# MultiHarp 150

Picosecond Histogram Accumulating
Real–time Processor
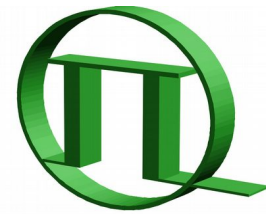
MHLib – Programming Library
for Custom Software Development
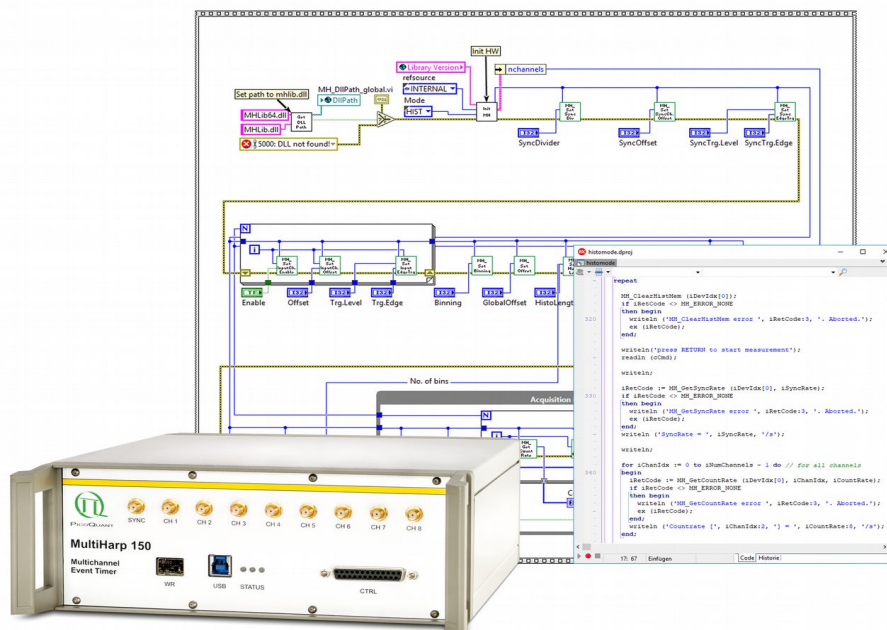
User's Manual

Version 2.0.0.0

Table of Contents

1.   Introduction

The MultiHarp 150 is a cutting edge Time-Correlated Single Photon Counting (TCSPC) system with USB 3.0 interface. Its new integrated design provides a flexible number of input channels at reasonable cost and allows innovative measurement approaches. The timing circuits allow high measurement rates up to 78 million counts per second (Mcps) with an excellent time resolution and a deadtime of only 650 ps. The modern USB interface provides very high throughput as well as 'plug and play' installation. The input triggers are adjustable for a wide range of input signals providing programmable level triggers for both negative and positive going signals. These specifications qualify the MultiHarp 150 for use with most common single photon detectors such as Single Photon Avalanche Diodes (SPADs) and Photomultiplier Tube (PMT) modules (via preamplifier). Depending on detector and excitation source the width of the overall Instrument Response Function (IRF) can be as small as 100 ps FWHM. The MultiHarp 150 is available with 4, 8, or 16 timing inputs. The use of these inputs is very flexible. In fluorescence lifetime applications the first channel is typically used as a synchronization input from a laser. The other inputs are then used for photon detectors. In coincidence correlation applications all inputs can be used for photon detectors.

The MultiHarp 150 can operate in various modes to adapt to different measurement needs. The standard histogram mode performs real–time histogramming in computer memory. Two different Time–Tagged–Time–Resolved (TTTR) modes allow recording 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 MultiHarp 150 is furthermore supported by a variety of accessories such as pre–amplifiers, signal adaptors and detector assemblies from PicoQuant. A significant novel feature of the MultiHarp 150 is support for White Rabbit, allowing time transfer and synchronization with sub-ns accuracy over long distances (see https://en.wikipedia.org/wiki/The_White_Rabbit_Project).

For more information on the MultiHarp 150 hardware and software please consult the MultiHarp 150 manual. For details on the method of Time–Correlated Single Photon Counting, please refer to our TechNote on TCSPC.

The MultiHarp 150 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 MultiHarp'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 8.1 and 10. Windows 7 should still work but is not actively supported.

The library supports custom programming in all major programming languages, notably C / C++, C#, Delphi / Pascal, Python, LabVIEW and MATLAB. This manual describes the installation and use of the MultiHarp programming library and explains the associated demo programs. Please read both this library manual and the MultiHarp manual before beginning your own software development with the DLL. The MultiHarp 150 is a sophisticated real–time measurement system. In order to work with the system using the DLL, sound knowledge in your chosen programming language is required.

2.　General Notes

This version of the MultiHarp 150 programming library is suitable for Windows 8 and 10. Windows 7 should still work but is no longer actively supported.

The library has been tested with MinGW 2.0 (32 bit), MinGW-W64 4.3.5 (64 bit), MSVC++ 6.0 (32 bit), Visual C++ 2013, 2015, and 2019 (32/64 bit), Visual C# 2013, 2015 and 2019 (32/64 bit), Mono 5.14.0. and 6.8.0, Delphi 10.1 and 10.2 (32/64 bit), Lazarus 1.8.4 and 2.0.8 + FreePascal 3.0.4 (32/64 bit), Python 3.7.4 and 3.8.2 (32/64 bit), LabVIEW 2017 (32/64 bit), and MATLAB R2019a (32/64 bit).

