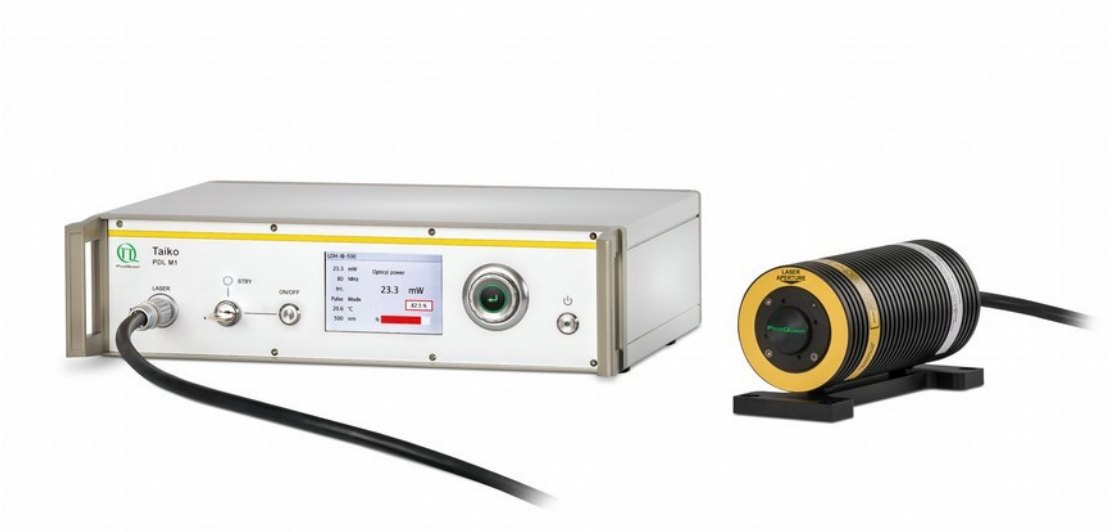
Taiko PDL M1

Application Programming Interface
Library for Software Developers

PicoQuant

Programming Interface for the Taiko PDL M1



Programming Reference Handbook

Document version 2.1.1

Table of Contents

1. Introduction ... 3

2. Library for Software Developers ... 4
   2.1. Covered Library and Hardware Versions ... 4
   2.2. General Notes ... 4
      2.2.1. Naming Conventions ... 4
      2.2.2. Calling Conventions ... 4
      2.2.3. Transferring Arguments and Memory Allocation ... 5
      2.2.4. Return Values ... 5
      2.2.5. Running Considerations ... 5
      2.2.6. Status Updates and Tagged Communication ... 5
   2.3. Using the Taiko API DLLs under Linux ... 6
      2.3.1. Requirements ... 6
      2.3.2. Device Access Permissions ... 6
      2.3.3. Using the Library and Demo Programs ... 7

3. List of API Functions ... 8
   3.1. Interface Functions ... 8
   3.2. Basic Device Functions ... 10
   3.3. Device Information Functions ... 12
   3.4. Laser Head Information Functions ... 14
   3.5. Status and Error Information Functions ... 16
   3.6. Laser Locking Functions ... 19
   3.7. Laser Emission Mode Functions ... 19
   3.8. Triggering and Gating Functions ... 21
   3.9. Pulse Frequency and Burst Setting Functions ... 23
   3.10. Functions for Temperature Settings ... 25
   3.11. Laser Head Functions for Pulse Power Settings ... 27
   3.12. Laser Head Functions for CW Power Settings ... 29
   3.13. Special Laser Head Functions ... 31
   3.14. Preset Functions ... 31

4. Demonstration Programs ... 33

5. Legal Terms ... 34
   5.1. Copyright ... 34
   5.2. Trademarks ... 34

6. Appendix ... 35
   6.1. Table of Common Constants ... 35
   6.2. Table of Return / Error Codes ... 35

6.3. Table of Assigned Status Bits....37
6.4. Table of Useful Status Masks....39
6.5. Table of Declared Tag Types....42
6.6. Table of Documented Tags....42
6.7. Table of Supported Temperature Scales....44
6.8. Table of Laser Head Feature Bits....44
6.9. Table of Laser Head Types....44
6.10. Index....45

1. Introduction

The Taiko is a smart, universal laser driver that can operate and monitor any picosecond pulsed laser head from the LDH-I Series. As a smart driver, the Taiko interfaces with a laser head to read out and display various operational parameters. These include the current emission wavelength, laser head temperature, repetition rate, current output intensity and pulse shape regime based on calibration data stored in the head.

Every LDH-I head is calibrated during manufacturing with regards to its intensity / output power curve, pulse shape regime, and temperature dependent wavelength shift. The Taiko is thus able to provide an indication of current output power and central wavelength during operation.

The Taiko laser driver can be controlled via two interface types: either through the local, single-button menu-driven system or remotely by software running on a PC (using a USB connection). A powerful, Windows based GUI written by PicoQuant is included in the Taiko package.

PicoQuant also provides an Application Programming Interface (API) that allows writing your own Windows-based control software for the Taiko. This reference handbook aims to provide an overview of all API functions available for such tasks.

2. Library for Software Developers

In addition to the powerful, general purpose Taiko control software included, you might want to create your own control sequences or graphical user interfaces that are tailored to your needs. This should be an easy task for an experienced software developer with the API provided as Windows™ dynamic link library.

The library is provided in two different "flavors": as x86 (32 bit) and x64 (64 bit) type. You can find out the library's version number by checking the file version displayed by the properties page of the Windows™ Explorer: right click on the file name, select "Properties" from the context menu and then navigate to the "Details" tab.

The major high and low word code the actual software version. The bit width of the target architecture is encoded in the third part of the version number (a. k. a. "minor high word"), while the minor low word contains the build number. A version number like "2.0.32.xxxx" stands for the software version 2.0, compiled for an x86 target architecture and with build number xxxx. Correspondingly, "2.0.64.xxxx" identifies the same library version, but compiled for an x64 architecture.

With the system software (GUI and DLL), we also provide "ready to use" library interfaces in C/C++ and Delphi including language specific declaration files and the import library "PDLM_Lib.lib". Developers who use other languages supporting access to DLLs may build their own interfaces analogue to the purchased, by simply adapting the declaration files to their desired language and linking their project with the aforementioned import library. It might be necessary to encapsulate the functions-to-call for convenience.

2.1. Covered Library and Hardware Versions

This handbook refers to library version 2.1.[target].[build > 4077] (or higher)

Please note that the version numbering convention used for this library will increase the minor version number only if functions have been discontinued. As work on the library is on going, you might see improved performance, stability, or functionality by simply substituting your library for the most recent version, as long as the major and minor version numbers are still the same. Also, make sure that you do not use a build number lower than the one you built and tested your software against. All library version with later build numbers should work as expected with your product.

The Taiko PDL M1 should have the firmware version 2.1.xxx (or higher)

Newer firmware version may provide improved performance or additional features. Please check the firmware release notes prior to updating. A special note will be provided in the description of API functions that require a higher firmware version.

2.2. General Notes

All functions exported by this library behave according to a few conventions, the most important of which are listed in the following sections. Since the library was implemented in C/C++, we chose to document it in the same language. In order to focus on the essentials, we omit storage classes, calling conventions and all compiler specific details for individual functions. Note that if you use Pascal, we used true booleans wherever appropriate.

2.2.1. Naming Conventions

Every API function name starts with the library preamble "PDLM_". Note that in this handbook, all functions have been sorted into logical groups for clarity's sake.

2.2.2. Calling Conventions

Note that all functions described here use the `stdcall` calling convention. Refer to the purchased demo code and to the developer's manual specific to your compiler for more detailed information.

2.2.3. Transferring Arguments and Memory Allocation

The transferring convention for all input arguments (marked with an “I”) is “by value” except for strings. For input string arguments as well as for all output arguments (marked with “O”), the transferring convention is “by reference”. Bi-directional arguments (marked with “B”) can be used for input as well as output arguments. Therefore they use the transferring convention “by reference” in either direction. Use the “var” – clause in Pascal resp. a pointer to the destination variable in C/C++ to implement outputs or bi-directionals.

Calling programs have to take care of memory allocation for output arguments. Refer to the C header files for a list of necessary maximal string or array lengths. All strings referred to by this document are strings of 8 bit characters (ISO-8859) and zero terminated. Note that all length information for strings are given as net sizes, so don't forget for the zero termination byte in C/C++.

2.2.4. Return Values

All functions return an error code (signed integer, 32 bit).

function returns:	0	:	success
	< 0	:	error

You should always check whether the return code of every function call is 0 (i.e. “PDLM_ERROR_NONE”). Note that the library interface function “**PDLM_DecodeError**” can convert any returned error code into a human-readable text string. Refer to Appendix 6.2 for a list of error codes.

2.2.5. Running Considerations

Most of the functions described here need an operational Taiko PDL M1 to work properly. Since the library is prepared to work with more than one connected device, you will have to identify the device you want to address by its USB channel index (`iDevIdx`, ranging from 0 to 7).

That index can be obtained from the Windows Device Manager. In a more generic way, you could build a loop that tries to open devices on all channels and – using the returned error code – compare to the serial number of the desired device. For an even more convenient approach to this task, we designed the special function "`PDLM_OpenGetSerNumAndClose`" (see section 3.2), which can get the state and serial number of a device eventhough it might already be opened.

Note that the open device operation establishes an exclusive access to the device! You cannot open a device if another program is already having access to it. However, an application may open more than one device and communicate with them quasi simultaneous. Do keep in mind that **the library is not thread-safe** by design.

2.2.6. Status Updates and Tagged Communication

Since the Taiko PDL M1 can be controlled via the local and a remote interface at the same time, any parameter or state changes have to be communicated the remote host software, no matter whether they were triggered manually or autonomously. This is required to ensure that the locally and remotely displayed status of the Taiko is always synchronized.

Any such change will result in the setting or deleting of an associated flag in the Taiko’s status word. As an example, changing a parameter such as the temperature will set a status flag with the symbolic name “`PDLM_DEVSTATE_PARAMETER_CHANGES_PENDING`”. A list of all status flags can be found in Appendix 6.3. Note that flags can be logically grouped and even evaluated together (e.g., all laser locking status flags).

The Taiko’s status word is regularly polled in the background by the API DLL using a timer (at least every 750 ms). This process is automatic and does not need to be triggered by the host software. Certain groups of flags can also lead the API DLL to send a windows message to the registered host application that is listening. Refer to the function "`PDLM_SetHWND`" for more details. If the host application supports handling these messages, then state changes can be updated asynchronously.

Information on state or parameter changes are transferred between the API DLL and host applications based on a tagged communication system. This system is a (none or weakly specified) transfer method that is, in particular, used to send status feedback from the Taiko to a host application.

When a host application detects that a parameter has changed either via a flag set in the status word or by receiving one of the above mentioned messages, it will call the function "**PDLM_GetQueuedChanges**". The function then returns a list of tagged values, each of which consists of its tag ID (an unsigned integer), and the actual value of the parameter. You may retrieve the name and data type code of the tagged value by a call to the function "PDLM_GetTagDescription". For more information on this subject, refer also to the demo code provided..

A great advantage of this communication type is that one does not have to query **all** variable parameter each time, which would otherwise negatively impact system performance. Instead, the generated list (of variable length) will only contain those tags corresponding to parameters that have actually changed. The host application can recognize which values have changed by either referring to the tag ID or the tag name and type.

As an added bonus, this type of communication is "future proofed". This means that if an older host application does not know about a certain tag, it can simply ignore that specific feedback entry. Conversely, a Taiko with an older firmware version will not be able to generate tags that would be introduced in newer versions. In both cases, compatibility is maintained as the Taiko and host application can still communicate usefully (albeit without access to the latest features).

2.3. Using the Taiko API DLLs under Linux

WARNING! The use of the Taiko API DLLs under Linux ist not straightforward, PicoQuant can provide only limited support and gives no warranty of success.

The Taiko API DLLs can be used under Linux via Wine, a free and open source compatibility layer that allows running software developed for Microsoft Windows™ under Linux. Providing an in-depth introduction to Wine lies outside of this manual's scope. Please refer to the official Wine User's Guide at https://wiki.winehq.org/Winelib_User's_Guide for detailed information on installing and using Wine.

2.3.1. Requirements

Supported hardware is at this time solely the "x86-64" CPU platform as found in the majority of recent PCs. Required is a PC with at least one free USB 2.0 port.

Note that Wine compatibility has been successfully tested under Linux Mint 19.2 (x86) with Wine-3.6, Ubuntu (x64) 18.04.04 with Wine-3.0, Ubuntu 20.04 (x64) with Wine-5.0, and Mint 19.3 (x64) with Wine-4.0. PicoQuant makes no warranties (implicit or otherwise) in regards to compatibility with other combinations of distributions and WINE versions.

