User Manual for LabVIEW Drivers Controlling Becker & Hickl SPC Modules

VERSION 1.0

September 2021

Content

.... 1
Introduction 7
Getting started 8
Advanced Package 10
Example VIs 10
SPC Communication 10
Intro-CheckConnection.vi 10
Intro-ReadRatesFromModules.vi 10
HW Histogram 10
HWhist_SimpleSingleCurve.vi 10
HWhist_SingleCurve.vi 11
HWhist_SingleCurveMultiModule.vi 11
HWhist_TimeSeriesStrictTiming.vi 11
HWhist_SimpleImageAcquisition-ScanSyncIn.vi 11
PC Memory 12
EventStream-GetPhotonInfo.vi 12
EventStream-StoreToSPC.vi 12
EventStream-Oscillloscope.vi 12
ImagingEventStream-SimpleFifoImage.vi 13
ImagingEventStream-FifoImage.vi 13
RemoteControlFifoImage.vi 13
Data Conversion 14
Data-ConvertSPCtoPhotInfo.vi 14
SubVIs for examples: 14
AskUserForModules.vi 14
GetDLLPath.vi 14
UserInput2SPCDataCluster.vi 14
UserInputMultiModule2SPCDataCluster.vi 15
SPCDataCluster2UserInput.vi 15
SPCDataCluster2TACAll&SysParPerModule.vi 15
EventDataToPhotonInfos.vi 15
CheckAndGetFifoData.vi 16
CollectDataAndSave.vi 16
ConvertModuleTypes.vi 16
CreateFixedLengthString.vi 17

SaveToLVSet.vi 17
LoadFromLVSet.vi 17
FalseColorIntensityAxis.vi 17
ChangeFileExtension.vi 17
ControlFifoImage.vi 18
GetControlRefnum.vi 18
WindowsProcesses.vi 18
File I/O VIs 18
PH Files: 18
PhRead.vi 18
PhWrite.vi 19
SDT Files: 19
SDT_ReadFile.vi 19
SDT_ReadSetUpBlock.vi 19
SDT_ReadMeasureInfoBlock.vi 19
SDT_ReadFileBlock.vi 20
SDT_DecompressData.vi 20
SDT_WriteFile.vi 20
SDT_Generate_bhfile_header.vi 21
SDT_GenerateInfoBlock.vi 21
SDT_GenerateMeasureInfoBlock.vi 21
SDT_GenerateBHFileBlockHeader.vi 22
SDT_MeasureBlockToString.vi 22
SDT_CreateModTypeString.vi 22
SDT_AddCheckSum.vi 22
SPC Files: 22
SPCReadFile.vi 23
SPCGetTimeScaleFromSETfile.vi 23
SPCGetSetUpParameter.vi 23
SPCWriteFile.vi 24
SPCWriteSETforSPC.vi 24
SPCCreateSetUpAsciiString.vi 24
SETParametersToSPCdata.vi 24
Low-Level wrappers 25
Initialisation functions 26
Initialize.vi 26

GetDLLPath.vi 27

SPC_get_init_status.vi 27

SPC_get_module_info.vi 28

SPC_test_id.vi 28

SPC_set_mode.vi 30

SPC_get_mode.vi 30

SPC_close.vi 31

SPC_get_error_string.vi 31

Setup functions: 31

SPC_get_parameters.vi 32

SPC_set_parameters.vi 40

SPC_get_parameter.vi 40

SPC_set_parameter.vi 40

SPC_get_eeprom_data.vi 41

SPC_get_adjust_parameters.vi 41

SPC_set_adjust_parameters.vi 41

SPC_read_parameters_from_inifile.vi 42

SPC_save_parameters_to_inifile.vi 42

SetTrigger.vi 43

Status functions: 43

SPC_test_state.vi 43

SPC_get_sync_state.vi 45

SPC_get_time_from_start.vi 45

SPC_get_break_time.vi 45

SPC_get_actual_coltime.vi 46

SPC_read_rates.vi 46

SPC_clear_rates.vi 47

SPC_get_sequencer_state.vi 47

SPC_read_gap_time.vi 47

SPC_get_scan_clk_state.vi 48

SPC_get_fifo_usage.vi 48

CheckState.vi 49

FindModulesInUse.vi 49

TestFillState.vi 49

Measurement control functions: 49

SPC_start_measurement.vi 50

SPC_pause_measurement.vi 50
SPC_restart_measurement.vi 51
SPC_stop_measurement.vi 52
SPC_set_page.vi 53
SPC_enable_sequencer.vi 53
Memory transfer functions: 54
SPC_configure_memory.vi 55
SPC_fill_memory.vi 57
SPC_read_data_block.vi 57
SPC_write_data_block.vi 58
SPC_read_fifo.vi 59
SPC_read_data_frame.vi 60
SPC_read_data_page.vi 61
SPC_read_block.vi 62
SPC_save_data_to_sdtfile.vi 63
ConfigFIFO.vi 65
AcqData.vi 65
EventStreamToDecayCurve.vi 65
PopulateSingleFrame.vi 66
Functions to manage photons streams: 66
SPC_init_phot_stream.vi 66
SPC_get_phot_stream_info.vi 68
SPC_get_photon.vi 68
SPC_close_phot_stream.vi 68
SPC_get_fifo_init_vars.vi 69
SPC_init_buf_stream.vi 69
SPC_add_data_to_stream.vi 72
SPC_read_fifo_to_stream.vi 72
SPC_get_photons_from_stream.vi 73
SPC_stream_start_condition.vi 74
SPC_stream_stop_condition.vi 75
SPC_stream_reset.vi 75
SPC_get_stream_buffer_size.vi 76
SPC_get_buffer_from_stream.vi 76
Advanced Package VIs 77
Advanced Application Vis 77

ImagingEventStream-FifoImage.vi ... 77
BH_EventStream-GetPhotonInfo.vi ... 78
BH_HWhist-SingleCurve.vi ... 78
PopulateDecayMatrix.vi ... 78
ScanCFDLimitLow.vi ... 78
DecayOverTimeShift.vi ... 78
Ini2SPCDataCluster.vi ... 78
SPCDataCluster2UserInput_WithCFG.vi ... 78
About.vi and BHAboutLogo.ctl ... 79
Functions folder ... 79
PieChart folder ... 79
DataAnalysis folder ... 79
Ini-Files folder ... 79
builds folder ... 79
Appendix ... 80
End-User License Agreement (EULA) of SPC-LVD ... 80
License Grant ... 80
Intellectual Property and Ownership ... 81
Termination ... 81
Governing Law ... 81

Introduction

Thank you for working with Becker&Hickl's SPC modules.

You are already familiar with B&H's own control software Spcm.exe. There you have the full control over all different module types and their operation modes.

Nevertheless, you feel that there is an important feature missing to bring your application to its expected heights.

B&H honours that and is offering you software drivers capable to control your SPC module down to the lowest level.

These drivers are written in National Instruments LabWindows / CVI and available on your computer at C:\Program Files (x86)\BH\SPCM\DLL\spcmXX.dll (XX standing for your operating system's bitness).

For programmer wanting to use National Instruments LabVIEW to control their SPC modules, B&H provides LV wrappers in the Becker&Hickl SPC Modules folder to call the above mentioned dll.

Also included are some higher-level convenience VIs as well as some simple examples to provide you with a starting point to your own LV-project.

Not included is the block diagram of the high level, fully operational compiled application.

This manual is about the use of these wrappers and examples, aimed at users with a solid understanding of programming technics in LabVIEW.

For training in LabVIEW, itself we recommend to contact National Instruments directly at *www.ni.com*.

Becker&Hickl suggests to start your own SPC module control development by having a look and running the examples provided in the Examples folder. They are a collection of executable simple LabVIEW VIs focused on different operation modes and SPC related tasks.

You will also find VIs to handle B&H's proprietary file formats *.sdt and *.spc. This will allow you to exchange measurement data with B&H's Spcm.exe in both directions. Do a measurement in Spcm.exe and write your own analysis software, or take a measurement with a LabVIEW application and visualize the results in Spcm.exe.

All examples and file I/O VIs are executable and have been tested on 150, 160 and 180 type modules. For older type modules some modification might be necessary.

Note: Becker&Hickl takes no responsibility for the use of the example VIs, nor is there any kind of support included for this VI collection.

Your modules should always be tested with our house application Spcm.exe

Getting started

The Becker&Hickl SPC Modules LabVIEW project comes as collection of files in a folders hierarchy installed together with your Spcm.exe software at C:\Program Files (x86)\BH\SPCM\BH-LabVIEW-SPC.

The VIs and the project are written in the LabVIEW2017 development environment. Upgrading to a newer development environment on your PC should not be a problem. For downgrading to older versions use the File>Save for Previous Version… option in the LV project window or get in contact with Becker&Hickl.

