

Multichannel Time–Correlated Single Photon Counting Systems and High-Speed Time Taggers



Version 4.0.0.0

Table of Contents

1. Introduction.....	4
2. General Notes.....	5
2.1. What's new in this Version.....	5
2.2. Warranty and Legal Terms.....	6
3. Installation of the Library.....	7
3.1. Requirements.....	7
3.2. Device Access Permissions.....	7
3.3. Installing the Library.....	7
3.4. Installing the Demo Programs.....	8
4. The Demo Applications.....	9
4.1. Functional Overview.....	9
4.2. The Demo Applications by Programming Language.....	10
5. Advanced Techniques.....	14
5.1. Using Multiple Devices.....	14
5.2. Efficient Data Transfer.....	14
5.3. Instant TTTR Data Processing.....	15
5.4. Working with Warnings.....	16
5.5. Hardware Triggered Measurements.....	16
5.6. Working with the External FPGA Interface.....	17
5.7. Working with Event Filtering.....	17
5.8. Synchronizing Devices with White Rabbit.....	18
6. Problems, Tips & Tricks.....	21
6.1. PC Performance Requirements.....	21
6.2. USB Interface.....	21
6.3. Troubleshooting.....	21
6.4. Version tracking.....	21
6.5. New Linux Versions.....	22
6.6. Software Updates.....	22
6.7. Bug Reports and Support.....	22
7. Appendix.....	23
7.1. Data Types.....	23
7.2. Functions Exported by MHLib.so.....	23
7.2.1. General Functions.....	24
7.2.2. Device Related Functions.....	24
7.2.3. Functions for Use on Initialized Devices.....	25
7.2.4. Special Functions for TTTR Mode.....	34
7.2.5. Special Functions for TTTR Mode with Event Filtering.....	35
7.2.6. Special Functions for White Rabbit.....	38

7.2.7. Special Functions for the External FPGA Interface.....	41
7.3. Warnings.....	43

1. Introduction

The MultiHarp is a cutting edge Time-Correlated Single Photon Counting (TCSPC) system with USB 3.0 interface. Its integrated design provides a flexible number of high performance input channels at reasonable cost and enables innovative measurement approaches. The timing circuits of each channel allow high measurement rates up to 78 million counts per second (Mcps) with an excellent time resolution and a world record dead-time of only 650 ps. Timing resolution (depending on the chosen model) can be down to 5 ps. The 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 for use with most common single photon detectors such as Single Photon Avalanche Diodes (SPADs) and Photomultiplier Tube (PMT) modules as well as superconducting nanowire detectors (via preamplifier). Depending on detector and excitation source the width of the overall Instrument Response Function (IRF) can be as small as 50 ps FWHM. The MultiHarp can be purchased in different versions with up to 64 timing inputs and one synchronization (sync) input. The use of these inputs is very flexible. In fluorescence lifetime applications the sync channel is typically used as a synchronization input from a laser. The other inputs are then used for photon detectors. In coincidence correlation applications all inputs can be used for photon detectors.

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

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

The MultiHarp 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 (see separate manual) and as a shared library for Linux described here.

The library supports custom programming in all major programming languages, notably C / C++, C#, Pascal, Python, Rust, 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 library. The MultiHarp is a sophisticated real-time measurement system. In order to work with the system using the library, sound knowledge in your chosen programming language is required.

2. General Notes

This version of the MultiHarp programming library for Linux is suitable for the “x86-64” processor architecture only. We dropped support for 32-bit systems due to the fact that hardly any Linux distribution still offers it.

The library has been tested with gcc 9.4.0 and 11.4.0, Mono 6.8.0 and 6.12.0, Rustc 1.79.0 (+cargo 1.79.0), Python 3.10.12 and 3.11.3, as well as with Lazarus 2.0.12. and 2.2.0 (FreePascal 3.2.0 and 3.2.2). The demos for LabVIEW and Matlab have only been tested under Windows using LabVIEW 2020 and MATLAB R2024b, due to our lack of the Linux versions. If you happen to test with Linux versions please let us know the results.

This manual assumes that you have read the MultiHarp 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. presently 4.0). The library actually has two further sub-version digits, so that the complete version number has four digits (e.g. presently 4.0.0.0). 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 compatibility in regard to the original interface of the corresponding major release. The rightmost digit of the complete version number usually increments to indicate bugfix releases of otherwise identical interface and functionality.

2.1. What’s new in this Version

The new version 4.0.0.0 of the library now supports the latest MultiHarp gateway of February 2025, significantly reducing the timing jitter (P models only). Furthermore, with this gateway the external FPGA interface (EFI) can now also be used via the SFP port at the front, previously used only for White Rabbit connections. This means that the external FPGA interface can now be used with the MultiHarp 150 P too. Another possible benefit is that the link to the external FPGA can now go over optical fiber (also for the MultiHarp 160) which allows longer distances without concerns about EMI.

A very useful new feature is also the API call `MH_SaveDebugDump`. It is provided to help debugging gateway issues by letting the user save a snapshot of the device’s internal FPGA states to a file that then can be submitted for support. Similarly useful for debugging your own code is that the generic error code `ERROR_INVALID_ARGUMENT` is now replaced by separate error codes for each argument so that you can pinpoint the spot of trouble more precisely. Finally, the new version makes the transition to x64 only, 32-bit computers are no longer supported.

Apart from the new features the API as found in v. 3.x remains mostly unchanged. Users upgrading existing code to use the new library will typically only need to change the part of their code where the version number is checked and consider the changes marked in red in section 7.2.

The new version 4.0.0.0 of the library also comes with an improved set of programming demos, most importantly a White Rabbit demo where one device remote-starts the other so that the resulting measurements are closely aligned in time (see section 5.8). There is now also a demo in Rust.

A related novelty is that in addition to MHLib there is now a relatively advanced high-level API package for Python called “snAPI”. It acts as a convenience layer on top of MHLib and readily provides data collection and file writing methods as well as many real-time analysis methods such as intensity and coincidence time traces, FCS and $g^{(2)}$ correlation. Note that snAPI is free of charge but it is a separate software package that you need to download from GitHub and install separately.

Important: Should you still be using a MHLib version older than 3.0.0.0 an upgrade is strongly recommended as in earlier versions the call of some White Rabbit functions might damage the content of the device EEPROM.

2.2. Warranty and Legal Terms

Disclaimer

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

License and Copyright Notice

With the MultiHarp hardware 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 hardware in its current version as of February 2025 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/wikis/home>) 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.

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

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

3. Installation of the Library

3.1. Requirements

Supported hardware is at this time solely the “x86-64” CPU platform as found in the majority of recent PCs. Support for 32-bit platforms has been dropped, for the simple reason that all major Linux distributions are no longer supporting it. Required is a PC with 3.0^{*)}, at least two CPU cores, 2 GHz CPU clock and 4 GB of memory. For optimal TTTR mode throughput to disk a fast solid state disk is recommended.

The library is designed to run on Linux kernel versions 5.0 or higher. It has been tested with the following distributions:

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

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

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

3.2. Device Access Permissions

For device access through libusb suitable permissions for the device must be granted to the normal user, otherwise only the super-user `root` will have access. Recent Linux distributions use udev to handle this. For automated setting of the device access permissions with udev you can add an entry to the set of rules files that are contained in `/etc/udev/rules.d`. Udev processes these files in alphabetical order. The default rule files usually carry names starting with a number. Don't change these files as they could be overwritten when you upgrade your system. Instead, put your custom rule for the MultiHarp in a separate file. The typical content of this file should be:

```
ATTR{idVendor}=="0d0e", ATTR{idProduct}=="0013", MODE="666"
```

A suitable rules file `MultiHarp.rules` is provided in the folder `library` under the unpacked MHLib distribution folder. The install script (see section 3.3) will copy it to the `/etc/udev/rules.d` folder. After installation you will need to disconnect and reconnect the device to get access.

If you have issues obtaining permissions recall that the name of the rules file is important. Each time a device is detected by the udev system, the files are read in alphabetical order until a match is found. Different Linux distributions may use different rule file names for various categories. If there happen to be later rules that are more general (applying to a whole class of devices) they may override your custom rule and the desired access rights. It is therefore important that you use a rules file named such that it gets evaluated after the general case. The default naming `MultiHarp.rules` most likely ensures this but if you see access problems you may want to check.