Using the library requires `libusb` (https://libusb.info/). The formally required version is 1.0 or higher, tested versions were 1.0.19, 1.0.20, 1.0.21 and 1.0.23. Libusb is typically installed by default on all major Linux distributions.

2.3.2. Device Access Permissions

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

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

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

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

```
ATTR{idVendor}=="0d0e", ATTR{idProduct}=="0012", MODE="666"
```

A udev install script is provided on the installation medium that was delivered with your Taiko. The script is named `install` and can be found in the subfolder `Taiko_Linux`. Note that the "exectue" flag for the script needs to be set:

```
# chmod +x install
```

Note that this requires root permissions.

The name of the rules file is important: Each time a device is detected by the udev system, the files are read in alphabetical order, line by line, until a match is found. Note that different distributions may use different rule file names for various categories. For instance, Ubuntu organizes the rules into further files: `20-names.rules`, `40-permissions.rules`, and `60-symlinks.rules`. In other distributions they are not separated by those categories, as you can see by studying `50-udev.rules`. Instead of editing the existing files, it is therefore usually recommended to put all of your modifications in a separate file like 10-udev.rules or 10-local.rules. The low number at the beginning of the file name ensures it will be processed before the default file. However, later rules that are more general (applying to a whole class of devices) may later override the desired access rights. This is the case for USB devices handled through Libusb. It is therefore important that you use a rules file for the Taiko that gets evaluated after the general case. The default naming `Taiko.rules` most likely ensures this but if you see problems you may want to check.

Note that the presence of the rules file may not be sufficient to instantly access your device. It may be sufficient to re-plug the devices but it may also be necessary to instruct udev to reload the rules. Note that there are different udev implementations with different command sets. On some distributions you must reboot to activate changes, on others you can reload rule changes and restart udev with these commands:

```
# udevcontrol reload_rules
```

```
# udevstart
```

2.3.3. Using the Library and Demo Programs

Running the `install` script (see section 2.3.2) will create a folder called `API` with multiple sub-folders, including `API/Taiko_Linux`, `API/Demos` and `API/Win32`.

The `API/Win32` sub-folder contains the complete run time environment of the Windows based Taiko remote GUI (including the required DLLs files). The various demonstartion programs can be found in the sub-folder `API/Demos/<language>`, where `<language>` stands for the respective programming language. The only exception being Python, where the files are located under `API/Demos`. The required library files are also included in each sub-folder so that no file copying is required.

The library files (32 bit Windows DLLs) can be found in `API/Taiko_Linux`. When developing your own program, the two files `pquwstub.DLL` and `PDLM_Lib.DLL` need to be copied into the same folder as your project in order to run it with Wine.

3. List of API Functions

This section provides an overview of all provided API functions, their arguments (including type), important (non-trivial) return values, as well as a short description of the function. Note that importing arguments are labeled with an “I”, while exporting ones with an “O”. Bidirectional arguments has the label “B” (i.e. “I/O”). Arguments prefaced with an asterisk (*) represent pointers.

3.1. Interface Functions

Unlike most other functions, the ones described here do not require a device context (as given by the `USBIdx`). They can be successfully run even when no operational Taiko device is available.

```
/* C/C++ */ int   PDLM_GetLibraryVersion        (char       *Version
                                                 uint32_t   uiBuffLen);
```

Arguments:	*Version	O	pointer to the output string buffer
	uiBuffLen	I	maximum string buffer length for transmission
Returns:	`PDLM_ERROR_BUFFER_TOO_SMALL`		if the provided buffer is to small
Description:	Provides the version number of the currently installed library as a string. The string is formatted as follows: `<major version:1>.<minor version:1>.<target:2>.<build:4>`, where `<target>` indicates the CPU word width (either 32 or 64 bits). Please consider holding reserves of up to two additional characters for `<build>` (refer also to Appendix 6.2, “`PDLM_LIBVERSION_MAXLEN`”). To ensure compatibility with the expected reference library, make sure that the first 7 characters match.		

```
/* C/C++ */ int   PDLM_LibIsRunningInWine       (uint32_t   *IsRunningInWine);
```

Arguments:	*IsRunningInWine	I	pointer to an unsigned integer variable that returns a boolean; true, if running in a Wine environment on a POSIX system
Returns:	`PDLM_ERROR_NONE`		(always)
Description:	This function returns the boolean information whether the library is running in a Wine environment, which may be relevant for support cases. Besides this, this function is solely informative.		

```
/* C/C++ */ int   PDLM_GetUSBDriverInfo         (char       *cName
                                                 uint32_t   uiNBuffLen
                                                 char       *cVersion
                                                 uint32_t   uiVBuffLen
                                                 char       *cDate
                                                 uint32_t   uiDBuffLen);
```

Arguments:	*cName	O	pointer to a string variable for the USB driver service name
	uiNBuffLen	I	maximum string buffer length for transmission of the service name
	*cVersion	O	pointer to a string variable for the USB driver version
	uiVBuffLen	I	maximum string buffer length for transmission of the driver version
	*cDate	O	pointer to a string variable for the USB driver date
	uiDBuffLen	I	maximum string buffer length for transmission of the driver date
Returns:	`PDLM_ERROR_BUFFER_TOO_SMALL`		if any of the provided buffers is to small
Description:	Provides information on the USB driver (driver service name, driver version, and driver date)		

```
/* C/C++ */ int   PDLM_DecodeError              (int        iErrCode
                                                 char       *cBuffer
                                                 uint32_t   *uiBuffLen);
```

Arguments:	iErrCode	I	the error number
	*cBuffer	O	pointer to the output string buffer
	*uiBuffLen	B	pointer to a variable that contains the maximum string buffer length if set to 0, the length of the error text is returned in this variable but no text is returned in cBuffer

Returns: `PDLM_ERROR_UNKNOWN_ERRORCODE` if the error code is not found
`PDLM_ERROR_BUFFER_TOO_SMALL` if the provided buffer is to small

Description: Provides a human readable error string for a given error code. See also Appendix 6.2 for a list of all error codes.

```
/* C/C++ */ int   PDLM_GetTagDescription        (uint32_t   Tag
                                                 uint32_t   *TypeCode
                                                 char       *cName);
```

Arguments:	Tag	I	the tag code
	*TypeCode	O	pointer to an unsigned integer variable that returns the type code of the tag
	*cName	O	pointer to a string variable that returns the name of the tag

Returns: `PDLM_ERROR_UNKNOWN_TAG` if no tag is registered for the given tag code

Description: Gets the *TypeCode and *cName as a formal description of the requested tag. Check the tables “Table of Declared Tag Types” in Appendix 6.5 and “Table of Documented Tags” in Appendix 6.6, respectively, for a list of valid tag types.

```
/* C/C++ */ int   PDLM_DecodePulseShape         (uint32_t   shape
                                                 char       *cBuffer
                                                 uint32_t   uiBuffLen);
```

Arguments:	shape	I	the tag code
	*cBuffer	O	pointer to the output string buffer
	uiBuffLen	I	maximum string buffer length for transmission

Returns: `PDLM_ERROR_BUFFER_TOO_SMALL` if the provided buffer is to small

Description: Provides a human-readable description of a pulse shape code. These codes indicate in which range the laser head is currently operating. Valid values are:

Value	Description
0	Broadened pulse regime (due to high power settings)
1	Narrow pulse regime or “single pulse” (laser diode is operating at a pulse width corresponding to its specification)
2	Sub-threshold (or “LED domain”). No lasing occurs, only spontaneous emission
3	Unknown pulse shape

```
/* C/C++ */ int   PDLM_DecodeLHFeatures          (uint32_t   LHFeatures
                                                  char       *cBuffer
                                                  uint32_t   uiBuffLen);
```

Arguments:	shape	I	the tag code
	*cBuffer	O	pointer to the output string buffer
	uiBuffLen	I	maximum string buffer length for transmission

Returns: `PDLM_ERROR_BUFFER_TOO_SMALL` if the provided buffer is to small

Description: Turns the bit encoded feature list of a laser head into a human-readable list with each field separated by a semi-colon (;). Note that a sufficiently big buffer needs to be provided. Since the feature set varies from laser head to laser head, a recommended fixed length cannot be provided. However, a block size of 256 bytes should be large enough for all currently available laser heads (subject to change).

```
/* C/C++ */ int   PDLM_DecodeSystemStatus        (uint32_t   state
                                                  char       *cBuffer
                                                  uint32_t   uiBuffLen);
```

Arguments:	state	I	the status code to decode
	*cBuffer	O	pointer to a string variable for the decoded status
	uiBuffLen	I	maximum string buffer length for transmission

Returns: `PDLM_ERROR_BUFFER_TOO_SMALL` if the provided buffer is to small

Description: decodes the status code to a human-readable string. Note that texts corresponding to each of the status bits set are separated by a semi-colon (;)

3.2. Basic Device Functions

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, mostly unrecoverable USB communication problem (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

```
/* C/C++ */ int   PDLM_OpenDevice                (int        USBIdx
                                                  char       *cSerNo);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*cSerNo	B	pointer to a string variable with a length of at least 8 characters to hold the device's serial number

Returns:	`PDLM_ERROR_WRONG_PARAMETER`	if the USBIdx is out of range [0..7]
	`PDLM_ERROR_DEVICE_BUSY_OR_BLOCKED`	if the device is busy (i.e. opened by another program)
	`PDLM_ERROR_USB_INAPPROPRIATE_DEVICE`	if the given serial number doesn't match the devices
	`PDLM_ERROR_USB_GET_DSCR_FAILED`	if USB descriptor couldn't be loaded
	`PDLM_ERROR_USBDRIVER_NO_MEMORY`	if driver gets out of memory
	`PDLM_ERROR_DEVICE_ALREADY_OPENED`	if the software tries to re-open a device that is already opened
	`PDLM_ERROR_OPEN_DEVICE_FAILED`	if the driver couldn't get a valid windows handle
	`PDLM_ERROR_USB_UNKNOWN_DEVICE`	if device is not a Taiko

Description: Exclusively opens the device associated with the given `USBIdx`. If `cSerNo` is empty, the function returns the device's serial number (e.g., "1234567"). Otherwise, the given and device serial numbers are compared. An error is returned if they don't match. Note that `cSerNo` might undefined (empty) in case of an erroneous termination. When running in a loop, consider re-initializing cSerNo each time.

```
/* C/C++ */ int   PDLM_CloseDevice              (int        USBIdx);
```

Arguments: USBIdx I this is the USB index [valid range: 0..7]

Description: Closes the device associated with the given `USBIdx`.

```
/* C/C++ */ int   PDLM_OpenGetSerNumAndClose    (int        USBIdx
                                                 char       *cSerNo);
```

Arguments: USBIdx I this is the USB index [valid range: 0..7]
*cSerNo B pointer to a string variable with a length of at least 8 characters to hold the device's serial number

Returns:

`PDLM_ERROR_WRONG_PARAMETER`	if the USBIdx is out of range [0..7]
`PDLM_ERROR_DEVICE_BUSY_OR_BLOCKED`	if the device is busy (i.e. opened by another program)
`PDLM_ERROR_USB_INAPPROPRIATE_DEVICE`	if the given serial number doesn't match the devices
`PDLM_ERROR_USB_GET_DSCR_FAILED`	if USB descriptor couldn't be loaded
`PDLM_ERROR_USBDRIVER_NO_MEMORY`	if driver gets out of memory
`PDLM_ERROR_DEVICE_ALREADY_OPENED`	if the software tries to re-open a device that is already opened
`PDLM_ERROR_OPEN_DEVICE_FAILED`	if the driver couldn't get a valid windows handle
`PDLM_ERROR_USB_UNKNOWN_DEVICE`	if device is not a Taiko

Description: Non-exclusively opens the device associated with the given `USBIdx`. This function will return a serial number even for blocked devices). If `cSerNo` is empty, the function returns the device's serial number (e.g., "1234567"). Otherwise, the given and device serial numbers are compared. An error is returned if they don't match. Note that `cSerNo` might undefined (empty) in case of an erroneous termination. When running in a loop, consider re-initializing cSerNo each time.

```
/* C/C++ */ int   PDLM_SetExclusiveUI           (int        USBIdx
                                                 uint32_t   mode);
```

Arguments: USBIdx I this is the USB index [valid range: 0..7]
mode I the desired UI access mode (see description for valid values)

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if mode codes other than allowed are set

Description: Sets the UI access mode. if set to "`PDLM_UI_COOPERATIVE`", the user can change settings directly on the device by using the push-dial (illuminated in green) while the calling application is still running. In this mode, both the local and remote interfaces can effect changes. On the other hand, setting the mode to "`PDLM_UI_EXCLUSIVE`", will restrict the ability to effect changes to the calling application (i.e. local interface is disabled, as indicated by an unlit push-dial).

Note: Setting the UI mode to "PDLM_UI_EXCLUSIVE" is a good way to ensure that the user cannot interfere with the operation of your application (when no user interaction is intended). However, make sure that you properly release the UI after the operation completes by setting the mode to "PDLM_UI_COOPERATIVE". Not doing so might leave the Taiko locked state in the exclusive mode, requiring turning the Taiko off and on again in the worst case.