There is also a library version for Linux (Intel processor architecture only) which is fully compatible with that for Windows so that applications can easily be ported across the two platforms. See the separate Linux manual.

This manual assumes that you have read the MultiHarp 150 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 (e.g. 2.0). The DLL actually has two further sub–version digits, so that the complete version number has four digits (e.g. 2.0.0.0). They are shown only in the Windows file properties. These sub–digits help to identify intermediate versions that may have been released for minor updates or bug fixes. The interface of releases with identical major version will remain the same. 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.

2.1.　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 MultiHarp 150 product you have purchased a license to use the MultiHarp 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 MultiHarp 150 hardware in its current version of April 2020 uses the White Rabbit PTP core v. 4.0 (https://www.ohwr.org/projects/wr-cores/wiki/wrpc-release-v40) licensed under the CERN Open Hardware Licence v1.1 and its embedded WRPC software (https://www.ohwr.org/projects/wrpc-sw/wiki/wiki) licensed under GPL Version 2, June 1991. The WRPC software was minimally modified and in order to meet the licensing terms the modified WRPC source code is provided as part of the MultiHarp software distribution media.

3.   Installation of the MHLib Software Package

MHLib and its demos will not be installed by the standard MultiHarp 150 software setup. The standard "interactive" MultiHarp 150 data acquisition software does not require the DLL, which is provided for custom application programming only. Vice versa, your custom program will only require the DLL and the driver, but not the standard MultiHarp 150 data acquisition software. Installing both the standard MultiHarp software and DLL–based custom programs on the same computer is possible, but only one program at a time can use the MultiHarp 150.

To install MHLib, please back up your work, then disconnect the MultiHarp device(s) and uninstall any previous versions of MHLib. Then run the setup program SETUP.EXE in the MHLib folder on the installation CD. If you received the setup files as a ZIP archive, please unpack them to a temporary directory on your hard disk and run SETUP.EXE from there. On some versions of Windows you may need administrator rights to perform the setup. If the setup is performed by an administrator but used from other accounts without full access permission to all disk locations, these restricted accounts may not be able to run the demos in the default locations they have been installed to. In such cases it is recommended that you copy the demo folder (or selected files from it) to a dedicated development folder, in which you have the necessary rights (e.g., in 'My Documents').

You also need to install the MultiHarp 150 device if you have not done so before (see your MultiHarp manual). The programming library will access the MultiHarp 150 through a dedicated device driver. The driver is installed together with the library by SETUP.EXE and is then instantly ready to use. Depending on your Windows version you may be prompted to confirm the final driver installation when the device is connected for the first time. Both the standard MultiHarp software distribution as well as the MHLib distribution media contain the driver and will install it in the dedicated location that Windows maintains for this purpose.

Note that multiple devices can be controlled through MHLib. After connecting the device(s) you can use the Windows Device Manager to check if they have been detected (under the USB tree) and the driver is correctly installed. On some Windows versions you may need administrator rights to perform setup tasks. Refer to your MultiHarp 150 manual for other installation details.

It is recommended to start your work with the MultiHarp 150 by using the standard interactive MultiHarp data acquisition software. This should give you a better understanding of the system's operation before attempting your own programming efforts. It also ensures that your optical / electrical setup is working. See the subfolder \demos in your MHLib installation folder for sample code that can be used as a starting point for your own programs. Please copy the demos to a working directory where you have write permission. Inside the installation folder this is not the case.

The MHLib package provides both 32-bit and 64-bit versions of the library. On a 64-bit Windows the setup program will install both versions of the DLL. On a 32-bit Windows it will only install the 32-bit version. Note that the 32-bit version of the DLL is named MHLib.dll while the 64-bit version is named MHLib64.dll. This is to avoid confusions between the two. As a consequence of the different names the demo code is version dependent and will be installed in two separate folders for 32-bit and 64-bit. For reference and comparison the demos will always be fully installed in both versions but obviously the 64-bit versions will not run on a 32-bit version of Windows.

If you wish to uninstall the library you should first of all back–up custom files you might have created in the installation folder. Do not manually delete any program files from the installation folder as it will render a clean uninstall impossible. Also do not delete any driver files manually.

To uninstall the library from your PC you may need administrator rights (depending on Windows version and security settings). Go to *Control Panel > Programs and Features* and select *PicoQuant – MultiHarp - MHLib vx.x* for un–installation. This will remove all files that were installed by the setup program but not the user data that may have been stored. If there was user data in the installation folders these will not be deleted. If intended, you will have to delete these files or folders manually.

Note that un–installation of the data acquisition software does not uninstall the device driver since other software may still need it. If need be you can delete the driver software from within Device Manager.

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 explanatory purposes and a starting point for your own work.

The demos are kept as simple as possible to maintain focus on the key issues of accessing the library. This is why most of the demos have a minimalistic user interface and / or must be run from the command line (cmd). 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 MultiHarp manual and try them out using the regular MultiHarp software.

For the reason of simplicity, the demos will always only use the first MultiHarp 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 demo code to match your configuration.