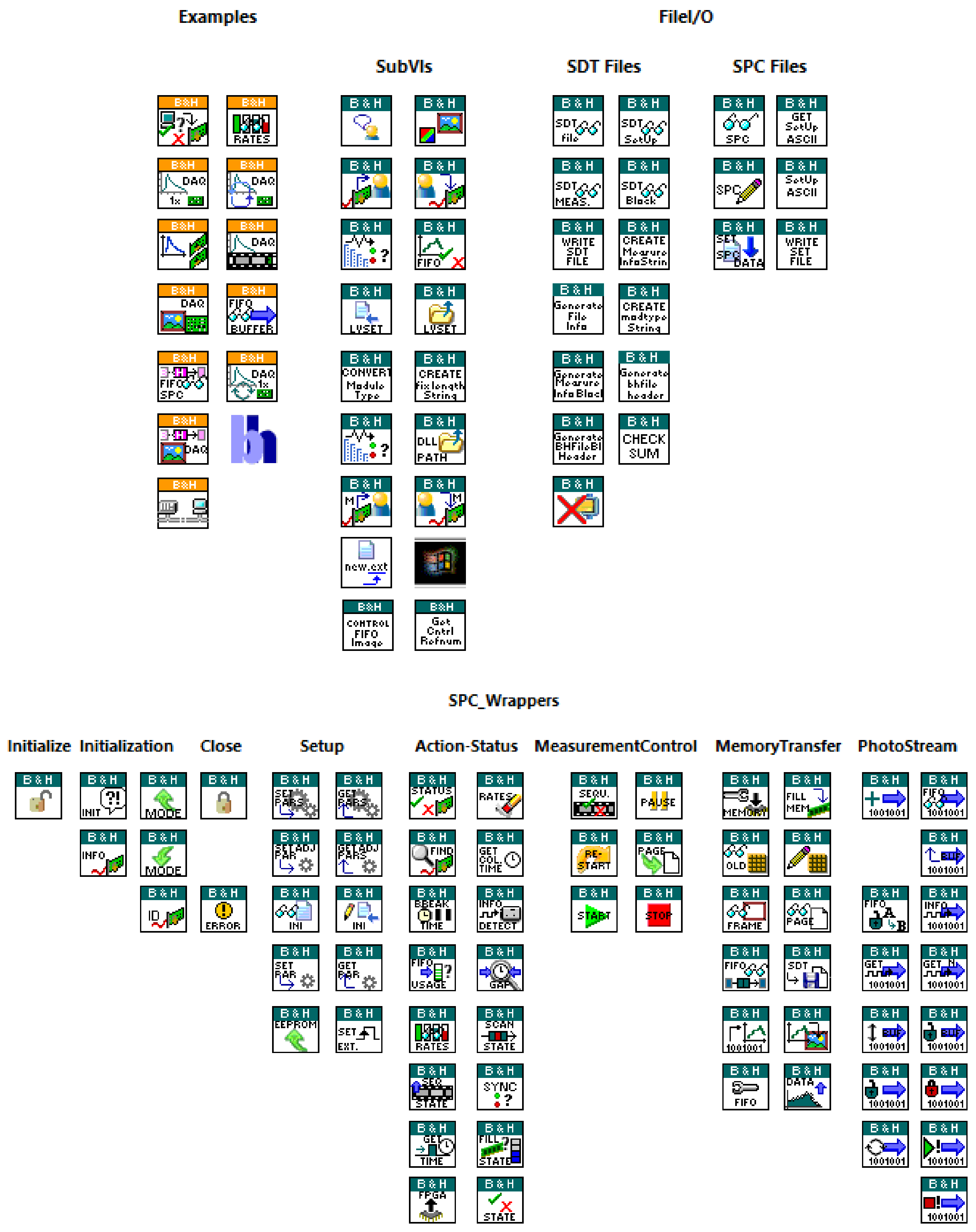
All LV-wrapper VIs in the \SPC_Wrappers folder depend on the spcmXX.dll located at C:\Program Files (x86)\BH\SPCM\DLL on your computer. There are 3 versions available. You will need to use **spcm32.dll** on Windows32 bit systems. For Windows64 bit systems you will use the **spcm32x64.dll** when programming in the LabVIEW 32bit environment or **spcm64.dll** when running LabVIEW 64bit.

We recommend to always upgrade your Spcm.exe to the newest version which will include new features and bug fixes.
With every new release there will be new versions of the spcmXX.dll. Since B&H reserves the right to change (with release notes) functions and their parameter sets at any time, we also recommend not to edit the wrapper VIs directly but keep them in their original form at their original location. Like this an upgrade of SPCM can overwrite the wrapper VIs without changing your code.

To start your own LabVIEW project, you should create a folder on your User\My Documents path, create a New Project on the LabVIEW start-up window. In the project window right click on My computer and choose Add>Folder (Auto-populating)… In the file select box navigate to C:\Program Files (x86)\BH\SPCM\BH-LabVIEW-SPC\ and click Select Folder.

You could do the same for the Examples folder, but we recommend to create a copy of this folder on your development path. You most likely will do changes on these VIs which you might not be allowed to do on the protected path of C:\Program File (x86). They also might be overwritten with newer Spcm.exe version distribution.

Becker & Hickl GmbH - SPC LabVIEW Drivers



For an overview over all VIs included in this package have a look at the block diagram of **\VI_Tree.vi.**

Advanced Package

Upon request Becker&Hickl distributes an advanced BH-LabVIEW-SPC-Full version package against a license fee.
The end-user has to agree to the EULA-license printed in the Appendix of this document.

Example VIs

When attempting to write a SPC control application we suggest to start off with one of our examples located in the \Examples folder.

Each example focuses on a specific operation mode or measurement data retrieval. Pick the one that closest matches your setup.

Of course, the first step should be to verify an error free communication with your SPC module. You can use **Intro-CheckConnection.vi** and **Intro-ReadRatesFromModules.vi** to do so.

Some examples are based on C examples located in **use_spcm.c**. There you might find more inspiration for your specific use-case.

For an introduction to the SPC operating modes, memory management and programming suggestions have a look at **spcmdll.pdf**.

SPC Communication

Intro-CheckConnection.vi

This example demonstrates how to establish a connection to SPC modules.

Intro-ReadRatesFromModules.vi

This example demonstrates how to get correct values from the initialised and selected SPC modules for different rates like the sync rate, CFD rate, TAC rate as well as ADC rate.

HW Histogram

HWhist_SimpleSingleCurve.vi

This example demonstrates how to use the Becker&Hickl LabVIEW wrapper VIs to acquire one single decay curve from one SPC module.

HWhist_SingleCurve.vi

This example demonstrates how to use the Becker&Hickl LabVIEW wrapper VIs to acquire single decay curves in a continuous loop.

The SPC parameters can be adjusted during the run. The rates and status flags are continuously updated.

It is also possible to save the acquired data to an SDT or CSV file.

HWhist_SingleCurveMultiModule.vi

This example demonstrates how to use the Becker&Hickl LabVIEW wrapper VIs to acquire single decay curves from multiple modules in a continuous loop.

The SPC parameters can be adjusted during the run. The rates and status flags are continuously updated.

It is also possible to save the acquired data to an SDT or CSV file.

HWhist_TimeSeriesStrictTiming.vi

This example demonstrates how to use the Becker&Hickl LabVIEW wrapper VIs to acquire decay curves in continuous flow mode using the sequencer and swapping banks.

Multiple modules are covered in a parallelized for loop.

Each bank's decay curves are saved to an SDT-file on disc.

The SPC parameters can be adjusted for every new run. The rates and status flags are continuously updated.

This VI is based on the C-example part 1 in **use_spcm.c**.

HWhist_SimpleImageAcquisition-ScanSyncIn.vi

This example demonstrates how to use the Becker&Hickl LabVIEW wrapper VIs to acquire one page of data in ScanSyncIn mode. The data is displayed as Sum of Decays and Intensity Image.

Here all parameters are hard wired. This VI is based on the C-example part 3 in **use_spcm.c**.

PC Memory

EventStream-GetPhotonInfo.vi

This example demonstrates how to use the Becker&Hickl LabVIEW wrapper VIs to acquire a decay curve using the FIFO Stream mode.

A measurement is stopped either when accumulating the desired number of photons or the collection time expired.

The SPC parameters can be adjusted during the run. The rates and status flags are continuously updated.

This VI is based on the C-example part 5 in **use_spcm.c**.

EventStream-StoreToSPC.vi

This example demonstrates how to use the Becker&Hickl LabVIEW wrapper VIs to acquire a decay curve using the FIFO mode.

A measurement is stopped either when accumulating the desired number of photons or the collection time expired.

After the measurement the EventData is interpreted as decay curve and saved to a SPC-file readable by Spcm.exe.

The rates and status flags are continuously updated.

Here the parameters can’t be updated during a run.

EventStream-Oscillloscope.vi

This example demonstrates how to use the Becker&Hickl LabVIEW wrapper VIs to acquire continuously decay curves in Fifo mode and display them on a timeline.

The Intensity TimeLine display could be used as guide for signal improving.

The SPC parameters can be adjusted during the run. The rates and status flags are continuously updated.

ImagingEventStream-SimpleFifoImage.vi

This example demonstrates how to use the Becker&Hickl LabVIEW wrapper VIs to acquire a single image in Fifo-Imaging mode.

The Event Data is interpreted and populates a 4D DecayMatrix which is then saved as SDT-file to disc.

ImagingEventStream-FifoImage.vi

The Block Diagram of this VI is not included in the collection. It is obtainable separately at Becker&Hickl.

The compiled executable FifoImage.exe demonstrates the full power of LabVIEW when controlling the SPC modules. Improved memory management allows for Fifo-Imaging recording of images up to 4096 x 4096 pixel with ADC-resolution of 64 or alternatively 512 x 512pixel with ADC-resolution of 4096. This is close to the LV limit for array sizes of 2^31.

This VI is properly LV-Event driven and the GUI experience is optimized. It can be remotely controlled.

If you are looking for a tested LabVIEW application to just hook in your personal analysis algorithm or display format, this would be your starting point.

RemoteControlFifoImage.vi

This VI demonstrates how a Becker&Hickl compiled application can be remotely controlled by using the LabVIEW VI-Server.

FifoImage.exe has opened a port at 3366. The functional global variable \Examples\SubVIs\ControlFifoImage.vi is obtaining and keeping references to selected front panel controls in FifoImage.exe and allows read and write functionality on those.

Remote controlling can be done from the LabVIEW development environment or a compiled application. With slight modifications controlling can be done over a network.

Data Conversion

Data-ConvertSPCtoPhotInfo.vi

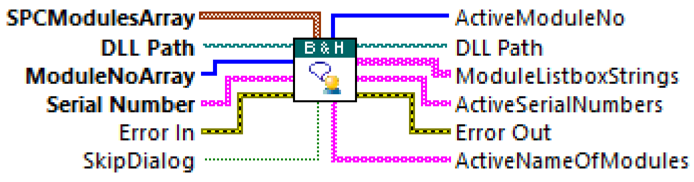
This VI demonstrates how a *.spc file filled with raw event data from a SPC_read_fifo call can be converted to a PhotInfo-cluster (structure) and written to a file in the Becker&Hickl propriety format *.ph.

