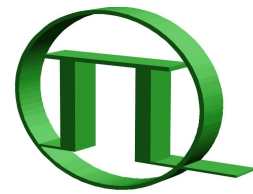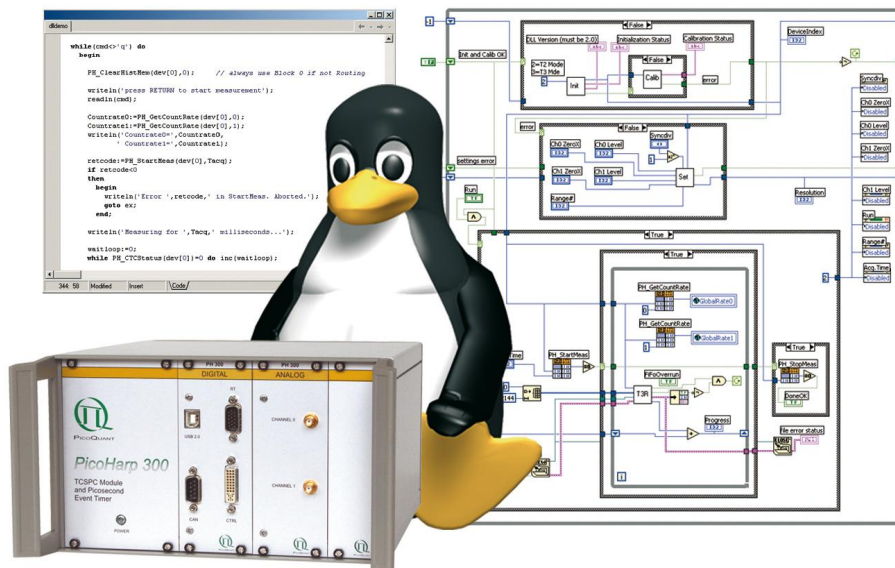# PicoHarp 300

Picosecond Histogram
Accumulating Real-time Processor

PicoQuant GmbH
Unternehmen für optoelektronische
Forschung und Entwicklung

## PHLib Programming Library
## for Custom Software Development
## under Linux

# User's Manual

Version 3.0 - January 2014

# Table of Contents

# 1. Introduction

The PicoHarp 300 is a compact and easy-to-use TCSPC system with USB interface. Its new design enhances functionality, keeps cost down, improves reliability and simplifies calibration. The timing circuit allows high measurement rates up to 10 Mcounts/s and provides a time resolution of 4 ps. The input channels are programmable for a wide range of input signals. They both have programmable Constant Fraction Discriminators (CFD). These specifications qualify the PicoHarp 300 for use with all common single photon detectors such as Single Photon Avalanche Photodiodes (SPAD), Photo Multiplier Tubes (PMT) and MCP-PMT modules (PMT and MCP-PMT via preamp). The time resolution is well matched to fast detectors and the overall Instrument Response Function (IRF) will not be limited by the PicoHarp electronics. Similarly, inexpensive and easy-to-use diode lasers such as the PDL 800-B with interchangeable laser heads can be used as an excitation source perfectly matched to the time resolution offered by the detector and the electronics. Overall IRF widths of 200 ps FWHM can be achieved with inexpensive PMTs and diode lasers. Down to 50 ps can be achieved with selected diode lasers and MCP-PMT detectors. 30 ps can be reached with femtosecond lasers. This permits lifetime measurements down to a few picoseconds with deconvolution e.g. via the FluoFit multiexponential Fluorescence Decay Fit Software. For more information on the PicoHarp 300 hardware and software please consult the PicoHarp 300 manual. For details on the method of Time-Correlated Single Photon Counting, please refer to our TechNote on TCSPC.

The PicoHarp 300 standard software for Windows provides functions such as the setting of measurement parameters, display of results, loading and saving of measurement parameters and histogram curves. Important measurement characteristics such as count rate, count maximum and position, histogram width (FWHM) are displayed continuously. While these features will meet many of the routine demands, advanced users may want to include the 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 this approach may be of interest. For this purpose a programming library is provided as a Dynamic Link Library (DLL) for Windows. As an alternative, a Linux version of the library is offered, which is subject of this manual. It is 100% API compatible with the Windows library, so that applications can be ported very easily.

The shared library PHLib.so supports custom programming under Linux in all major programming languages, notably C/C++, C#, Pascal, MATLAB and LabVIEW. This manual describes the installation and use of the PicoHarp programming library PHLib.so and explains the associated demo programs. Please read both this manual and the PicoHarp manual before beginning your own software developement with the library. The PicoHarp 300 is a sophisticated real-time measurement system. In order to work with the system using the library, sound knowledge in your chosen programming language is required.

# 2. General Notes

This release of the PicoHarp 300 programming library is suitable for x86 systems with Linux versions 3.0. and higher. There are separate versions for 32 and 64 bit. The library uses Libusb so that no kernel driver is required. Note that the library is provided as a binary object only.

This manual assumes that you have read the PicoHarp 300 manual and that you have experience with the chosen programming language. References to the PicoHarp manual will be made where necessary.

This version of the library supports histogramming mode and both TTTR modes but TTTR mode is only usable if the TTTR option was purchased and installed in the device firmare memory.

Users who purchased a license for any older version of the library will receive free updates when they are available. For this purpose, please register by sending email to *info@picoquant.com* with your name, your PicoHarp 300 serial number and the email address you wish to have the update information sent to.

Users upgrading from earlier versions of the PicoHarp 300 library or moving over from the Windows DLL need to adapt their programs. Some changes are usually necessary to accommodate new measurement modes and improvements. However, the required changes are mostly minimal and will be explained in the manual (especially check section 6, 7 and the notes marked red in section 10).

Note that despite of our efforts to keep changes minimal, data structures, program flow and function calls may still change in future versions without advance notice. Users must maintain appropriate version checking in order to avoid incompatibilities. There is a function call tat you can use to retrieve the version number (see section 10). Similarly, the development of Linux is highly dynamic and distributions may vary considerably. This manual can therefore only try and describe a snapshot valid at the time of writing.