None of the demos currently show how to use White Rabbit. Nevertheless it should be possible to get an understanding on how this is done by looking at the White Rabbit Dialog in the regular MultiHarp software, which uses the same basic library routines as documented in section 7.2. Please contact PicoQuant if you need further help using White Rabbit.

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

Histogramming Mode Demos

These demos show how to use the standard measurement mode for on–board histogramming. These are the simplest demos and the best starting point for your own experiments. In case of LabVIEW there are basic and advanced demos, the latter being more sophisticated and allowing interactive input of most parameters on the fly.

TTTR Mode Demos

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

The MultiHarp 150 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 MultiHarp manual. In TTTR mode it is also possible to record external TTL signal transitions as markers in the TTTR data stream (see the MultiHarp 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 fairly demanding both in programming skills and computer performance. See the section about TTTR mode in your MultiHarp manual.

Note that you must not call any of the `MH_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 in the collected data.

Details on how to interpret and process the TTTR records can be studied in the advanced LabVIEW demos and in the file demo code installed together with the regular MultiHarp software. The same code can (with minor adaptions) be used to process the TTTR records on the fly.

4.2. The Demo Applications by Programming Language

As outlined above, there are demos for C / C++, Delphi / Pascal, Python, LabVIEW and MATLAB. For each of these programming languages 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 MultiHarp 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 MultiHarp library as your first attempt at programming. You will also need some knowledge about Windows DLL concepts and calling conventions. The ultimate reference for details about how to use the DLL is in any case the source code of the demos and the header files of MHLib (mhlib.h and mhdefin.h).

Be warned that wrong parameters and / or variables, invalid pointers and buffer sizes, inappropriate calling sequences etc. may crash your application and / or your complete computer. The latter is quite unlikely but it may happen, even for relatively safe operating systems, because you are accessing a kernel mode driver through MHLib. This driver has high privileges at kernel level, that all have the power to do damage if used inappropriately. Make sure to backup your data and / or perform your development work on a dedicated machine that does not contain valuable data. Note that the 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, preferably in the order shown by the demos.

Note that for the 64-bit versions different names apply. The main 64-bit DLL file is named MHLib64.dll and the 64-bit link library is named MHLib64.lib. This is to avoid confusion between the two versions. As a consequence of the different names the demo code is version dependent and will be installed in two separate folders: One for 32-bit and one for 64-bit. Nevertheless some of the demos are actually identical for 32-bit and for 64-bit and the code handles the distinction between 32-bit and 64-bit versions internally. In the following we use only the 32-bit library names without the suffix 64.

