

---

# **phconvert Documentation**

*Release 0.7.3+49.g4dcb7b4*

**Antonino Ingargiola**

**Jun 04, 2017**



---

## Contents

---

<b>1</b>	<b>Module hdf5</b>	<b>3</b>
<b>2</b>	<b>Module loader</b>	<b>7</b>
<b>3</b>	<b>Module pqreader</b>	<b>9</b>
<b>4</b>	<b>Module bhreader</b>	<b>13</b>
<b>5</b>	<b>Indices and tables</b>	<b>17</b>
	<b>Python Module Index</b>	<b>19</b>



**Version** 0.7.3+49.g4dcb7b4 ([release notes](#))

phconvert is a python 2 & 3 library which helps writing valid [Photon-HDF5](#) files. This document contains the API documentation for phconvert.

The phconvert library contains two main modules: *hdf5* and *loader*. The former contains functions to save and validate Photon-HDF5 files. The latter, contains functions to load other formats to be converted to Photon-HDF5.

The phconvert repository contains a set the notebooks to convert existing formats to Photon-HDF5 or to write Photon-HDF5 from scratch:

- [phconvert notebooks \(read online\)](#).

In particular see notebook [Writing Photon-HDF5 files \(read online\)](#) as an example of writing Photon-HDF5 files from scratch.

Finally, phconvert repository contains a [JSON specification](#) of the Photon-HDF5 format which lists all the valid field names and corresponding data types and descriptions.

Contents:



The module `hdf5` defines functions to save and validate Photon-HDF5 files. The main two functions in this module are:

- `save_photon_hdf5()` to save data from a dictionary to Photon-HDF5.
- `assert_valid_photon_hdf5()` to validate if a HDF5 file is valid Photon-HDF5.

This module also provides functions to save free-form dict to HDF5 (`dict_to_group()`) and read a HDF5 group into a dict (`dict_from_group()`). Finally there are utility functions to easily print HDF5 nodes and attributes (`print_children()`, `print_attrs()`).

For more info see: [Writing Photon-HDF5 files](#).

## List of functions

Main functions to save and validate Photon-HDF5 files.

```
phconvert.hdf5.save_photon_hdf5(data_dict, h5_fname=None, h5file=None, user_descr=None,
                                overwrite=False, compression={'complevel': 6, 'complib': 'zlib'},
                                close=True, validate=True, warnings=True, skip_measurement_specs=False,
                                require_setup=True, debug=False)
```

Saves the dict `data_dict` in the Photon-HDF5 format.

This function requires the data to be saved as `data_dict` argument. The data needs to have the hierarchical structure of a Photon-HDF5 file. For the purpose, we use a standard python dictionary: each key is a Photon-HDF5 field name and each value contains data (e.g. array, string, etc..) or another dictionary (in which case, it represents an HDF5 sub-group). Similarly, sub-dictionaries contain data or other dictionaries, as needed to represent the hierarchy of Photon-HDF5 files.

Features of this function:

- Checks that all field names are valid Photon-HDF5 field names.
- Checks that all field types match the Photon-HDF5 specs (scalar, array, or string).

- Populates automatically the identity group with filename, software, version and file creation date.
- Populates automatically the provenance group with info on the original data file (if it can be found on disk): creation and modification date, path.
- Computes field *acquisition\_duration* when not provided (single-spot data only).

Minimal fields required to create a Photon-HDF5 file:

- /description* (string)
- /photon\_data/timestamps* (array)
- /photon\_data/timestamps\_specs/timestamps\_unit* (scalar float)
- /setup/num\_pixels* (int): number of detectors
- /setup/num\_spots* (int): number of excitation/detection spots
- /setup/num\_spectral\_ch* (int): number of detection spectral bands
- /setup/num\_polarization\_ch* (int): number of detected polarization states
- /setup/num\_split\_ch* (int): number of beam split channels
- /setup/modulated\_excitation* (bool): True if there is any form of intensity or polarization modulation or interleaved excitation (PIE or nsALEX). This field has become obsolete in version 0.5 and maintained only for compatibility.
- /setup/excitation\_alternated* (array of bool): New in version 0.5. Values are True if the respective excitation source is intensity-modulated. In us-ALEX both sources are alternated, while in PAX measurements only one source is alternated.
- /setup/lifetime* (bool): True if dataset contains TCSPC data.

See also [Writing Photon-HDF5 files](#).

As a side effect *data\_dict* is modified by adding the key ‘\_data\_file’ containing a reference to the pytables file.