```
/* C/C++ */ int   PDLM_GetExclusiveUI          (int        USBIdx
                                                uint32_t   *mode);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*mode	O	the desired UI access mode (see description for valid values)

Description: Reads out the current UI access mode state (either "PDLM_UI_COOPERATIVE" or "PDLM_UI_EXCLUSIVE")

3.3. Device Information Functions

All of the following functions require you to identify a device by its USBIdx. These functions can also commonly return with PDLM_ERROR_USB_IOCTL_FAILED. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

```
/* C/C++ */ int   PDLM_GetUSBStrDescriptor     (int        USBIdx
                                                char       *Descr
                                                uint32_t   uiBuffLen);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*Descr	O	pointer to a string variable for the string descriptor
	uiBuffLen	I	maximum string buffer length to transmit

Returns:	PDLM_ERROR_BUFFER_TOO_SMALL	if the provided buffer is to small
	PDLM_ERROR_USB_GET_DSCR_FAILED	if USB descriptor couldn't be loaded

Description: Returns a string with concatenated USB string descriptors for the device associated to the USBIdx, separated by a semi-colon (;).

```
/* C/C++ */ int   PDLM_GetHardwareInfo         (int        USBIdx
                                                char       *Infos
                                                uint32_t   uiBuffLen);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*Infos	O	pointer to a string variable for the hardware information
	uiBuffLen	I	maximum string buffer length to transmit

Returns:	PDLM_ERROR_BUFFER_TOO_SMALL	if the provided buffer is to small

Description: Returns a string containing the hardware info, as usually shown in the About box/Support info texts. This information mainly identifies the hardware product type and version.

```
/* C/C++ */ int   PDLM_CreateSupportRequestText (int        USBIdx
                                                 char       *cPreamble
                                                 char       *cCallingSW
                                                 uint32_t   uiOptions
                                                 uint32_t   uiBuffLen
                                                 char       *cBuffer);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*cPreamble	I	pointer to a string for the preamble text to include (e.g., open the about box and refer to all text before "<snip>")
	*cCallingSW	I	pointer to a string to identify the calling software to include (e.g., open the About box and refer to the paragraph "Calling Software")
	uiOptions	I	bitset of options, that the caller can choose and combine from (see description for values)
	uiBuffLen	I	the maximum string buffer length to transmit
	*cBuffer	O	pointer to a string variable to take the SupportRequestText

Returns: `PDLM_ERROR_BUFFER_TOO_SMALL` if the provided buffer is to small

Description: Generates a string containing common hardware, software, and environment information, as can be usually found in the “About...” or the support information boxes. This sting contains all relevant information about the device associated to the given `USBIdx` (including version numbers, environment, feature and option lists). The output string can be customized via the `uiOptions` bitset:

Name	Bit Value	Effect
`PDLM_SUPREQ_OPT_NO_PREAMBLE`	`0x01`	if included, the preamble will be suppressed
`PDLM_SUPREQ_OPT_NO_TITLE`	`0x02`	if included, the title will be suppressed
`PDLM_SUPREQ_OPT_NO_CALLING_SW_INDENT`	`0x04`	if included, the info on calling software will not be indented
`PDLM_SUPREQ_OPT_NO_SYSTEM_INFO`	`0x08`	if included, the system info will be suppressed

```
/* C/C++ */ int   PDLM_GetFWVersion          (int        USBIdx
                                              char       *Version
                                              uint32_t   uiBuffLen);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*Version	O	pointer to a string variable for the firmware version
	uiBuffLen	I	the maximum string buffer length to transmit

Returns: `PDLM_ERROR_BUFFER_TOO_SMALL` if the provided buffer is to small

Description: Reads out the firmware version of the device associated with the given `USBIdx`.

```
/* C/C++ */ int   PDLM_GetFPGAVersion        (int        USBIdx
                                              char       *Version
                                              uint32_t   uiBuffLen);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*Version	O	pointer to a string variable for the firmware version
	uiBuffLen	I	the maximum string buffer length to transmit

Returns: `PDLM_ERROR_BUFFER_TOO_SMALL` if the provided buffer is to small

Description: Reads out the firmware version of the device associated with the given `USBIdx`.

```
/* C/C++ */ int   PDLM_GetDeviceData         (int         USBIdx
                                              TDeviceData *Data
                                              uint32_t    size);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*Data	O	packed structure containing device information (see description for detailed list)
	size	I	size of the structure

Description: This function retruns some of the information that is also generated by the function "PDLM_CreateSupportRequestText", but as not yet formatted, raw data in a packed structure. Use this function if you want to build up your own set of information on the device in use.T The structure contains the following information:

Type	Name	Description
uint32_t	SN	Serial number
uint32_t	ArtNo	Article number
char	Name[PDLM_DEV_STRING_LENGTH]	Product name (e.g., "Taiko")
char	Type[PDLM_DEV_STRING_LENGTH]	Product type (e.g., "PDL M1")
char	Date[PDLM_DEV_STRING_LENGTH]	Date of manufacturing
char	VersPCB[PDLM_DEV_STRING_LENGTH]	PCB version number (e.g., "078.2005.0104")
_TVersNum	VersDev	Version numbers for the device
uint16_t	Major	Major firmware version number
uint16_t	Minor	Minor firmware version number
char	Notes[PDL_DEV_STRING_LENGTH]	Version notes (e.g., "beat", "pre-release", "release")

3.4. Laser Head Information Functions

All of the following functions require you to identify a device by its USBIdx. These functions can also commonly return with PDLM_ERROR_USB_IOCTL_FAILED. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device. Furthermore, functions in this group all refer to a plugged-in laser head. A missing head will result in an error return code.

```
/* C/C++ */ int   PDLM_GetLHVersion          (int        USBIdx
                                              char       *Version
                                              uint32_t   uiBuffLen);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*Version	O	pointer to a string variable for the laser head version
uiBuffLen	I	the maximum string buffer length to transmit

Returns: PDLM_ERROR_BUFFER_TOO_SMALL if the provided buffer is too small

Description: This function provides the version number (as a string) of the laser head connected to the device associated with the given USBIdx.

```
/* C/C++ */ int   PDLM_GetLHData             (int        USBIdx
                                              TLaserData *pData
                                              uint32_t   size);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*Data	O	packed structure containing laser head information (see description for detailed list)
size	I	size of the structure

Description: Reads out detailed information about the laser head currently connected to the device associated with the given USBIdx as a packed structure. The structure contains the following information:

Type	Name	Description
uint32_t	SN	Serial number, as unsigned integer (not a string)
uint32_t	Features	Bitset summarizing the laser head features. Use "PDLM_DecodeLHFeatures" to decode.
uint32_t	FreqMin	Minimum frequency for the laser head (in Hz). Same information can be obtained via the function "PDLM_GetFrequencyLimits"
uint32_t	FreqMax	Maximum frequency for the laser head (in Hz). Same information can be obtained via the function "PDLM_GetFrequencyLimits"
uint32_t	CwPowerMax	Maximum power for CW mode in µW (only for information purposes; **do not use in calculations**!)
uint32_t	PulsePowerMax	Maximum power for pulsed mode in µW (only for information purposes; **do not use in calculations**!)
uint16_t	WavelengthNominal	Nominal laser head wavelength (in 1/10 nm)
uint16_t	CaseTempMax (: 10)	Maximum case temperature (in 1/10 °C) **Note:** shares the same uint16_t with the next two named entries (bit masked)
uint16_t	Protection (: 1)	Laser protection classification: 0 → class 3; 1 → class 4
uint16_t	CwCurrentPolarity (: 1)	Indicates current polarity in CW mode: 0 → positive, 1 → negative **Note:** not used for the Taiko PDL M1, reserved for future use
uint16_t	(: 4)	Reserved for future use **Note:** shares the same uint16_t with the previous three entries (bit masked)
uint16_t	LHTypeCtrlVoltage (: 12)	Maximum driver voltage for this laser head in cV (1 cV = 10 mV)
uint16_t	(: 4)	Reserved for future use **Note:** shares the same uint16_t with the previous entry (bit masked)
uint16_t	LHMaxVoltage (:12)	Maximum voltage for this individual laser head diode in cV (1 cV = 10 mV)
uint16_t	(: 4)	Reserved for future use **Note:** shares the same uint16_t with the previous entry (bit masked)
uint16_t	CurrentTEP12V	Current consumption of TEP 12V power supply in mA. **Note:** currently not used for the Taiko PDL M1, reserved for future devices
uint16_t	laserType	Identification label for the Taiko laser head type (see Appendix 6.9 for a list of values)
TLHVersNumu int16_t	laserVersion	Structure of two uint16_t values, named major and minor. Can be directly accessed by the function "PDLM_GetLHVersion"
uint16_t	calibratedWarrantHours	Duration of the guaranteed validity of the laser heads calibration data (in hours)

```
/* C/C++ */ int   PDLM_GetLHInfo          (int                    USBIdx
                                           TLaserInfo             *pInfo
                                           uint32_t               size);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*pInfo	O	packed structure containing laser head information in text form (see description for detailed list)
size	I	size of the structure

Description: This function provides additional information (as text strings in a packed structure) about the laser head connected to the device associated with the given `USBIdx`. The packed structure contains the following data:

Type	Name	Description
char	LType[PDLM_LDH_STRING_LENGTH]	Designation of laser head (e.g., “LDH-IX-B-405”)
char	date[PDLM_LDH_STRING_LENGTH]	Date of manufacturing (format: yyyy-mm-dd)
char	LClass[PDL_DEV_STRING_LENGTH]	Laser Class that is applicable to this head (e.g., “3R”)

```
/* C/C++ */ int   PDLM_GetLHFeatures          (int        USBIdx
                                               uint32_t   *LHFeatures);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*LHFeatures	O	all laser head features as a bit encoded `uint32_t` value

Description: This function generates a bit encoded `uint32_t` value that contains all laser head features. Checking that the connected head supports a specific feature can be done through masking with a bit wise AND operation. For example: to check if the laser head supports “burst mode”, you could run the following IF statement in C:

```
if ((*LHFeatures & PDLM_LHFEATURE_BURST_CAPABILITY) > 0) { ... }
```

Determining the type of installed intensity sensor could be done as follows:

```
iType = ((*LHFeatures & PDLM_LHFEATURE_INTENSITY_SENSOR_TYPE >> 24);
```

A list of valid `PDLM_LHFEATURE_` values is given in Appendix 6.8.

3.5. Status and Error Information Functions

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

```
/* C/C++ */ int   PDLM_SetHWND            (int        USBIdx
                                            HWND       hwnd);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
hwnd	I	the handle of the calling application’s main window

Returns:

`PDLM_ERROR_USB_REGNTFY_FAILED`	if registration fails
`PDLM_ERROR_USB_INVALID_HANDLE`	if no valid USB handle could be obtained

Description: Transmits the handle of the message-loop-holding window (generally the main window of the calling application) to the DLL. This enables the DLL to asynchronously post messages with device feedback to the host software via windows messages.

The class of this window should implement and register handlers for messages that are posted as notifications on several occasions. Sent notifications usually contain this window handle, a message ID identifying the responsible event handler, a `WPARAM`-typed short parameter named `wParam`, and a `LPARAM`-typed long parameter named `lParam`.

When the DLL posts one of the following notification messages, the `USBIdx` is included in the `wParam`, while the `lParam` transmits the current status word (as `uint32_t`). Nine different messages (all of type notification) are defined, see table below. Note that `WM_PDLM_BASE` has always a value of `0x1200`.

Name	Value	Status change trigger
`WM_ON_PENDING_ERRORS`	`WM_PDLM_BASE + 0x01`	On 0 →1 `PDLM_DEVSTATE_ERRORMSG_PENDING`
`WM_ON_LOCKING_CHANGE`	`WM_PDLM_BASE + 0x02`	On any `PDLM_DEVSTATEMASK_LOCKED`
`WM_ON_LASERHEAD_CHANGE`	`WM_PDLM_BASE + 0x03`	On 0 → 1 `PDLM_DEVSTATE_LASERHEAD_CHANGED`
`WM_ON_LASER_NOT_OPERATIONAL_CHANGE`	`WM_PDLM_BASE + 0x04`	On 0 → 1 or 1 → 0 `PDLM_DEVSTATEMASK_LASER_NOT_OPERATIONAL`
`WM_ON_DEVICE_NOT_OPERATIONAL_CHANGE`	`WM_PDLM_BASE + 0x05`	On 0 → 1 or 1 → 0 `PDLM_DEVSTATEMASK_DEVICE_NOT_OPERATIONAL`
`WM_ON_PARAMETER_CHANGE`	`WM_PDLM_BASE + 0x07`	On any `PDLM_DEVSTATE_PARAMETER_CHANGES_PENDING`
`WM_ON_EXCLUSIVE_UI_CHANGE`	`WM_PDLM_BASE + 0x08`	On any `PDLM_DEVSTATE_EXCLUSIVE_SW_OP_GRANTED`
`WM_ON_WARNINGS_CHANGE`	`WM_PDLM_BASE + 0x09`	On any `PDLM_DEVSTATEMASK_ALL_WARNINGS`
`WM_ON_OTHER_STATES_CHANGE`	`WM_PDLM_BASE + 0xFF`	On any other status changes