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 mhlib.h, mhdefin.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 "mhdefin.h"
  #include "mhlib.h"
}
```

In order to make the exports of MHLib.dll known to the rest of your application you may use MHLib.exp or link directly with the import library MHLib.lib. MHLib.lib was created for MSVC 6.0 or higher, with symbols decorated in Microsoft style. The DLL also (additionally) exports all symbols undecorated, so that other compilers should be able to use them conveniently, provided they understand the Microsoft LIB format or they can create their own import library. The MinGW compiler understands the Microsoft format. With Borland compilers you can use the utility IMPLIB to create your own import library.

To test any of the demos, consult the MultiHarp manual for setting up your MultiHarp 150 and establish a measurement setup that runs correctly and generates useable test data. 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.

The C demos are designed to run in a cmd console. They need no command line input parameters. They create their output files in their current working directory (*.out). The output files will be ASCII‑readable in case of the standard histogramming demos. For this demo, the ASCII files will contain multiple columns of integer numbers representing the counts from the 65,536 histogram bins. You can use any editor or a data visualiza‑

tion 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 MultiHarp 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 MHLib 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 MH_Initialize to a value of 3 if you want T3 mode. Note that you probably also need to adjust the sync divider and the resolution in this case.

The C# Demos

The C# demos are provided in the `Csharp` subfolder. They have been tested with MS Visual Studio as well as with Mono.

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

With the C# demos you also need to check whether the hard-coded settings are suitable for your actual instrument setup. The demos are designed to run in a cmd console. 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 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 Delphi / Lazarus Demos

Users of Delphi or FreePascal / Lazarus please refer to the `Delphi` folder. The source code for Delphi and Lazarus is the same. 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.dll` 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. `MHLib.dll` was created with symbols decorated in Microsoft style. It additionally exports all symbols undecorated, so that you can call them from Delphi with the plain function name. Please check the function parameters of your code against mhlib.h in the demo directory whenever you update to a new DLL version.

The Delphi / Lazarus demos are also designed to run in a cmd 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 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 MultiHarp 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 MHLib 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 `MH_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).

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 DLL version please check the function parameters of your existing code against `mhlib.h` in the MHLib 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 (`*.out`). The output file will be readable text in case of the standard histogramming demo. The 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 TTTR mode 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 MultiHarp 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 MHLib 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 `MH_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 src sub-folder inside the LabVIEW20xx folders. They can be run either with 32 bit or 64 bit LabVIEW. The correct DLL (mhlib.dll for 32 bit, mhlib64.dll for 64 bit respectively) is selected automatically, provided that it is located in the designated Windows folder (i.e. SysWOW64 and/or System32). This should be the case after correct installation of the library package. The original demo code was created with LabVIEW 2017, accordingly also a LabVIEW project file (MultiHarp.lvproj) and two executables (MultiHarpHisto.exe and MultiHarpT3.exe; both in build sub-folder) are provided for that version. For backward compatibility the source code was also converted to LabVIEW 2010.

The first demo (`1_SimpleDemo_MHHisto.vi`) is very simple, demonstrating the basic usage and calling sequence of the provided SubVIs encapsulating the DLL functionality, which are assembled inside the LabVIEW library `mhlib_x86_x64_UIThread.llb` (in the folder `_lib/PQ/MultiHarp`). The demo starts by calling some of these library functions to setup the hardware in a defined state and continues with a measurement in histogramming mode by calling the corresponding library functions inside a while-loop. Histograms and count rates for all available hardware channels are displayed on the front panel in a waveform graph (you might have to select `AutoScale` for 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 LabVIEW 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 (SubVIs, Types). General helper functions and type-definitions as well as DLL encapsulating libraries (*.llb) can be found in the _lib folder (containing further sub-folders) inside the demo folder. In order to facilitate the usage of the DLL functions, additional VIs called `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 DLL functions and their functional context.

Please note:

In addition to the library used by the demos (`mhlib_x86_x64_UIThread.llb`) a second library is included allowing the DLL calls to be executed in any thread of LabVIEWs threading engine (`mhlib_x86_x64_Any-Thread.llb`). This library is intended for time critical applications where user actions on the front panel (like e.g., resizing or moving) must not affect the execution of a data acquisition thread containing these DLL functions (please refer to "Multitasking in LabVIEW": http://zone.ni.com/reference/en-XX/help/371361P-01/lvconcepts/multitasking_in_labview/). When using this library you have to make sure that all DLL functions are called in a sequential order to avoid errors or even program crashes. Also be aware that DLL functions in `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 but any version from 6.5 on should work. Note that recent versions of MATLAB require a compiler to be installed for working with DLLs. We tested with MATLAB R2019a and MinGW. For your specific version of MATLAB, please check the documentation of the MATLAB command `loadlibrary` as to which compilers it supports. 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 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 Multi-Harp 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 MHLib 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.

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 `MH_Initialize` to a value of 3 if you want T3 mode. At the same time you need to modify your program for an appropriate sync divider and a suitable range (resolution).

5. Advanced Techniques

5.1. Using Multiple Devices

The library is designed to work with multiple MultiHarp devices (up to 8). The demos always use the first device found. If you have more than one MultiHarp and you want to use them together you need to modify the code accordingly. At the API level of MHLib the devices are distinguished by a device index (0 .. 7). The device order corresponds to the order in which Windows enumerates the devices. If the devices were plugged in or switched on sequentially when Windows 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 `MH_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 `MH_GetSerialNumber` any time later on a device you have successfully opened. The serial number of a physical MultiHarp 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 `MH_OpenDevice` or `MH_GetSerialNumber`.

As outlined above, if you have more than one MultiHarp 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 MultiHarp 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. Windows 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 or White Rabbit, the approach is somewhat cumbersome.

5.2. Efficient Data Transfer

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

In order to achieve the highest throughput, the MultiHarp 150 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 MultiHarp 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 `MH_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 return after a timeout of approximately 1 ms if no more data is available. The actual time to return can be slightly longer (2..3 ms) due to USB overhead and unpredictable Windows latencies.