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

3.3. Installing the Library

The library package is distributed as a zip archive. The shared library as such is provided as a binary file. It supports all MultiHarp models. Starting with version 4.0.0.0 it bears the name `libmhlb.so` and resides by default under `/opt/picoquant/mhlb/`. This is not a strict requirement but it is where the demo programs will look for the library files and therefore it is recommended to use this location.

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

Note that earlier versions were by default installed in `usr/local/lib64/mhl50/` with symlinks to `usr/lib/` or `usr/lib64`. If you used such older versions you may want to check these historic locations and remove the old files in order to avoid conflicts.

The shell script `install` in the `library` distribution directory does the default directory creation and installation in one step. You run it at the command prompt from within the `library` directory. Note that this requires root permissions. As a normal user you must run it preceded with `sudo` like so:

```
sudo ./install
```

The install script will copy the library files (but not the demos) to `opt/picoquant/mhlib/`. The new library name is now `libmhlib.so` and in contrast to the old versions the install script will add the installation path to `picoquant-mhlib.conf` under `/etc/ld.so.conf` and subsequently run `ldconfig`. This ensures that `libmhlib.so` can be found by executables at runtime from anywhere. Note, however, that at compile time of such an executable the linker must still be given the library path. The demo projects are readily set up to look in `opt/picoquant/mhlib/`. The install script also installs a `udev` rule file for device access permissions (see section 3.2). After installation (and still in the `library` folder) you may want to run `./chkinst` to verify the library is installed properly. The same script is also useful to check if there was a previous version 4.x.x.x. installed in `opt/picoquant/mhlib/` and see which it was.

If `chkinst` shows the expected library version and no error is reported then the library is ready to use and can be tested with the demos provided (see sections 3.4 and 4). You may also want to inspect the various files in `opt/picoquant/mhlib/` to read the license terms, check definitions of constants, function signatures, etc.

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

```
chcon -t texrel_shlib_t opt/picoquant/mhlib/libmhlib.so
```

3.4. Installing the Demo Programs

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

4. The Demo Applications

4.1. Functional Overview

Please note that all demo code provided is correct to the best of our knowledge. However, we must disclaim all warranties as to fitness for a particular purpose of this code. It is provided 'as is' for no more than 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. 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 Windows.

For the reason of simplicity, most of the demos will always only use the first MultiHarp device they find, although the library can support multiple devices. In selected programming languages (C, C#) there is an advanced demo showing how to use multiple devices in TTTR mode. If you wish to use some other demo with more than one MultiHarp you need to modify the code accordingly. See section 5.1 on this topic.

As of MHLib version 4.0.0.0 there is now also a demo showing how to use White Rabbit (see section 5.8). As this is an advanced demo it is presently available only in C.

For the more general applications there are demos in C / C++, C#, Delphi / Pascal, Python, LabVIEW and MATLAB. For each of these programming languages / systems there are different demo versions for the different 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 a simple and an advanced demo, the latter being more sophisticated and allowing interactive input of most parameters on the fly. In some programming languages (C, C#) there are also advanced demos to show hardware triggered measurements.

TTTR Mode Demos

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

The MultiHarp 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 LVTTTL 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 more demanding both in programming skills and computer performance. Also consider the speed performance of your programming language. Interpreted Python and Matlab, for example, are very slow. For more information on TTTR mode consult the corresponding section in your 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 advanced demo `tttrmode_instant_processing` (C, Python, Delphi, C#). You may also consult the file demo code installed together with the regular MultiHarp software.

4.2. The Demo Applications by Programming Language

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

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 shared library concepts and related Linux conventions. The ultimate reference for details about how to use the library is in any case the source code of the demos and the header files of the library (`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 makes sense to play safe. Make sure to backup your data and / or perform your development work on a dedicated machine that does not contain valuable data. Note that the library is not re-entrant 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.