### Parameters

- **data\_dict** (*dict*) – the dictionary containing the photon data. The keys must strings matching valid Photon-HDF5 paths. The values must be scalars, arrays, strings or another dict.
- **h5\_fname** (*string or None*) – file name for the output Photon-HDF5 file. If None and *h5file* is also None, the file name is taken from *data\_dict['\_filename']* with extension changed to ‘.hdf5’.
- **h5file** (*pytables.File or None*) – an already open and writable HDF5 file to use as container. This argument can be used to complete an HDF5 file already containing some arrays, or to update an already existing Photon-HDF5 file in-place. For more info see note below.
- **user\_descr** (*dict or None*) – dictionary of descriptions (strings) for user-defined fields. The keys must be strings representing the full HDF5 path of each field. The values must be binary (i.e. encoded) strings restricted to the ASCII set.
- **overwrite** (*bool*) – if True, a pre-existing HDF5 file with same name is overwritten. If False, save the new file by adding the suffix “new\_copy” (and if a “\_new\_copy” file is already present overwrites it).
- **compression** (*dict*) – a dictionary containing the compression type and level. Passed to *pytables.tables.Filters()*.



- **close** (*bool*) – If True (default) the HDF5 file is closed before returning. If False the file is left open.
- **validate** (*bool*) – if True, after saving perform a validation step raising an error if the specs are not followed.
- **warnings** (*bool*) – if True, print warnings for important optional fields that are missing. If False, don't print warnings.
- **skip\_measurement\_specs** (*bool*) – if True don't print any warning for missing measurement\_specs group.
- **require\_setup** (*bool*) – if True, raises an error if some mandatory fields in /setup are missing. If False, allows missing setup fields (or missing setup altogether). Use False when saving only detectors' dark counts.
- **debug** (*bool*) – if True prints additional debug information.

For description and specs of the Photon-HDF5 format see: <http://photon-hdf5.readthedocs.org/>

---

**Note:** The argument *h5file* accepts an already open HDF5 file for storage. This allows completing a partially written file (for example containing only *photon\_data* arrays) or correcting and already complete Photon-HDF5 file. When using *h5file*, you need to pass a full *data\_dict* structure as usual. If you don't want update an array, put in *data\_dict* a reference to the existing pytables array (instead of using a numpy array). Fields containing numpy arrays will be overwritten. Fields containing pytables Array (including CArray or EArray) will be left unmodified. In either cases the TITLE attribute is always updated.

---

```
phconvert.hdf5.assert_valid_photon_hdf5(datafile, warnings=True, verbose=False,
                                       strict_description=True, require_setup=True,
                                       skip_measurement_specs=False)
```

Asserts that *datafile* follows the Photon-HDF5 specs.

If the input *datafile* does not follow the specifications, it raises the `Invalid_PhotonHDF5` exception, with a message indicating the cause of the error.

This function checks that:

- all fields are valid Photon-HDF5 names
- all fields have valid descriptions
- all mandatory fields are present
- if /setup/lifetime is True (i.e. 1), assures that *nanotimes* and *nanotimes\_specs* are present

#### Parameters

- **datafile** (*string* or *tables.File*) – input data file to be validated
- **warnings** (*bool*) – if True, print warnings for important optional fields that are missing. If False, don't print warnings.
- **verbose** (*bool*) – if True print details about the performed tests.
- **strict\_description** (*bool*) – if True consider a non-conforming description (TITLE) a specs violation.
- **require\_setup** (*bool*) – if True, raises an error if some mandatory fields in /setup are missing. If False, allows missing setup fields (or missing setup altogether).
- **skip\_measurement\_specs** (*bool*) – if True don't print any warning for missing measurement\_specs group.

## Utility functions

Utility functions to work with HDF5 files in pytables.

`phconvert.hdf5.print_children` (*group*)

Print all the sub-groups in *group* and leaf-nodes children of *group*.

**Parameters** `group` (*pytables group*) – the group to be printed.

`phconvert.hdf5.print_attrs` (*node*, *which*='user')

Print the HDF5 attributes for *node\_name*.

**Parameters**

- **node** (*pytables node*) – node whose attributes will be printed. Can be either a group or a leaf-node.
- **which** (*string*) – Valid values are ‘user’ for user-defined attributes, ‘sys’ for pytables-specific attributes and ‘all’ to print both groups of attributes. Default ‘user’.

`phconvert.hdf5.dict_from_group` (*group*, *read*=True)

Return a dict with the content of a PyTables *group*.

`phconvert.hdf5.dict_to_group` (*group*, *dictionary*)

Save *dictionary* into HDF5 format in *group*.