As noted above, the transfer is implemented efficiently without excessive CPU use. Nevertheless, assuming large block sizes, the transfer takes some time. Windows therefore gives the unused CPU time to other pro-cesses 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 `MH_ReadFiFo`) and another for processing or stor-ing 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 tech-nique. Greatest care must be taken not to access the MHLib routines from different threads without strict con-trol of mutual exclusion and maintaining the right sequence of function calls. However, the technique also al-lows throughput improvements of 50% .. 100% and advanced programmers may want to use it. It might be in-teresting to note that this is how TTTR mode is implemented in the regular MultiHarp software, where sus-tained count rates over 9 Mcps (to disk) can be achieved with USB 2.0 and even up to 90 Mcps with USB 3.0.

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 order to find out how many USB controllers you have and which one the individual USB sockets belong to, you can use Microsoft's tool `usbview.exe`. In case of using mul-tiple devices it is also beneficial for overall throughput if you use multi‑threading in order to fetch and store data from the individual devices in parallel. Again, re‑entrance issues must be observed carefully in this case, at least for all calls accessing the same device.

5.3. Working with 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 MultiHarp software. In order to obtain and use these warnings also in your custom software you may want to use the library routine `MH_GetWarnings`. This may help inexperienced users to notice possible mistakes before stating a measure-ment 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 `MH_GetWarnings` does not obtain the count rates on its own, be-cause the corresponding calls take some time and might waste too much processing time. It is therefore nec-essary that the library routines for count rate retrieval (on all channels) have been called before `MH_Get-Warnings` 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 (`MH_GetSyncRate` and `MH_GetCountRate`) but in case of performance critical applications it is more effi-cient to use `MH_GetAllCountRates` retrieving all rates in one call.

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

6. Problems, Tips & Tricks

6.1. PC Performance Requirements

Performance requirements for the DLL are the same as with the standard MultiHarp software. The MultiHarp device and its software interface are a complex real–time measurement system demanding appropriate performance both from the host PC and the operating system. This is why a fairly modern CPU and sufficient memory are required. At least a dual core, 2 GHz processor, 4 GB of memory and a fast hard disk (preferably SSD) are recommended. However, as long as you do not use TTTR mode, these issues should not be of severe impact.

6.2. USB Interface

In order to deliver maximum throughput, the MultiHarp 150 uses state–of–the–art USB bulk transfers. This is why the MultiHarp must rely on having a USB host interface matched to the device speed. USB 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 MultiHarp. Do not run other bandwidth demanding devices on the same USB interface when working with the MultiHarp. USB cables must be qualified for the USB speed you are using. 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.

6.3. Troubleshooting

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

The DLL will access the MultiHarp device through a dedicated device driver. You need to make sure the device driver has been installed correctly. The driver is installed by the setup program using standard Windows Plug&Play mechanisms. In addition both the standard MultiHarp software distribution as well as the MHLib distribution media contain the driver in the subfolder `\Driver`. You can use the Windows Device Manager to check if the board has been detected and the driver is installed. On some Windows versions you may need administrator rights to perform hardware setup tasks. Please consult the MultiHarp manual for hardware related problem solutions.

The next step, if hardware and driver are working, is to make sure you have the right DLL version installed. It comes with its own setup program that must be executed as Administrator. In the Windows Explorer you can also right click `MHLib64.DLL` and `MHLib.DLL` (in `\Windows\System32` or `\Windows\SysWOW64`) and check the version number (under *Properties*).

To get started, ensure that your setup is working by running the regular MultiHarp software. After closing the regular MultiHarp software, try the readily compiled demos supplied with the DLL. For first tests take one of 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 demos may not work as expected. Only the advanced LabVIEW demos allow to enter most of the settings interactively.

6.4.   Access permissions

On some Windows versions you may need administrator rights to perform the DLL setup. If the setup is performed by an administrator but used from other accounts without full access permission to all disk locations, these restricted accounts may no be able to run the demos in the default locations they have been installed to. In such cases it is recommended that you copy the demo directory or selected files from it to a dedicated development directory in which you have the necessary rights. Otherwise the administrator must give full access to the demo directory. On some Windows versions it is possible to switch between user accounts without shutting down the running applications. It is not possible to start a MultiHarp program if any other program accessing the device is running in another user account that has been switched away. Doing so may cause crashes or loss of data.

6.5.   Version tracking

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

6.6.   Software Updates

We 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 MultiHarp 150 TCSPC system has gone through extensive testing. It builds on over 20 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 MultiHarp.

If you should observe errors or bugs caused by the MultiHarp system please try to find a reproducible error situation. Email a detailed description of the problem and all relevant circumstances, especially the versions of the software you were using, the version of Windows, and a summary of add-on hardware installed in your PC, to support@picoquant.com. Please run `msinfo32` to obtain a listing of your PC configuration and attach the summary file to your error report. Your feedback will help us to improve the product and documentation.

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

7. Appendix

7.1. Data Types

The MultiHarp programming library `MHLib.DLL` is written in C and its data types correspond to standard C / C++ data types as follows:

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

Note that the format for the decimal point may depend on your Windows settings at run–time of the MultiHarp software (usually national language dependent).

Note also that on platforms other than the Intel architecture byte swapping may occur when the MultiHarp data files are read there for further processing. We recommend using the native Intel architecture environ-ment consistently.

The distribution pack includes a set of demo programs (source code) for various programming languages to show how access to MultiHarp data files can be implemented. These demos also show how to process TTTR records and the related code fragments can be used for real-time processing of freshly collected data as well. They will be installed in the subfolder `\Filedemo` under the chosen installation folder.

7.2. Functions Exported by **MHLib.DLL**

See **mhdefin.h** for predefined constants given in capital letters here. Return values < 0 denote errors. See **errorcodes.h** for the error codes. On 32-bit platforms all functions must be called with `_stdcall` convention. On 64-bit platforms this defaults to the Microsoft x64 calling convention. Note that `MHLib` is a multi-device library with the capability to control more than one MultiHarp simultaneously. For that reason all device specific functions (i.e. the functions from section 7.2.2 on) take a device index as first argument. Note that functions taking a channel number as an argument expect the channels enumerated 0..N-1 while the in-teractive MultiHarp software as well as the physical front panel enumerates the channels 1..N. This is due to internal data structures and for consistency with earlier products.

7.2.1.   General Functions

These functions work independent from any device.

int MH_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 MH_GetErrorString (char* errstring, int errcode);

arguments:	errstring:	pointer to a buffer for at least 40 characters
	errcode:	error code returned from a MH_xxx function call
return value:	=0	success
	<0	error

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

7.2.2.   Device Related Functions

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

int MH_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 MH_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 MH_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 = White Rabbit master with generic partner
		3 = White Rabbit slave with generic partner
		4 = White Rabbit grand master with generic partner
		5 = use 10 MHz + PPS from GPS receiver