```
/* C/C++ */ int   PDLM_GetSystemStatus          (int        USBIdx
                                                 uint32_t   *mode);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*state	O	pointer to an unsigned integer variable for the status code

Description: This function reads out the state code (bit-coded). Refer to the "Table of all assigned status bits" in the Appendix 6.3 and 6.4 for a list of useful status bit masks.

```
/* C/C++ */ int   PDLM_GetQueuedChanges         (int        USBIdx
                                                 TTagValue  *TagValList
                                                 uint32_t   *uiListLen);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*TagValList	O	pointer to a variable, holding a tag value list (array)
*uiListLen	B	pointer to an unsigned integer variable to enter the length of the tag value list provided (max. number of elements). Upon return, the number of transferred elements is accessable.

Returns:

`PDLM_ERROR_BUFFER_TOO_SMALL`	if the provided buffer is too small
`PDLM_ERROR_DLL_MEMORY_ALLOCATION_ERROR`	if a memory allocation error occurred

Description: This function returns a list of all queued changes as an array (of `TTagValue` type) containing the respective tags as well as their associated values.

Refer to the function "**`PDLM_GetTagValueList`**" for more details.

```
/* C/C++ */ int   PDLM_GetTagValueList          (int        USBIdx
                                                 uint32_t   uiListLen
                                                 PTagValue  pTagValList);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	uiListLen	I	Number of elements to retrieve from in the list of tagged values
	pTagValList	B	pointer to an array of TTagValue typed fields. initialize the fields with the tags of the values to be retrieved and get the desired values after return.

Returns:	
`PDLM_ERROR_BUFFER_TOO_SMALL`	if the provided buffer is too small
`PDLM_ERROR_DLL_MEMORY_ALLOCATION_ERROR`	if a memory allocation error occurred

Description: This function takes a pointer to an array of `TTagValue` typed fields, initialized with the tags of the desired values as input templates and returns it filled with the current values associated with the tags.

`TTagValue` contains a field "Value" of `TValueType`, which is a union of various typed fields. To interpret such a value, use the function "`PDLM_GetTagDescription`", which will provide you with information on both base type and scaling of the value.

For example, if you work in pulsed mode and want to query the currently emitted optical power, you could insert a template initialized with the tag "`PDLM_TAG_PulsePower`", "`PDLM_TAG_PulsePowerNanowatt`" or "`PDLM_TAG_PulsePowerPermille`", depending on desired kind of visualization and further processing. "`PDLM_TAG_PulsePower`" returns the value as a float, scaled in Watt (W), while "`PDLM_TAG_PulsePowerNanowatt`" will return it as an unsigned integer, scaled in nW. Using "`PDLM_TAG_PulsePowerPermille`" will also yield an unsigned integer, but scaled in per mil (in relation to the maximum power).

```
/* C/C++ */ int   PDLM_GetQueuedError           (int        USBIdx
                                                 int        *ErrCode);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	* ErrCode	O	pointer to an integer variable, returning the deepest error code

Description: If an error situation was not directly produced by a call to a function (e.g., laser head overheating), the situation is registered and an error code is queued. To signal that new elements are the queue, the most significant bit is set in the status code ("`PDLM_STATE_ERROR_MESSAGE_PENDING`"). With this bit set, the user can get the queued codes by executing one or multiple calls to this function. Each call returns the deepest code (FIFO), until the queue is purged. Once the most recent error code element is retreived, the signaling status flag is reset to 0.

```
/* C/C++ */ int   PDLM_GetQueuedErrorString     (int        USBIdx
                                                 int        ErrCode
                                                 char       *ErrText);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	ErrCode	I	the error code, that has to be decoded
	*ErrText	O	pointer to the output string buffer

Description: Decodes the give error code into a human-readable text string. Make sure that the `ErrText` buffer can accommodate a number of characters that is at least equal to `PDLM_HW_ERRORSTRING_MAXLEN+1`

3.6. Laser Locking Functions

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

```
/* C/C++ */ int    PDLM_GetLocked              (int        USBIdx
                                                uint32_t   *Locked);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	*Locked	O	pointer to an unsigned integer variable to return the locking state

Description: Returns the over-all locking state of the Taiko: `PDLM_LASER_UNLOCKED` (0) or `PDLM_LASER_LOCKED` (1). Further information on why the laser is locked can be obtained by inspecting the status code.

```
/* C/C++ */ int    PDLM_SetSoftLock            (int        USBIdx
                                                uint32_t   SoftLocked);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	SoftLocked	I	the desired soft locking state (boolean)

Description: Sets the soft locking state of the Taiko. Valid values are `PDLM_LASER_UNLOCKED` (0) or `PDLM_LASER_LOCKED` (1)

```
/* C/C++ */ int    PDLM_GetSoftLock            (int        USBIdx
                                                uint32_t   *SoftLocked);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	SoftLocked	O	pointer to an unsigned integer variable to return the soft locking state

Description: Returns the current soft locking state of the Taiko. Note that even if the value returned equals PDLM_LASER_UNLOCKED, the Taiko might be locked for other reasons. In such a case, make sure to inspect the status code to find out more.

3.7. Laser Emission Mode Functions

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

```
/* C/C++ */ int    PDLM_SetLaserMode           (int        USBIdx
                                                uint32_t   mode);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	mode	I	the desired laser emission mode (see description for valid values)

Returns:		
	`PDLM_ERROR_FEATURE_NOT_AVAILABLE`	if the mode is not allowed for this type of laser
	`PDLM_ERROR_ILLEGAL_VALUE`	if mode codes other than mentioned below are set or if the mode is not allowed in current trigger mode

Description: This function sets the laser emission mode. Depending on the connected laser head type, some modes will not be available. Please note that laser heads cannot be switched to burst mode as long as the device is triggered externally.

Note: after switching the laser emission mode, several other values will also be automatically changed. These include the optical output power, which is set to the latest valid settings. The Taiko always enforces safe operating values by applying the limits stored in the connected laser head. For example, optical power in CW mode may be not suited or even totally out of bounds for pulsed mode operation.

List of valid laser mode codes:

Mode	Value	Note
PDLM_LASER_MODE_CW	0x00000000	Continuous wave (CW) mode
PDLM_LASER_MODE_PULSE	0x00000001	Pulsed mode
PDLM_LASER_MODE_BURST	0x00000002	Burst mode

```
/* C/C++ */ int   PDLM_GetLaserMode          (int        USBIdx
                                              uint32_t   *mode);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*mode	O	pointer to an unsigned integer variable that returns the current laser emission mode

Description: Use this function to query the current laser emission mode (for valid return values, see the table in the description of "PDLM_SetLaserMode".

```
/* C/C++ */ int   PDLM_SetLDHPulsePowerTable  (int        USBIdx
                                              uint32_t   TableIdx);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
TableIdx	I	code (index) of the desired pulse power table (see description for valid values

Returns:

PDLM_ERROR_FEATURE_NOT_AVAILABLE	if TableIdx is equal to 1 and the connected laser head does not support max. power mode
PDLM_ERROR_ILLEGAL_VALUE	if TableIdx is larger than 1

Description: This function sets the current pulse power table code. After changing the pulse power table, you should perform a call to the "PDLM_GetPulsePowerLimits" function to get the power range for the current frequency.

Note: after changing the pulse power table, the current power is set to zero for safety reasons.

List of valid TableIdx codes:

TableIdx	Value	Description
PDLM_LDH_LINEAR_PULSE_TABLE	0	Linear power mode
PDLM_LDH_MAX_POWER_PULSE_TABLE	1	Max. power mode

```
/* C/C++ */ int   PDLM_GetLDHPulsePowerTable  (int        USBIdx
                                              uint32_t   *TableIdx);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
TableIdx	O	pointer to an unsigned integer variable returning the code (index) of the currently set pulse power table

Description: Use this function to query the current pulse power table code. If this value changed, you should perform a call to the `PDLM_GetPulsePowerLimits` function to get the power range for the current frequency.

3.8. Triggering and Gating Functions

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

```
/* C/C++ */ int    PDLM_SetTriggerMode          (int        USBIdx
                                                 uint32_t   mode);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
mode	I	the code of the desired trigger mode (see description for valid values)

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if mode has a value other than the allowed ones

Description: Sets the code of the desired trigger mode of the device. If trigger mode is set to one of the external modes, the user should also set appropriate values for the trigger level

Note: when switching from internal to external triggering, the context of "intensity" changes. Since the Taiko doesn't "know" the characteristics of the external trigger signal, it can't go on using a table-driven power linearization at a given frequency. Instead, the "intensity" control switches over to controlling the diode voltage (internally called "PosVar"). The new range goes from the lowest minimum voltage value recorded in the power tables of all frequencies up to the maximum voltage allowed. This allows using a wider part of the lower range of the per mil scale for driving the laser head in the sub-threshold (LED) domain than with internally triggered pulses. The maximum, 1000 per mil, corresponds to a voltage that is set individually for each head and is carefully chosen to prevent damage to it.

Table of valid trigger mode values:

Mode	Value	Device is...
`PDLM_TRIGGER_INTERNAL`	`0x00000000`	triggered internally
`PDLM_TRIGGER_EXTERNAL_FALLING_EDGE`	`0x00000001`	triggered externally on falling edge
`PDLM_TRIGGER_EXTERNAL_RISING_EDGE`	`0x00000002`	triggered externally on rising edge

```
/* C/C++ */ int    PDLM_GetTriggerMode          (int        USBIdx
                                                 uint32_t   *mode);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*mode	O	pointer to an unsigned integer variable, returning the code of the current trigger mode

Description: Reads out the current trigger mode of the device associated with the given `USBIdx`.

```
/* C/C++ */ int    PDLM_GetTriggerLevelLimits   (int        USBIdx
                                                 float      *MinLevel
                                                 float      *MaxLevel);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*MinLevel	O	pointer to a float variable (single precision), returning the device's trigger level lower limit in Volt
*MaxLevel	O	pointer to a float variable (single precision), returning the device's trigger level upper limit in Volt

Description: This function reads out the lower and upper level limits of the external trigger signal in Volt.

```
/* C/C++ */ int   PDLM_SetTriggerLevel          (int        USBIdx
                                                 float      Level);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	Level	I	the device's desired trigger level in Volt

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if Level is outside of the limits

Description: Sets the external trigger level in Volt

```
/* C/C++ */ int   PDLM_GetTriggerLevel          (int        USBIdx
                                                 float      *Level);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	*Level	O	pointer to a float variable (single precision), returning the device's current trigger level in Volt

Description: Reads out the current external trigger level in Volt.

```
/* C/C++ */ int   PDLM_GetExtTriggerFrequency   (int        USBIdx
                                                 uint32_t   *ExtFreq);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	*ExtFreq	O	pointer to an unsigned integer variable, returning the device's current external trigger frequency in Hertz

Description: Gets the external trigger frequency in Hz. Note that a call to this function is not intended to replace an actual frequency measurement. The value returned by this function provides a rough idea (i.e. order of magnitude) of the external trigger signal's frequency. The base resolution of the implemented counter is 80 Hz, so that is strongly recommended to trust only readings, that are significantly higher than 8 kHz.

```
/* C/C++ */ int   PDLM_SetFastGate              (int        USBIdx
                                                 uint32_t   mode);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	mode	I	the code of the desired fast gate mode (boolean)

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if mode has a value other than the two allowed ones

Description: sets the code of the desired fast gate mode of the device. Valid values are `PDLM_DISABLE` (0) and `PDLM_ENABLE` (1). If the fast gate mode is set to enabled, you should also set the appropriate impedance for the fast gate.

```
/* C/C++ */ int   PDLM_GetFastGate              (int        USBIdx
                                                 uint32_t   *mode);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	*mode	O	pointer to an unsigned integer variable, returning the code of the current fast gate mode

Description: Reads out the code of the fast gate mode of the device with the associated `USBIdx`. Valid (boolean) values are: enabled (1) and disabled (0).