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"
}
```

To test any of the demos, consult the MultiHarp manual for setting up your MultiHarp and establish a measurement setup that runs correctly and generates useable test data. This is best done with the regular MultiHarp software under Windows. Compare the settings (notably sync divider, binning and trigger levels) with those used in the demo and use the values that work in your setup when building and testing the demos. Observe the mode input variable going into `MH_Initialize`. It makes a difference if you run T2 or T3 mode. For instance, T2 mode will not allow you to work with high sync rates. For meaningful measurements you will need to adjust the sync divider and the resolution (binning) dependent on your choice of mode.

The C demos are designed to run in a console or terminal window. They need no command line input parameters. They create their output files in their current working directory (`*.out`). The output files will be ASCII-readable in case of the standard histogramming demos. For this demo, the ASCII files will contain multiple columns of integer numbers representing the counts from the 65,536 histogram bins. You can use any editor or a data visualization program to inspect the ASCII histograms. For the TTTR modes the output is stored in binary format for performance reasons. The binary files must be read by dedicated programs according to the format they were written in. The file read demos provided for the 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.

The C# Demos

The C# demos are provided in the `Csharp` subfolder. They have been tested with Mono.

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

With the C# demos you also need to check whether the hard-coded settings are suitable for your actual instrument setup. The demos are designed to run in a terminal window. They need no command line input parameters. They create their output files in their current working directory. The output files will be ASCII in case of the histogramming demo and some of the advanced demos. In the simplest TTTR mode demo the output is stored in binary format for simplicity and performance reasons. The ASCII files of the histogramming demo will contain single or multiple columns of integer numbers representing the counts from the histogram channels. You can use any editor or a data visualization program to inspect the ASCII histograms. The binary files must be read by dedicated programs according to the format they were written in. The file read demos provided for the MultiHarp TTTR data files (.PTU) and the advanced demo `tttrmode_instant_processing` can be used as a starting point to learn this. The file read demos cannot be used directly on the demo output because they expect a file header the demos do not generate. This is intentional in order to keep the MHLib demos focused on the key issues of using the library.

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

The Pascal/ Lazarus Demos

Users of FreePascal / Lazarus please refer to the `Pascal` folder. The source code for Delphi (Windows) and Lazarus is essentially 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.so` known to your application you have to declare each function in your Pascal code as 'external'. This is already prepared in the demo source code.

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

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

The Python Demos

The Python demos are in the `Python` folder. Python users should start their work in histogramming mode from `histomode.py`. The code should be fairly self explanatory. If you update to a new library version please check the function parameters of your existing code against `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. The output file will be readable text in case of the standard histogramming demo and some of the advanced demos. The histogramming demo output files will contain columns of integer numbers representing the counts from the histogram channels. You can use any data visualization program to inspect the histograms. In the simplest TTTR mode demo the output is stored in binary format for performance reasons. The binary files must be read by dedicated programs according to the format they were written in. The file read demos for the regular MultiHarp TTTR data files (.PTU) and the advanced demo `tttrmode_instant_processing` can be used as a starting point to learn this. The file read demos cannot be used directly on the demo output because they expect a file header the demos do not generate. This is intentional in order to keep the MHLib demos focused on the key issues of using the library. Note that even if it may be tempting to directly use the advanced demo `tttrmode_instant_processing` you should not do this routinely. It creates very large files and throughput with interpreted Python is very poor.

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

The LabVIEW Demos

The LabVIEW demos for Linux are identical with the LabVIEW demos for Windows. They automatically detect the operating system and its “bitness” (32 vs 64) and accordingly select the appropriate library name and path. Unfortunately we do not have LabVIEW for Linux, so this feature is untested under Linux. Please kindly report success or error if you happen to work with LabVIEW for Linux.

The first LabVIEW demo (`1_SimpleDemo_MHHisto.vi`) is very simple, demonstrating the basic usage and calling sequence of the provided SubVIs encapsulating the library functionality, which are assembled inside the LabVIEW library `mhlib_x86_x64_UIThread.llb`. The demo starts by calling some of these library functions to setup the hardware in a defined state and continues with a measurement in histogramming mode by calling the corresponding library functions inside a while-loop. Histograms and count rates for all available hardware channels are displayed on the front panel in a waveform graph (you might have to select `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 (`Acq. 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 (here relating to the installed MHLib package for Windows). General helper functions and type-definitions as well as encapsulating LabVIEW libraries (*.llb) can be found in the `_lib` folder (containing further sub-folders) inside the demo folder. In order to facilitate the use of all library functions, additional VIs called `MH_AllDllFunctions_xxx.vi` have been included. These VIs are not meant to be executed but should only give a structured overview of all available library functions and their required context.

Please note:

In addition to the library used by the demos (`mhlib_x86_x64_UIThread.llb`) a second LabVIEW library (llb) is included allowing the library calls to be executed in any thread of LabVIEW's threading engine (`mhlib_x86_x64_AnyThread.llb`). This llb is intended for time critical applications where user actions on the front panel (like e.g., resizing or moving) must not affect the execution of a data acquisition thread containing these library functions (please refer to "Multitasking in LabVIEW": <https://www.ni.com/docs/en-US/bundle/labview/page/multitasking-in-labview.html>). When using this llb you have to make sure that all library functions are called in a sequential order to avoid errors or even program crashes. Also be aware that library functions in `mhlib_x86_x64_AnyThread.llb` have the same names as in `mhlib_x86_x64_UIThread.llb` and opening both libraries at the same time would lead to name conflicts. Therefore, only experienced users should use `mhlib_x86_x64_AnyThread.llb`.

The MATLAB Demos

The MATLAB demos are provided in the `MATLAB` folder. They are contained in `.m` files. You need to have a MATLAB version that supports the `loadlibrary` and `calllib` commands. The earliest version we have tested in this regard was MATLAB 7.3 (under Windows) but any version from 6.5 on should work. For your specific version of MATLAB, please check the documentation of the MATLAB command `loadlibrary` as to whether and how it works. Be careful about the header file name specified in `loadlibrary`. The names are case sensitive and spelling errors will lead to an apparently successful load - but later no library calls will work.

The MATLAB demos are designed to run inside the MATLAB console. They need no command line input parameters. They create output files in their current working directory. The output file will be ASCII in case of the histogramming demo. In TTTR mode the output is stored in binary format for simplicity and performance reasons. You can use any data visualization program to inspect the ASCII histograms. The binary files from TTTR mode must be read by dedicated programs according to the format they were written in. The file read demos for the regular MultiHarp TTTR data files (.PTU) can be used as a starting point. They cannot be used directly on the binary demo output because they expect a file header the demos do not generate. This is intentional in order to keep the MHLib demos focused on the key issues of using the library. The file demo code can (with minor adaptations) in principle be used to process the TTTR records on the fly. However, MATLAB scripts are relatively slow compared to properly compiled code. This may impose throughput limits. You might want to consider compiling Mex files instead. In this case please study the advanced demos `tttrmode_instant_processing` (C, Python, Delphi, C#) which can be used as a starting point to learn this.

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

The Rust Demo

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

5. Advanced Techniques

5.1. Using Multiple Devices

The library is designed to work with multiple MultiHarp devices (up to 8). Most of the demos use only the first device found. In selected programming languages (C, C#) there is an advanced demo showing how to use multiple devices in TTTR mode. If you wish to use some other demo with more than one MultiHarp 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. Filtering out specific serial numbers is shown in the White Rabbit demo (see the .c files under `demos/C/advanced/tttrmode_white_rabbit`).

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 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 typically return after about 10 ms, or even less if no more data is available. However, the actual time to return can sometimes also be slightly longer due to USB overhead and unpredictable operating system latencies, especially when the PC or the USB connection is slow.

As noted above, the transfer is implemented efficiently without excessive CPU load. Nevertheless, assuming large block sizes, the transfer takes some time. Linux therefore gives the unused CPU time to other processes or threads i.e. it waits for completion of the transfer without burning CPU time. This wait time is what can also be used for doing 'useful things' in terms of any desired data processing or storing within your own application. The proper way of doing this is to use multi-threading. In this case you design your program with two threads, one for collecting the data (i.e. working with `MH_ReadFiFo`) and another for processing or storing the data. Multiprocessor systems can benefit from this technique even more. Of course you need to provide an appropriate data queue between the two threads and the means of thread synchronization. Thread priorities are another issue to be considered. Finally, if your program has a graphic user interface you may need a third thread to respond to user actions reasonably fast. Again, this is an advanced technique and it cannot be demonstrated in all detail here. Currently only the most advanced LabVIEW demo uses this technique. Greatest care must be taken not to access the MHLib routines from different threads without strict control of mutual exclusion and maintaining the right sequence of function calls. However, the technique also allows significant throughput improvements and advanced programmers may want to use it. It might be interesting to note that this is how TTTR mode is implemented in the regular MultiHarp software, where sustained count rates over 9 Mcps 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 case of using multiple devices it is also beneficial for overall throughput if you use multi-threading in order to fetch and store data from the individual devices in parallel. Again, re-entrance issues must be observed carefully in this case, at least for all calls accessing the same device.