```
                               6 = use 10 MHz + PPS + time via UART from GPS receiver
                               7 = White Rabbit master with MultiHarp as partner
                               8 = White Rabbit slave with MultiHarp as partner
                               9 = White Rabbit grand master with MultiHarp as partner

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 MultiHarp manual for more information on the measurement modes, external clock, and White Rabbit (WR). Note that selecting WR as a clock source requires that a WR connection has actually been established beforehand. Unless the WR connection is established by a WR startup script this will require a two stage process initially initializing with internal clock source, then setting up the WR connection by means of the WR routines described below, then initializing again with the desired WR clock mode.

7.2.3. Functions for Use on Initialized Devices

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

```
int MH_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 MH_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 MH_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 MH_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 value returned in binsteps is the number of steps. The range of values you can pass to MH_SetBinning is then 0..binsteps-1.

```
int MH_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 MH_GetNumOfModules (int devidx, int* nummod);
```

arguments: devidx: device index 0..7
                    nummod: pointer to an integer,
                                        returns the number of installed modules

return value: =0   success
                    <0   error

Note: This routine is only an accessory for retrieval of hardware version details via `MH_GetModuleInfo` which must be called
          separately for each module. The value returned in `nummod` is the number of modules. The range of values you can pass to
          `MH_GetModuleInfo` is then `0..nummod-1`.

```
int MH_GetModuleInfo (int devidx, int modidx, int* modelcode, int* versioncode);
```

arguments: devidx: device index 0..7
                    modidx: module index 0..nummod-1 (see MH_GetNumOfModules)
                    modelcode: pointer to an integer,
                                        returns the model of the module identified by modidx
                    versioncode: pointer to an integer,
                                        returns the versioncode of the module identified by modidx

return value: =0   success
                    <0   error

Note: This routine is for retrieval of hardware version details and must be called separately for each module. Get the number of
          modules via `MH_GetNumOfModules`. You only need this information for support enquiries.

```
int MH_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. Call it immediately after receiving an error code <0 from any library call or after de-
          tecting a `FLAG_SYSERROR` from `MH_GetFlags`. In case of `FLAG_SYSERROR` please provide this information for support.

```
int MH_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 < 78 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 MH_GetCountRate are internally corrected for the divider setting and deliver the ex-
          ternal (undivided) rate. The sync divider should not be changed while a measurement is running.

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

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

int MH_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 MH_SetSyncDeadTime (int devidx, int on, int deadtime);   **//new since v1.1**

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. The corresponding hardware functionality is regularly available in devices manufactured after September 2019. Earlier devices will need a firmware upgrade to provide this feature. See the distribution media folder `gateware_update`. You can use MH_GetFeatures or evaluate the return code of MH_SetSyncDeadTime to determine if the feature is available. 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. Note that when an extended dead-time is set then it will also affect the 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 MH_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
	mac_edge:	0 = falling, 1 = rising
return value:	=0	success
	<0	error

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

int MH_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` as obtained through MH_GetNumOfInputChannels().

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

arguments: devidx: device index 0..7
                    channel: input channel index 0..nchannels-1
                    enable: desired enable state of the input channel
                                       0 = disabled
                                       1 = enabled

return value: =0   success
                    <0   error

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

```
int MH_SetInputDeadTime (int devidx, int channel, int on, int deadtime);        //new since v1.1
```

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. The corresponding hardware functionality is regularly available in devices manufactured after September 2019. Earlier devices will need a firmware upgrade to provide this feature. See the distribution media folder `gateware_update`. You can use MH_GetFeatures or evaluate the return code of MH_SetSyncDeadTime to determine if the feature is available. 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 MH_SetStopOverflow (int devidx, int stop_ovfl, unsigned int stopcount);
```

arguments: devidx: device index 0..7
                    stop_ofl: 0 = do not stop,
                                   1 = do stop on overflow
                    stopcount: count level at which should be stopped
                                       minimum = STOPCNTMIN
                                       maximum = STOPCNTMAX

return value: =0   success
                    <0   error

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

```
int MH_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: 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 MH_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 MH_SetSyncChannelOffset and/or MH_SetInputChannelOffset.

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

arguments:	devidx:	device index 0..7
	lencode:	histogram length code
		minimum = 0
		maximum = MAXLENCODE (default)
	actuallen:	pointer to an integer,
		returns the current length (time bin count) of histograms
		calculates as 1024 times lencode to the power of 2
return value:	=0	success
	<0	error

Note: This sets the number of bins of the collected histograms. The histogram length obtained with MAXLENCODE is 65536 which is also the default after initialization if MH_SetHistoLen is not called.

```
int MH_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 MH_SetMeasControl (int devidx, int meascontrol, int startedge, int stopedge);
```

arguments:	devidx:	device index 0..7
	meascontrol:	measurement control code
		0 = MEASCTRL_SINGLESHOT_CTC
		1 = MEASCTRL_C1_GATED
		2 = MEASCTRL_C1_START_CTC_STOP
		3 = MEASCTRL_C1_START_C2_STOP
		4 = MEASCTRL_WR_M2S
		5 = MEASCTRL_WR_S2M
	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. software controlled acquisition time. The other modes 1..5 allow hardware triggered measurements through TTL signals at the control port or through White Rabbit. Note that this needs custom software. For a guideline please see the demo set for the C language.

int MH_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.

nt MH_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 MH_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 must in any case be called
after a measurement, also if the measurement has expired on its own.

int MH_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 MH_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 obtained through MH_SetHistoLen().
The maximum input channel index must correspond to nchannels-1 as obtained through MH_GetNumOfInputChannels().
Note that MH_GetHistogram cannot be used with the shortest two histogram lengths of 1024 and 2048 bins. You need to
use MH_GetAllHistograms in this case.