**Disclaimer**

PicoQuant GmbH disclaims all warranties with regard to this software including all implied warranties of merchantability and fitness. 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 or performance of this software. Demo code is provided 'as is' without any warranties as to fitness for any purpose.

**License and Copyright Notice**

With this product you have purchased a license to use the PicoHarp 300 programming library. You have not purchased any other rights to the software itself. The software is protected by copyright and intellectual property laws. You may not distribute the software to third parties or reverse engineer, decompile or disassemble the software or part thereof. You may use and modify demo code to create your own software. Original or modified demo code may be re-distributed, provided that the original disclaimer and copyright notes are not removed from it.

*PicoHarp* is a registered trademark of PicoQuant GmbH.

Other trademarks, mostly related to operating systems, programming languages and their components etc. are property of their respective owners and are used here for explanatory purposes only, to the owner's benefit and without the intent to infringe.

PHLib.so dynamically links with Libusb.so to acces the PicoHarp USB device. Libusb for Linux is licensed under the LGPL which allows a fairly free use even in commercial projects. For details and precise terms please see http://libusb.sourceforge.net/.

# 3. Firmware Update

Note: You can skip this section if you bought your PicoHarp with the DLL option readily installed.

The PicoHarp 300 programming library requires a firmware update of your PicoHarp 300, unless you already bought it with the DLL option installed. The update must be performed by DLLUPD.EXE under Windows. This file is provided to you (typically be email) only if you purchased an upgrade for the DLL option. It is compiled specifically for the serial number of your PicoHarp and cannot be used on others. The firmware update only needs to be done once. If necessary, perform the following steps to install the update:

(see your PicoHarp manual for steps 1..3 listed below)

1.  Make sure your PicoHarp 300 is powered and connected correctly through USB 2.0.
2.  Check that the standard PicoHarp 300 software for Windows runs correctly.
3.  Make sure to exit the PicoHarp 300 software.
4.  Start the program DLLUPD.EXE from a temporary disk location.
5.  Follow the instructions. Do not interrupt the actual update progress, it may take a minute or so. The program will report successful completion.

After successful completion of the upgrade your PicoHarp is ready to use PHLib. See the sections below for hints how to install and use it.

# 4. Installation of the PHLib Software Package

### 4.1 Requirements

Supported hardware is at this time the x86 platform only (32 or 64 bit). Recommended is a PC with at least 1 GHz CPU clock and 1 GB of memory. For serious use of the TTTR measurement modes a speedy hard disk is recommended.

The PicoHarp 300 library is designed to run on Linux kernel versions 3.x. It has been tested with the following Linux distributions:

OpenSuSE 12.1, 12.3 and 13.1
Kubuntu 12.04

Using the PicoHarp 300 library requires libusb 0.1.12 or the combination of Libusb 1.0 and Libusb-Compat (see http://libusb.sourceforge.net/). The latter combination is part of all recent Linux distributions and should typically be readily installed on your system.

It is recommended to start your work with the PicoHarp 300 by using the standard interactive PicoHarp data acquisition software under Windows. This should give you a better understanding of the instrument's operation before attempting your own programming efforts. It also ensures that your optical/electrical setup is working. If you are planning to use a router, try to get everything working without router first to avoid additional complications.

### 4.2 Libusb Access Permissions

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

        usbfs /proc/bus/usb usbfs defaults  0  0

This should routinely be the case if you installed any of the tested distributions.

The permissions for the device files used by libusb must be adjusted for user access. Otherwise only root can use the devices. The device files are located in /proc/bus/usb/. Any manual change would not be permanent, however. The permissions will be reset after reboot or replugging the device. A much more elegant way of finding the right files and setting the suitable permissions is by means of hotplugging scripts or udev. Which mechanism you can use depends on the Linux distribution you have. Recent distributions have replaced hotplug with udev.

**Hotplug**

For automated setting of the device file permissions with hotplug you have to add an entry to the hotplug "usermaps". These files are located in /etc/hotplug/. If the entire hotplug directory does not exist, you probably don't have the hotplug package installed. This may be because your distribution is rather old (you may still be able to install hotplug) or it is very new and uses udev instead of hotplug (see section below).

Every line in the hotplug usermap files is for one device, the line starts with a name that will also be used for a handler script. Here we are trying to get a PicoHarp300 device working, so let's call the entry "PicoHarp300". The next entry is always 0x0003. The following two entries are the vendor ID (0x0e0d for PicoQuant) and the product ID (0x0003 for the PicoHarp 300). The remaining fields are all set to 0.

The line for the PicoHarp should look like below. The whole string should be on one line. Here it only wraps around due to limited page width:

```
PicoHarp300    0x0003    0x0e0d 0x00003  0x0000     0x0000
               0x00      0x00      0x00      0x00      0x00          0x00      0x00000000
```

The handler script "PicoHarp300" gets started when the device is connected or disconnected. It looks like this:

```
#!/bin/bash

if [ "${ACTION}" = "add" ] && [ -f "${DEVICE}" ]
then
     chown root "${DEVICE}"
     chgrp users "${DEVICE}"
     chmod 666 "${DEVICE}"
fi
```

When the PicoHarp gets switched on or connected, the hotplug system will recognize it and will run the script, which will then change the permissions on the device file associated with the device to 666, which means that everybody can both read and write from/to this device. Note that this may be regarded as a security gap. If you whish tighter access control you can create a dedicated group for PicoHarp users and give access only to that group.

Both a usemap file *PicoHarp300.usermap* and the handler script *PicoHarp300* are provided in the hotplug folder on the PHLib distribution media. Copying both of them to  /etc/hotplug/usb/  should be sufficient to set up acces conrol via hotplugging for your PicoHarp.