SubVIs for Examples:

\Examples\SubVIs

These VIs are used by the included examples VIs. Their study might also be of interest for LV-programmer using the LV-wrappers directly for their application.

AskUserForModules.vi



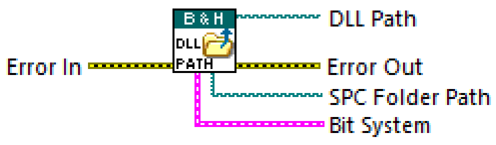
This VI presents the user with a dialog displaying all available modules with type and serial number information.

One or several modules can be chosen to continue with the application flow.

The dialog can be skipped if only one module is present or the choice already known.

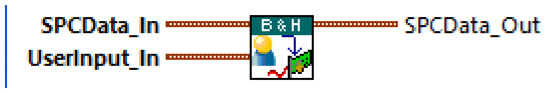
The input values are provided by the **Initialize.vi**.

GetDLLPath.vi



All wrapper VIs for spcmdll calls need the file path to the dll. This VI provides this path according to the operation systems bitness and the LabVIEW bitness.

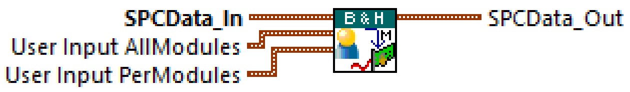
UserInput2SPCDataCluster.vi



Only a small portion of all parameters in SPCData need adjustment in a specified measurement environment.

This VI transfers these data from UserInput cluster into SPCData cluster.

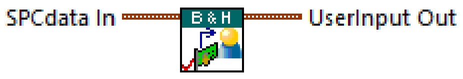
UserInputMultiModule2SPCDataCluster.vi



Only a small portion of all parameters in SPCData need adjustment in a specified measurement environment.

This VI transfers these data from UserInput All Modules and UserInput Per Modules cluster into SPCData cluster.

SPCDataCluster2UserInput.vi

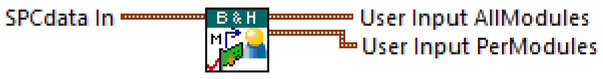


Only a small portion of all parameters in SPCData need adjustment in a specified measurement environment.

This VI transfers these data from SPCData cluster into UserInput cluster.

Use this VI to display the actual parameters accepted by the modules.

SPCDataCluster2TACAll&SysParPerModule.vi

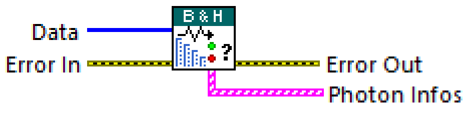


Only a small portion of all parameters in SPCData need adjustment in a specified measurement environment.

This VI transfers these data from the SPCData cluster into the UserInput All Modules and UserInput PerModules cluster.

Use this VI to display the actual parameters accepted by the modules.

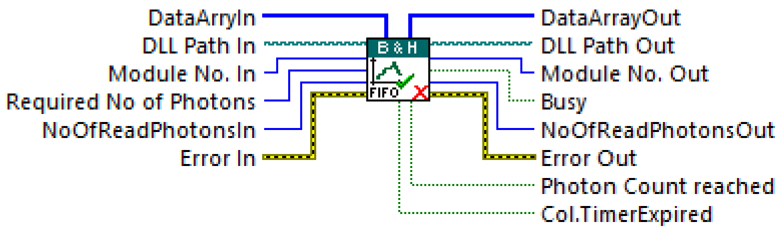
EventDataToPhotonInfos.vi



This VI analyses a given photon data array for photon information like micro time, macro time, marker information, photon validity, macro time overflow and routing channel number.

Note, that the input data needs to be in U32 form in the right word order. (Reversed to the **SPC_read_fifo.vi** Data Buffer output.)

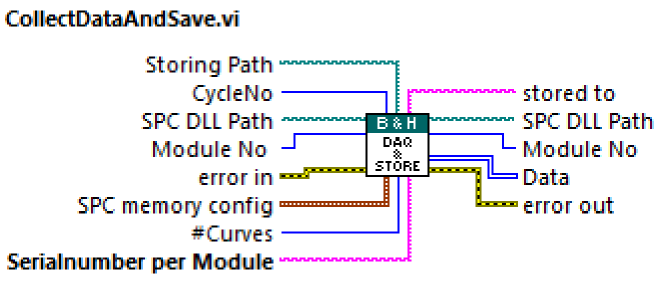
CheckAndGetFifoData.vi



This convenience VIs wraps **CheckState.vi** and **SPC_read_fifo.vi** in one VI to provide a clearer code.

Attention: this VI runs Inline to avoid memory allocation of SubVi calls.

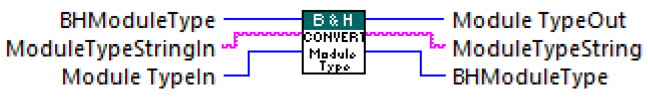
CollectDataAndSave.vi



This convenience VIs wraps **SPC_read_data_page.vi** and **SPC_save_data_to_sdtfile.vi** in one VI to provide a clearer code.

Note: This VI is set to execute as preallocated clone to improve performance when run in parallel loops for multi module setups.

ConvertModuleTypes.vi

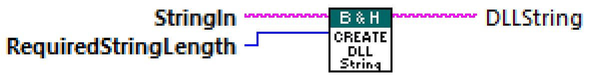


The module type can be coded in spcmdll in three different styles. This convenience VI converts each of these styles into the other two.

Warning: Only ever connect one of the input terminals.

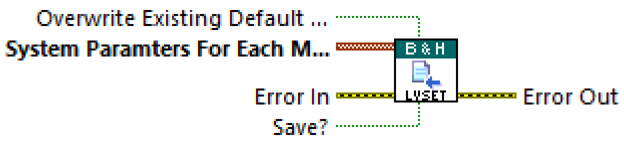
Use this VI when providing information to an SDT- or SET-file.

CreateFixedLengthString.vi



Use this convenience VI when providing a string of a certain length (filled with 0x00) is required. Typically, when writing an SDT-file.

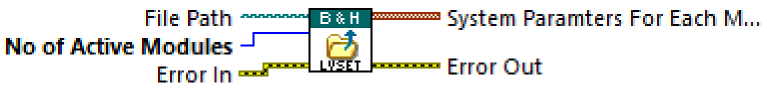
SaveToLVSet.vi


Overwrite Existing Default ...
System Paramters For Each M...
Error In
Save?
B & H
LVSET
Error Out

This convenience VI stores the given system parameters into a LVSET file. The default file which is also used as the file to store the last used settings for each module is located in "C:\BHdata\SPCM\SPCDefaultVal.lvset".

The user has the choice to overwrite this file or to use another file location to store the settings.

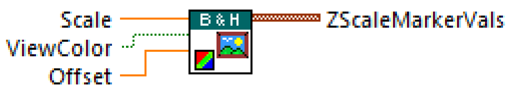
LoadFromLVSet.vi



This convenience VI loads the user setting parameter such as TAC range, Sync Threshold and CFD Limit Low from a given LVSET file.

Is no file specified the default file will be used, located at: "C:\BHdata\SPCM\SPCDefaultVal.lvset"

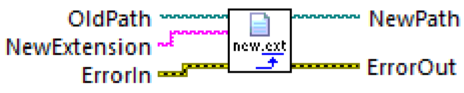
FalseColorIntensityAxis.vi



This convenience VI creates the colour markers for an intensity graph in rainbow style (blue to red).

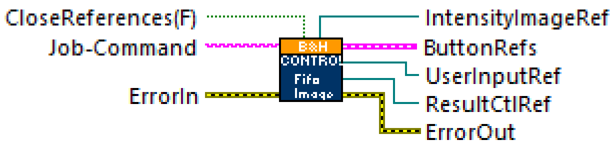
Use this VI to rescale the Z-axis in an Intensity image.

ChangeFileExtension.vi



A convenience VI simplifying the process of changing a file extension.

ControlFifoImage.vi



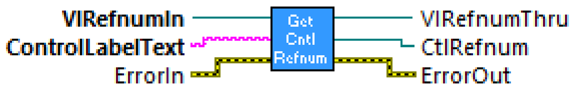
This VI searches and provides refnums to controls/indicators in Examples**ImagingEventStream-FifoImage.vi**.

Use this VI to remotely control FifoImage.exe from your own VI.

An example use scenario is shown in Examples**RemoteControlFifoImage.vi**

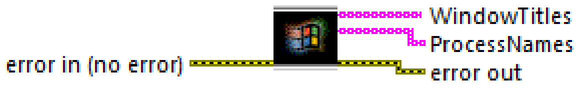
FifoImage.exe opens Port 3366

GetControlRefnum.vi



Gets the Refnum for a control or indicator on the FrontPanel, its Clusters or TabControls of a VI by VIRefnumIn.

WindowsProcesses.vi



Returns WindowTitles and lower case(!) ProcessNames of all running Processes.

File I/O VIs

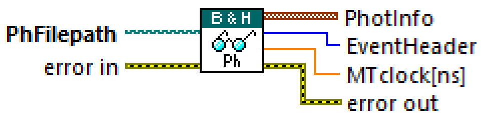
These VIs will assist you in reading and writing B&H propriety file formats *.sdt and *.spc.

Note, that there is a low-level wrapper VI \SPC_Wrappers\MemoryTransfer\SPC_save_data_to_sdtfile.vi for writing single decay curves to file.

PH Files:

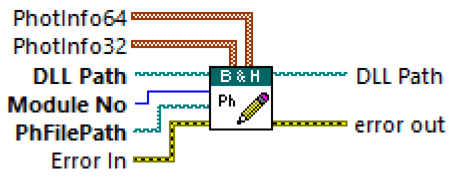
\FileIO\PH

PhRead.vi



This Vi demonstrates how a *.ph file in the Becker&Hickl propriety format can be read to a PhotInfo-cluster (structure).

PhWrite.vi



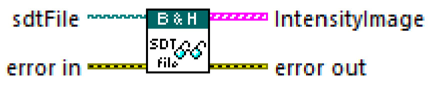
This VI demonstrates how a PhotInfo-cluster (structure) can be written to a *.ph file in the Becker&Hickl propriety format readable by SPCM.exe

Both cluster types (PhotInfo32 and PhotInfo64) are supported.