```
int MH_GetAllHistograms(int devidx, unsigned int *chcount);
```

arguments:	devidx:	device index 0..7
	chcount:	buffer for a multidimensional array of the form
		unsigned int histograms[num_channels][histolen]
return value:	=0	success
	<0	error

Note: This can be used as a replacement for multiple calls to `MH_GetHistogram` when all histograms are to be retrieved in the most time-efficient way. The multidimensional array receiving the data must be shaped according to the number of input channels of the device and the chosen histogram length. Written in C notation this would be something like unsigned int histobuf[numinputchannels][numhistogrambins].

```
int MH_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 MH_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 MH_Initialize or MH_SetSyncDivider to get a stable rate meter reading.
Similarly, wait at least 100 ms to get a new reading. This is the gate time of the counter.

```
int MH_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 MH_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` as obtained through MH_GetNumOfInputChannels().

```
int MH_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[num_channels] receiving the input rates
return value:	=0	success
	<0	error

Note: This can be used as replacement of `MH_GetSyncRate` and `MH_GetCountRate` when all rates need to be retrieved in an efficient manner.

```
int MH_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 mhdefin.h (e.g. FLAG_OVERFLOW) to extract individual bits through a bitwise AND.

```
int MH_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.

```
int MH_GetStartTime(int devidx, unsigned int* timedw2, unsigned int* timedw1,
  unsigned int* timedw0);
```

arguments:	devidx:	device index 0..7
	timedw2:	most significant dword of the time value
	timedw1:	2nd m.s. dword of the time value
	timedw0:	least significant dword of the time value
return value:	=0	success
	<0	error

Note: This can be used to retrieve the start time of a measurement with picosecond resolution. It relates always to the start of the most recent measurement, be it completed or only just started. The result is to be interpreted in the sense of a unix time, i.e. elapsed picoseconds since January 1st 1970 00:00:00 UTC (Universal Time). Note that the actual resolution is the device's base resolution. Actual accuracy depends on the chosen time base, e.g., a White Rabbit grandmaster can be very accurate. With less accurate clocks the high resolution result can still be meaningful in a relative sense, e.g. between two devices synchronized over White Rabbit. With internal clocking the accuracy only reflects that of the PC clock. The retrieval via 3 dwords is due to the limited range of all other standard number formats. A simple conversion to double (with loss of accurcy but sufficient with ordinary PC clock) can be done as follows:

```
// timedw0 = 0xFFFFFFFF corresponds to 4294967295 ps = 4.294967295 ms
// timedw1 = 0x00000001 corresponds to 4294967296 ps = 4.294967296 ms
// therefore:
unsigned __int64 time64 = (((unsigned __int64)timedw2)<<32) + timedw1;
double unixtime_in_sec = (double)time64 * 4.294967296 * 1e-3;
```

```
int MH_GetWarnings (int devidx, int* warnings);
```

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

Note: Prior to this call you must call either MH_GetAllCountRates or call MH_GetSyncRate and MH_GetCoutRate for all channels. Otherwise the received warnings will at least partially not be meaningful.

```
int MH_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 MH_GetWarnings
return value:             =0                 success
                          <0                 error
```

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

```
int MH_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 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 MH_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: CPU time during wait for completion will be yielded to other processes / threads. The call will return after a timeout period of
            approximately 1 ms if no more data could be fetched. The actual time to return may vary towards 2..3 ms due to USB over-
            head and Windows latencies. Buffer must not be accessed until the call returns.

```
int MH_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 MH_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 MH_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 ±20ns.

7.2.5. Special Functions for White Rabbit


```
int MH_WRabbitGetMAC (int devidx, char* mac_addr);
```

```
arguments:          devidx:         device index 0..7
                    mac_addr:       pointer to an array of six bytes to receive the MAC address

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

Note: MHLib v1.0 was unnecessarily writing a 7th byte of value 0 here, this has been fixed since v1.1


```
int MH_WRabbitSetMAC (int devidx, char* mac_addr);
```

```
arguments:          devidx:         device index 0..7
                    mac_addr:       pointer to an array of six bytes holding the MAC address

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

Note: The MAC address must be unique, at least with in the network you are using.


```
int MH_WRabbitGetInitScript (int devidx, char* initscript);
```

```
arguments:          devidx:         device index 0..7
                    initscript:     pointer to buffer for at least 256 characters

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

Note: This can be used to retrieve the WR initialization script (if any) from EEPROM. Lines are separated by newline characters. For details on script syntax etc. see the MultiHarp manual and the White Rabbbit documentation.


```
int MH_WRabbitSetInitScript(int devidx, char* initscript);
```

```
arguments:          devidx:         device index 0..7
                    initscript:     pointer to buffer with init script, max 256 characters

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

Note: This can be used to place a WR initialization script in device EEPROM. Lines are separated by newline characters. For details on script syntax etc. see the MultiHarp manual and the White Rabbbit documentation.

```
int MH_WRabbitGetSFPData(int devidx, char* sfpnames, int* dTxs, int* dRxs, int* alphas);
```

arguments:	devidx:	device index 0..7
	sfpnames:	pointer to character array of the form: char sfpnames[4][20]
	dTxs:	pointer to integer array of the form: int dTxs[4]
	dRxs:	pointer to integer array of the form: int dRxs[4]
	alphas:	pointer to integer array of the form: int alphas[4]
return value:	=0	success
	<0	error