**Udev**

As noted above, recent distributions replace hotplug with udev. You won't find /etc/hotplug in this case. Instead, you can use udev to create the devices so they are readable and writable by non-priveleged users. The udev rules are contained in files in /etc/udev/rules.d.  Udev processes these files in alphabetical order. The default file is usually called  50-udev.rules.  Don't change this file as it could be overwritten when you upgrade udev. Instead, write your custom rule for the PicoHarp in a separate file. The contents of this file for the handling of the PicoHarp 300 should be:

```
ATTR{idVendor}=="0d0e", ATTR{idProduct}=="0003", MODE="0666"
```

A suitable rules file *PicoHarp.rules* is provided in the udev folder on the PHLib distribution media. You can simply copy it to the rules.d folder. The install script in the same folder does just this. Note that the name of the rules file is important. Each time a device is detected, the files are read in alphabetical order, line by line, until a match is found. Note that different distributions may use different rule file

names for various categories. For instance, Kubuntu organizes the rules into further files: 20-names.rules, 40-permissions.rules, and 60-symlinks.rules. In Fedora they are not separated by those categories, as you can see by studying 50-udev.rules. Instead of editing the existing files, it is therefore usually recommended to put all of your modifications in a separate file like 10-udev.rules or 10-local.rules. The low number at the beginning of the file name ensures it will be processed before the default file. However, later rules that are more general (applying to a whole class of devices) may later override the desired acecss rights. This is the case for USB devices handled through Libusb. It is therefore important that you use a rules file for the PicoHarp that gets evaluated after the general case. The default naming "PicoHarp300.rules" most likely ensures this.

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

# udevcontrol reload_rules
# udevstart

## 4.3 Installing the Library

The library is distributed as a binary file. By default it resides under */usr/local/lib/ph300* (32 bit) or */usr/local/lib64/ph300* (64 bit). This is not a strict requirement but it is where the demo programs will look for the library files and therefore it is recommended to use this location. You can create the directory */usr/local/lib/ph300 (32 bit) or /usr/local/lib64/ph300 (64 bit)* and copy all files from the directory *library* on the distribution media to that location (p*hlib.so, phlib.h, phdefin.h* and *errorcodes.h*). The shell script *install* in the *library* distribution directory does the directory creation and installation in one step. As root, just issue it at the command prompt from within the *library* directory. After installing, the library is ready to use and can be tested with the demos provided.

Note that Lazarus and Mono are somewhat picky as to what they accept as a library name. Both expect a name starting with *lib.* Lazarus also does not easily allow linking with a library that is not in the default path. In order to solve both problems, the install script for PHLib creates a link making it also appear under the alias */usr/lib/libph300.so* (32 bit) or */usr/local/lib64/libph300.so* (64 bit). This is the path the Lazarus and Mono demos use for accessing phlib.so.

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*. In this case, observe the special requirements for Lazarus and Mono.

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

chcon -t texrel_shlib_t /usr/local/lib/ph300/phlib.so

## 4.4 Installing the Demo Programs

The demos can be installed by simply copying the entire directory *demos* from the distribution media to a disk location of your choice. This need not be under the root account but you need to ensure proper file permissions. While the C compiler for the C demos, as well as Mono for C# are part of all recent linux distributions, you will probably need to obtain and install Lazarus/Pascal, Matlab or LabVIEW for Linux separately if you wish to use these programming environments.

# 5. New in this Version

Version 3.0 of  PHLib constitutes a major overhaul with respect to the previous version 2.3. It provides new routines to change the time offset of the input channels, which eliminates the need for cable delay adjustments. There are also new routines to obtain information on hardware features and debug information. Furthermore, there is now a programmable holdoff time for marker signals in TTTR mode and the markers can be enabled/disabled individually. Multistop in histogramming and T3 mode can now be disabled if necessary. Finally, the policy of return values has been changed to carry only error/ success information and never any other data. The latter are now always passed by reference. The new and changed routines are marked in red in section 10. Users upgrading from earlier versions of the PHLib will need to adapt their programs. It is important to note here one more time that you must maintain appropriate version checking in order to avoid crashes or malfunction due to such changes. There is a function call tat you can use to retrieve the version number (see section 10).

# 6. The Demo Applications - Functional Overview

Please note that all demo code provided is correct to our best knowledge, however, we must disclaim all warranties as to fitness for a particular purpose of this code. It is provided 'as is' for no more than explanatory purposes and a starting point for your own work. For any practical work the demo programs will almost always require modification.

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

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

There are demos for C, C#, Pascal, LabVIEW and MATLAB. The demos are 99% identical to those for Windows, so that code can easily be ported between the platforms. For each of the programming languages/systems there are different demo versions for various measurement modes:

**Standard Mode Demos**

These demos show how to use the standard measurement mode for on-board histogramming. These are the simplest demos and the best starting point for your own experiments. In case of LabVIEW the standard mode demo is already fairly sophisticated and allows interactive input of most parameters. The standard mode demos will not initialize or use a router that may be present. Please do not connect a router for these demos.

**Routing Demos**

Multi channel measurement (routing) is possible in standard histogramming mode and in TTTR mode. It requires that a PHR 40x or PHR 800 router and multiple detectors are connected. The routing demos show how to perform such measurements in histogramming mode and how to access the histogram data of the individual detector channels. The concept of routing in histogramming mode is quite simple and similar to standard histogramming. In TTTR mode it is also very simple using the same library routines. Therefore, no dedicated demo is provided for routing in TTTR mode. To get started see the section about routing in your PicoHarp 300 manual. If you need the routing functionality in any of the other demos you need to copy the relevant code fragments from the routing demos.

**TTTR Mode Demos**

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

The PicoHarp 300 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. TTTR mode always implicitly performs routing if a router and multiple detectors are used. It is also possible to record external TTL signal transitions as markers in the TTTR data stream e.g. for image scanning applications (see the PicoHarp manual).

Note: TTTR mode is not part of the standard PicoHarp product. It must be purchased as a separate firmware option that gets burned into the ROM of the device. An upgrade is possible at any time. It always includes both T2 and T3 mode.

Because TTTR mode requires real-time processing and/or real-time storing of data, the TTTR demos are fairly demanding both in programming skills and computer performance. See the section about TTTR mode in your PicoHarp manual as well as section 8 here.