SDT Files:

\FileIO\SDT

SDT_ReadFile.vi

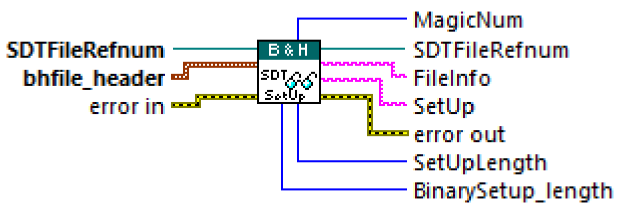


This VI reads a SDT-file from disc. The header and block data are interpreted and displayed. The usage and information in the graph depend on the type of data stored in the SDT-file.

For a description of the *.sdt file format see: **SPC_data_file_structure.h**

Use this VI to display data stored previously in an SDT-file or to create your own analysis on this data.

SDT_ReadSetUpBlock.vi



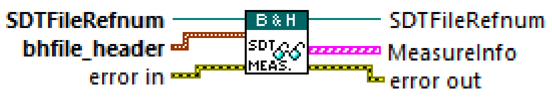
This VI is a SubVI to **SDT_ReadFile.vi**.

It reads the ASCII- and first part of the binary-SetUp block in an SDT-file from disc.

See also a different approach at \FileIO\SPC****SPCGetTimeScaleFromSETfile.vi.

For a description of the *.sdt file format see: **SPC_data_file_structure.h**

SDT_ReadMeasureInfoBlock.vi

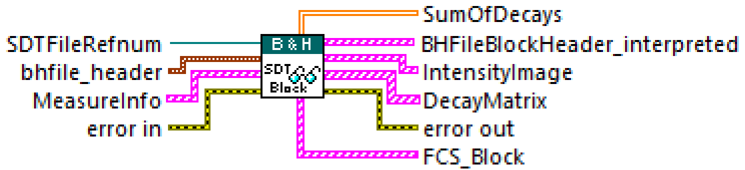


This VI is a SubVI to **SDT_ReadFile.vi**.

It reads the MeasureInfo block in an SDT-file from disc.

For a description of the *.sdt file format see: **SPC_data_file_structure.h**

SDT_ReadFileBlock.vi

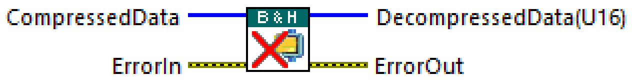


This VI is a SubVI to **SDT_ReadFile.vi**.

It reads the BHFileBlockHeader and data blocks in an SDT-file from disc.

For a description of the *.sdt file format see: **SPC_data_file_structure.h**

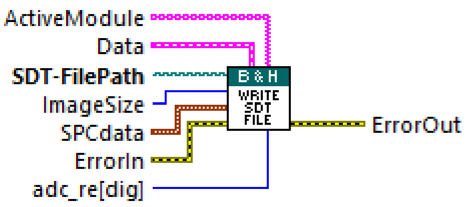
SDT_DecompressData.vi



Use this VI to decompress ZIPed data blocks in SDT-files. We utilize LV’s own **Unzip.vi** but need to write the data block to disc for that.

Special care needs to be taken not to decompress the data into an array larger than x8000 0000 = d2.147.483.648 which is the LV array’s largest index value. (I32)

SDT_WriteFile.vi



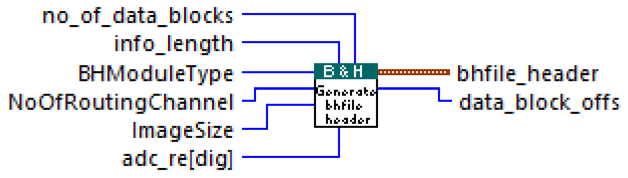
This VI writes an SDT-file to disc. The SetUp and the MeasureInfo blocks are created and the appended with the data blocks.

This VI is limited to writing a DecayMatrix from a FIFO-Image recording and optional an IntensityImage to file. For single mode recordings use the spcmdll VI \SPC_Wrappers\MemoryTransfer\SPC_save_data_to_sdtfile.vi.

For a description of the format for the Data input cluster to this VI see the usage in \Examples**ImagingEventStream-SimpleFifoImage.vi**. Leave either of the cluster input arrays empty to not save them to disc.

For a description of the *.sdt file format see: **SPC_data_file_structure.h**

SDT_Generate_bhfile_header.vi

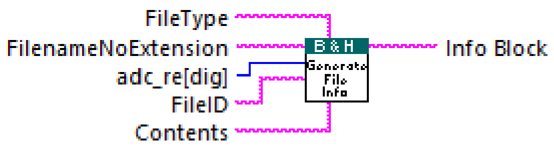


This VI is a SubVI to **SDT_WriteFile.vi**.

It populates the bhfile_header cluster ready to be written to file.

For a description of the *.sdt file format see: **SPC_data_file_structure.h**

SDT_GenerateInfoBlock.vi

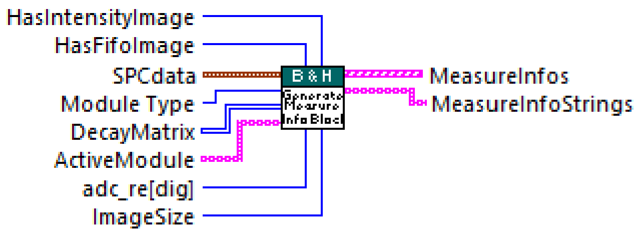


This VI is a SubVI to **SDT_WriteFile.vi**. and **SPCWriteSETfoSPC.vi**.

It formats the Info Block ready to be written to file.

For a description of the *.sdt file format see: **SPC_data_file_structure.h**

SDT_GenerateMeasureInfoBlock.vi

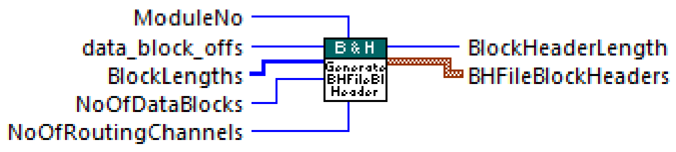


This VI is a SubVI to **SDT_WriteFile.vi**.

It populates the arrays of MeasureInfos cluster and generates strings ready to be written to file.

For a description of the *.sdt file format see: **SPC_data_file_structure.h**

SDT_GenerateBHFileBlockHeader.vi

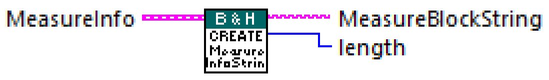


This VI is a SubVI to **SDT_WriteFile.vi**.

It populates the clusters BHFileBlockHeader ready to be written to file.

For a description of the *.sdt file format see: **SPC_data_file_structure.h**

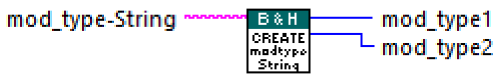
SDT_MeasureBlockToString.vi



This VI is a SubVI to **SDT_GenerateMeasureInfoBlock.vi**. It formats the cluster MeasureInfo to a string ready to be written to file.

Particular care is taken to format MeasureInfo's strings to the correct length.

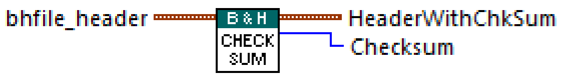
SDT_CreateModTypeString.vi



This VI is a SubVI to **SDT_GenerateMeasureInfoBlock.vi**.

It formats the ASCII-string describing the module type to two U64 binary integers.to be inserted at the MeasureInfo cluster's parameters 'MeasureInfo_PartB.mod_type1' and 'MeasureInfo_PartB.mod_type2'.

SDT_AddCheckSum.vi



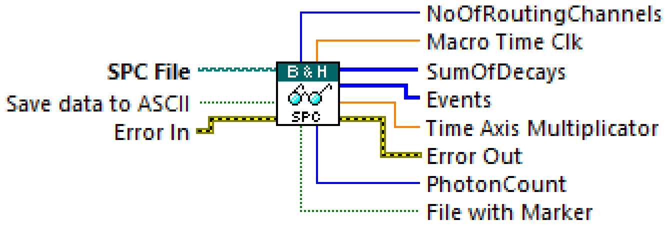
Use this VI to calculate and insert the correct U16 chksum into the bhfile_header cluster when writing an SDT- or SET-file.

The sum over all bhfile_header's words needs to add up to x55AA with overflow ignored.

SPC Files:

\FileIO\SPC Files

SPCReadFile.vi



This VI reads a SPC-/SET-file pair from disc. The header and event data are interpreted and displayed. The graph's Time Axis is scaled using the information stored in the SET-file.

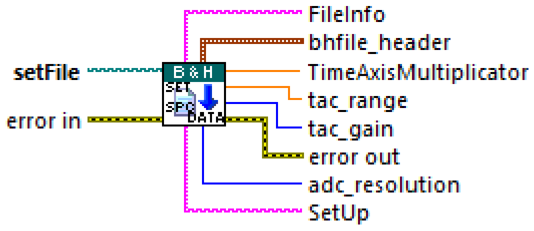
Optional the Events can be written to a *.csv file.

Note: Events are always 2 word wide, and received in reversed byte and word order.

For a description of the *.set file format see: **SPC_data_file_structure.h**

Use this VI to display data stored previously in a SPC-/SET-file pair or to create your own analysis on this data.

SPCGetTimeScaleFromSETfile.vi

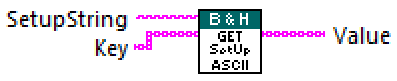


This VI demonstrates how parameters can be retrieved from a SDT- or SET-file's SetUp-block.

See also a different approach at \FileIO\SDT\SDT_ReadSetUpBlock.vi.

For a description of the *.set file format see: **SPC_data_file_structure.h**

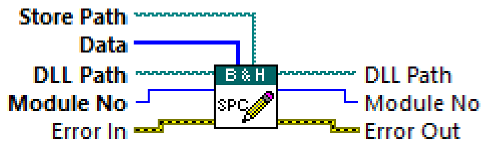
SPCGetSetUpParameter.vi



This VI demonstrates how specific parameters can be retrieved from an ASCII SetUp-block.