5.3. Instant TTTR Data Processing

As outlined earlier, collecting TTTR mode streams is time critical when data rates are high. This is why such streams are often just written to disk and then only subsequently post-processed. Nevertheless there are circumstances where it is desirable to process the data instantly "on the fly" as it arrives. This may be for the purpose of an instant preview or for data reduction. The advanced LabVIEW demo nicely demonstrates how to obtain an instant preview. This requires interpreting and bitwise dissecting the TTTR data records as well as correcting for overflows. In order to demonstrate this also for other programming languages there are advanced demos in the subfolders `tttrmode_instant_processing` (C, Python, Delphi, C#). These demos do not write binary output but instead perform an instant processing and write the results out in ASCII. Please note well that this is done purely for educational purposes. Instant processing and writing the results out in ASCII is time consuming and dramatically reduces the achievable throughput. Furthermore, the resulting files are many times larger than the original binary data. Any meaningful application derived from these demos should therefore not write out individual data records but perform some sort of application specific data analysis for preview and/or data reduction. Typical and meaningful examples are histogramming (see subfolders `t3rmode_instant_histogramming` in C, Python, Delphi and C#) or intensity over time traces as shown in the LabVIEW demo. Please note also that such real-time processing requires a suitable choice of programming language. For instance, interpreted Python and Matlab scripts are many times slower than natively compiled code. Ultimate performance is obtained only with a proper compiled language such as C or Pascal. Furthermore, true efficiency (and maximum throughput) can in such a scenario only be achieved by making use of parallel processing on multiple CPUs. This requires programming with multiple threads. In this case you should design your program with at least two threads, one for collecting the data (i.e. working with `MH_ReadFiFo`) and another (or more) for processing, displaying, or storing the data (see also section 5.2). This is not trivial and requires some programming experience.

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

5.4. Working with Warnings

The library provides routines for obtaining and interpreting warnings about critical measurement conditions. The mechanism and warning criteria are the same as those used in the regular MultiHarp software for Windows. 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 starting a measurement or even during the measurement.

It is important to note that the generation of warnings is dependent on the current count rates and the current measurement settings. It was decided that `MH_GetWarnings` does not obtain the count rates on its own, because the corresponding calls take some time and might waste too much processing time. It is therefore necessary that the library routines for count rate retrieval (on all channels) have been called before `MH_GetWarnings` is called. Since most interactive measurement software periodically retrieves the rates anyhow, this is not a serious complication. Note that there are library calls for retrieval of individual count rates (`MH_GetSyncRate` and `MH_GetCountRate`) but in case of performance critical applications it is more efficient 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_GetWarningsText` you can mask out individual warnings by means of the bit masks defined in `mhdefin.h`. See the appendix section 7.3 for a description of the individual warnings.

5.5. Hardware Triggered Measurements

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

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

The MultiHarp's data acquisition can be controlled in various ways. Default is the internal CTC (counter timer circuit). In that case the measurement will take the duration set by the `tacq` parameter passed to `MH_StartMeas`. The other way of controlling the histogram boundaries (in time) is by external LVTTTL signals fed to the control connector pins C1 and C2 (see appendix section *Connectors* of the MultiHarp manual). In that case it is possible to have the acquisition started and stopped when specific signals occur. It is also possible to combine external starting with stopping through the internal CTC. The exact behavior of this scheme is controlled by the parameters supplied to the call of `MH_SetMeasControl`. Dependent on the parameter `meascontrol` the following modes of operation can be obtained:

Symbolic Name	Value	Function
MEASCTRL_SINGLESOT_CTC	0	Default value. Acquisition starts by software command and runs until CTC expires. The duration is set by the <code>tacq</code> parameter passed to <code>MH_StartMeas</code> .
MEASCTRL_C1_GATE	1	Data is collected for the period where C1 is active. This can be the logical high or low period dependent on the value supplied to the parameter <code>startedge</code> .
MEASCTRL_C1_START_CTC_STOP	2	Data collection is started by a transition on C1 and stopped by expiration of the internal CTC. Which transition actually triggers the start is given by the value supplied to the parameter <code>startedge</code> . The duration is set by the <code>tacq</code> parameter passed to <code>MH_StartMeas</code> .

MEASCTRL_C1_START_C2_STOP	3	Data collection is started by a transition on C1 and stopped by by a transition on C2. Which transitions actually trigger start and stop is given by the values supplied to the parameters <code>startedge</code> and <code>stopedge</code> .
MEASCTRL_WR_M2S	4	For White Rabbit only. The WR master shall remote-start measurements on the WR slave. This setting must be made identically on master and slave.
MEASCTRL_WR_S2M	5	For White Rabbit only. The WR slave shall remote-start measurements on the WR master. This setting must be made identically on master and slave.
MEASCTRL_SW_START_SW_STOP	6	This setting permits controlling the duration of measurements purely by software and thereby overcoming the limit of 100h imposed by the hardware CTC. Note that in this case the results of <code>MH_GetElapsedMeasTime</code> will be less accurate. Note also that this feature requires gateway of at least April 2022.

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

Please study the demo code for external hardware triggering and observe the polling loops required to detect the beginning and end of a measurement. Be aware that the speed of you computer and the delays introduced by the operating system's task switching impose some limits on how fast you can run this scheme.

5.6. Working with the External FPGA Interface

The external FPGA interface (EFI) permits retrieving TTTR mode data at substantially higher bandwidth than via USB. Furthermore, since the data is streamed directly to an FPGA, it permits custom data processing in real-time, way beyond the capabilities of a PC in terms of speed and latency.

The new version 4.0.0.0 of the MHLib library supports, among other things, a new gateway (of February 2025). With this gateway the external FPGA interface can now also be used via the SFP port at the front, previously used only for White Rabbit connections. This means that the external FPGA interface can now be used with the MultiHarp 150 P too. Another possible benefit is that the link to the external FPGA can now go over optical fiber (also for the MultiHarp 160) which allows longer distances without concerns about EMI.

In order to enable and use the EFI from the software side, there are a set of dedicated library routines. They are listed in section 7.2 for completeness. However, since using the EFI is an advanced topic in its own, also involving a large amount of FPGA programming details, there is a separate manual for this. It is provided in the MultiHarp software distribution as part of the EFI gateway and software pack. The most recent EFI pack can be downloaded from the MultiHarp 160 product page at <https://www.picoquant.com/products/category/tc-spc-and-time-tagging-modules/multiharp-160>. Please do not be confused, it can also be used for the MultiHarp 150 provided it has up-to-date gateway as mentioned above.

5.7. Working with Event Filtering

Filtering TTTR data streams in hardware is a novel feature available from MHLib v. 3.1.0.0. This helps to reduce USB bus load in TTTR mode by eliminating photon events that carry no information of interest as typically found in many coincidence correlation experiments. Please read the MultiHarp manual for more details.

Note that this new feature requires suitable gateway. Devices shipped after April 2022 will have this readily installed. For older devices you can request an update. Please note that the budget models MultiHarp 150 N cannot be upgraded for this feature. Availability of the feature can be probed with `MH_GetFeatures`, the bit `FEATURE_EVNT_FILT` will be 1 if it is available.

There are two types of event filters. The Row Filters are implemented in the local FPGA processing a row of input channels. Each Row Filter can act only on the input channels within its own row and never on the sync channel. The Main Filter is implemented in the main FPGA processing the aggregated events arriving from the row FPGAs. The Main Filter can therefore act on all channels of the MultiHarp device including the sync channel. Since the Row Filters and Main Filter form a daisychain, the overall filtering result depends on their combined action. Both filters are by default disabled upon device initialization and can be independently enabled when needed.

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

In addition to the filter parameters explained so far it is possible to mark individual channels for use. Used channels will take part in the filtering process. Unused channels will be suppressed altogether. Furthermore, it is possible to indicate if a channel is to be passed through the filter unconditionally, whether it is marked as 'use' or not. The events on a channel that is marked neither as 'use' nor as 'pass' will not pass the filter, provided the filter is enabled.

As outlined earlier, the Row Filters and Main Filter form a daisychain and the overall filtering result depends on their combined action. It is usually sufficient and easier to use the Main Filter alone. The only reasons for using the Row Filter(s) are early data reduction, so as to not overload the Main Filter, and the possible need for more complex filters, e.g. with different time ranges.

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

5.8. Synchronizing Devices with White Rabbit

For a first understanding of what White Rabbit (WR) is and how the MultiHarp supports it, please read the section on the White Rabbit Dialog in the regular MultiHarp software manual. The dialog described there is used to establish a White Rabbit connection and uses the same basic library routines for WR as documented in section 7.2.6 here. Even though it would therefore be possible to implement this step of making a White Rabbit connection in the WR demo code, it was decided not to do this for the following reasons: The initialization code would clutter the demo and it would unduly strain the EEPROMs of the devices because some of it involves writing data into them. Indeed it is not necessary to do this each time again because once there is valid initialization data and a startup script placed in the EEPROMs of the devices they will automatically establish the WR connection at power-up.

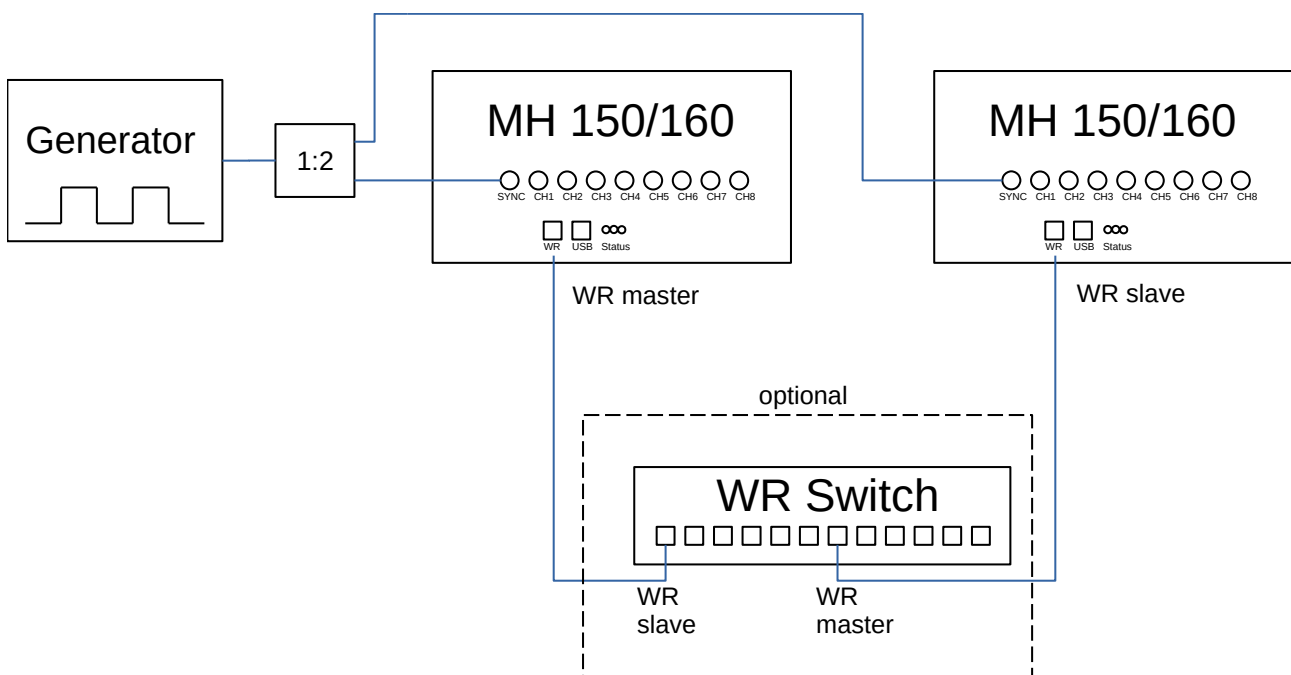
For understanding of the next paragraphs please study the White Rabbit demo code in the MHLib installation folder `demos\C\advanced\tttrmode_withe_rabbit`. The demo consists of two pieces, `wr_master.c` and `wr_slave.c`, which upon build will result in two executables. It assumes that the WR connection has been appropriately prepared and only begins with opening the master and slave devices and initializing them with the suitable choice of external clock reference code. The special feature of this demo is that it shows how one MultiHarp device can remote-start another so that their measurements are beginning closely aligned in time. This works for any measurement mode, however, as the most interesting use case for this is TTTR mode, the WR demo implements the latter.