Note: This can be used to retrieve the SFP module calibration data (if any) from EEPROM.
           For details on SFP module calibration see the MultiHarp manual and the White Rabbbit documentation.


```
int MH_WRabbitSetSFPData(int devidx, char* sfpnames, int* dTxs, int* dRxs, int* alphas);
```

arguments:	devidx:	device index 0..7
	sfpnames:	pointer to character array of the form: char sfpnames[4][20]
	dTxs:	pointer to integer array of the form: int dTxs[4]
	dRxs:	pointer to integer array of the form: int dRxs[4]
	alphas:	pointer to integer array of the form: int alphas[4]
return value:	=0	success
	<0	error

Note: This can be used to place the SFP module calibration data in EEPROM.
           For details on SFP module calibration see the MultiHarp manual and the White Rabbbit documentation.


```
int MH_WRabbitInitLink(int devidx, int link_on);
```

arguments:	devidx:	device index 0..7
	link_on:	0 = off, 1 = on
return value:	=0	success
	<0	error

Note: This can be used to switch the WR link on and off. For details on WR link setup see the MultiHarp manual and the White
           Rabbbit documentation.


```
int MH_WRabbitSetMode(int devidx, int bootfromscript, int reinit_with_mode, int mode);
```

arguments:	devidx:	device index 0..7
	bootfromscript:	boot from script in EEPROM, 0 = yes, 1 = no
	reinit_with_mode:	0 = probe if previous mode set is completed
		1 = re-initialize with new mode
	mode:	0 = off, 1 = Slave, 2 = Master, 3 = Grandmaster
return value:	=0	success
	<0	error

Note: This can be used to make the WR core boot from the init script in EEPROM. It can also be used to select the WR mode and
           probe for completion. For details on WR link setup see the MultiHarp manual and the White Rabbbit documentation.


```
int MH_WRabbitSetTime(int devidx, unsigned int timehidw, unsigned int timelodw);
```

arguments:	devidx:	device index 0..7
	timehidw:	unix time in sec, most significant dword
	timelodw:	unix time in sec, least significant dword
return value:	=0	success
	<0	error

Note: This can be used to set the current UTC time of a MultiHarp's WR core configured as WR master. If a slave is connected it
           will be set to the same time. For details on WR time handling see the White Rabbbit documentation.

int MH_WRabbitGetTime(int devidx, unsigned int* timehidw, unsigned int* timelodw,
unsigned int* subsec16ns);

arguments:	devidx:	device index 0..7
	timehidw:	unix time in sec, most significant dword
	timelodw:	unix time in sec, least significant dword
	subsec16ns:	unix time sub-seconds in steps of 16 ns
return value:	=0	success
	<0	error

Note: This can be used to retrieve the current UTC time of a MultiHarp's WR core. For details on WR time handling see the White Rabbbit documentation.

int MH_WRabbitGetStatus(int devidx, int* wrstatus);

arguments:	devidx:	device index 0..7
	wrstatus:	pointer to an integer receiving the status
return value:	=0	success
	<0	error

Note: The status must be interpreted as a bit field. Use the bit masks WR_STATUS_XXX as defined in mhdefin.h. For details on WR status see the White Rabbbit documentation.

int MH_WRabbitGetTermOutput(int devidx, char* buffer, int* nchar);

arguments:	devidx:	device index 0..7
	buffer:	pointer to a text buffer of at least 513 characters
	nchar:	pointer to an integer receiving the actual text length
return value:	=0	success
	<0	error

Note: When the MultiHarp's WR core has received the commend gui (should be the last line of the init script) it sends terminal output describing its state. This routine can then be used to retrieve that terminal output as a null terminated string. This needs to be done repeatedly. The output will contain escape sequences for control of text color, screen refresh, etc. In order to present it correctly these escape sequences must be interpreted and translated to the corresponding control mechanisms of the chosen display scheme. To take care of this the data can be sent to a terminal emulator. Note that this is read-only. There is currently no way of injecting commands to the WR core's console prompt.

7.3. Warnings

The following is related to the warnings (possibly) generated by the library routine `MH_GetWarnings`. The mechanism and warning criteria are the same as those used in the regular MultiHarp 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 `MH_GetSyncrate` and `MH_Get-Coutrate` has been called (the latter for all channels) before `MH_GetWarnings` is called.

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

Warning	Histo Mode	T2 Mode	T3 Mode
WARNING_SYNC_RATE_ZERO No counts are detected at the sync input. In histogramming and T3 mode this is crucial and the measurement will not work without this signal.	√		√
WARNING_SYNC_RATE_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 75 MHz. This is close to the TDC limit. Sync events will be lost above 78 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 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 disabled. Examples are measurements where the FIFO can absorb all data of interest before it overflows.	√	√	√

Warning	Histo Mode	T2 Mode	T3 Mode
WARNING_INPT_RATE_RATIO This warning is issued in histogramming and T3 mode when the rate at any input channel is higher than 5% of the sync rate. This is the classical pile-up criterion. It will lead to noticeable dead-time artefacts. There are rare measurement scenarios where this condition is expected and the warning can be disabled. Examples are antibunching measurements or rapid-FLIM 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 disabled. In histogramming and T3 mode: If the pulse rate at the sync input is below 75 MHz then a SyncDivider >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 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 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.	√	√	√

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.