you should use T3 mode. If the signal at the sync input is from a photon detector (coincidence correlation etc.) a divider > 1 will lead to unexpected results. There are rare measurement scenarios where this condition is intentional and the warning can be ignored or disabled. In histogramming and T3 mode: If the pulse rate at the sync input is below 81 MHz then a sync divider > 1 is not needed. The measurement may yield unnecessary jitter if the sync source is not very stable.	√	√	√
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: 　Span = Resolution * Length Length is 32768 in T3 mode. In histogramming mode it depends on the chosen histogram length (default is 65536). Events outside this span will not be recorded. There are some measurement scenarios where this condition is intentional and the warning can be ignored or 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 (see calculation above) 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 ignored or disabled.	√		√
WARNING_COUNTS_DROPPED This warning is issued when the front end of the data processing pipeline was not able to process all events that came in. This will occur typically only at very high count rates during intense bursts of events.	√	√	√

WARNING_USB20_SPEED_ONLY The PicoHarp 330 is designed for USB 3.0 superspeed (5Gbits/s). This warning is issued when the device is connected only at the speed of USB 2.0 (480Mbits/s). This works but will result in severely limited throughput. Check USB ports and cables in use. The same issue is indicated by the USB status LED showing yellow instead of green.	√	√	√

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.