The demo uses hardcoded settings for the serial numbers of the used devices, for the input trigger levels, etc. You need to change these to match your specific setup. Using specific serial numbers has the benefit that for test purposes one can execute them on the same computer without risking a confusion of WR master and WR slave. Note that the term master and slave here primarily means the two device's relation in terms of their WR connection. The ability of one remote-starting the other (or more) is independent of their roles in the WR connection. In this demo we make the WR master remote-start the WR slave, which could well be reversed but would then look counter-intuitive. The places where this decision is encoded are the calls of `MH_SetMeasControl(devNum_WR_Master, MEASCTRL_WR_M2S, 0, 0)` in `wr_master.c` and `MH_SetMeasControl(devNum_WR_Slave1, MEASCTRL_WR_M2S, 0, 0)` in `wr_slave.c`, where

`MEASCTRL_WR_M2S` means that the WR master shall remote-start the WR slave(s). The demo only uses one slave for which a point-to-point fiber connection is sufficient. It could easily be extended for multiple slaves which then requires a WR switch to connect them all.

The next point of interest in the demo code are the calls to `MH_StartMeas` for master and slave. It is plausible that the master is started by software but it may seem strange that the same is done at the slave side. The magic is that because of the earlier setting of `MEASCTRL_WR_M2S` the slave is aware that it shall be remote-started over WR. It therefore interprets `MH_StartMeas` such as to prepare (arm) for remote start rather than start by itself. Important and directly related is now the function `waitForRemoteStart` that exists only on the slave side. This is where the slave waits for its start over WR. While it would in principle be possible to implement this wait inside `MH_StartMeas`, it would cause that function to block forever if the remote start does not arrive. It was therefore decided to implement `waitForRemoteStart` as a polling loop in the user code where it can easily be interrupted if need be. Once both devices are started, both master and slave will begin fetching their TTTR data and write it to disk, each side writing a file of their own. The executables can be run at remote locations or on the same computer. In their data collection loops both sides are polling for the end of the measurement by calling `MH_CTCStatus`. When the expiration is detected they still do a few more rounds to make sure all data has actually been retrieved from the FIFOs.

For speed reasons only binary data is saved by master and slave. Only after the measurement is completed the binary files are processed and converted to ASCII files. The purpose is that then the two files can be easily compared in order to verify the precision of the WR synchronization. Such a comparison of course requires comparable event data being fed to master and slave device. In order to achieve this, the setup shown in the following figure was used.



A high precision, low jitter signal generator (e.g. Stanford Research Systems model CG635) delivers a continuous pulse train of fixed period, e.g. 1 MHz. The box labelled 1:2 is a power splitter (reflection-free T-pad) that delivers the same signal to master and slave. Assuming that the measurement will be done in T2 mode, the signals can go to the Sync inputs of the devices but any other input channel is fine too. The WR link can be a direct fiber between master and slave but optionally one or more WR switches can sit inbetween. This would allow connecting more slaves. Running the demo code with such a setup would result in two output files where the same set of event records should appear with time tags differing only by the timing uncertainty (jitter) of the MultiHarp and the WR connection and some small residual offset. If you are planning to use WR to synchronize two MultiHarps it is recommended to replicate this little experiment to get familiar with the concept and to verify your setup. When the measurement is completed you can use the Python script `eval.py` to compare the output files. It calculates and plots the event time differences between master and slave. Furthermore, it calculates the rms jitter (i.e. standard deviation) of the event time differences between master and slave. For comparison it also calculates the local rms jitter of the master's and the slave's pulse period.

An interesting detail in the demo code is the call to `MH_GetStartTime` on both sides. It returns the start time of the measurement in picoseconds. 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. As there is no data type to hold the possibly large numbers it is passed in the form of three unsigned integers. In order to do a proper conversion into a single time value a suitable large integer library must be used. Here for the purpose of this demo we just display the raw data. Although in the demo the master and slave performed a WR synchronized measurement, their time alignment can only be accurate to within the same WR TAI cycle (16ns). A correction by means of `MH_GetStartTime` on the two sides can be used to improve the alignment to within ± 3 ns. Even in scenarios where master and slave were started manually with a large unknown delay, the result of `MH_GetStartTime` on the two sides can be used to determine and correct their relative offset so that the time tag data can be aligned to within a few ns.

6. Problems, Tips & Tricks

6.1. PC Performance Requirements

Performance requirements for the library are the same as with the standard MultiHarp software for Windows. 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 reasonably modern CPU and sufficient memory are required. At least a dual core, 2 GHz processor, 4 GB of memory and a fast hard disk 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 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 controller when working with the MultiHarp. USB cables must be qualified for the USB speed you are using. Old and cheap cables often do not meet this requirement and can lead to errors and malfunction. Similarly, many PCs have poor internal USB cabling, so that USB sockets at the front of the PC are often unreliable. Obscure USB errors may also result from subtle damages to USB cables, caused e.g., by sharply bending or crushing them.

6.3. Troubleshooting

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

Under Linux the MultiHarp programming library will access the MultiHarp device through Libusb. You need to make sure Libusb has been installed correctly. Normally this is readily provided by all recent Linux distributions. You can use `lsusb` to check if the device has been detected and is accessible. Please consult the MultiHarp manual for hardware related problem solutions. Note that an attempt at opening a device that is currently used by another process will result in the error code `ERROR_DEVICE_BUSY` being returned from `MH_OpenDevice`. Opening the device may also fail due to insufficient access rights (permissions). This may appear as if the device is not present at all. In this case look at the output of `lsusb`. The MultiHarp should appear with its vendor ID `0D0E` and the device ID `0013`. If the device is actually listed there and you still cannot open it then you probably have not set the right access permissions. See section 3.2 to fix this.

As a next step, try the readily compiled demos supplied with the library. For first tests take the standard histogramming demos. If this is working, your own programs should work as well. Note that the hard coded settings may not be compatible with your experimental setup. Then the pre-compiled demos may not work as expected. In this case you need to change the settings and then rebuild the executable unless you use a scripted language.

6.4. Version tracking

While PicoQuant will always try to maintain a maximum of continuity in further hardware and software development, changes for the benefit of technical progress cannot always be avoided. It may therefore happen, that data structures, calling conventions or program flow will change. In order to design programs that will recognize such changes with a minimum of trouble we strongly recommend that you make use of the functions provided for version retrieval of hardware and library. In any case your software should issue a warning if it detects versions other than those it was tested with. There is a function call that you can use to retrieve the library version number (see section 7.2). Note that this call returns only the major two digits of the version (e.g. 4.0). The library actually has two further sub-version digits, so that the complete version number has four digits (e.g. 4.0.0.0). 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 compatibility in regard to the original interface of the corresponding major release. The very last digit is typically incremented upon bugfixes without functional changes.

6.5. New Linux Versions

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

6.6. Software Updates

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

6.7. Bug Reports and Support

The MultiHarp TCSPC system has gone through extensive testing. It builds on over 25 years of experience with several predecessor models and the feedback of hundreds of users. Nevertheless, it is a fairly complex 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 observe errors or bugs caused by the MultiHarp system please try to find a reproducible error situation. Then email a detailed description of the problem and how to reproduce it, including all relevant circumstances to support@picoquant.com. Alternatively you can also use our support page at www.picoquant.com/contact/support. Please include a listing of your PC configuration including hardware, OS version, versions of used tools, etc, and attach it to your error report. Your feedback will help us to improve the product and documentation.

A very useful new feature of MHLib v. 4.0 is the API call `MH_SaveDebugDump`. It is provided to help debugging gateway issues by letting the user save a snapshot of the device's internal FPGA states to a file that then can be submitted for support. Please implement this feature in your custom code whenever feasible and invoke `MH_SaveDebugDump` immediately after detecting a `FLAG_SYSEERROR` from `MH_GetFlags` and in case of errors in `MH_Initialize`. Then provide the saved file(s) for support.

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. For the MultiHarp you can do so very easily by citing our reference publications:

1. Wahl M., Roehlicke T., Kulisch S., Rohilla S., Kraemer B., Hocke A.C.: Photon arrival time tagging with many channels, sub-nanosecond deadtime, very high throughput, and fiber optic remote synchronization. *Review of Scientific Instruments*, 91, 013108 (2020)
REFERENCE PUBLICATION FOR THE MULTIHARP 150, [preprint available on ArXiv](#)
2. Terhaar R., Rödiger J., Häußler M., Wahl M., Gehring H., Wolff M.A., Beutel F., Hartmann W., Walter N., Hanke J., Hanne P., Walenta N., Diedrich M., Perlot N., Tillmann N., Röhlicke T., Ahangarianabhari M., Schuck C., and Pernice W.H.P. : Ultrafast quantum key distribution using fully parallelized quantum channels. *Optics Express*, Vol. 31, Issue 2, pp. 2675-2688 (2023)
REFERENCE PUBLICATION FOR THE MULTIHARP 160, [preprint available on ArXiv](#)

At our Website we also maintain a large bibliography of publications referring to our instruments. It may serve as a reference for you and other potential users. See <http://www.picoquant.com/scientific/references>. Please kindly submit your publication references for addition to this list.

7. Appendix

7.1. Data Types

The MultiHarp programming library is written in C and its data types correspond to C / C++ data types with bit-widths as follows:

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

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

7.2. Functions Exported by `MHLib.so`

See `mhdefin.h` for predefined constants given in capital letters here. Return values < 0 denote errors. See `errorcodes.h` for the error codes. 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 interactive 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. As of version 4.0 of MHLib it is possible to pass the channel number -1 into such functions and thereby make them act on all channels simultaneously.

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 refsresource);
```