Provide the Key up to and including the format specifier and trailing comma.

SPCWriteFile.vi

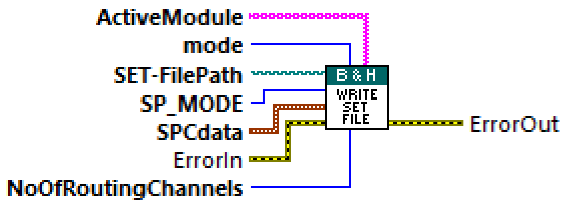


This VI demonstrates how an Event Data stream retrieved by \Public\Configure\Memory Transfer**SPC_read_fifo.vi** can be written to a SPC-file.

The required four byte header can be obtained using the \SPC_Wrappers\PhotoStream\SPC_get_fifo_init_vars.vi.

Note: In order to be readable by Spcm.exe SPC-files need to be accompanied by a corresponding SET-file.

SPCWriteSETforSPC.vi

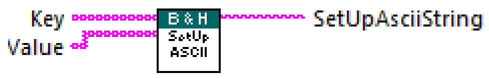


This VI demonstrates how a SET-file as pair for a corresponding SPC-file can be generated and written to disc.

Note: In order to be readable by Spcm.exe SPC-files need to be accompanied by a corresponding SET-file.

For a description of the *.set file format see: **SPC_data_file_structure.h**

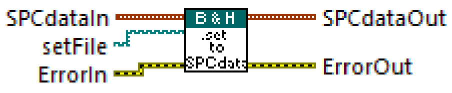
SPCCreateSetUpAsciiString.vi



This VI demonstrates how specific parameters can be entered into an ASCII SetUp-block replacing their default values.

Provide the Key up to and including the format specifier and trailing comma.

SETParametersToSPCdata.vi



This VI aids to read a full set of system parameters from a .set-file or the header of a .sdt into the SPCdata cluster. Usefull when wanting to copy all parameters established in SPCM.exe to the LV-environment.

Note, that some parameters like **mode** are not mapped. To correctly set these parameters use SPC_set- and get_parameters.vi before calling this VI.

Low-Level Wrappers

Low-level wrappers allow the LabVIEW programmer to call C-procedures provided in the spcm.dll corresponding to your systems bitness. These C-procedures are interacting with and collecting data from the SPC modules present at your computer.

You can use the low-level wrappers to create your own control and/or data acquisition application in LabVIEW. Allowing you for example to create specific analysis methods for your setup, create your own visualization tool or do automation tasks.

You need to provide the path to the spcm.dll to all low-level wrapper VIs. The default path to the correct bitness dll can be obtain through the **GetDLLPath.vi**

All spcm.dll C-procedures return an error status. Return values >= 0 indicate error free

execution. A value < 0 indicates that an error occurred during execution.

The low-level wrapper VIs include the **SPC_get_error_string.vi** which retrieves the string explanation for an error code and inserts it to the LV error cluster.

Included in this section are also some medium-level ‘convenience’ VIs. They will assist the LV programmer in properly setting up parameters for the low-level wrappers, or do the file I/O with *.sdt or ‘.spc files.

For examples on how to use these low-level wrappers have a look at the Examples section of this manual.

Calls to C-procedures in spcm.dll strongly rely on C structures and defines. In the LV environment Type Defs of clusters and enums are provided in the SPC_Wrappers\SPC_Controls folder.

Although the conversion task between the C and LV environment has been done already we add here a list that might be interesting to LV users.

Type in C programs	Type in LabView
char	signed 8-bit integer, byte (I8)
unsigned char	unsigned 8-bit integer, unsigned byte (U8)
short	signed 16-bit integer, word (I16)
unsigned short	unsigned 16-bit integer, unsigned word (U16)
long, int	signed 32-bit integer, long (I32)
unsigned long, int	unsigned 32-bit integer, unsigned long (U32)
__int64	signed 64-bit integer, quad (I64)
unsigned __int64	unsigned 64-bit integer, unsigned quad (U64)
float	4-byte single, single precision (SGL)
double	8-byte double, double precision (DBL)
char *	C string pointer
float *	Pointer to Value (Numeric, 4-byte single)

Naming convention:

All LV-wrapper that are calling a C-procedure directly are named identically to the C-procedure, that is SPC_***.vi.

All other convenience VIs or examples don't use the SPC_ prefix.

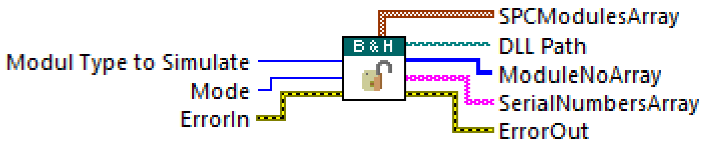
The now following VI descriptions are based on the description for the C-procedures found in **spcmdll.pdf**.

LV-programmer might also want to familiarize themselves with information provided in the files **spcm_def.h** (Include file containing Types definitions and Functions Prototypes) and **use_spcm.c** (Simple examples of using SPC DLL functions for SPC modules).

Initialisation Functions

\SPC_Wrappers\Initialisation

Initialize.vi



This is a convenience VI combining several spcmdll calls all needed to successfully initialize communication to SPC modules.

Before a measurement is started the measurement parameter values must be written into the internal structures of the DLL functions (not directly visible from the user program) and sent to the control registers of the SPC module(s). This is accomplished by the function **SPC_init**. The function

- checks whether DLL is correctly registered
- reads the parameter values from the specified file ini_file
- checks and recalculates the parameters depending on hardware restrictions and adjusts parameters from the EEPROM on the SPC module(s)
- sends the parameter values to the internal structures of the DLL
- sends the parameter values to the SPC control registers
- performs a hardware test of the SPC module(s)

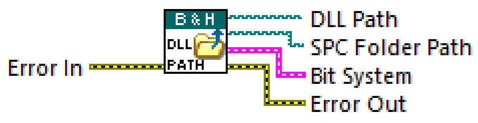
The module will be initialised, but only when it is not in use (locked) by other application.

After successful initialisation the module is locked to prevent that other application can access it. The user should check initialisation status of all modules he wants to use by calling SPC_get_init_status function.

If, for some reasons, the module which was locked must be initialised, it can be done using the function SPC_set_mode with the parameter 'force_use' = 1. (The default in the LV-wrappers)

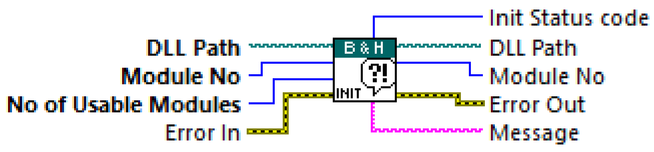
The initialisation file is an ASCII file. We recommend either to use the file spcm.ini or to start with spcm.ini and introduce the desired changes.

GetDLLPath.vi



All wrapper VIs for spcmdll calls need the filepath to the dll. This VI provides this path according to the operation systems bitness and the LabVIEW bitness.

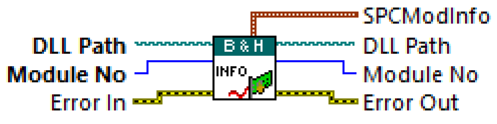
SPC_get_init_status.vi



The procedure returns the initialisation result code set by the function SPC_init. The possible values are shown below (see also spcm_def.h):

INIT_ SPC_OK	0 no error
INIT_ SPC_NOT_DONE	-1 init not done
INIT_ SPC_WRONG_EEP_CHKSUM	-2 wrong EEPROM checksum
INIT_ SPC_WRONG_MOD_ID	-3 wrong module identification code
INIT_ SPC_HARD_TEST_ERR	-4 hardware test failed
INIT_ SPC_CANT_OPEN_PCI_CARD	-5 cannot open PCI card
INIT_ SPC_MOD_IN_USE y	-6 module in use (locked) - cannot initialise
INIT_ SPC_WINDRVR_VER	-7 incorrect WinDriver version
INIT_ SPC_WRONG_LICENSE	-8 corrupted license key
INIT_SPC_FIRMWARE_VER	-9 incorrect firmware version of DPC/SPC module
INIT_SPC_NO_LICENSE	-10 license key not read from registry
INIT_SPC_LICENSE_NOT_VALID	-11 license is not valid for SPCM DLL
INIT_SPC_LICENSE_DATE_EXP	-12 license date expired
INIT_SPC_XILINX_ERR	-1xx Xilinx chip configuration error - where xx = Xilinx error code

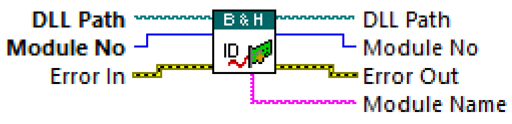
SPC_get_module_info.vi



After calling the SPC_init function the SPCModInfo internal structures for all 8 modules are filled. This function transfers the contents of the internal structure of the DLL into a structure of the type SPCModInfo (see spcm_def.h) which has to be defined by the user. The parameters included in this structure are described below.

short module_type	SPC module type (see spcm_def.h)
short bus_number	PCI bus number of the module
short slot_number	slot number on PCI bus ‘bus_number’ occupied by the module
short in_use	-1 used and locked by other application, 0 - not used, 1 - in use
short init	set to initialisation result code
unsigned short	base_adr base I/O address on PCI bus

SPC_test_id.vi



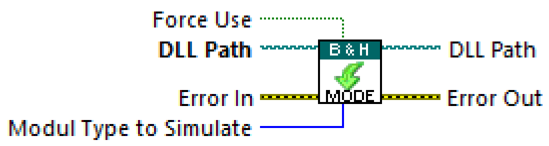
The procedure can be used to check whether an SPC module is present and which type of module it is. It is a low-level procedure that is called also during the initialisation in SPC_init.

The procedure returns a module type value of module 'mod_no'. Possible module type values are defined in spcm_def.h file.