# 7. The Demo Applications by Programming Language

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

This manual explains the special aspects of using the PicoHarp programming library, it does NOT teach you how to program in the chosen programming language. We strongly recommend that you do not choose a development with the PicoHarp library as your first attempt at programming. Please study the documentation of your chosen programming language, especially on how to call routines in dynamic link libraries. 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 PHlib (*phlib.h, phdefin.h* and *errorcodes.h*).

Be warned that wrong parameters and/or variables, invalid pointers and buffer sizes, inappropriate calling sequences etc. may crash your application and/or your complete computer. Make sure to backup your data and/or perform your development work on a dedicated machine that does not contain valuable data. Note that the library is not re-entrant. This means, it cannot be accessed from multiple, concurrent processes or threads at the same time. All calls must be made sequentially in the order shown in the demos.

**The C/C++ Demos**

The demos are provided in the 'C' subfolder. The code is in plain C to provide the smallest common denominator for C and C++. Consult *phlib.h*, *phdefin.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 "phdefin.h"
#include "phlib.h"
}
```

To test any of the demos, consult the PicoHarp manual for setting up your PicoHarp 300 and establish a measurement setup that runs correctly and generates useable test data. Compare the settings (notably sync divider, range and CFD levels) with those used in the demo and use the values that work in your setup when building and testing the demos.

All C demos can be compiled with gcc. They come with a makefile so that a simple call of make in the respective source folder should readily build the application. PHlib and Libusb are linked dynamically.

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

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

### The C# Demos

The C# demos are provided in the 'Csharp' subfolder. They have been tested with Mono 2.10.6 and 2.10.9 (see http://mono-project.com/) under Windows and Linux. The only difference is  the library name, which in principle could also be unified.

Calling a native DLL (unmanaged code) from C# requires the DllImport attribute and correct type specification of the parameters. Not all types are easily portable. Especially C strings require special handling. See Calling.txt.

With the C# demos you also need to check wether the hardcoded settings are suitable for your actual instrument setup. The demos are designed to run in a console window. They need no command line input parameters. They create their output files in their current working directory (*.out). The output files will be ASCII in case of the standard and routing demos. For continuous and TTTR mode the output is stored in binary format for performance reasons. The ASCII files will contain single or multiple columns of integer numbers representing the counts from the 65536 histogram channels. You can use any editor or a data visualization program to inspect the ASCII histograms. The binary files must be read by dedicated programs according to the format they were written in.

### The Pascal Demos

Pascal users please refer to the *Lazarus* branch of the *demo* directory. Lazarus is a rapid application development system for Linux and Windows based on the Free Pascal language. It feels very much like Delphi for Windows but allows developing cross-platform portable programs for Linux and Windows. The code for the respective demo is in the Delphi project file for that demo (*.DPR). In fact, the same project can be used with Delphi for Windows, only by changing calling convention and the library path and name. Lazarus accesses the DPR file through the corresponding LPI file. In order to make the exports of the library 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 supplied demos were tested with Lazarus 0.9.30 and 1.0.8.

Note that Lazarus is somewhat picky as to what it accepts as a library name. It expects a name starting with `lib`. It also does not easily allow linking with a library that is not in the default path. In order to solve both problems, the install script for PHLib creates a link making it also appear under the alias `libph300.so`. This is what the Lazarus demos use for accessing phlib.so.

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

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

**The LabVIEW Demos**

The LabVIEW demo VIs are provided in the 'Labview' directory. They are contained in LabVIEW libraries (*.llb). The top level VI is always 'PicoHarp.vi'. Note that the sub-VIs in the various demos are not always identical, even though their names may be the same. The files are saved for LabVIEW 8.0 for Linux. Newer versions will probably work but have not been tested.

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

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

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

Internally the TTTR mode demo also requires a special note: each TTTR record as returned in the buffer of PH_TTReadData actually is a DWORD (32bit). However, LabVIEW stores DWORD data (U32) always in big endian format. On the x86 platform (little endian) this results in reversed bytes compared  to C programs. For consistency with the demo programs for reading TTTR data this byte reversing of the data going to disk is avoided in the demo by declaring the buffer for PH_TTReadData as a byte array (hence 4 times longer than the DWORD array). You may instead want to work with a U32 array if your goal is not storing data to disk but doing some on-line analysis of the TTTR records. In this case you must initialize the array with 65536 x U32  and change the type of buffer in the library calls of PH_TTReadData to U32.

The LabVIEW demos access the library routines via the 'Call Library Function' of LabVIEW. For details refer to the LabVIEW documentation. Consult phlib.h and section 10 of this manual for the parameter types etc.

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

**The MATLAB Demos**

The MATLAB demos are provided in the 'Matlab' directory. They are contained in m-files. You need to have a MATLAB version that supports the 'calllib' function. We have tested with MATLAB 7.3 but any version from 6.5 should work.  Be very careful about the header file name specified in 'loadlibrary'. This name is case sensitive and a wrong spelling will lead to an apparently successful load but later no library calls will work.

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

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

# 8. Advanced Techniques

**Using Multiple Devices**

Starting from version 2.0 the library is designed to work with multiple PicoHarp devices (up to 8). The demos always use the first device found. If you have more than one PicoHarp and you want to use them together you need to modify the code accordingly. At the API level of PHLib the devices are distinguished by a device index (0..7). The device order corresponds to the order Libusb enumerates the devices. This can be somewhat unpredictable. It may therefore be difficult to know which physical device corresponds to the given device index. In order to solve this problem, the library routine PH_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 PH_GetSerialNumber any time later on a device you have successfully opened. The serial number of a physical PicoHarp device can be found at the back of the housing. It is a 8 digit number starting with 0100. The leading zero will not be shown in the serial number strings retrieved through PH_OpenDevice or PH_GetSerialNumber.

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

Note that an attempt at opening a device that is currently used by another process will result in the error code ERROR_DEVICE_BUSY being returned from PH_OpenDevice. Unfortunately this cannot be distinguished from a failure to open the device due to insufficient access rights (permissions). Such a case also results in  ERROR_DEVICE_BUSY.