arguments:	devidx:	device index 0..7
	mode:	measurement mode
		0 = histogramming mode
		2 = T2 mode
		3 = T3 mode
	refsresource:	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 base resolution of a device is its best possible resolution as determined by the hardware. It also corresponds to the timing resolution in T2 mode. In T3 and Histogramming mode it is possible to "bin down" the resolution by means of MH_SetBinning. The value returned in binsteps is the number of permitted binning 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 accepting 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 <code>MH_GetNumOfModules</code>)
	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 detecting a `FLAG_SYSERROR` from `MH_GetFlags`. In case of `FLAG_SYSERROR` please provide this information for support.

```
int MH_SaveDebugDump(int devidx, char* filepath); // new since v. 4.0
```

arguments:	devidx:	device index 0..7
	filepath:	pointer to a string holding the destination path including a trailing path delimiter
return value:	=0	success
	<0	error

Note: Use this call to obtain and save hardware debug information. You can call it immediately after receiving an error code <0 from any library call. It is of particular value after detecting a `FLAG_SYSERROR` from `MH_GetFlags` and in case of errors in `MH_Initialize`. Please provide the saved file(s) 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 timing jitter. The readings obtained with `MH_GetCountRate` are internally corrected for the divider setting and deliver the external (undivided) rate. The sync divider should not be changed while a measurement is running, the recorded data will then likely be corrupted.

```
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

Note: The hardware uses a 10 bit DAC that can resolve the level value only in steps of about 2.34 mV.

```
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_SetSyncChannelEnable (int devidx, int enable); // new since v3.1
```

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

Note: This is really only useful in T2 mode. Histogramming and T3 mode need an active sync signal.

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

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

Note: This call is primarily intended for the suppression of afterpulsing artefacts of some detectors. The corresponding hardware functionality is regularly available in devices manufactured after September 2019. Earlier devices need a firmware upgrade to provide this feature. 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 occurring 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); // changed since v4.0
```

arguments: devidx: device index 0..7
 channel: input channel index 0..nchannels-1, or -1 for all channels
 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`. The hardware uses a 10 bit DAC that can resolve the level value only in steps of about 2.34 mV. As of version 4.0 of MHLib it is possible to pass the channel number -1 and thereby make the setting for all channels simultaneously.

```
int MH_SetInputChannelOffset (int devidx, int channel, int value); // changed since v4.0
```

arguments: devidx: device index 0..7
 channel: input channel index 0..nchannels-1, or -1 for all channels
 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`. As of version 4.0 of MHLib it is possible to pass the channel number -1 and thereby make the setting for all channels simultaneously.

```
int MH_SetInputChannelEnable (int devidx, int channel, int enable); // changed since v4.0
```

arguments: devidx: device index 0..7
 channel: input channel index 0..nchannels-1, or -1 for all channels
 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()`. As of version 4.0 of MHLib it is possible to pass the channel number -1 and thereby make the setting for all channels simultaneously.

```
int MH_SetInputDeadTime (int devidx, int channel, int on, int deadtime); // changed since v4.0
```

arguments: devidx: device index 0..7
 channel: input channel index 0..nchannels-1, or -1 for all channels
 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 need a firmware upgrade to provide this feature. 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 occurring 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. As of version 4.0 of MHLib it is possible to pass the channel number -1 and thereby make the setting for all channels simultaneously.

```
int MH_SetInputHysteresis (int devidx, int hystcode); // new since v3.0
```

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

return value: =0 success
 <0 error

Note: This call is intended for the suppression of noise or pulse shape artefacts of some detectors by setting a higher input hysteresis. The corresponding functionality is regularly available in devices manufactured after March 2021. Earlier devices will need a firmware upgrade to provide this feature. You can use the library call `MH_GetFeatures` or evaluate the return code of `MH_SetInputHysteresis` to determine if the feature is available. Note that this setting affects all timing inputs (sync and channels) simultaneously.

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

arguments: devidx: device index 0..7
 stop_ovfl: 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_ovfl` 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 = binsteps-1 (see `MH_GetBaseresolution`)

return value: =0 success
 <0 error

Note: Binning only applies in Histogramming and T3 Mode. The binning code corresponds to repeated doubling, i.e.

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

```
int 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 i.e. 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_SINGLESOT 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 6 = MEASCTRL_SW_START_SW_STOP // new since v3.1
	startedge:	edge selection code 0 = falling 1 = rising
	stopedge:	edge selection code 0 = falling 1 = rising
return value:	=0	success
	<0	error

Note: This sets the measurement control mode and must be called before starting a measurement. The default after initialization (if this function is not called) is 0, i.e. CTC controlled acquisition time. The modes 1..5 allow hardware triggered measurements through LVTTTL 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. MEASCTRL_SW_START_SW_STOP permits controlling the duration of measurements purely by software and thereby overcoming the limit of 100 h imposed by the hardware CTC. Note that in this case the results of MH_GetElapsedMeasTime will be less accurate. Note also that MEASCTRL_SW_START_SW_STOP requires gateware of at least April 2022. The parameters startedge and stopedge are relevant only for the control codes 1 through 3.

```
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.

```
int 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

Note: If beforehand MEASCTRL_SW_START_SW_STOP was set via MH_SetMeasControl, the parameter tacq will be ignored and the measurement will run until MH_StopMeas is called. This can be used to overcome the limit of 100 h imposed by the hardware CTC. However, the results of MH_GetElapsedMeasTime will in this case be less accurate as it can only use the timers of the operating system. Similarly, MH_StartMeas will behave differently if beforehand MEASCTRL_C1_XXX or MEASCTRL_WR_M2S or MEASCTRL_WR_S2M were set. Calling MH_StartMeas at the device to be hardware- or remote-started will then not actually start a measurement but only arm the device to wait for hardware/remote start. In these cases the user code must subsequently poll MH_CTCStatus for status==0 to learn whether the measurement has actually started, before it can begin polling for status==1 to learn whether the measurement has ended.

```
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* ctccstatus);
```

arguments:	devidx:	device index 0..7
	ctccstatus	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 actual len 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. For speed reasons this would be preferred anyhow.

```
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 <code>int cntrates[num_channels]</code> 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. Make sure that the array `cntrates` is large enough for the number of input channels your device has. The safest approach is to dimension it for `MAXINPCHAN`, i.e. 64 channels.

```
int MH_GetFlags (int devidx, int* flags);
```

arguments:	devidx:	device index 0..7
	flags:	pointer to an integer returns the current status flags (a bit pattern)


```

return value:      =0          success
                  <0          error

```

Note: Use the predefined bit mask values in `mhdefn.h` (e.g. `FLAG_OVERFLOW`) to extract individual bits through a bitwise AND. The possible flags are:

`FLAG_OVERFLOW` `0x0001`

This flag occurs in histo mode only. It indicates that a histogram measurement has reached the maximum count as specified via `MH_SetStopOverflow`.

`FLAG_FIFOFULL` `0x0002`

This flag occurs in TTTR mode only. It indicates that the main USB data FIFO has run full. The measurement will then have to be aborted as data integrity is no longer maintained.