The x0x and x3x versions are not distinguished by the id but by the EEPROM data and the function SPC_init. SPC_test_id will return the correct values only if SPC_init has been called.

```
/* supported SPC module types - returned value from SPC_test_id */

#define M_SPC600 600 PCI version of 400

#define M_SPC630 630 PCI version of 430

#define M_SPC700 700 PCI version of 500

#define M_SPC730 730 PCI version of 530

#define M_SPC130 130 PCI special version of 630

#define M_SPC830 830 version of 730 with large memory, Fifo mode

#define M_SPC140 140 130 with large memory, Fifo mode, Scan modes

#define M_SPC930 930 830 with Camera mode

#define M_SPC150 150 140 with Fifo mode, Scan modes, Fifo Imaging, Cont. Flow

#define M_DPC230 230 DPC-230

#define M_SPC131 131 130 with with extended memory = SPC-130-EM

#define M_SPC151 151 150N = 150 with faster discriminators and reduced timing wobble

#define M_SPC160 160 160 - special module for optical tomography

#define M_SPC161 161 SPC-160X = SPC-160 with TAC range 25-2500 ns (instead of 50-5000 ns)

#define M_SPC162 162 SPC-160PCIE = SPC-160 on PCI-EX bus

#define M_SPC180 180 SPC-180N - as SPC-160 but with faster data transfer on PCI-EX bus

                    //                    (using SmartLogic IP_CORE on Xilinx Artix 7)

#define M_SPC181 181 SPC-180NX = SPC-180N with TAC range 25-2500 ns (instead of 50-5000 ns)

#define M_SPC182 182 SPC-180NXX = SPC-180N with TAC range 12.5-125 ns (instead of 50-5000 ns)

#define M_SPC185 185 SPC-180N-USB - as SPC-180N with USB-3 interface

#define M_SPC186 186 SPC-180NX-USB - as SPC-180N-USB with TAC range 25-2500 ns (instead of 50-5000 ns)

#define M_SPC187 187 SPC-180NXX-USB - as SPC-180N-USB with TAC range 12.5-125 ns (instead of 50-5000
ns)

#define M_TDC104 104 TDC-104 - four TDC channels module
```

SPC_set_mode.vi



The procedure is used to change the mode of the DLL operation between the hardware mode and the simulation mode. It is also used to switch the DLL to the simulation mode if hardware errors occur during the initialisation.

Table 'in_use' should contain entries for all 8 modules:

0 – means that the module will be unlocked and not used longer

1 – means that the module will be initialised and locked

When the Hardware Mode is requested for each of 8 possible modules:

-if 'in_use' entry = 1: the proper module is locked and initialised (if it wasn't) with the

initial parameters set (from ini_file) but only when it was not locked by another

application or when 'force_use' = 1.

-if 'in_use' entry = 0: the proper module is unlocked and cannot be used further.

When one of the simulation modes is requested for each of 8 possible modules:

-if 'in_use' entry = 1: the proper module is initialised (if it wasn't) with the initial

parameters set (from ini_file).

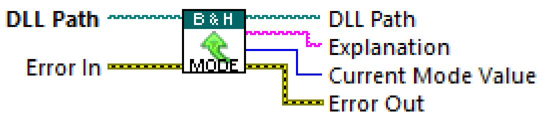
-if 'in_use' entry = 0: the proper module is unlocked and cannot be used further.

Errors during the module initialisation can cause that the module is excluded from use.

Use the function SPC_get_init_status and/or SPC_get_module_info to check which modules are correctly initialised and can be use further.

Use the function SPC_get_mode to check which mode is actually set. Possible 'mode' values are defined in the spcm_def.h file.

SPC_get_mode.vi

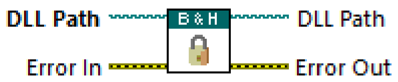


The procedure returns the current DLL operation mode. Possible 'mode' values are defined in the spcm_def.h file:

```
#define SPC_HARD          0   /* hardware mode */
#define SPC_SIMUL600    600 /* simulation mode of SPC600 module */
#define SPC_SIMUL630    630 /* simulation mode of SPC630 module */
#define SPC_SIMUL700    700 /* simulation mode of SPC700 module */
```

```
#define SPC_SIMUL730    730 /* simulation mode of SPC730 module */
#define SPC_SIMUL130    130 /* simulation mode of SPC130 module */
#define SPC_SIMUL830    830 /* simulation mode of SPC830 module */
#define SPC_SIMUL140    140 /* simulation mode of SPC140 module */
#define SPC_SIMUL930    930 /* simulation mode of SPC930 module */
#define SPC_SIMUL150    150 /* simulation mode of SPC150 module */
#define DPC_SIMUL230    230 /* simulation mode of DPC230 module */
#define SPC_SIMUL131    131 /* simulation mode of SPC131 (= SPC-130-EM) module */
#define SPC_SIMUL151    151 /* simulation mode of SPC150N module */
#define SPC_SIMUL160    160 /* simulation mode of SPC160 module */
```

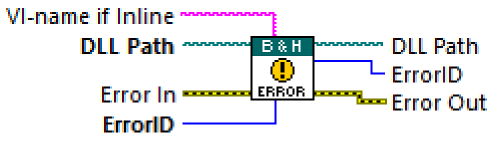
SPC_close.vi



The procedure frees buffers allocated via DLL and set the DLL state as before SPC_init call.

SPC_init is the only procedure which can be called after SPC_close.

SPC_get_error_string.vi



The procedure copies the string which contains the explanation of the SPC DLL error provided by ‘error_id’to the LV ErrorCluster’s source field.

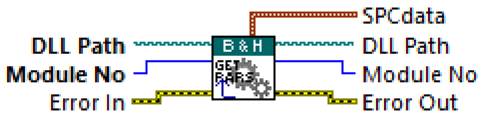
For debugging aid, this VI adds the name of the calling VI (usually the LV-SPCM DLL wrapper) to the error description in the source field.

Wire manually the name of the wrapper to ‘VI-name if Inline’ when the wrapper’s name can’t be found on the Call Chain list.

Setup Functions:

\SPC_Wrappers\Setup

SPC_get_parameters.vi



After calling the SPC_init function (see above) the measurement parameters from the initialisation file are present in the module and in the internal data structures of the DLLs. To give the user access to the parameters, the function SPC_get_parameters is provided. This function transfers the parameter values from the internal DLL structures of the module 'mod_no' into a structure of the type SPCdata (see spcm_def.h) which has to be defined by the user. The parameter values in this structure are described below and in spcm_def.h.