---

Module loader

---

This module contains functions to load each supported data format. Each loader function loads data from a third-party formats into a python dictionary which has the structure of a Photon-HDF5 file. These dictionaries can be passed to `phconvert.hdf5.save_photon_hdf5()` to save the data in Photon-HDF5 format.

The loader module contains high-level functions which “fill” the dictionary with the appropriate arrays. The actual decoding of the input binary files is performed by low-level functions in other modules (`smreader.py`, `pqreader.py`, `bhreader.py`). When trying to decode a new file format, these modules can provide useful examples.

```
phconvert.loader.nsalex_bh(filename_spc, donor=4, acceptor=6, alex_period_donor=(10, 1500),
                           alex_period_acceptor=(2000, 3500), excitation_wavelengths=(5.32e-07,
                                           6.35e-07), detection_wavelengths=(5.8e-07, 6.8e-07),
                           allow_missing_set=False, tcspc_num_bins=None, tcspc_unit=None)
```

Load a .spc and (optionally) .set files for ns-ALEX and return 2 dict.

The first dictionary can be passed to the `phconvert.hdf5.save_photon_hdf5()` function to save the data in Photon-HDF5 format.

**Returns** the first contains the main photon data (timestamps, detectors, nanotime, ...); the second contains the raw data from the .set file (it can be saved in a user group in Photon-HDF5).

**Return type** Two dictionaries

```
phconvert.loader.nsalex_ht3(filename, donor=0, acceptor=1, alex_period_donor=(150,
                                   1500), alex_period_acceptor=(1540, 3050),
                             excitation_wavelengths=(5.23e-07, 6.28e-07),
                             detection_wavelengths=(5.8e-07, 6.8e-07))
```

Load a .ht3 file containing ns-ALEX data and return a dict.

This dictionary can be passed to the `phconvert.hdf5.save_photon_hdf5()` function to save the data in Photon-HDF5 format.

```
phconvert.loader.usalex_sm(filename, donor=0, acceptor=1, alex_period=4000, alex_offset=750,
                            alex_period_donor=(2850, 580), alex_period_acceptor=(930,
                                        2580), excitation_wavelengths=(5.32e-07, 6.35e-07),
                            detection_wavelengths=(5.8e-07, 6.8e-07), software='LabVIEW
                            Data Acquisition usALEX')
```

Load a .sm us-ALEX file and returns a dictionary.

This dictionary can be passed to the `phconvert.hdf5.save_photon_hdf5()` function to save the data in Photon-HDF5 format.

This module contains functions to load and decode files from PicoQuant hardware.

The three main functions to decode PTU, HT3 and PT3 files are respectively:

- `load_ptu()`
- `load_ht3()`
- `load_pt3()`

These functions return the arrays timestamps, detectors, nanotimes and an additional metadata dict.

Other lower level functions are:

- `ptu_reader()` which loads metadata and raw t3 records from PTU files
- `ht3_reader()` which loads metadata and raw t3 records from HT3 files
- `pt3_reader()` which loads metadata and raw t3 records from PT3 files
- `process_t3records()` which decodes the t3 records returning timestamps (after overflow correction), detectors and TCSPC nanotimes.

Note that the functions performing overflow/rollover correction can take advantage of numba, if installed, to significantly speed-up the processing.

## List of functions

High-level functions to load and decode several PicoQuant file formats:

`phconvert.pqreader.load_ht3(filename, ovfunc=None)`  
Load data from a PicoQuant .ht3 file.

### Parameters

- **filename** (*string*) – the path of the HT3 file to be loaded.

- **ovcfunc** (*function or None*) – function to use for overflow/rollover correction of timestamps. If None, it defaults to the fastest available implementation for the current machine.

**Returns** A tuple of timestamps, detectors, nanotimes (integer arrays) and a dictionary with metadata containing at least the keys ‘timestamps\_unit’ and ‘nanotimes\_unit’.

`phconvert.pqreader.load_pt3(filename, ovcfunc=None)`  
 Load data from a PicoQuant .pt3 file.

**Parameters**

- **filename** (*string*) – the path of the PT3 file to be loaded.
- **ovcfunc** (*function or None*) – function to use for overflow/rollover correction of timestamps. If None, it defaults to the fastest available implementation for the current machine.

**Returns** A tuple of timestamps, detectors, nanotimes (integer arrays) and a dictionary with metadata containing at least the keys ‘timestamps\_unit’ and ‘nanotimes\_unit’.

## Low-level functions

These functions are the building blocks for loading and decoding the different files:

`phconvert.pqreader.ht3_reader(filename)`  
 Load raw t3 records and metadata from an HT3 file.

`phconvert.pqreader.pt3_reader(filename)`  
 Load raw t3 records and metadata from a PT3 file.