As outlined above, if you have more than one PicoHarp and you want to use them together you need to modify the demo code accordingly. This requires briefly the following steps: Take a look at the demo code where the loop for opening the device(s) is. In most of the demos all the available devices are opened. You may want to extend this so that you 1) filter out devices with a specific serial number and 2) do not hold open devices you don't actually need. The latter is recommended because a device you hold open cannot be used by other programs.

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

**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 300 uses USB 2.0 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 this permits data throughput as high as 5 Mcps and leaves time for the host to perform other useful things, such as on-line data analysis or storing data to disk.

In TTTR mode the data transfer process is exposed to the library user in a single function PH_TTReadData that accepts a buffer address where the data is to be placed, and a transfer block size. This block size is critical for efficient transfers. The larger the block size, the better the transfer efficiency. This is because setting up and completing a transfer costs some fixed amount of time, independent of the block size. The maximum transfer block size permitted by PHLib is 131072 (128k event records). It may not under all circumsances be sensible to use the maximum size. The demos use a medium size of 32768 records.

As noted above, the transfer is implemented efficiently without using the CPU excessively. Nevertheless, assuming large block sizes, the transfer takes some time. The operating system therefore gives the unused CPU time to other processes or threads, i.e., it waits for completion of the transfer without burning CPU time. This wait time is what can also be used for doing 'useful things' in

terms of any desired data processing or storing within your own application. The best way of doing this is to use multithreading. In this case you design your program with two threads, one for collecting data (i.e. working with PH_TTReadData) and another for processing or storing the data. Multiprocessor systems can benefit from this technique even more. Of course you need to provide an appropriate data queue between the two threads and the means of thread synchronization. Thread priorities are another issue to be considered. Finally, if your program has a graphic user interface you may need a third thread to respond to user actions reasonably fast. Again, this is an advanced technique and it cannot be demonstrated in detail here. Greatest care must be taken not to access PHLib from different threads without strict control of mutual exclusion and maintaining the right sequence of function calls. However, the technique allows throughput improvements of 50..100% and advanced programmers may want to use it. It might be interesting to note that this is how TTTR mode is implemented in the regular PicoHarp software for Windows, where sustained count rates over 5 millions of  counts/sec (to disk) can be achieved on modern PCs.

When working with multiple PicoHarp devices, the overall USB throughput is limited by the host controller or any hub the devices must share. You can increase overall throughput if you connect the individual devices to separate host controllers without using hubs. If you install additional USB controller cards you should prefer PCI-express models. Traditional PCI can become a bottleneck in itself. However, modern mainboards often have multiple USB host controllers, so you may not even need extra controller cards. In order to find out how many USB controllers you have and which bus the individual devices are attached to, you can use lsusb. In case of using multiple devices it is also beneficial for overall throughput if you use multithreading 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.

**Working with Very Low Count Rates**

As noted above, the transfer block size is critical for efficient transfers. The larger the block size, the better the transfer efficiency. This is because setting up a transfer costs some fixed amount of time, independent of the block size. However, it may not under all circumsances be ideal to use the maximum size. A large block size takes longer to fill. If the count rates in your experiment are very low, it may be better to use a smaller block size. This ensures that the transfer function returns more promptly. It should be noted that the PicoHarp has a "watchdog" timer that terminates large transfers prematurely so that they do not wait forever. The timeout period is approximately 80 ms. This results in PH_TTReadData returning less than requested (possibly even zero). This helps to avoid complete stalls even if the maximum transfer size is used with low or zero count rates. However, for fine tuning of your application it may still be of interest to use a smaller block size. The block size must be an integer muliple of 512.

**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 software for Windows (see also section 11). In order to obtain and use these warnings in your custom software you may want to use the library routine PH_GetWarnings. This may help inexperienced users to notice possible mistakes before stating a measurement or even during the measurement.
It is important to note that the generation of warnings is dependent on the current count rates and the current measurement settings. It was decided that PH_GetWarnings does not obtain the count rates on its own, because the corresponding USB calls take some time and might waste too much processing time. It is therefore necessary that PH_GetCoutrate has been called for all channels before PH_GetWarnings is called. Since most interactive measurement software periodically calls PH_GetCoutrate anyhow, this is not a serious complication and avoids redundant USB calls.
The routine PH_GetWarnings only delivers the cumulated warnings in the form of a bit field. In order to translate this into readable information you can use PH_GetWarningsText. Before passing the bit field into  PH_GetWarningsText you can mask out individual warnings by means of the bit masks defined in phdefin.h.

# 9. Data Types

The PicoHarp programming library PHLib is written in C and its data types correspond to standard C/C++ data types as follows:

| | |
|---|---|
| char | 8 bit, byte (or character in ASCII) |
| short int | 16 bit, signed integer |
| unsigned short int | 16 bit, unsigned integer |
| int | 32 bit, signed integer |
| unsigned int | 32 bit, unsigned integer |
| long int | 32 bit, signed integer |
| unsigned long int | 32 bit, unsigned integer |
| float | 32 bit, floating point |
| double | 64 bit, floating point |

These types are supported by all major programming languages.

# 10.  Functions exported by PHLib

See **phdefin.h** for predefined constants given in capital letters here. Return values <0  denote errors.
See **errorcodes.h** for the error codes.

**General Functions**

These functions work independent from any device.

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

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

return value:         =0                success
                      <0                error
Note:                 This function is provided to obtain readable error strings that explain
the                   cause of the error better than the numerical error code. Use these in er-
ror                   handling message boxes etc.
```

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

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

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

Note:      This is the only function you may call before opening and initializing a device. Use it to ensure compatibility of the library with
           your own application. Take this seriously if you do not want to end up in a mess when new versions come out.