```
/* C/C++ */ int   PDLM_SetFastGateImp          (int        USBIdx
                                                uint32_t   mode);
```

Arguments: USBIdx I this is the USB index [valid range: 0..7]
mode I the code of the desired fast gate impedance (see description for valid values)

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if mode has a value other than the two allowed ones

Description: Use this function to set the desired impedance for the fast gate input via the appropriate code. Table with valid impedance codes:

Code	Value	Impedance
`PDLM_GATEIMP_10k_OHM`	`0x00000000`	High gating impedance (10k Ω)
`PDLM_GATEIMP_50_OHM`	`0x00000001`	Low gating impedance (50 Ω)

```
/* C/C++ */ int   PDLM_GetFastGateImp          (int        USBIdx
                                                uint32_t   *mode);
```

Arguments: USBIdx I this is the USB index [valid range: 0..7]
*mode O pointer to an unsigned integer variable, returning the code of the current fast gate impedance

Description: Returns the code for the fast gate input impedance currently set for the device associated with the given `USBIdx`.

```
/* C/C++ */ int   PDLM_SetSlowGate             (int        USBIdx
                                                uint32_t   mode);
```

Arguments: USBIdx I this is the USB index [valid range: 0..7]
mode I the code of the desired slow gate mode (boolean)

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if mode has a value other than the two allowed ones

Description: This function sets the slow gate mode of the device associated with the given `USBIdx` depending on the provides code. Valid (boolean) codes are: enabled (1) and disabled (0).

```
/* C/C++ */ int   PDLM_GetSlowGate             (int        USBIdx
                                                uint32_t   *mode);
```

Arguments: USBIdx I this is the USB index [valid range: 0..7]
*mode O pointer to an unsigned integer variable, returning the code of the current slow gate mode

Description: Reads out the code of the slow gate mode of the device associated with the given `USBIdx`. Valid (boolean) values are: enabled (1) and disabled (0).

3.9. Pulse Frequency and Burst Setting Functions

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

```
/* C/C++ */ int   PDLM_GetFrequencyLimits      (int        USBIdx
                                                uint32_t   *MinFreq
                                                uint32_t   *MaxFreq);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*MinFreq	O	pointer to an unsigned integer variable, returning the device's lower frequency limit in Hz
*MaxFreq	O	pointer to an unsigned integer variable, returning the device's upper frequency limit frequency in Hz

Description: Calling this function returns the lower and upper pulse frequency limits of the laser head currently connected to the device associated with the given `USBIdx`.

Note: These frequency limits vary from laser head to laser head. It is recommended to call this function every time a laser head change has occurred.

```
/* C/C++ */ int   PDLM_SetFrequency            (int        USBIdx
                                                uint32_t   freq);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
freq	I	the device's desired base oscillator frequency in Hz

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if freq is outside of the limits

Description: Use this function to set the base oscillator frequency (in Hz) of the device associated with the given `USBIdx`. The pulse frequency of both pulsed and burst modes depend on this value.

```
/* C/C++ */ int   PDLM_GetFrequency            (int        USBIdx
                                                uint32_t   *freq);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*freq	O	pointer to an unsigned integer variable, returning the device's base oscillator frequency in Hz

Description: Reads out the current base oscillator frequency (in Hz) of the device associated with the given `USBIdx`. The pulse frequency of both pulsed and burst modes depend on this value.

```
/* C/C++ */ int   PDLM_SetBurst                (int        USBIdx
                                                uint32_t   BurstLength
                                                uint32_t   PeriodLength);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
BurstLength	I	the desired burst length, in pulses (see description for valid range)
PeriodLength	I	the desired period length, in pulses (see description for valid range)

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if `BurstLength` or `PeriodLength` is outside of the limits

Description: Use this function to write the burst definition. Note that the `BurstLength` and `PeriodLength` are defined in pulses. Limits are defined as follows:

```
                2   <=  BurstLength   <   (2^24 - 1) = 16777215

(BurstLength + 1)   <=  PeriodLength  <=   16777215
```

Although patterns with very long pulse pauses (long period length) can be defined, you should look for alternative ways to implement the desired behavior, such as using external triggering or gating. It can be hard to determine during a measurement whether the laser driver is still working in a valid burst cycle or just shut off due to an error.

```
/* C/C++ */ int   PDLM_GetBurst                (int        USBIdx
                                                uint32_t   *BurstLength
                                                uint32_t   *PeriodLength);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*BurstLength	O	pointer to an unsigned integer variable, returning the device's burst length in pulses
	*PeriodLength	O	pointer to an unsigned integer variable, returning the device's period length in pulses

Description: Reads out the currently set burst definition (consisting of the two variables `BurstLength` and `PeriodLength`.

3.10. Functions for Temperature Settings

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

Note that many of these functions can be called with an arbitrary temperature scale ID code, that does not need to match the currently set ScaleID of the device's GUI. Keep in mind that all internal calculations and settings are performed in °C (regardless of ScaleID setting). Rounding is done to tenths of a degree Celsius. This might lead to some strange stepping and rounding effects when displaying the laser head temperature in Fahrenheit units.

```
/* C/C++ */ int   PDLM_SetTempScale            (int        USBIdx
                                                uint32_t   ScaleID);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	ScaleID	I	the code of the desired temperature scale (see description for valid values)

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if ScaleID has a value other than the allowed ones

Description: This function sets the code for the temperature scale as currently used in the GUI of the device associated with the given `UBSIdx`. Three ScaleID values are supported:

Code	Name	Notes
0	`PDLM_TEMPERATURESCALE_CELSIUS`	Displays temperature in °C
1	`PDLM_TEMPERATURESCALE_FAHRENHEIT`	Displays temperature in °F
2	`PDLM_TEMPERATURESCALE_KELVIN`	Displays temperature in K

```
/* C/C++ */ int   PDLM_GetTempScale            (int        USBIdx
                                                uint32_t   *ScaleID);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	* ScaleID	O	pointer to an unsigned integer variable returning the code of the temperature scale currently set

Description: Reads out the current temperature scale code (see the description of the function "**`PDLM_SetTempScale`**" for a list of values).

```
/* C/C++ */ int   PDLM_GetLHTargetTempLimits   (int        USBIdx
                                                uint32_t   ScaleID
                                                float      *MinTemp
                                                float      *MaxTemp);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	ScaleID	I	the code of the desired temperature scale (see description of "PDLM_SetTempScale" for a list of valid values)
	*MinTemp	O	pointer to a float variable returning the laser diode's target temperature lower limit, in units of the desired temperature scale.
	*MaxTemp	O	pointer to a float variable returning the laser diode's target temperature upper limit, in units of the desired temperature scale.

Returns: PDLM_ERROR_ILLEGAL_VALUE if ScaleID has a value other than the allowed ones

Description: Reads out the minimum and maximum temperature limits stored in the laser head that is connected to the device associated with the given USBIdx. The returned values are in units corresponding to the given ScaleID.

```
/* C/C++ */ int   PDLM_SetLHTargetTemp          (int        USBIdx
                                                 uint32_t   ScaleID
                                                 float      TargTemp);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	ScaleID	I	the code of the desired temperature scale (see description of "PDLM_SetTempScale" for a list of valid values)
	TargTemp	I	the laser diode's desired target temperature in units of the desired temperature scale.

Returns: PDLM_ERROR_ILLEGAL_VALUE if ScaleID has a value other than the allowed ones

Description: Use this function to change the set-value for the temperature (in units corresponding to the chosen ScaleID) of the laser diode connected to the device associated with the given USBIdx.

```
/* C/C++ */ int   PDLM_GetLHTargetTemp          (int        USBIdx
                                                 uint32_t   ScaleID
                                                 float      *TargTemp);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	ScaleID	I	the code of the desired temperature scale (see description of "PDLM_SetTempScale" for a list of valid values)
	*TargTemp	O	pointer to a float variable returning the laser diode's target temperature as currently set, in units of the desired temperature scale.

Returns: PDLM_ERROR_ILLEGAL_VALUE if ScaleID has a value other than the allowed ones

Description: Reads out the current set-value for the temperature (in units corresponding to the chosen ScaleID) of the laser diode connected to the device associated with the given USBIdx.

```
/* C/C++ */ int   PDLM_GetLHCurrentTemp         (int        USBIdx
                                                 uint32_t   ScaleID
                                                 float      *CurrTemp);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	ScaleID	I	the code of the desired temperature scale (see description of "PDLM_SetTempScale" for a list of valid values)
	*CurrTemp	O	pointer to a float variable returning the laser diode temperature as currently measured, in units of the desired temperature scale.

Returns: PDLM_ERROR_ILLEGAL_VALUE if ScaleID has a value other than the allowed ones

Description: Reads out the current temperature (in units corresponding to the chosen ScaleID) of the laser diode connected to the device associated with the given USBIdx.

```
/* C/C++ */ int   PDLM_GetLHCaseTemp          (int        USBIdx
                                               uint32_t   ScaleID
                                               float      *CaseTemp);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	ScaleID	I	the code of the desired temperature scale (see description of "PDLM_SetTempScale" for a list of valid values)
	*CaseTemp	O	pointer to a float variable returning the case temperature as currently set, in units of the desired temperature scale.

Returns: PDLM_ERROR_ILLEGAL_VALUE if ScaleID has a value other than the allowed ones

Description: Reads out the current temperature (in units corresponding to the chosen ScaleID) of the laser diode housing.

```
/* C/C++ */ int   PDLM_GetLHWavelength        (int        USBIdx
                                               float      *Wavelength);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*Wavelength	O	pointer to a float variable (single precision), returning the laser head's wavelength in nm

Description: Reads out the estimated temperature-shifted wavelength of the laser diode. Please note that this is only an estimation based on the data obtained when the laser head was calibrated. It is not the result of a direct wavelength measurement at the function call time.

Note: Wavelength tuning is an indirect result of temperature setting and requires the laser head to support the feature "PDLM_LHFEATURE_WL_TUNABLE". Therefore, only a "get" function is available.

3.11. Laser Head Functions for Pulse Power Settings

All of the following functions require you to identify a device by its USBIdx. These functions can also commonly return with PDLM_ERROR_USB_IOCTL_FAILED. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

The functions described in this section are setting or reading the optical output power in pulsed or burst modes for the connected laser head (in various units of W or permille of the current maximum). Obviously, the lower and upper limits (fMinPower and fMaxPower) will differ from one laser head to the next as well as with any change of the LDH power table in use(see PDLM_SetLDHPulsePowerTable) or repetition rate. It is therefore recommended to always read out the limits after a new head has been connected or after changing the power table index or the repetition rate.

Note: Even though it is tempting to always use absolute W values for power settings, this might cause some issues. For a given laser head, a power setting of 0.15 W might be fine, while this value might be too high for another. It is therefore recommended to use "per mille" settings (in relation to fMaxPower).

```
/* C/C++ */ int   PDLM_GetPulsePowerLimits    (int        USBIdx
                                               float      *fMinPower
                                               float      *fMaxPower);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*fMinPower	O	pointer to a float variable returning the laser diode's lower power limit (pulsed mode), in W.
	*fMaxPower	O	pointer to a float variable returning the laser diode's upper power limit (pulsed mode), in W.

Description: Reads out the minimum and maximum power limits in pulsed mode (in units of W) of the laser diode currently connected to the device associated with the given USBIdx.

```
/* C/C++ */ int   PDLM_SetPulsePower          (int        USBIdx
                                               float      fPower);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	fPower	I	the desired optical output power (pulsed mode), in W

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if fPower lies outside of the head's valid range

Description: Sets the desired (in W, pulsed mode) optical output power of the laser diode connected to the device associated with the given `USBIdx`.

```
/* C/C++ */ int   PDLM_GetPulsePower          (int        USBIdx
                                               float      *fPower);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*fPower	O	pointer to a float variable (single precision), returning the laser optical output power (pulsed mode) in W

Description: Reads out the currently set optical output power (in W, pulsed mode) for the laser head connected to the device associated with the given `USBIdx`.

```
/* C/C++ */ int   PDLM_SetPulsePowerPermille  (int        USBIdx
                                               uint32_t   permille);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	permille	I	the desired optical output power (pulsed mode) setting based on a per mille (1/1000) setting of fMaxPower

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if permille lies outside of the head's valid range

Description: Sets the desired (in per mille of maximum power, pulsed mode) optical output power of the laser diode connected to the device associated with the given `USBIdx`. Valid range for a per mille is 0 to 1000, logically.

```
/* C/C++ */ int   PDLM_GetPulsePowerPermille  (int        USBIdx
                                               uint32_t   *permille);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*permille	O	pointer to an unsigned integer variable, returning the laser doide's power setting (in pulsed mode) as a per mille of fMaxPower

Description: Reads out the currently set optical output power (as a per mille of `fMaxPower`, pulsed mode) for the laser head connected to the device associated with the given `USBIdx`.