`phconvert.pqreader.process_t3records(t3records, time_bit=10, dtime_bit=15, ch_bit=6, special_bit=True, ovcfunc=None)`  
 Extract the different fields from the raw t3records array.

The input array of t3records is an array of “records” (a C struct). It packs all the information of each detected photons. This function decodes the different fields and returns 3 arrays containing the timestamps (i.e. macro-time or number of sync, ns resolution), the nanotimes (i.e. the micro-time or TCSPC time, ps resolution) and the detectors.

Assuming the t3records are in little-endian order, the fields are assumed in the following order:

Optional special bit	detectors	nanotimes	timestamps	
MSB				LSB

- the lowest *time\_bit* bits contain the timestamps
- the next *dtime\_bit* bits contain the nanotimes
- the next *ch\_bit* contain the detector number
- if *special\_bit = True*, the highest bit is the special bit.

The returned timestamps are overflow-corrected, and therefore should be monotonically increasing. Each overflow event is marked by a special detector (or a special bit) and this information is used for the correction. These overflow “events” **are not removed** in the returned arrays resulting in spurious detectors. This choice has been made for safety (you can always go and check where there was an overflow) and for efficiency (removing a few elements requires allocating a new array that is potentially expensive for big data files). Under normal usage the additional detectors take negligible space and can be safely ignored.

**Parameters**

- **t3records** (*array*) – raw array of t3records as saved in the PicoQuant file.
- **time\_bit** (*int*) – number of bits in the t3record used for timestamps (or macro-time).
- **dtime\_bit** (*int*) – number of bits in the t3record used for the nanotime (TCSPC time or micro-time)
- **ch\_bit** (*int*) – number of bits in the t3record used for the detector number.
- **special\_bit** (*bool*) – if True the t3record contains a special bit for overflow correction. This special bit will become the MSB in the returned detectors array. If False, it assumes no special bit in the t3record.
- **ovcfunc** (*function or None*) – function to perform overflow correction of timestamps. If None use the default function. The default function is the numba-accelerated version if numba is installed otherwise it is function using plain numpy.

### Returns

A 3-element tuple containing the following 1D arrays (all of the same length):

- **timestamps** (*array of int64*): the macro-time (or number of sync) of each photons after overflow correction. Units are specified in the file header.
- **nanotimes** (*array of uint16*): the micro-time (TCSPC time), i.e. the time lag between the photon detection and the previous laser sync. Units (i.e. the bin width) are specified in the file header.
- **detectors** (*arrays of uint8*): detector number. When *special\_bit = True* the highest bit in *detectors* will be the special bit.





This module contains functions to load and decode files from Becker & Hickl hardware.

The high-level function in this module are:

- `load_spc()` which loads and decodes the photon data from SPC files.
- `load_set()` which returns a dictionary of metadata from SET files.

## Becker & Hickl SPC Format

The structure of the SPC format is here described.

### SPC-600/630

SPC-600/630 files have a record of 48-bit (6 bytes) in little endian (<) format. The first 6 bytes of the file are an header containing the *timestamps\_unit* (in 0.1ns units) in the two central bytes (i.e. bytes 2 and 3). In the following drawing each char represents 2 bits:

```

bit: 64          48          0
      0000 0000 XXXX XXXX XXXX XXXX XXXX XXXX
           '-----' '---' '---' '-----'
field names:      a      c      b      d

      0000 0000 XXXX XXXX XXXX XXXX XXXX XXXX
           '-----' '---' '---' '-----'
numpy dtype:      a      c      b      field0

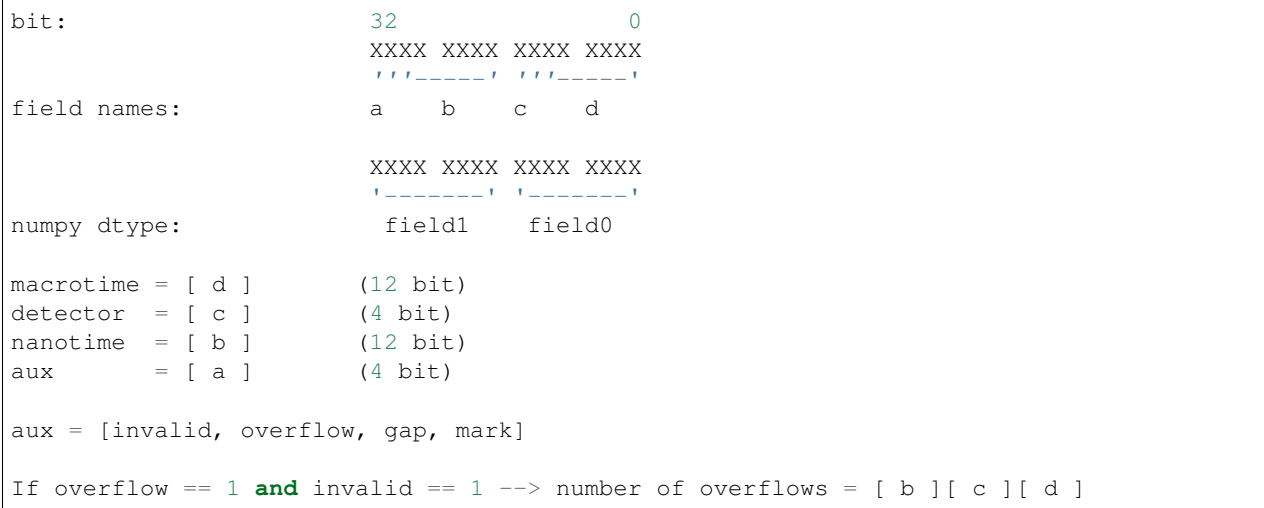
macrotime = [ b ] [ a ] (24 bit)
detector   = [ c ] (8 bit)
nanotime   = [ d ] (12 bit)

overflow bit: 13, bit_mask = 2^(13-1) = 4096