**Open/Close/Initialize Functions**

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

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

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

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

```
int PH_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 PH_Initialize (int devidx, int mode);
```

```
arguments:            devidx:           device index 0..7
                      mode:             0 = histogramming,
                                        2 = T2_Mode
                                        3 = T3_Mode

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

### Functions for Initialized Devices

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

```
int PH_GetHardwareInfo (int devidx, char* model, char* partnum, char* vers);   changed in v.3.0
```

```
arguments:          devidx:         device index 0..7
                    model:          pointer to a buffer for at least 16 characters
                    partnum:        pointer to a buffer for at least  8 characters
                    vers:           pointer to a buffer for at least  8 characters
return value:       =0              success
                    <0              error
```

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

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


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

```
int PH_GetBaseResolution (int devidx, double* resolution);                          changed in v.3.0
```

```
arguments:          devidx:         device index 0..7
                    resolution:     base resolution of the device (passed by reference)
return value:       =0              success
                    <0              error
```

```
int PH_GetFeatures (int devidx, int* features);                                    new in v.3.0
```

```
arguments:          devidx:         device index 0..7
                    features:       features of this device (a bit pattern, passed by
                                    reference)
return value:       =0              success
                    <0              error
```

Note:      Use the predefined bit mask values in phdefin.h to probe for a specific feature

```
int PH_Calibrate (int devidx);
```

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

```
int PH_SetInputCFD (int devidx, int channel, int level, int zerocross);     changed in v.3.0
```

```
arguments:          devidx:         device index 0..7
                    channel:        number of the input channel (0 or 1)
                    level:          CFD discriminator level in millivolts
                                        minimum = DISCRMIN
                                        maximum = DISCRMAX

                    level:          CFD zero cross in millivolts
                                        minimum = ZCMIN
                                        maximum = ZCMAX
return value:       =0              success
                    <0              error
```

Note:      Values are passed as a positive number although the electrical signals are actually negative.

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

arguments:          devidx:          device index 0..7
                    div:             input rate divider applied at channel 0
                                     (1, 2, 4, or 8)

return value:       =0               success
                    <0               error

Note:       The sync divider must be used to keep the effective sync rate at values <= 10 MHz. It should only be used with sync sources of
            stable period. The readings obtained with PH_GetCountRate are corrected for the divider setting and deliver the external
            (undivided) rate.

```
int PH_SetSyncOffset (int devidx, int offset);                                       new in v.3.0
```

arguments:          devidx:          device index 0..7
                    offset:          offset (time shift) in ps for that channel
                                         minimum = SYNCOFFSMIN
                                         maximum = SYNCOFFSMAX

return value:       =0               success
                    <0               error

Note:       This function can replace an adjustable cable delay. A positive offset corresponds to inserting a cable in the sync input.
            See also PH_SetRoutingChannelOffset.

```
int PH_SetStopOverflow (int devidx, int stop_ovfl, 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
                                     (max. 65,535)

return value:       =0               success
                    <0               error

Note:       This setting determines if a measurement run will stop if any channel reaches the maximum set by stopcount. If stop_ofl is 0 the
            measurement will continue but counts above 65,535 in any bin will be clipped.

```
int PH_SetBinning (int devidx, int binning);                                      changed in v.3.0
```

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

return value:       =0               success
                    <0               error

Note:       The binning code corresponds to a power of 2, i.e.

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

```
int PH_SetMultistopEnable (int devidx, int enable);                               new in v.3.0
```

arguments:          devidx:          device index 0..7
                    enable:          0 = disable
                                     1 = enable (default)

return value:       =0               success
                    <0               error

Note:       This is only for special applications where the multistop feature of the PicoHarp is causing complications in statistical analysis.
            Ususally it is not required to call this funktion. By default, multistop is enabled after PH_Initialize.

```
int PH_SetOffset (int devidx, int offset);
```

| arguments: | devidx: | device index 0..7 |
| | offset: | offset in picoseconds (histogramming and T3 mode only) |
| | | minimum = OFFSETMIN |
| | | maximum = OFFSETMAX |
| return value: | =0 | success |
| | <0 | error |

Note:     The true offset is an approximation of the desired offset by the nearest multiple of the base resolution. This offset only acts on the difference between ch1 and ch0 in histogramming and T3 mode. Do not confuse it with the input offsets.

```
int PH_ClearHistMem (int devidx, int block);
```

| arguments: | devidx: | device index 0..7 |
| | block: | block number to clear |
| return value: | =0 | success |
| | <0 | error |

```
int PH_StartMeas (int devidx, int tacq);
```

| arguments: | devidx: | device index 0..7 |
| | tacq: | acquisition time in milliseconds |
| | | minimum = ACQTMIN |
| | | maximum = ACQTMAX |
| return value: | =0 | success |
| | <0 | error |

```
int PH_StopMeas (int devidx);
```

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

Note:     Can also be used before the CTC expires but for internal housekeeping it MUST be called any time you finish a measurement, even if data collection was stopped internally, e.g. by expiration of the CTC or an overflow.

```
int PH_CTCStatus (int devidx, int* ctcstatus);                           changed in v.3.0
```

| arguments: | devidx: | device index 0..7 |
| | ctcstatus: | CTC status (passed by reference) |
| | | =0 acquisition time still running |
| | | >0 acquisition time has ended |
| return value: | =0 | success |
| | <0 | error |

```
int PH_GetHistogram (int devidx, unsigned int* chcount, int block);      changed in v.3.0
```

| arguments: | devidx: | device index 0..7 |
| | chcount: | pointer to an array of at least HISTCHAN double words (32bit)where the histogram data can be stored |
| | block: | block number to fetch |
| | | (block > 0   meaningful only with routing) |
| return value: | =0 | success |
| | <0 | error |

Note:     The current version counts only up to 65,535 (16 bits). This may change in the future.

```
int PH_GetResolution (int devidx, double* resolution);                   changed in v.3.0
```

| arguments: | devidx: | device index 0..7 |
| | resolution: | resolution at current binning (passed by reference) |

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



int PH_GetCountRate (int devidx, int channel, int* rate);                changed in v.3.0

arguments:           devidx:             device index 0..7
                     channel:            number of the input channel (0 or 1)
                     rate:               current pulse rate at this channel (passed by reference)
return value:        =0                  success
                     <0                  error
```

Note:       The hardware rate meters employ a gate time of 100 ms.  You must allow at least 100 ms after PH_Initialize or
            PH_SetDyncDivider to get a valid rate meter reading. Similarly, wait at least 100 ms to get a new reading. The readings are
            corrected for the sync divider setting and deliver the external (undivided) rate. The gate time cannot be changed. The readings
            may therefore be inaccurate or fluctuating when the rates are very low. If accurate rates are needed you must perform a full
            blown measurement and sum up the recorded events.