```
/* C/C++ */ int   PDLM_SetPulsePowerNanowatt  (int        USBIdx
                                               uint32_t   nanowatt);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	nanowatt	I	the desired optical output power (pulsed mode) setting in nW

Returns: `PDLM_ERROR_ILLEGAL_VALUE` if nanowatt lies outside of the head's valid range

Description: Sets the desired (in nW, pulsed mode) optical output power of the laser diode connected to the device associated with the given `USBIdx`. Note: use this function only if there is a strong demand for integer arithmetics. Consider the rounding of the float value in W to whole nW values.

```
/* C/C++ */ int   PDLM_GetPulsePowerNanowatt  (int        USBIdx
```

```
                                                    uint32_t   *nanowatt);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*nanowatt	O	pointer to an unsigned integer variable, returning the laser doide's power setting (pulsed mode) in nW

Description: Reads out the currently set optical output power (in nW, pulsed mode) for the laser head connected to the device associated with the given `USBIdx`.

```
/* C/C++ */ int   PDLM_GetPulseShape            (int        USBIdx
                                                 uint32_t   *shape);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*shape	O	pointer to an unsigned integer, returning a code describing the laser pulse shape (see description for a list of values)

Description: Returns a code that describes the pulse shape regime the laser diode (connected to the device associated with given `USBIdx`) operates in. Table of valid return values:

Value	Description
0	Broadened pulse regime (due to high power settings)
1	Narrow pulse regime or “single pulse” (laser diode is operating at a pulse width corresponding to its specification)
2	Sub-threshold (or “LED domain”). No lasing occurs, only spontaneous emission
3	Unknown pulse shape

3.12. Laser Head Functions for CW Power Settings

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

The functions described in this section are setting or reading the optical output power in continuous wave (CW) mode for the connected laser head (in various units of W or permille of the current maximum). Obviously, the lower and upper limits (`MinPower` and `MaxPower`) will differ from one laser head to the next. It is therefore recommended to always read out the limits after a new head has been connected.

Note: even though it is tempting to always use absolute W values for power settings, this might cause some issues. For a given laser head, a power setting of 0.15 W might be fine, while this value might be too high for another. It is therefore recommended to use “per mille” settings (in relation to `MaxPower`).

```
/* C/C++ */ int   PDLM_GetCwPowerLimits         (int        USBIdx
                                                 float      *MinPower
                                                 float      *MaxPower);
```

Arguments:

USBIdx	I	this is the USB index [valid range: 0..7]
*MinPower	O	pointer to a float variable returning the laser diode's lower power limit (CW mode), in W.
*MaxPower	O	pointer to a float variable returning the laser diode's upper power limit (CW mode), in W.

Description: Reads out the minimum and maximum power limits in CW mode (in units of W) of the laser diode currently connected to the device associated with the given `USBIdx`.

```
/* C/C++ */ int   PDLM_SetCwPower              (int        USBIdx
                                                float      fPower);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	fPower	I	the desired optical output power (CW mode), in W
Returns:	PDLM_ERROR_ILLEGAL_VALUE		if fPower lies outside of the head's valid range

Description: Sets the desired (in W, CW mode) optical output power of the laser diode connected to the device associated with the given USBIdx.

```
/* C/C++ */ int   PDLM_GetCwPower              (int        USBIdx
                                                float      *fPower);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*fPower	O	pointer to a float variable (single precision), returning the laser optical output power (CW mode) in W

Description: Reads out the currently set optical output power (in W, CW mode) for the laser head connected to the device associated with the given USBIdx.

```
/* C/C++ */ int   PDLM_SetCwPowerPermille      (int        USBIdx
                                                uint32_t   permille);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	permille	I	the desired optical output power (CW mode) setting based on a per mille (1/1000) setting of fMaxPower
Returns:	PDLM_ERROR_ILLEGAL_VALUE		if permille lies outside of the head's valid range

Description: Sets the desired (in per mille of maximum power, CW mode) optical output power of the laser diode connected to the device associated with the given USBIdx. Valid range for a per mille is 0 to 1000, logically.

```
/* C/C++ */ int   PDLM_GetCwPowerPermille      (int        USBIdx
                                                uint32_t   *permille);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	*permille	O	pointer to an unsigned integer variable, returning the laser doide's power setting (CW mode) as a per mille of fMaxPower

Description: Reads out the currently set optical output power (as a per mille of fMaxPower, CW mode) for the laser head connected to the device associated with the given USBIdx.

```
/* C/C++ */ int   PDLM_SetCwPowerMicrowatt     (int        USBIdx
                                                uint32_t   microwatt);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	microwatt	I	the desired optical output power (CW mode) setting, in µW
Returns:	PDLM_ERROR_ILLEGAL_VALUE		if microwatt lies outside of the head's valid range

Description: Sets the desired (in µW, CW mode) optical output power of the laser diode connected to the device associated with the given USBIdx. Note: Use this function only if there is a strong demand for integer arithmetics. Consider the rounding of the float value in w to µW values.

```
/* C/C++ */ int   PDLM_GetCwPowerMicrowatt      (int        USBIdx
                                                 uint32_t   *microwatt);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	*microwatt	O	pointer to an unsigned integer variable, returning the laser doide's power setting (CW mode) in µW

Description: Reads out the currently set optical output power (in µW, CW mode) for the laser head connected to the device associated with the given `USBIdx`.

3.13. Special Laser Head Functions

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

```
/* C/C++ */ int   PDLM_SetLHFan                 (int        USBIdx
                                                 uint32_t   FanValue);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	FanValue	I	the desired fan operation mode (boolean)

Description: Set the fan operation mode (on/off) of the laser head connected to the device associated with the given `USBIdx`. Valid modes are: disabled (0) / enable (1).

```
/* C/C++ */ int   PDLM_GetLHFan                 (int        USBIdx
                                                 uint32_t   *FanValue);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	*Fanvalue	O	pointer to an unsigned integer that returns the current operational state of the laser head fan

Description: Reads out the current state of the laser head fan (enabled / disabled) that is connected to the device associated with the given `USBIdx`.

3.14. Preset Functions

All of the following functions require you to identify a device by its `USBIdx`. These functions can also commonly return with `PDLM_ERROR_USB_IOCTL_FAILED`. This signals a severe, probably unrecoverable USB communication issue (e.g., connection lost). Should this occur, it is recommended to close and re-initiate the connection to the device.

```
/* C/C++ */ int   PDLM_StorePreset              (int        USBIdx
                                                 uint32_t   PsIdx
                                                 char       *PsInfo
                                                 uint32_t   size);
```

Arguments:			
	USBIdx	I	this is the USB index [valid range: 0..7]
	PsIdx	I	this is the preset index [valid range 1..9]
	*PsInfo	I	An optional user-defined string to be stored along the preset values
	size	I	length of the PsInfo string

Returns: `PDLM_ERROR_ILLEGAL_INDEX` if the preset index is out of range

Description: Calling this function stores the current device settings (including laser head serial number) into the given preset slot (`PsIdx`). The optional variable `PsInfo` can hold a user-defined string.

```
/* C/C++ */ int   PDLM_GetPresetInfo          (int        USBIdx
                                               uint32_t   PsIdx
                                               char       *PsInfo
                                               uint32_t   size);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	PsIdx	I	this is the preset index [valid range 1..9]
	*PsInfo	O	pointer to a string where user-defined string that was stored along with the preset values will be returned
	size	I	length of the input buffer

Returns:	PDLM_ERROR_ILLEGAL_INDEX	if the preset index is out of range
	PDLM_ERROR_BUFFER_TOO_SMALL	if the buffer is too small

Description: This function returns a preview of the PsInfo string, without loading the whole preset. This can be used to generate selection list, for example.

```
/* C/C++ */ int   PDLM_GetPresetText          (int        USBIdx
                                               uint32_t   PsIdx
                                               char       *PsText
                                               uint32_t   size);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	PsIdx	I	this is the preset index [valid range 1..9]
	*PsText	O	pointer to a sting, returning a summary of the stored preset values as a text string
	size	I	length of the input buffer

Returns:	PDLM_ERROR_ILLEGAL_INDEX	if the preset index is out of range
	PDLM_ERROR_BUFFER_TOO_SMALL	if the buffer is too small

Description: This function returns a text string summarizing the stored preset values for the given PsIdx.

```
/* C/C++ */ int   PDLM_RecallPreset           (int        USBIdx
                                               uint32_t   PsIdx);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	PsIdx	I	this is the preset index [valid range 1..9]

Returns:	PDLM_ERROR_ILLEGAL_INDEX	if the preset index is out of range

Description: Recalls the preset values stored in the slot corresponding to PsIdx for the device associated with the USBIdx. Note that the serial number of the connected laser head has to match the one stored in the preset. If this is not the case, a run time error will be generated and the preset will not be restored.

Additionally, this function call terminates successfully after a valid parameter has been passed. The queued changes will occur after the call has ended. If errors occur during this switching process, new run time errors will be generated that have to be captured.

Important: the recall function will overwrite any current settings irrevocably!