```

## SPC-134/144/154/830

SPC-134/144/154/830 files have a record of 32-bits (4 bytes) in little endian (<) format. The first 4 bytes of the file are an header containing the *timestamps\_unit* (in 0.1ns units) in first two bytes. In the following drawing each char represents 2 bits:



## List of functions

High-level functions to load and decode Becker & Hickl SPC/SET pair of files:

`phconvert.bhreader.load_spc(fname, spc_model='SPC-630')`

Load data from Becker & Hickl SPC files.

**Parameters** `spc_model` (*string*) – name of the board model. Valid values are 'SPC-630', 'SPC-134', 'SPC-144', 'SPC-154' and 'SPC-830'.

**Returns** 3 numpy arrays (timestamps, detector, nanotime) and a float (timestamps\_unit).

`phconvert.bhreader.load_set(fname_set)`

Return a dict with data from the Becker & Hickl .SET file.

## Low-level functions

These functions are the building blocks for decoding Becker & Hickl files:

`phconvert.bhreader.bh_set_identification(fname_set)`

Return a dict containing the IDENTIFICATION section of .SET files.

The both keys and values are native strings (binary strings on py2 and unicode strings on py3).

`phconvert.bhreader.bh_set_sys_params(fname_set)`

Return a dict containing the SYS\_PARAMS section of .SET files.

The keys are native strings (traditional strings on py2 and unicode strings on py3) while values are numerical type or byte strings.

`phconvert.bhreader.bh_decode(s)`

Replace code strings from .SET files with human readable label strings.

phconvert.bhreader.**bh\_print\_sys\_params** (*sys\_params*)  
Print a summary of the Becker & Hickl system parameters (.SET file).



## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



**p**

`phconvert.bhreader`, 13

`phconvert.hdf5`, 3

`phconvert.loader`, 7

`phconvert.pqreader`, 9





## A

assert\_valid\_photon\_hdf5() (in module phconvert.hdf5), 5

## B

bh\_decode() (in module phconvert.bhreader), 14

bh\_print\_sys\_params() (in module phconvert.bhreader), 14

bh\_set\_identification() (in module phconvert.bhreader), 14

bh\_set\_sys\_params() (in module phconvert.bhreader), 14

## D

dict\_from\_group() (in module phconvert.hdf5), 6

dict\_to\_group() (in module phconvert.hdf5), 6

## H

ht3\_reader() (in module phconvert.pqreader), 10

## L

load\_ht3() (in module phconvert.pqreader), 9

load\_pt3() (in module phconvert.pqreader), 10

load\_set() (in module phconvert.bhreader), 14

load\_spc() (in module phconvert.bhreader), 14

## N

nsalex\_bh() (in module phconvert.loader), 7

nsalex\_ht3() (in module phconvert.loader), 7

## P

phconvert.bhreader (module), 13

phconvert.hdf5 (module), 3

phconvert.loader (module), 7

phconvert.pqreader (module), 9

print\_attrs() (in module phconvert.hdf5), 6

print\_children() (in module phconvert.hdf5), 6

process\_t3records() (in module phconvert.pqreader), 10

pt3\_reader() (in module phconvert.pqreader), 10

## S

save\_photon\_hdf5() (in module phconvert.hdf5), 3

## U

usalex\_sm() (in module phconvert.loader), 7