```
int PH_GetFlags (int devidx, int* flags);                                changed in v.3.0

arguments:           devidx:             device index 0..7
                     flags:              current status flags
                                         (a bit pattern, passed by reference)
return value:        =0                  success
                     <0                  error
```

Note:       Use the predefined bit mask values in phdefin.h (e.g. FLAG_OVERFLOW) to extract individual bits through a bitwise AND.
            It is also recommended to check for FLAG_SYSERROR to detect possible hardware failures. In that case you may want to call
            PH_GetHardwareDebugInfo and submit the results for support.

```
int PH_GetElapsedMeasTime (int devidx, double* elapsed);                 changed in v.3.0

arguments:           devidx:             device index 0..7
                     elapsed:            elapsed measurement time in ms (passed by reference)
return value:        =0                  success
                     <0                  error



int PH_GetWarnings (int devidx, int* warnings);                          changed in v.3.0

arguments:           devidx:             device index 0..7
                     warnings:           warnings, (passed by reference)
                                         bitwise encoded (see phdefin.h)
return value:        =0                  success
                     <0                  error
```

Note:               Must call PH_GetCoutRates for all channels prior to this call.

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

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



int PH_GetHardwareDebugInfo (int devidx, char* debuginfo);                new in v.3.0

arguments:           devidx:             device index 0..7
                     debuginfo:          pointer to a buffer for at least 16384 characters
return value:        =0                  success
                     <0                  error
```

Note:       It is recommended to use PH_GetFlags and check for FLAG_SYSERROR to detect possible hardware failures. In that case you
            may want to call PH_GetHardwareDebugInfo and submit the results for support.

### Special Functions for TTTR Mode

To use these functions, you must have purchased the TTTR mode option. You can use
PH_GetFeatures to probe for it.

```
int PH_ReadFiFo (int devidx, unsigned int* buffer, int count, int* nactual); changed in v.3.0
```

```
arguments:          devidx:             device index 0..7
                    buffer:             pointer to an array of count dwords (32bit)
                                           where the TTTR data can be stored
                    count:              number of TTTR records to be fetched
                                           (max TTREADMAX)

                    nactual:            number of dwords actually read (passed by reference)
return value:       =0                  success
                    <0                  error
```

Note:        Must not be called with count larger than buffer size permits. CPU time during wait for completion will be yielded to other
             processes/threads. Function will return after a timeout period of ~80 ms, even if not all data could be fetched. Return value
             indicates how many records were fetched. Buffer must not be accessed until the function returns.

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

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

Note:        PicoHarp devices prior to hardware version 2.0 support only the first three markers. Default after Initialize is "all rising".

```
int PH_SetMarkerEnable (int devidx, int en0, int en1, int en2, int en3);        new in v.3.0
```

```
arguments:          devidx:             device index 0..7
                    me<n>:              enabling of marker signal <n>,
                                           0 = disabled,
                                           1 = enabled
return value:       =0                  success
                    <0                  error
```

Note:        PicoHarp devices prior to hardware version 2.0 support only the first three markers. Default after Initialize is "all enabled".

```
int PH_SetMarkerHoldofftime (int devidx, int holdofftime);                      new in v.3.0
```

```
arguments:          devidx:             device index 0..7
                    holdofftime:        holdofftime in nanoseconds
return value:       =0                  success
                    <0                  error
```

Note:        This setting can be used to clean up glitches on the marker signals. When set to X ns then after detecting a first marker edge the
             next marker will not be accepted before X ns. Observe that the internal granularity of this time is only about 50 ns. The holdoff
             time is set equally for all marker inputs but the holdoff logic acts on each marker independently.

### Special Functions for Routing

These functions require a PHR 402 / 403 / 800.

```
int PH_GetRoutingChannels (int devidx, int* rtchannels);                  changed in v.3.0

arguments:          devidx:          device index 0..7
                    rtchannels:      number of routing channels (passed by reference)
return value:       =0               success
                    <0               error



int PH_EnableRouting (int devidx, int enable);

arguments:          devidx:          device index 0..7
                    enable:          routing state control code
                                        1 = enable routing
                                        0 = disable routing
return value:       =0               success
                    <0               error
```

Note:       This function can also be used to detect the presence of a router.

```
int PH_GetRouterVersion (int devidx, char* model, char* vers);

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



int PH_SetRoutingChannelOffset (int devidx, int offset);

arguments:          devidx:          device index 0..7
                    offset:          offset (time shift) in ps for that channel
                                        minimum = CHANOFFSMIN
                                        maximum = CHANOFFSMAX
return value:       =0               success
                    <0               error
```

Note:       This function can be used to compensate small timing delays between the individual routing channels. It is similar to PH_SetSyncOffset and can replace cumbersome cable length adjustments but compared to PH_SetSyncOffset the adjustment range is relatively small. A positive number corresponds to inserting cable in that channel.

```
int PH_SetPHR800Input (int devidx, int channel, int level, int edge);

arguments:          devidx:          device index 0..7
                    channel:         router channel to be programmed (0 .. 3)
                    level:           trigger voltage level in mV (−1600 .. 2400)
                    edge:            trigger edge
                                        0 = falling edge,
                                        1 = rising edge
return value:       =0               success
                    <0               error
```

Note 1:     Not all channels may be present.