`FLAG_SYNC_LOST` `0x0004`

This flag may occur in T3 mode and in histo mode. It indicates that the sync signal has been lost which in this case is critical as the function of T3 mode and histo mode relies on an uninterrupted sync signal.

`FLAG_REF_LOST` `0x0008`

This flag will occur when the MultiHarp is programmed to use an external reference clock and this reference clock is lost.

`FLAG_SYSERROR` `0x0010`

This flag indicates an error of the hardware or internal software. The user should in this case call the library routine `MH_GetDebugInfo` and provide the result to PicoQuant support.

`FLAG_ACTIVE` `0x0020`

This flag indicates that a measurement is running.

`FLAG_CNTS_DROPPED` `0x0040`

This flag indicates that counts were dropped at the first level FIFO following the TDC of an input channel. This occurs typically only at extremely high count rates. Dependent on the application this may or may not be considered critical.

```

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. Note that when `MEASCTRL_SW_START_SW_STOP` is used (controlling the duration of measurements purely by software) the results of `MH_GetElapsedMeasTime` will be less accurate.

```

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 in ps

return value:    =0          success
                  <0          error

```

Note: This can be used to retrieve the start time of a measurement with high 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.

```
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 <code>MH_GetWarnings</code>
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: The call will return typically after 10 ms and even less if no more data could be fetched. The call may occasionally take longer due to USB overhead and operating system latencies, especially when the PC or the USB connection is slow. Buffer must not be accessed until the call returns. Note that even when `MH_CTCStatus` reports expiration of the measurement time there may still be data in the FIFO. In order to fully retrieve this residue it may be necessary to call `MH_ReadFiFo` a few times more.

```
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 LVTTTL 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 LVTTTL 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 meaningful in TTTR mode only. It is not normally required but it can be used to deal with glitches on the marker lines. Markers following a previous marker within the hold-off time will be suppressed. Note that the actual hold-off time is only approximated to about ± 20 ns.

```
int MH_SetOfICompression (int devidx, int holdtime); // new since v3.1
```

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

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

Note: This setting is meaningful in TTTR mode only. It is not normally required but it can be useful when data rates are very low and there are more overflows than photons. The hardware will then count overflows and only transfer them to the FIFO when `holdtime` has elapsed. The default value is 2 ms as of MHLib v3.1. Previously it used to be 0 (no compression). If you are implementing a real-time preview and data rates are very low you may observe “stutter” when `holdtime` is chosen too large because then there is nothing coming out of the FIFO for longer times. Indeed this is aggravated by the fact that the FIFO has a transfer granularity of 16 records. Supposing a data stream without any regular event records (i.e. only overflows) this means that effectively there will be transfers only every $16 \cdot \text{holdtime}$ ms. Whenever there is a true event record arriving (photons or markers) the previously accumulated overflows will instantly be transferred. This may be the case merely due to dark counts, so the “stutter” would rarely occur. In any case you can switch overflow compression off by setting `holdtime` 0.

7.2.5. Special Functions for TTTR Mode with Event Filtering

Starting from version 3.1.0.0 the library supports event filtering in hardware (see section 5.7). This helps to reduce USB bus load in TTTR mode by eliminating photon events that carry no information of interest as typically found in many coincidence correlation experiments. Please read the MultiHarp manual for details. Note that this new feature requires suitable gateway. Devices shipped after April 2022 will have this readily installed. For older devices you can request an update. Please note that the budget models MultiHarp 150 N cannot be upgraded for this feature. Availability of the feature can be probed with `MH_GetFeatures`, the bit `FEATURE_EVNT_FILT` will be 1 if it is available.

```
int MH_SetRowEventFilter(int devidx, int rowidx, int timerange, int matchcnt,
                        int inverse, int usechannels, int passchannels); // new since v3.1
```

arguments:

devidx:	device index 0..7
rowidx:	index of the row of input channels, counts bottom to top (ROWIDXMIN..ROWIDXMAX)
timerange:	time distance in ps to other events to meet filter condition (TIMERANGEMIN..TIMERANGEMAX)
matchcnt:	number of other events needed to meet filter condition (MATCHCNTMIN..MATCHCNTMAX)
inverse:	set regular or inverse filter logic 0 = regular, 1 = inverse
usechannels:	integer bitfield with bit0 = leftmost input channel,.. bit7 = rightmost input channel, bit8 and higher must be 0 bit value 1 = use this channel, bit value 0 = ignore this channel
passchannels:	integer bitfield with bit0 = leftmost input channel,.. bit7 = rightmost input channel, bit8 and higher must be 0 bit value 1 = unconditionally pass this channel, bit value 0 = pass this channel subject to filter condition

return value:

=0	success
<0	error

Note: This sets the parameters for one Row Filter implemented in the local FPGA processing that row of input channels. Each Row Filter can act only on the input channels within its own row and never on the sync channel. The value `timerange` determines the time window the filter is acting on. Note that `timerange` acts both ways in time so that the window width is actually $2 * \text{timerange}$. The parameter `matchcnt` specifies how many other events must fall into the chosen time window for the filter condition to act on the event at hand. The parameter `inverse` inverts the filter action, i.e. when the filter would regularly have eliminated an event it will then keep it and vice versa. For the typical case, let it be not inverted. Then, if `matchcnt` is 1 we will obtain a simple 'singles filter'. This is the most straight forward and most useful filter in typical quantum optics experiments. It will suppress all events that do not have at least one coincident event within the chosen time range, be this in the same or any other channel marked as 'use' in this row. The bitfield `passchannels` is used to indicate if a channel is to be passed through the filter unconditionally, whether it is marked as 'use' or not. The events on a channel that is marked neither as 'use' nor as 'pass' will not pass the filter, provided the filter is enabled. The parameter settings are irrelevant as long as the filter is not enabled. The output from the Row Filters is fed to the Main Filter. The overall filtering result depends on their combined action. Only the Main Filter can act on all channels of the MultiHarp device including the sync channel. It is usually sufficient and easier to use the Main Filter alone. The only reasons for using the Row Filter(s) are early data reduction, so as to not overload the Main Filter, and the possible need for more complex filters, e.g. with different time ranges.

```
int MH_EnableRowEventFilter(int devidx, int rowidx, int enable); // new since v3.1
```

arguments:

devidx:	device index 0..7
rowidx:	index of the row of input channels, counts bottom to top (ROWIDXMIN..ROWIDXMAX)
enable:	desired enable state of the filter 0 = disabled 1 = enabled

return value:

=0	success
<0	error

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

```
int MH_SetMainEventFilterParams(int devidx, int timerange, int matchcnt, int inverse); // new since v3.1
```

arguments:

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

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

Note: This sets the parameters for the Main Filter implemented in the main FPGA processing the aggregated events arriving from the row FPGAs. The Main Filter can therefore act on all channels of the MultiHarp device including the sync channel. The value `timerange` determines the time window the filter is acting on. Note that `timerange` acts both ways in time so that the window width is actually $2 * \text{timerange}$. The parameter `matchcnt` specifies how many other events must fall into the chosen time window for the filter condition to act on the event at hand. The parameter `inverse` inverts the filter action, i.e. when the filter would regularly have eliminated an event it will then keep it and vice versa. For the typical case, let it be not inverted. Then, if `matchcnt` is 1 we obtain a simple 'singles filter'. This is the most straight forward and most useful filter in typical quantum optics experiments. It will suppress all events that do not have at least one coincident event within the chosen time range, be this in the same or any other channel. In order to mark individual channel as 'use' and/or 'pass' please use `MH_SetMainEventFilterChannels`. The parameter settings are irrelevant as long as the filter is not enabled. Note that the Main Filter only receives event data that passes the Row Filters (if they are enabled). The overall filtering result therefore depends on the combined action of both filters. It is usually sufficient and easier to use the Main Filter alone. The only reasons for using the Row Filters are early data reduction, so as to not overload the Main Filter, and the possible need for more complex filters, e.g. with different time ranges.