```
/* C/C++ */ int   PDLM_ErasePreset            (int        USBIdx
                                               uint32_t   PsIdx);
```

Arguments:	USBIdx	I	this is the USB index [valid range: 0..7]
	PsIdx	I	this is the preset index [valid range 1..9]

Returns:	PDLM_ERROR_ILLEGAL_INDEX	if the preset index is out of range

Description: Deletes all stored preset data in the slot corresponding to the PsIdx of the device associated with the given USBIdx. Important: The erase function will delete any data existing in that slot irrevocably!

4. Demonstration Programs

Please note that all of the demonstration code provided is correct to our best knowledge. However, it is provided “as is” with no warranties as to fitness for purpose. A link to the directory containing the demonstration code and programs was created during the Taiko PDL M1 software installation. Click on that link to open this folder in the Windows Explorer .

Two demos are currently included (a “simple” and a “complex” one) that aim to insights into some of the API’s feature set. In order to keep the demo code concise and accessible, both application examples have a minimalistic text-based interface (using the Windows console for input and output).

The “simple” demo code is available for five languages, including C/C++, C#, Delphi, Python, and LabVIEW. It is meant to highlight the tagged communication mode between laser driver and host application. After starting the demo, it will connect to the Takio driver of your choice and read out it current status. Upon performing any changes to intensity, repetition rate or temperature setting via the local interface (i.e. using the push dial), the “simple” demo will notice these changes and update the display accordingly. Pressing “x” will end the demo and release the Taiko.

The “complex” demo code is only available in Delphi and LabVIEW. This application will connect to a Taiko and allow controlling it through a simple text-based interface. However, the LabVIEW “complex” demo provides no text-based menu and will only display information on changed done through the local interface (demonstrating the use windows messages).

Follow the on-screen prompts to effect any desired changes. Note that the application can be closed by pressing “x”. The application will the release the Taiko.

The demo programs illustrate the typical structure of a Taiko session:

- Get library version and check it comparing to system constant LIB_VERSION_REFERENCE
  (optional)
- Open the device on the desired USB channel
  (mandatory)
- Get firmware version and USB string descriptors (just for information and service purposes)
  (optional)
- Get last error detected by firmware and decode it if necessary
  (optional)
- Insert implementation of your desired behavior here
  …
  …
- Close the device to release it

  (mandatory)

5. Legal Terms

5.1. Copyright

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

5.2. Trademarks

Other 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 the purposes of identification or explanation and to the owner's benefit, without intent to infringe.

6. Appendix

6.1. Table of Common Constants

Constant Name	Length
PDLM_LIBVERSION_MAXLEN	12
PDLM_USB_INDEX_MIN	0
PDLM_USB_INDEX_MAX	7
PDLM_MAX_USBDEVICES	8
PDLM_HW_ERRORSTRING_MAXLEN	47
PDLM_HW_INFO_MAXLEN	36
PDLM_DEV_STRING_LENGTH	16
PDLM_LDH_STRING_LENGTH	16

6.2. Table of Return / Error Codes

Note: A human-readable error string can be queried for each error code by calling the function "**PDLM_DecodeError**".

Important: A number of error numbers do exist between "PDLM_ERROR_HW_ERROR_OFFSET" and "PDLM_ERROR_HW_MAX_ERROR_NUM". These are **not** listed here as they are dependent on the hardware version. However, these error numbers can still be decoded by using the function "**PDLM_DecodeError**".

Error String	Error Code
PDLM_ERROR_NONE	0
PDLM_ERROR_DEVICE_NOT_FOUND	-1
PDLM_ERROR_NOT_CONNECTED	-2
PDLM_ERROR_ALREADY_CONNECTED	-3
PDLM_ERROR_WRONG_USBIDX	-4
PDLM_ERROR_ILLEGAL_INDEX	-5
PDLM_ERROR_ILLEGAL_VALUE	-6
PDLM_ERROR_USB_MSG_INTEGRITY_VIOLATED	-7
PDLM_ERROR_ILLEGAL_NODEINDEX	-8
PDLM_ERROR_WRONG_PARAMETER	-9
PDLM_ERROR_INCOMPATIBLE_FW	-10
PDLM_ERROR_WRONG_SERIALNUMBER	-11
PDLM_ERROR_WRONG_PRODUCTMODEL	-12
PDLM_ERROR_BUFFER_TOO_SMALL	-13
PDLM_ERROR_INDEX_NOT_FOUND	-14
PDLM_ERROR_FW_MEMORY_ALLOCATION_ERROR	-15
PDLM_ERROR_FREQUENCY_TOO_HIGH	-16
PDLM_ERROR_DEVICE_BUSY_OR_BLOCKED	-17
PDLM_ERROR_USB_INAPPROPRIATE_DEVICE	-18
PDLM_ERROR_USB_GET_DSCR_FAILED	-19

Error String	Error Code
PDLM_ERROR_USB_INVALID_HANDLE	-20
PDLM_ERROR_USB_INVALID_DSCRBUF	-21
PDLM_ERROR_USB_IOCTL_FAILED	-22
PDLM_ERROR_USB_VCMD_FAILED	-23
PDLM_ERROR_USB_NO_SUCH_PIPE	-24
PDLM_ERROR_USB_REGNTFY_FAILED	-25
PDLM_ERROR_USBDRIVER_NO_MEMORY	-26
PDLM_ERROR_DEVICE_ALREADY_OPENED	-27
PDLM_ERROR_OPEN_DEVICE_FAILED	-28
PDLM_ERROR_USB_UNKNOWN_DEVICE	-29
PDLM_ERROR_EMPTY_QUEUE	-30
PDLM_ERROR_FEATURE_NOT_AVAILABLE	-31
PDLM_ERROR_UNINITIALIZED_DATA	-32
PDLM_ERROR_DLL_MEMORY_ALLOCATION_ERROR	-33
PDLM_ERROR_UNKNOWN_TAG	-34
PDLM_ERROR_OPEN_FILE	-35
PDLM_ERROR_FW_FOOTER	-36
PDLM_ERROR_FIRMWARE_UPDATE	-37
PDLM_ERROR_FIRMWARE_UPDATE_RUNNING	-38
PDLM_ERROR_INCOMPATIBLE_HARDWARE	-39
PDLM_ERROR_VALUE_NOT_AVAILABLE	-40
PDLM_ERROR_USB_SET_TIMED_OUT	-41
PDLM_ERROR_USB_GET_TIMED_OUT	-42
PDLM_ERROR_USB_SET_FAILED	-43
PDLM_ERROR_USB_GET_FAILED	-44
PDLM_ERROR_USB_DATA_SIZE_TOO_BIG	-45
PDLM_ERROR_FW_VERSION_CHECK	-46
PDLM_ERROR_WRONG_DRIVER	-47
PDLM_ERROR_WINUSB_STORED_ERROR	-48
PDLM_ERROR_UNKNOWN_ERRORCODE	-999
PDLM_ERROR_HW_ERROR_OFFSET	-1000
PDLM_ERROR_HW_MAX_ERROR_NUM	-2999
PDLM_ERROR_FUNCTION_IS_PQ_INTERNAL	-9000
PDLM_ERROR_FUNCTION_NOT_IMPLEMENTED_YET	-9999

6.3. Table of Assigned Status Bits

String	Bit Code	Description
PDLM_DEVSTATE_INITIALIZING	0x00000001	Device is initializing during boot up
PDLM_DEVSTATE_DEVICE_UNCALIBRATED	0x00000002	If the device has no valid data in eeprom
PDLM_DEVSTATE_COMMISSIONING_MODE	0x00000004	During commissioning. All errors coming from device/laserhead are ignored
PDLM_DEVSTATE_LASERHEAD_SAFETY_MODE	0x00000008	Laser head safety mode
PDLM_DEVSTATE_FW_UPDATE_RUNNING	0x00000010	Firmware update is in progress
PDLM_DEVSTATE_DEVICE_DEFECT	0x00000020	At least one part of the device hardware is defective
PDLM_DEVSTATE_DEVICE_INCOMPATIBLE	0x00000040	The firmware cannot control the read device version
PDLM_DEVSTATE_BUSY	0x00000080	Device is busy during costly calculations, etc
PDLM_DEVSTATE_EXCLUSIVE_SW_OP_GRANTED	0x00000100	Only the host software can manipulate the device
PDLM_DEVSTATE_PARAMETER_CHANGES_PENDING	0x00000200	At least one parameter of the device has changed
PDLM_DEVSTATE_LASERHEAD_CHANGED	0x00000800	When a new laser head was connected
PDLM_DEVSTATE_LASERHEAD_MISSING	0x00001000	No laser head connected
PDLM_DEVSTATE_LASERHEAD_DEFECT	0x00002000	Laser head defective
PDLM_DEVSTATE_LASERHEAD_UNKNOWN_TYPE	0x00004000	The laser type cannot be controlled by the laser driver
PDLM_DEVSTATE_LASERHEAD_DECALIBRATED	0x00008000	Laser head calibration expired, data may no longer be valid
PDLM_DEVSTATE_LASERHEAD_DIODE_TEMP_TOO_LOW	0x00010000	Laser head temperature is below set point
PDLM_DEVSTATE_LASERHEAD_DIODE_TEMP_TOO_HIGH	0x00020000	Laser head temperature is above set point
PDLM_DEVSTATE_LASERHEAD_DIODE_OVERHEATING	0x00040000	Laser head diode overheated
PDLM_DEVSTATE_LASERHEAD_CASE_OVERHEATING	0x00080000	Laser head case overheated
PDLM_DEVSTATE_LASERHEAD_FAN_RUNNING	0x00100000	Laser head fan is running
PDLM_DEVSTATE_LASERHEAD_INCOMPATIBLE	0x00200000	The firmware cannot control the laser version read
PDLM_DEVSTATE_LOCKED_BY_EXPIRED_DEMO_MODE	0x00400000	Laser will be locked when demo mode expired
PDLM_DEVSTATE_LOCKED_BY_ON_OFF_BUTTON	0x00800000	Laser was off by On/Off button
PDLM_DEVSTATE_SOFTLOCK	0x01000000	Laser was turned off by host software
PDLM_DEVSTATE_KEYLOCK	0x02000000	Laser is off by keylock
PDLM_DEVSTATE_LOCKED_BY_SECURITY_POLICY	0x04000000	Laser is off due to Laser Class IV - rules

String	Bit Code	Description
PDLM_DEVSTATE_INTERLOCK	0x08000000	Laser is off because interlock is unplugged
PDLM_DEVSTATE_LASERHEAD_PULSE_POWER_INACCURATE	0x10000000	Laser temperature differs from what it was calibrated on
PDLM_DEVSTATE_ERRORMSG_PENDING	0x80000000	Error message pending in error queue, not laser head related

6.4. Table of Useful Status Masks

These are semantic groups of more than one status bit with their associated notification conditions.

States / Statemasks	Bit Pattern	Results on Changes
PDLM_DEVSTATEMASK_DEVICE_NOT_OPERATIONAL = (PDLM_DEVSTATE_INITIALIZING | PDLM_DEVSTATE_FW_UPDATE_RUNNING | PDLM_DEVSTATE_DEVICE_DEFECT | PDLM_DEVSTATE_DEVICE_INCOMPATIBLE)	0x00000071	fires notification "WM_ON_DEVICE_NOT_OPERATIONAL_CHANGE" if changing from 0 to (>0) or from (>0) to 0 but not from (>0) to another value (>0)
PDLM_DEVSTATE_EXCLUSIVE_SW_OP_GRANTED	0x00000100	fires notification "WM_ON_EXCLUSIVE_UI_CHANGE" on any change
PDLM_DEVSTATE_PARAMETER_CHANGES_PENDING	0x00000200	fires notification "WM_ON_PARAMETER_CHANGE" if changing from 0 to (>0)
PDLM_DEVSTATE_LASERHEAD_CHANGED	0x00000800	fires notification "WM_ON_LASERHEAD_CHANGE" if changing from 0 to (>0)
PDLM_DEVSTATEMASK_LASER_NOT_OPERATIONAL = (PDLM_DEVSTATE_LASERHEAD_SAFETY_MODE | PDLM_DEVSTATE_LASERHEAD_MISSING | PDLM_DEVSTATE_LASERHEAD_DEFECT | PDLM_DEVSTATE_LASERHEAD_UNKNOWN_TYPE | PDLM_DEVSTATE_LASERHEAD_DIODE_OVERHEATING | PDLM_DEVSTATE_LASERHEAD_CASE_OVERHEATING | PDLM_DEVSTATE_LASERHEAD_INCOMPATIBLE)	0x002C7008	fires notification "WM_ON_LASER_NOT_OPERATIONAL_CHANGE" if changing from 0 to (>0) or from (>0) to 0 but not from (>0) to another value (>0)
PDLM_DEVSTATEMASK_LOCKED = (PDLM_DEVSTATE_LOCKED_BY_EXPIRED_DEMO_MODE | PDLM_DEVSTATE_LOCKED_BY_ON_OFF_BUTTON | PDLM_DEVSTATE_SOFTLOCK | PDLM_DEVSTATE_KEYLOCK | PDLM_DEVSTATE_LOCKED_BY_SECURITY_POLICY | PDLM_DEVSTATE_INTERLOCK)	0x0FC00000	fires notification "WM_ON_LOCKING_CHANGE" on any change

States / Statemasks	Bit Pattern	Results on Changes
PDLM_DEVSTATE_ERRORMSG_PENDING	0x80000000	fires notification "WM_ON_PENDING_ERRORS" if changing from 0 to (>0)
PDLM_DEVSTATEMASK_WARNINGS_ONLY = (PDLM_DEVSTATE_DEVICE_UNCALIBRATED | PDLM_DEVSTATE_LASERHEAD_DECALIBRATED | PDLM_DEVSTATE_LASERHEAD_PULSE_POWER_INACCURATE)	0x10008002	a notification will be fired together with other flags, see below: PDLM_DEVSTATEMASK_ALL_WARNINGS