Note 2:     Invalid combinations of level and edge may lock up all channels!

```
int PH_SetPHR800CFD (int devidx, int channel, int dscrlevel, int zerocross);

arguments:          devidx:          device index 0..7
                    channel:         router CFD channel to be programmed (0 .. 3)
```

Page 22

```
                dscrlevel:        discriminator level in mV (0 .. 800)
                zerocross:        zero crossing level in mV (0 .. 20)

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

# 11. Warnings

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

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

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

| Warning | Histo Mode | T2 Mode | T3 Mode |
|---|:---:|:---:|:---:|
| **WARNING_INP0_RATE_ZERO**<br><br>No counts are detected at input channel 0. In histogramming and T3 mode this is the sync channel and the measurement will not work without that signal. | √ | | √ |
| **WARNING_INP0_RATE_TOO_LOW**<br><br>The count rate at input channel 0 is below 1 kHz and the sync divider is >1. In histogramming and T3 mode this is the sync channel and the measurement will not work without a signal <1 kHz if the sync divider is >1. If your sync rate is really so low then you do not need a sync divider >1. | √ | | √ |
| **WARNING_INP0_RATE_TOO_HIGH**<br><br>You have selected T2 mode and the count rate at input channel 0 is higher than 5 MHz. The measurement will inevitably lead to a FiFo overrun. There are rare measurement scenarios where this condition is expected and the warning can be disabled. Examples are very short measurements where the FiFo can absorb all data. | | √ | |
| **WARNING_INP1_RATE_ZERO**<br><br>No counts are detected at input channel 1. In histogramming and T3 mode this is the photon event channel and the measurement will yield nothing without this signal. You may sporadically see this warning if your detector has a very low dark counts. In that case you may want to disable this warning. | √ | | √ |
| **WARNING_INP1_RATE_TOO_HIGH**<br><br>If you have selected T2 mode then this warning means the count rate at input channel 1 is higher than 5 MHz. The measurement will inevitably lead to a FiFo overrun. There are rare measurement scenarios where this condition is expected and the warning can be disabled. Examples are very short measurements where the FiFo can absorb all data.<br><br>In histogramming and T3 mode this warning is issued when the input rate is >10 MHz. This will probably lead to deadtime artefacts. There are rare measurement scenarios where this condition is expected and the warning can be disabled. | √ | √ | √ |

| Warning | Histo Mode | T2 Mode | T3 Mode |
|---|:---:|:---:|:---:|
| **WARNING_INP_RATE_RATIO**<br><br>This warning is issued in histogramming and T3 mode when the rate at input 1 is over 5% of the rate at input 0. This is the classical pile-up criterion. It will lead to noticeable dead-time artefacts. There are rare measurement scenarios where this condition is expected and the warning can be disabled. Examples are antibunching measurements. | √ | | √ |
| **WARNING_DIVIDER_GREATER_ONE**<br><br>You have selected T2 mode and the sync divider is set larger than 1. This is probably not intended. The sync divider is designed primarily for high sync rates from lasers and requires a fixed pulse rate at channel 0. In that case you should use T3 mode. If the signal at channel 0 is from a photon detector (coincidence correlation etc.) a divider >1 will lead to unexpected results. There are rare measurement scenarios where this condition is intentional and the warning can be disabled. | | √ | |
| **WARNING_TIME_SPAN_TOO_SMALL**<br><br>This warning is issued in histogramming and T3 mode when the sync period (1/Rate[ch0]) 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:<br>   Histogramming mode: Span = Resolution * 65536<br>   T3 mode:  Span = Resolution * 4096<br>Events outside this span will not be recorded. There are some measurement scenarios where this condition is intentional and the warning can be disabled. | √ | | √ |
| **WARNING_OFFSET_UNNECESSARY**<br><br>This warning is issued in histogramming and T3 mode when an offset >0 is set even though the sync period (1/Rate0) can be covered by the measurement time span without using an offset. The offset may lead to events getting discarded. There are some measurement scenarios where this condition is intentional and the warning can be disabled. | √ | | √ |

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

Note that all count rate dependent warnings relate to the PicoHarp's native input channels. If you are using a router, remember that all routing channels share the PicoHarp's input channel 1 and hence create a cumulated common rate at that channel.

# 12.   Problems, Tips & Tricks

**PC Performance Issues**

Performance issues with the library under Linux are the same as with the standard PicoHarp software for Windows. The PicoHarp and its software interface are a complex real-time measurement system demanding considerable performance both from the host PC and the operating system. This is why a reasonably modern CPU (1 GHz min.) and at least 1 GB of memory are recommended. The USB 2.0 interface must be configured correctly and use only high speed rated components. If you  intend to use TTTR mode with streaming to disk you should also have a fast modern hard disk. However, as long as you do not use TTTR mode, performance issues should not be of severe impact.

**Troubleshooting**

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

Under Linux the library will access the PicoHarp device through libusb. You need to make sure that 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 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 PH_OpenDevice. Unfortunately this cannot be distinguished from a failure to open the device due to insufficient access rights (permissions). Such a case also results in ERROR_DEVICE_BUSY.

You should also make sure your PicoHarp has the right firmware to use the library. The library option is not free, a license must be purchased. If you do not have it, you can order an upgrade at any time.

To get started, 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 setings may not be compatible wih your experimental setup. Then the pre-compiled demo may not work as expected. Only the LabVIEW demo allows to enter the settings interactively.

**Version Tracking**

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

**New Kernel Versions and Linux Distributions**

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 it 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.

**Software Updates**

We constantly improve and update the sofware 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 register for email notification on software updates. If you wish to receive such notification, please email your name and the serial number of your PicoQuant product(s) to *info@picoquant.com.* You will then receive update information with links for download of any new software release.

**Support**

The PicoHarp 300 TCSPC system has gone through several iterations of hardware and software improvement as well as extensive testing. Nevertheless, it is fairly challenging technology and some glitches may still occur under the myriads of possible PC configurations, operating systems and application circumstances. We therefore offer you our support in any case of trouble with the system and ask your help to identify any such problems. Do not hesitate to contact PicoQuant in case of difficulties with your PicoHarp or the programming library. However, please note that the Linux library is a free supplement to the Windows version and its development is work carried out voluntarily by single Linux enthusiasts. It therefore implies no warranties for other than voluntary support.

Should you observe errors or unexpected behaviour related to the PicoHarp system, please try to find a precise and reproducible error situation.  E-mail a detailed description of the problem and relevant circumstances to *info@picoquant.com.* Please include all PC and operating system details. Your feedback will help us to improve the product and documentation.