```
int MH_SetMainEventFilterChannels(int devidx, int rowidx, int usechannels, int passchannels); // new since v3.1
```

```
arguments:      devidx:      device index 0..7
                 rowidx:      index of the row of input channels, counts bottom to top
                               (ROWIDXMIN..ROWIDXMAX)
                 usechannels:  integer bitfield with bit0 = leftmost input channel,..
                               bit7 = rightmost input channel,
                               if rowindex is 0 then bit8 = sync channel,
                               bit9 and higher must be 0
                               bit value 1 = use this channel,
                               bit value 0 = ignore this channel
                 passchannels: integer bitfield with bit0 = leftmost input channel,..
                               bit7 = rightmost input channel,
                               if rowindex is 0 then bit8 = sync channel
                               bit9 and higher must be 0
                               bit value 1 = unconditionally pass this channel,
                               bit value 0 = pass this channel subject to filter condition

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

Note: This selects the Main Filter channels for one row of input channels. Doing this row by row is to address the fact that the various device models have different numbers of rows. The bitfield `usechannels` is used to indicate if a channel is to be used by the filter. The bitfield `passchannels` is used to indicate if a channel is to be passed through the filter unconditionally, whether it is marked as 'use' or not. The events on a channel that is marked neither as 'use' nor as 'pass' will not pass the filter, provided the filter is enabled. The settings for the sync channel are meaningful only in T2 mode and will be ignored in T3 mode. The channel settings are irrelevant as long as the filter is not enabled. The Main Filter receives its input from the Row Filters. If the Row Filters are enabled, the overall filtering result therefore depends on the combined action of both filters. Only the Main Filter can act on all channels of the MultiHarp device including the sync channel. It is usually sufficient and easier to use the Main Filter alone. The only reasons for using the Row Filter(s) are early data reduction, so as to not overload the Main Filter, and the possible need for more complex filters, e.g. with different time ranges.

```
int MH_EnableMainEventFilter(int devidx, int enable); // new since v3.1
```

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

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

Note: When the filter is disabled all events will pass. This is the default after initialization. When it is enabled, events may be filtered out according to the parameters set by way of `MH_SetMainEventFilterParams` and `MH_SetMainEventFilterChannels`. Note that the Main Filter only receives event data that passes the Row Filters (if they are enabled). The overall filtering result therefore depends on the combined action of both filters. It is usually sufficient and easier to use the Main Filter alone. The only reasons for using the Row Filters are early data reduction, so as to not overload the Main Filter, and the possible need for more complex filters, e.g. with different time ranges.

```
int MH_SetFilterTestMode(int devidx, int testmode); // new since v3.1
```

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

return value: =0 success
 <0 error

Note: One important purpose of the event filters is to reduce USB load. When the input data rates are higher than the USB bandwidth, there will at some point be a FIFO overrun. It may under such conditions be difficult to empirically optimize the filter settings. Setting filter test mode disables all data transfers into the FIFO so that a test measurement can be run without interruption by a FIFO overrun. The library routines `MH_GetRowFilteredRates` and `MH_GetMainFilteredRates` can then be used to monitor the count rates after the Row Filter and after the Main Filter. When the filtering effect is satisfactory the test mode can be switched off again to perform the regular measurement.

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

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

return value: =0 success
 <0 error

Note: This call retrieves the count rates after the Row Filters before entering the Main Filter. A measurement must be running to obtain valid results. Allow at least 100 ms to get a new reading. This is the gate time of the rate counters. Make sure that the array `cntrates` is large enough for the number of input channels your device has. The safest approach is to dimension it for `MAXINPCHAN`, i.e. 64 channels.

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

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

return value: =0 success
 <0 error

Note: This call retrieves the count rates after the Main Filter before entering the FIFO. A measurement must be running to obtain valid results. Allow at least 100 ms to get a new reading. This is the gate time of the rate counters. Make sure that the array `cntrates` is large enough for the number of input channels your device has. The safest approach is to dimension it for `MAXINPCHAN`, i.e. 64 channels.

7.2.6. Special Functions for White Rabbit

```
int MH_WRabbitGetMAC (int devidx, unsigned char* mac_addr); // changed since v4.0
```

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, unsigned char* mac_addr); // changed since v4.0
```

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 Rabbit 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 Rabbit 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 Rabbit 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 Rabbit 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 Rabbit 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 Rabbit 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 Rabbit 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 Rabbit 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 Rabbit 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 command `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.

7.2.7. Special Functions for the External FPGA Interface

The functions in this category are provided for use with the External FPGA Interface (EFI) of the MultiHarp. In order to determine their availability you can use `MH_GetFeatures` in conjunction with the macro `FEATURE_EXT_FPGA` defined in `mhdefin.h`. For all further details on how to work with the EFI please see the separate manual on the topic.

```
int MH_ExtFPGAInitLink (int devidx, int linknumber, int on); // new since v3.0
```

arguments:	devidx:	device index 0..7	
	linknumber:	index 0..8 of the link to be initialized	
	on:	0 = off, 1 = EFI REAR, 2 = EFI SFP	// changed since v4.0
return value:	=0	success	
	<0	error	

Note: Sets the state of a link to the external FPGA. The number of usable links depends on the configuration of the MultiHarp in use. Using EFI REAR with the MultiHarp 160, the base unit contains the links zero to two and every expansion unit adds two links. Using EFI SFP only link zero can be used.

```
int MH_ExtFPGAGetLinkStatus (int devidx, int linknumber, unsigned int* status); // new since v3.0
```

arguments:	devidx:	device index 0..7	
	linknumber:	index 0..8 of the link to be queried	
	status:	pointer to unsigned int buffer to receive the status	
return value:	=0	success	
	<0	error	

Note: The number of usable links depends on the configuration of the MultiHarp 160 in use. The status is reported for each link independently. The meaning of the status is dependent on the external FPGA and is further defined in the EFI programming guide.

```
int MH_ExtFPGASetMode (int devidx, int mode, int loopback); // new since v3.0
```

arguments:	devidx:	device index 0..7	
	mode:	stream mode code to be set, see <code>mhdefin.h</code>	
	loopback:	loopback mode code to be set, see <code>mhdefin.h</code>	
return value:	=0	success	
	<0	error	

Note: For details on the meaning of the mode and loopback values see the EFI programming guide.

```
int MH_ExtFPGAResetStreamFifos (int devidx); // new since v3.0
```

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

Note: This function should typically be called after each call of the `MH_Initialize()` function. For details see the EFI programming guide.

```
int MH_ExtFPGAUserCommand (int devidx, int write, unsigned int addr, unsigned int* data);  
                                                                    // new since v3.0
```

arguments:	devidx:	device index 0..7
	write:	0 = read, 1 = write
	addr:	an "address" for the data in the external FPGA
	data:	pointer to location of data to write or to receive

return value:	=0	success
	<0	error

Note: This function is provided to allow data transfer to and from the external FPGA. The "address" may be understood as a command code associated with the data. The meaning of such user commands is specific to the custom EFI design and must be implemented there in order to work here at the software level. The primary objective is to facilitate control mechanisms but data transfer is also possible, albeit with limited speed.

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 (see section 5.4).

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_GetCoutrate` 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 Modes	T2 Mode	T3 Mode
WARNING_SYNC_RATE_ZERO No pulses are detected at the sync input. In histogramming and T3 mode this is crucial and the measurement will not work without this signal.	√		√
WARNING_SYNC_RATE_VERY_LOW The detected pulse rate at the sync input is below 100 Hz and cannot be determined accurately. Other warnings may not be reliable under this condition.	√		√
WARNING_SYNC_RATE_TOO_HIGH The pulse rate at the sync input (after the divider) is higher than 75 MHz. This is close to the sustainable front end speed. 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 85 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_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_DIVIDER_TOO_SMALL The pulse rate at the sync input (after the divider) is higher than 75 MHz. This is close to the sustainable front end speed. Sync events will be lost above 78 MHz. To avoid this, increase the sync divider.	√		√
WARNING_TIME_SPAN_TOO_SMALL This warning is issued in histogramming and T3 mode when the sync period (1/SyncRate) is longer than the start to stop time span that can be covered by the histogram or by the T3 mode records. You can calculate this time span as follows: Span = Resolution * 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.	√	√	√
WARNING_USB20_SPEED_ONLY This warning appears when the MultiHarp's USB connection is running only at USB 2.0 speed. For proper performance it should be running at USB 3.0 super speed. Check the cabling and the USB port in use. The same issue is indicated by the USB status LED showing yellow instead of green.	√	√	√

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

All information given here is reliable to our best knowledge. However, no responsibility is assumed for possible inaccuracies or omissions. Specifications and external appearances are subject to change without notice.



PicoQuant GmbH
 Rudower Chaussee 29 (IGZ)
 12489 Berlin
 Germany

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