PDLM_DEVSTATEMASK_UNHANDLED = (PDLM_DEVSTATE_COMMISSIONING_MODE | PDLM_DEVSTATE_BUSY | PDLM_DEVSTATE_LASERHEAD_DIODE_TEMP_TOO_LOW | PDLM_DEVSTATE_LASERHEAD_DIODE_TEMP_TOO_HIGH | PDLM_DEVSTATE_LASERHEAD_FAN_RUNNING)	0x00130084	fires notification "WM_ON_OTHER_STATES_CHANGE" on any change
PDLM_DEVSTATEMASK_ILLEGAL_STATES	0x60000400	these state flags (unused dummys) are always ignored
	0xFFFFFFFF	
PDLM_DEVSTATEMASK_ALL_WARNINGS = (PDLM_DEVSTATE_DEVICE_UNCALIBRATED | PDLM_DEVSTATE_COMMISSIONING_MODE | PDLM_DEVSTATE_LASERHEAD_SAFETY_MODE | PDLM_DEVSTATE_FW_UPDATE_RUNNING | PDLM_DEVSTATE_DEVICE_DEFECT | PDLM_DEVSTATE_DEVICE_INCOMPATIBLE | PDLM_DEVSTATE_LASERHEAD_DEFECT | PDLM_DEVSTATE_LASERHEAD_UNKNOWN_TYPE | PDLM_DEVSTATE_LASERHEAD_DECALIBRATED | PDLM_DEVSTATE_LASERHEAD_DIODE_OVERHEATING | PDLM_DEVSTATE_LASERHEAD_CASE_OVERHEATING | PDLM_DEVSTATE_LASERHEAD_INCOMPATIBLE | PDLM_DEVSTATE_LOCKED_BY_EXPIRED_DEMO_MODE | PDLM_DEVSTATE_LASERHEAD_PULSE_POWER_INACCURATE)	0x106CE07E	fires notification "WM_ON_WARNINGS_CHANGE" on any change; All Flags produce "!"-Warnings, except: this one produces a "C"-Warning this one produces an "i"-Warning

States / Statemasks	Bit Pattern	Results on Changes
PDLM_DEVSTATEMASK_LASERHEAD_STATUS_FLAGS = (PDLM_DEVSTATE_LASERHEAD_SAFETY_MODE | PDLM_DEVSTATE_LASERHEAD_CHANGED | PDLM_DEVSTATE_LASERHEAD_DEFECT | PDLM_DEVSTATE_LASERHEAD_UNKNOWN_TYPE | PDLM_DEVSTATE_LASERHEAD_DECALIBRATED | PDLM_DEVSTATE_LASERHEAD_DIODE_TEMP_TOO_LOW | PDLM_DEVSTATE_LASERHEAD_DIODE_TEMP_TOO_HIGH | PDLM_DEVSTATE_LASERHEAD_DIODE_OVERHEATING | PDLM_DEVSTATE_LASERHEAD_CASE_OVERHEATING | PDLM_DEVSTATE_LASERHEAD_FAN_RUNNING | PDLM_DEVSTATE_LASERHEAD_INCOMPATIBLE | PDLM_DEVSTATE_LOCKED_BY_EXPIRED_DEMO_MODE | PDLM_DEVSTATE_LOCKED_BY_SECURITY_POLICY)	0x047FE808	If a laser head is disconnected, all laser head related flags are reset (with the exception of PDLM_DEVSTATE_LASERHEAD_MISSING)

6.5. Table of Declared Tag Types

Tag	Code	Notes
PDLM_TAGTYPE_BOOL	0x00000001	
PDLM_TAGTYPE_UINT	0x00010001	
PDLM_TAGTYPE_UINT_ENUM	0x00010002	for list-driven values
PDLM_TAGTYPE_UINT_DAC	0x00010003	for any directly given raw DAC value
PDLM_TAGTYPE_UINT_IN_TENTH	0x00010101	for temperatures in tenth of a celsius degree
PDLM_TAGTYPE_UINT_IN_PERCENT	0x00010201	for (positive only values of) milli Volts, milli Watts, etc.
PDLM_TAGTYPE_UINT_IN_PERMILLE	0x00010301	for permille values in power or current interpolation tables
PDLM_TAGTYPE_UINT_IN_PERTHOUSAND	0x00010302	for (positive only) milli Volts, milli Watts, etc.
PDLM_TAGTYPE_UINT_IN_PERMYRIAD	0x00010401	for current interpolation tables (a hundredth of a percent)
PDLM_TAGTYPE_UINT_IN_PERMILLION	0x00010601	for cw power values in 10^{-6} Watt = µW
PDLM_TAGTYPE_UINT_IN_PERBILLION	0x00010901	for pulse power values in 10^{-9} Watt = nW
PDLM_TAGTYPE_UINT_IN_PERTRILLION	0x00010C01	for wavelength values in 10^{-12} m = pm
PDLM_TAGTYPE_UINT_IN_PERQUADTRILLION	0x00010F01	for pulse energy values in 10^{-15} joules = femto joule
PDLM_TAGTYPE_INT	0x00110001	
PDLM_TAGTYPE_INT_IN_PERTHOUSAND	0x00110302	for (negative values included) milli Volts, etc.
PDLM_TAGTYPE_SINGLE	0x01000001	
PDLM_TAGTYPE_VOID	0xFFFFFFFF	

6.6. Table of Documented Tags

Please note that more tags have been defined for internal use only. These are not listed in this table. Current PDLM_TAG_COUNT is 54. This number and list of tags is subject to change without notification.

Tag	Code	Notes
PDLM_TAG_NONE	0x00000000	
PDLM_TAG_LaserMode	0x00000020	
PDLM_TAG_LDH_PulsePowerTable	0x00000021	
PDLM_TAG_TriggerMode	0x00000030	
PDLM_TAG_TriggerLevelRaw	0x00000040	in DAC steps
PDLM_TAG_TriggerLevelRawLoLimit	0x00000041	in DAC steps
PDLM_TAG_TriggerLevelRawHiLimit	0x00000042	in DAC steps
PDLM_TAG_TriggerLevel	0x00000048	in V
PDLM_TAG_TriggerLevelLoLimit	0x00000049	in V

Tag	Code	Notes
PDLM_TAG_TriggerLevelHiLimit	0x0000004A	in V
PDLM_TAG_FastGate	0x00000050	
PDLM_TAG_FastGateImp	0x00000060	
PDLM_TAG_SlowGate	0x00000070	
		the temperatures passed along with the following tags are ambiguous! Units depend on current TempScale value
PDLM_TAG_TargetTempRaw	0x00000090	in tenth of a Celsius degree
PDLM_TAG_TargetTempRawLoLimit	0x00000091	in tenth of a Celsius degree
PDLM_TAG_TargetTempRawHiLimit	0x00000092	in tenth of a Celsius degree
PDLM_TAG_CurrentTempRaw	0x00000094	in tenth of a Celsius degree
PDLM_TAG_CaseTempRaw	0x00000095	in tenth of a Celsius degree
PDLM_TAG_TargetTemp	0x00000098	in arbitrary temperature units
PDLM_TAG_TargetTempLoLimit	0x00000099	in arbitrary temperature units
PDLM_TAG_TargetTempHiLimit	0x0000009A	in arbitrary temperature units
PDLM_TAG_CurrentTemp	0x0000009C	in arbitrary temperature units
PDLM_TAG_CaseTemp	0x0000009D	in arbitrary temperature units
PDLM_TAG_TempScale	0x0000009F	identifies temperature unit currently in use
PDLM_TAG_Frequency	0x000000A8	in Hz
PDLM_TAG_FrequencyLoLimit	0x000000A9	in Hz
PDLM_TAG_FrequencyHiLimit	0x000000AA	in Hz
PDLM_TAG_PulsePowerPermille	0x000000B4	
PDLM_TAG_PulseShape	0x000000B5	
PDLM_TAG_PulsePower	0x000000B8	in W
PDLM_TAG_PulsePowerLoLimit	0x000000B9	in W
PDLM_TAG_PulsePowerHiLimit	0x000000BA	in W
PDLM_TAG_PulsePowerNanowatt	0x000000BC	in nW
PDLM_TAG_PulsePowerVoltage	0x000000BE	In mV
PDLM_TAG_PulseEnergy	0x000000BF	in fJ
PDLM_TAG_CwPowerPermille	0x000000C4	
PDLM_TAG_CwPower	0x000000C8	in W
PDLM_TAG_CwPowerLoLimit	0x000000C9	in W
PDLM_TAG_CwPowerHiLimit	0x000000CA	in W
PDLM_TAG_CwPowerMicroWatt	0x000000CC	in µW

Tag	Code	Notes
PDLM_TAG_BurstLen	0x000000D0	
PDLM_TAG_BurstPeriod	0x000000E0	
PDLM_TAG_LDH_Fan	0x000000F0	is also published by status flag
PDLM_TAG_UI_Exclusive	0x00000100	is also published by status flag

6.7. Table of Supported Temperature Scales

Temperature Scale Name	Value
PDLM_TEMPERATURESCALE_CELSIUS	0x00000000
PDLM_TEMPERATURESCALE_FAHRENHEIT	0x00000001
PDLM_TEMPERATURESCALE_KELVIN	0x00000002

6.8. Table of Laser Head Feature Bits

These are used for both the `Features` field of the structure `laserData_t` and for the function "`PDLM_GetLHFeatures`". Note that there is one more feature bit, that is implicitly always set: "`PDLM_LHFEATURE_PULSE_CAPABILITY`", as all LDH-I laser heads are pulsed ones.

Feature Name	Bit Code	Notes
PDLM_LHFEATURE_CW_CAPABILITY	0x00000001	Is set if laser head supports cw operation mode
PDLM_LHFEATURE_PULSE_MAXPOWER	0x00000002	
PDLM_LHFEATURE_BURST_CAPABILITY	0x00000010	Is set if laser head supports burst mode
PDLM_LHFEATURE_EXTERNAL_TRIGGERABLE_BURSTS	0x00000040	Is set if laser head supports external triggering of bursts
PDLM_LHFEATURE_EXTERNAL_TRIGGERABLE_PULSES	0x00000080	Is set if laser head supports external triggering of pulses
PDLM_LHFEATURE_WL_TUNABLE	0x00000100	Is set if laser head includes calibrated data for temperature-dependent wavelength shifts
PDLM_LHFEATURE_COOLING_FAN	0x00010000	Is set if laser head features a cooling fan
PDLM_LHFEATURE_SWITCHABLE_FAN	0x00020000	Is set if the cooling fan can be switched on/off
PDLM_LHFEATURE_INTENSITY_SENSOR_TYPE	0x0F000000	Contains four bits that encode the type of the intensity sensor

6.9. Table of Laser Head Types

These can be found in the field `laserType` of the structure `laserData_t`.

Name	Code	Description
LASER_TYPE_UNDEFINED	0x0000	
LASER_TYPE_LDH	0x0010	Laser diode

Name	Code	Description
LASER_TYPE_LDH_FSL	0x0018	Laser diode with Fast Switched Laser mode, implemented via fast gate
LASER_TYPE_LED	0x0020	only spontaneous LED emission (no lasing!)
LASER_TYPE_TA_SHG	0x0030	With tapered fiber amplifier and second harmonic generation
LASER_TYPE_FIBER	0x0040	Fiber
LASER_TYPE_FIBER_FSL	0x0048	Fiber with Fast Switched Laser mode
LASER_TYPE_FA	0x0050	Fiber amplifier
LASER_TYPE_FA_SHG	0x0060	Fiber amplifier with second harmonic generation
LASER_TYPE_BRIDGE	0x00F0	Bridge for older generation laser heads

6.10. Index

PDLM_CloseDevice....11
PDLM_CreateSupportRequestText....12
PDLM_DecodeError....5, **9,** 36
PDLM_DecodeLHFeatures....9, **15**
PDLM_DecodePulseShape....**9**
PDLM_DecodeSystemStatus....**10**
PDLM_ErasePreset....**33**
PDLM_GetBurst....**25**
PDLM_GetCwPower....**30**
PDLM_GetCwPowerLimits....**29**
PDLM_GetCwPowerMicrowatt....**31**
PDLM_GetCwPowerPermille....**30**
PDLM_GetDeviceData....**13**
PDLM_GetExclusiveUI....**12**
PDLM_GetExtTriggerFrequency....**22**
PDLM_GetFastGate....**22**
PDLM_GetFastGateImp....**23**
PDLM_GetFPGAVersion....**13**
PDLM_GetFrequency....**24**
PDLM_GetFrequencyLimits....**15, 24**
PDLM_GetFWVersion....**13**
PDLM_GetHardwareInfo....**12**
PDLM_GetLaserMode....**20**
PDLM_GetLDHPulsePowerTable....**21**
PDLM_GetLHCaseTemp....**27**
PDLM_GetLHCurrentTemp....**26**
PDLM_GetLHData....**14**
PDLM_GetLHFan....**31**
PDLM_GetLHFeatures....**16**
PDLM_GetLHInfo....**16**

PDLM_GetLHTargetTemp....**26**
PDLM_GetLHTargetTempLimits....**26**
PDLM_GetLHVersion....**14,** 15
PDLM_GetLHWavelength....**27**
PDLM_GetLibraryVersion....**8**
PDLM_GetLocked....**19**
PDLM_GetPresetInfo....**32**
PDLM_GetPresetText....**32**
PDLM_GetPulsePower....**28**
PDLM_GetPulsePowerLimits....**27**
PDLM_GetPulsePowerNanowatt....**29**
PDLM_GetPulsePowerPermille....**28**
PDLM_GetPulseShape....**29**
PDLM_GetQueuedChanges....6, **17**
PDLM_GetQueuedError....**18**
PDLM_GetQueuedErrorString....**18**
PDLM_GetSlowGate....**23**
PDLM_GetSoftLock....**19**
PDLM_GetSystemStatus....**17**
PDLM_GetTagDescription....6, **9**
PDLM_GetTagValueList....**18**
PDLM_GetTempScale....**25**
PDLM_GetTriggerLevel....**22**
PDLM_GetTriggerLevelLimits....**21**
PDLM_GetTriggerMode....**21**
PDLM_GetUSBDriverInfo....**8**
PDLM_GetUSBStrDescriptor....**12**
PDLM_LibIsRunningInWine....**8**
PDLM_OpenDevice....**10**
PDLM_OpenGetSerNumAndClose....5, **11**
PDLM_RecallPreset....**32**
PDLM_SetBurst....**24**
PDLM_SetCwPower....**30**
PDLM_SetCwPowerMicrowatt....**30**
PDLM_SetCwPowerPermille....**30**
PDLM_SetExclusiveUI....**11**
PDLM_SetFastGate....**22**
PDLM_SetFastGateImp....**23**
PDLM_SetFrequency....**24**
PDLM_SetHWND....5, **16**
PDLM_SetLaserMode....**19**
PDLM_SetLDHPulsePowerTable....**20,** 27

PDLM_SetLHFan....31
PDLM_SetLHTargetTemp....26
PDLM_SetPulsePower....28
PDLM_SetPulsePowerNanowatt....28
PDLM_SetPulsePowerPermille....28
PDLM_SetSlowGate....23
PDLM_SetSoftLock....19
PDLM_SetTempScale....25
PDLM_SetTriggerLevel....22
PDLM_SetTriggerMode....21
PDLM_StorePreset....31

This page was intentionally left blank

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
www.picoquant.com