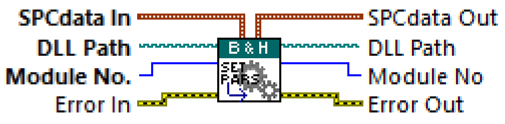
This LV-Driver project includes a type def cluster for these parameters at *SPC_Wrappers/SPC_Controls/SPCdata.ctl*

```
typedef struct _SPCdata{    /* structure for library data  */
 unsigned short base_adr;  /* base I/O address on PCI bus */
 short        init;     /* set to initialisation result code */
 float cfd_limit_low;   /* for SPCx3x(140,15x,131-7,16x,18x) -500 .. 0mV ,for SPCx0x 5 .. 80mV
                        for DPC230 = CFD_TH1 threshold of CFD1 -510 ..0 mV
                        for SPC-QC-104 = CFD_TH1_TDC threshold of CFD1 -500 ..0 mV */
 float cfd_limit_high;  /* 5 ..80 mV, default 80 mV , not for SPC130,140,15x,131-7,16x,18x,930
                        for DPC230 = CFD_TH2 threshold of CFD2 -510 ..0 mV
                        for SPC-QC-104 = CFD_TH2_TDC threshold of CFD2 -500 ..0 mV */
 float cfd_zc_level;    /* SPCx3x(140,15x,131-7,16x,18x) -96 .. 96mV, SPCx0x -10 .. 10mV
                        for DPC230 = CFD_TH3 threshold of CFD3 -510 ..0 mV
                        for SPC-QC-104 = CFD_TH3_TDC threshold of CFD3 -500 ..0 mV */
 float cfd_holdoff;     /* SPCx0x: 5 .. 20 ns, other modules: no influence
                        for DPC230 = CFD_TH4 threshold of CFD4 -510 ..0 mV
                        for SPC-QC-104 = CFD_ZC3_TDC Zero Cross level of CFD3 -96 ..96 mV */
 float sync_zc_level;   /* SPCx3x(140,15x,131-7,16x,18x): -96 .. 96mV, SPCx0x: -10..10mV
                        for DPC230 = CFD_ZC1 Zero Cross level of CFD1 -96 ..96 mV
                        for SPC-QC-104 = CFD_ZC4_TDC Zero Cross level of CFD4 -96 ..96 mV */
 float sync_holdoff;    /* 4 .. 16 ns ( SPC130,140,15x,131-7,16x,18x,930: no influence)
                        for DPC230 = CFD_ZC2 Zero Cross level of CFD2 -96 ..96 mV
                        for SPC-QC-104 = CFD_ZC2_TDC Zero Cross level of CFD2 -96 ..96 mV */
 float sync_threshold;  /* SPCx3x(140,15x,131-7,16x,18x): -500 .. -20mV, SPCx0x: no influence
                        for DPC230 = CFD_ZC3 Zero Cross level of CFD3 -96 ..96 mV
                        for SPC-QC-104 = CFD_TH4_TDC threshold of CFD4 -500 ..0 mV */
 float tac_range;       /* 50 .. 5000 ns, for SPC161(181) 25 .. 2500,
                    for SPC150NX 25 .. 2500, for SPC150NX-12(NXX), 182(NXX)  12.5 ..  125,

                    for DPC230 = DPC range in TCSPC and Multiscaler mode

                        0.16461 .. 1e7 ns

                        for SPC-QC-104 = TDC range 1.024 ns .. 67µs  */

 short sync_freq_div;   /* 1,2,4 ( SPC130,140,15x,131-7,16x,18x, 930, 6x0, DPC230),

                      (other modules (SPC7x0) : 1,2,4,8,16) */

 short tac_gain;       /* 1 .. 15    not for DPC230, TDC104 */

 float tac_offset;     /* 0 .. 100%, for SPC16x,150N(151),132-7,18x  0 .. 50%

               for DPC230 = TDC offset in TCSPC and Multiscaler mode -100 .. 100% */

 float tac_limit_low;   /* 0 .. 100%  not for DPC230 */

                 // for DPC590 = SYNC_FREQ  1 .. 100 MHz

 float tac_limit_high;  /* 0 .. 100%

                        for DPC230 = CFD_ZC4 Zero Cross level of CFD4 -96 ..96 mV

                        for SPC-QC-104 = CFD_ZC1_TDC Zero Cross level of CFD1 -96 ..96 mV */

 short adc_resolution;  /* 6,8,10,12 bits, default 10 ,

                        (additionally 0,2,4 bits for SPC830,140,15x,131-7,16x,18x,930 )

               for DPC230 = no of points of decay curve in TCSPC and Multiscaler mode

                          0,2,4,6,8,10,12,14,16  bits */

 short ext_latch_delay; /* 0 ..255 ns, (SPC130, DPC230 : no influence) */

                 /* SPC140,15x,131-7,16x,930: only values 0,10,20,30,40,50 ns are possible */

                 /* TDC104:  -57.344 .. 65.536 ns in 16 steps, step = 8.192ns  */

 float collect_time;    /* 1e-7 .. 100000s , default 0.01s */

 float display_time;    /* 0.1 .. 100000s , default 1.0s, obsolete, not used in DLL */

 float repeat_time;     /* 1e-7 .. 100000s , default 10.0s, not for DPC230 */

 short stop_on_time;    /* 1 (stop) or 0 (no stop) */

 short stop_on_ovfl;    /* 1 (stop) or 0 (no stop), not for DPC230  */

 short dither_range;    /* possible values - 0, 32,   64,   128,  256

                     have meaning:  0, 1/64, 1/32, 1/16, 1/8

                     value 256 (1/8) only for SPC6x0,7x0, 830

                     not for DPC230 */

 short count_incr;      /* 1 .. 255, not for DPC230  */

 short mem_bank;        /* for SPC130,600,630, 15x,131-7,16x,18x :  0 , 1 , default 0

                        other SPC modules: always 0

                        DPC230 : bit 1 - DPC 1 active, bit 2 - DPC 2 active

                       */

 short dead_time_comp;  /* 0 (off) or 1 (on), default 1, not for DPC230   */
```

```
unsigned short scan_control; /* SPC505(535,506,536) scanning(routing) control word,
                      other SPC modules always 0 */
unsigned short routing_mode;     /* DPC230  bits 0-7 - control bits
                   SPC15x,830,140,131-7,16x,18x )
                    - bits 6 - in FIFO_32M mode,
                           = 0 (default) Marker 3 not used,
                           = 1 waiting for Marker 3 to start collection timer,
                              ( used in accumulated Mosaic measurements)
                    - bits 7 - in FIFO_32M mode,
                           = 0 (default) Frame pulses on Marker 2,
                           = 1 Frame pulses on Marker 3,
                    - bits 8 - 11 - enable(1)/disable(0), default 0
                             of recording Markers 0-3 entries in FIFO mode
                    - bits 12 - 15 - active edge 0(falling), 1(rising), default 0
                              of Markers 0-3 in FIFO mode
                   other SPC modules - not used  */
float tac_enable_hold;  /* SPC230 10.0 .. 265.0 ns - duration of TAC enable pulse ,
                   DPC230 - macro time clock in ps, default 82.305 ps,
                   other SPC modules always 0 */
short pci_card_no;      /* module no on PCI bus (0-7)  */
unsigned short mode;    /* for SPC7x0      , default 0
                    0 - normal operation (routing in),
                    1 - block address out, 2 -  Scan In, 3 - Scan Out
                   for SPC6x0      , default 0
                    0 - normal operation (routing in)
                    2 - FIFO mode 48 bits, 3 - FIFO mode 32 bits
                   for SPC130      , default 0
                    0 - normal operation (routing in)
                    2 - FIFO mode 32 bits
                   for SPC140 , default 0
                    0 - normal operation (routing in)
                    1 - FIFO mode 32 bits, 2 -  Scan In, 3 - Scan Out
                    5 - FIFO_mode 32 bits with markers ( FIFO_32M ), with FPGA v. > B0
                   for SPC15x,16x,18x, default 0
                    0 - normal operation (routing in)
                    1 - FIFO mode 32 bits, 2 -  Scan In, 3 - Scan Out
                    5 - FIFO_mode 32 bits with markers ( FIFO_32M )
                  for SPC830,930 , default 0
                    0 - normal operation (routing in)
                    1 - FIFO mode 32 bits, 2 -  Scan In, 3 - Scan Out
                    4 - Camera mode ( only SPC930 )
                    5 - FIFO_mode 32 bits with markers ( FIFO_32M ),
                              SPC830 with FPGA v. > C0
                  for DPC230 , default 8
                    6 - TCSPC FIFO
                    7 - TCSPC FIFO Image mode
                    8 - Absolute Time FIFO mode
                    9 - Absolute Time FIFO Image mode
                  for SPC131-7 , default 0
                    0 - normal operation (routing in)
                    1 - FIFO mode 32 bits
                  for TDC104 , default 0
                    0 - normal operation (routing in)
                    1 - FIFO mode 32 bits,
                   13 - FIFO_Absolute Times mode 32 bits
                   */
unsigned long scan_size_x;  /* for SPC7x0,830,140,15x,16x,18x,930 modules in scanning modes 1 .. 65536,
                        default 1, not for DPC230  */
unsigned long scan_size_y;  /* for SPC7x0,830,140,15x,16x,18x,930 modules in scanning modes 1 .. 65536,
                        default 1, not for DPC230  */
unsigned long scan_rout_x;  /* number of X routing channels in Scan In & Scan Out modes, not for DPC230
                     for SPC7x0,830,140,15x,16x,18x,930 modules
                   1 .. 128, ( SPC7x0,830 ), 1 .. 16 (SPC140,15x,16x,18x,930), default 1 */
unsigned long scan_rout_y;  /* number of Y routing channels in Scan In & Scan Out modes, not for DPC230
                     for SPC7x0,830,140,15x,16x,18x, 930 modules
                   1 .. 128, ( SPC7x0,830 ), 1 .. 16 (SPC140,15x,16x,18x,930), default 1 */
  /* INT(log2(scan_size_x)) + INT(log2(scan_size_y)) +
    INT(log2(scan_rout_x)) + INT(log2(scan_rout_y)) <= max number of scanning bits
               max number of scanning bits depends on current adc_resolution:
                    12 (10 for SPC7x0,140,15x,16x,18x)  -          12
                    14 (12 for SPC7x0,140,15x,16x,18x)  -          10
                    16 (14 for SPC7x0,140,15x,16x,18x)  -           8
                    18 (16 for SPC7x0,140,15x,16x,18x)  -         6
                    20 (18 for SPC140,15x,16x,18x)     -         4
                    22 (20 for SPC140,15x,16x,18x)     -         2
                    24 (22 for SPC140,15x,16x,18x)     -         0
                    */
unsigned long  scan_flyback;   /* for SPC7x0,830,140,15x,16x,18x,930 modules in Scan Out or Rout Out mode,
                         default & minimum = 1, not for DPC230  */
                    /* bits 15-0  Flyback X in number of pixels
                       bits 31-16 Flyback Y in number of lines */
unsigned long  scan_borders;   /* for SPC7x0,830,140,15x,16x,18x,930 modules in Scan In mode,
                         default 0, not for DPC230  */
                    /* bits 15-0  Upper boarder, bits 31-16 Left boarder */
unsigned short scan_polarity;    /* for SPC7x0,830,140,15x,16x,18x,930 modules in scanning modes,
                         default 0, not for DPC230  */
      /* bit 0 - polarity of HSYNC (Line), bit 1 - polarity of VSYNC (Frame),
        bit 2 - pixel clock polarity
        bit = 0 - falling edge(active low)
        bit = 1 - rising  edge(active high)
       for SPC140,15x,16x,18x,830 in FIFO_32M mode
        bit = 8 - HSYNC (Line) marker disabled (1) or enabled (0, default )
              when disabled, line marker will not appear in FIFO photons stream */
unsigned short pixel_clock;   /* for SPC7x0,830,140,15x,16x,18x,930 modules in Scan In mode, or DPC230 in Image modes
                  pixel clock source, 0 - internal,1 - external, default 0
           for SPC140,15x,16x,18x,830 in FIFO_32M mode it disables/enables pixel markers
                           in photons stream */
unsigned short line_compression;   /* line compression factor for SPC7x0,830,140,15x,16x,18x,930 modules
                    in Scan In mode,   1,2,4,8,16,32,64,128, default 1*/
unsigned short trigger;    /* external trigger condition -
        bits 1 & 0 mean :   00 - ( value 0 ) none(default),
                     01 - ( value 1 ) active low,
                     10 - ( value 2 ) active high
      when sequencer is enabled on SPC130,6x0,15x,16x,18x,131-7 modules additionally
       bits 9 & 8 of the value mean:
        00 - trigger only at the start of the sequence,
        01 ( 100 hex, 256 decimal ) - trigger on each bank
        11 ( 300 hex, 768 decimal ) - trigger on each curve in the bank
```

```
     for SPC15x,16x,18x, 131-7, 140 and SPC130 (FPGA v. > C0) multi-module configuration
        bits 13 & 12 of the value mean:
      x0 - module does not use trigger bus ( trigger defined via bits 0-1),
      01 ( 1000 hex, 4096 decimal ) - module uses trigger bus as slave
                       ( waits for the trigger on master),
      11 ( 3000 hex, 12288 decimal ) - module uses trigger bus as master
                   ( trigger defined via bits 0-1),
                   ( only one module can be the master )
     */
float pixel_time;    /* pixel time in sec for SPC7x0,830,140,15x,16x,18x,930 modules in Scan In mode,
             50e-9 .. 1.0 , default 200e-9 */
unsigned long ext_pixclk_div;  /* divider of external pixel clock for SPC7x0,830,140,15x,16x,18x modules
              in Scan In mode, 1 .. 0x3fe, default 1*/
float rate_count_time;    /* rate counting time in sec  default 1.0 sec
             for SPC130,830,930,15x,16x,18x,131-7 can be : 1.0, 250ms, 100ms, 50ms
             for SPC140 fixed to 50ms
             for DPC230 - 1.0sec,
                 0.0 - don't count rate outside the measurement, */
short macro_time_clk;    /*  macro time clock definition for SPC130,140,15x,16x,18x,131-7,830,930 in FIFO mode
             for SPC130, SPC140,15x,16x,18x,131-7:
               0 - 50ns (default), 25ns for SPC15x,16x,18x,131-7 & 140 with FPGA v. > B0 ,
               1 - SYNC freq., 2 - 1/2 SYNC freq.,
               3 - 1/4 SYNC freq., 4 - 1/8 SYNC freq.
             for SPC830:
               0 - 50ns (default), 1 - SYNC freq.,
             for TDC-104:
               0 - 2.048ns in FIFO mode, 0.004ns in FIFO Absolute Time mode
             for SPC930:
               0 - 50ns (default), 1 - SYNC freq., 2 - 1/2 SYNC freq.*/
short add_select;    /* selects ADD signal source for all modules except SPC930 & DPC230 :
            0 - internal (ADD only), 1 - external */
short test_eep;       /* test EEPROM checksum or not  */
short adc_zoom;    /* selects ADC zoom level for module SPC830,140,15x,16x,18x,131-7,930 default 0
            bit 4 = 0(1) - zoom off(on ),
            bits 0 - 3 zoom level =
              0 - zoom of the 1st 1/16th of ADC range,
                    15 - zoom of the 16th 1/16th of ADC range */
unsigned long img_size_x;  /* image X size ( SPC140,15x,16x,18x,830 in FIFO_32M, SPC930 in Camera mode ),
                         1 .. 1024, default 1 */
unsigned long img_size_y;  /* image Y size ( SPC140,15x,16x,18x,830 in FIFO_32M, SPC930 in Camera mode ),
                     actually equal to img_size_x ( quadratic image ) */
unsigned long img_rout_x;  /* no of X routing channels ( SPC140,15x,16x,18x,830 in FIFO_32M, SPC930 in Camera mode ),
                         1 .. 16, default 1 */
unsigned long img_rout_y;  /* no of Y routing channels ( SPC140,15x,16x,18x,830 in FIFO_32M, SPC930 in Camera mode ),
                         1 .. 16, default 1 */
short xy_gain;      /* selects gain for XY ADCs for module SPC930, 1,2,4, default 1 */
short master_clock;  /*  use Master Clock( 1 ) or not ( 0 ), default 0,
                    only for SPC140,15x,16x,18x,131-7 multi-module configuration
                - value 2 (when read) means Master Clock state was set by other application
                     and cannot be changed */
short adc_sample_delay; /* ADC's sample delay, only for module SPC930
                   0,10,20,30,40,50 ns (default 0 ) */
short detector_type;    /*  for module SPC930 :
                   detector type used in Camera mode, 1 .. 9899, default 1,
               normally recognised automatically from the corresponding .bit file
                    1 - Hamamatsu Resistive Anode 4 channels detector
                    2 - Wedge & Strip 3 channels detector
               for module DPC230 :
                 type of active inputs : bit 1 - TDC1, bit 2 - TDC2,
                   bit value 0 , CFD inputs active,
                   bit value 1 , TTL inputs active */

unsigned long  chan_enable;   /* for module DPC230/330 - enable(1)/disable(0) input channels
                     bits 0-7   - en/disable TTL channel 0-7 in TDC1
                     bits 8-9   - en/disable CFD channel 0-1 in TDC1
                     bits 12-19 - en/disable TTL channel 0-7 in TDC2
                     bits 20-21 - en/disable CFD channel 0-1 in TDC2
                     */
unsigned long  chan_slope;   /* for module DPC230 - active slope of input channels
                       1 - rising, 0 - falling edge active
                     bits 0-7   - slope of TTL channel 0-7 in TDC1
                     bits 8-9   - slope of CFD channel 0-1 in TDC1
                    bits 12-19 - slope of TTL channel 0-7 in TDC2

                    bits 20-21 - slope of CFD channel 0-1 in TDC2

                    */

unsigned long  chan_spec_no;    /* for module DPC230/330 - channel numbers of special inputs

                              default 0x8813 in imaging modes

        bits 0-4 - reference chan. no ( TCSPC and Multiscaler modes)

            default = 19, value:

            0-1 CFD chan. 0-1 of TDC1,   2-9 TTL chan. 0-7 of TDC1

           10-11 CFD chan. 0-1 of TDC2, 12-19 TTL chan. 0-7 of TDC2

        bits  8-10 - frame clock TTL chan. no ( imaging modes ) 0-7, default 0

        bits 11-13 - line  clock TTL chan. no ( imaging modes ) 0-7, default 1

        bits 14-16 - pixel clock TTL chan. no ( imaging modes ) 0-7, default 2

        bit  17    - TDC no for pixel, line, frame clocks ( imaging modes )

                    0 = TDC1, 1 = TDC2, default 0

        bits 18-19 - not used

        bits 20-23 - active channels of TDC1 for DPC-330 Hardware Histogram modes

        bits 24-27 - active channels of TDC2 for DPC-330 Hardware Histogram modes

        bits 28-31 - not used

        */

unsigned long tdc_control;     /*   up to v.5.0 (before 12.12.2022) it was x_axis_type

                    TDC-104 module: bits 16 - 21 = control bits for TDC-104 :

                      bits 16-18 - enable/disable (1/0) routing in Fifo modes for module's inputs IN1-3

                      bits 19    - enable/disable (1/0) routing in FIFO Absolute Time mode (TDC_ABS) for module's input IN4

                        !! for not Fifo modes bits 16-19 have fixed value 1 1 1 0 ( in hardware)

                      bit 20 = 0 - Multiphoton Detection, = 1 single photon

                      bit 21 = 1 - Suppress IN4 signal in TDC_ABS mode

                    - module SPC930 only for compatibility bits 0-3 = X axis representation,

                              0 - time (default ), 1 - ADC1 Voltage,

                              2 - ADC2 Voltage, 3 - ADC3 Voltage, 4 - ADC4 Voltage

                    - other SPC modules - not used */

float tdc_offset[4];   /*  for TDC-104 module, offset for 4 TDC channels in ns  0 - 32.256 in 0.512 steps */

char  reserve[56];     /* keep always total size = 256 B */

}SPCdata;
```

SPC_set_parameters.vi



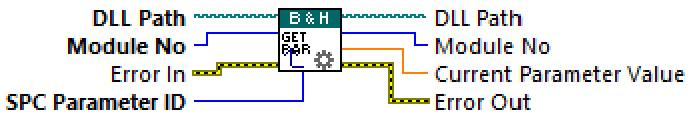
The procedure sends all parameters from the 'SPCdata' structure to the internal DLL structures of the module 'mod_no' and to the control registers of the SPC module 'mod_no'.

The new parameter values are recalculated according to the parameter limits, hardware restrictions (e.g. DAC resolution) and the SPC module type. Furthermore, cross dependencies between different parameters are taken into account to ensure the correct hardware operation.

This VI also calls SPC_get_parameters after the call to SPC_set_parameters to provide the actual values after recalculation.

If an error occurs at a particular parameter, the procedure does not set the rest of the parameters and returns with an error code.

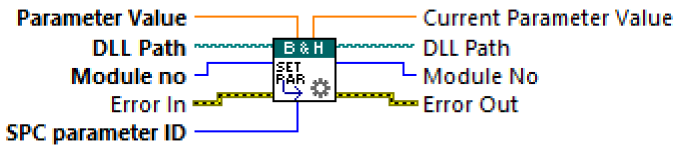
SPC_get_parameter.vi



The procedure loads 'value' with the actual value of the requested parameter from the internal DLL structures of the module 'mod_no'. The par_id values are defined in spcm_def.h file as SPC_PARAMETERS_KEYWORDS.

LV users may use the type def \SPC_Wrappers\SPC_Controls\par_id.ctl

SPC_set_parameter.vi



The procedure sets the specified hardware parameter. The value of the specified parameter is transferred to the internal DLL structures of the module 'mod_no' and to the SPC module 'mod_no'.

If 'mod_no' = -1, parameter will be changed for all SPC modules which are actually in use.

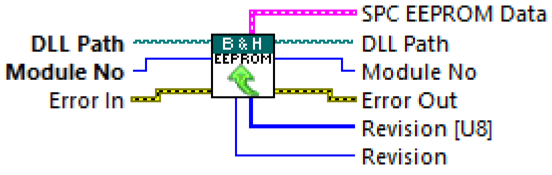
The new parameter value is recalculated according to the parameter limits, hardware restrictions (e.g. DAC resolution) and SPC module type. Furthermore, cross dependencies between different parameters are taken into account to ensure the correct hardware operation.

It is recommended to read back the parameters after setting it to get their real values after recalculation. This VI is actually calling SPC_get_parameter after the call to SPC_set_parameter to provide the actaual value after recalculation. But only if mod_no is not equal to -1.

The par_id values are defined in spcm_def.h file as SPC_PARAMETERS_KEYWORDS.

LV users may use the type def \SPC_Wrappers\SPC_Controls\par_id.ctl

SPC_get_eeprom_data.vi

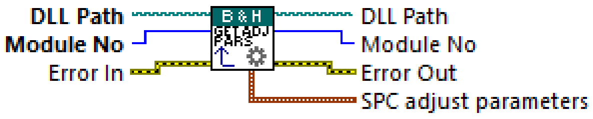


The structure "eep_data" is filled with the contents of EEPROM of SPC module ‘mod_no’.

The EEPROM contains production data and adjust parameters of the module. The structure "SPC_EEP_Data" is defined in the file spcm_def.h.

Normally, the EEPROM data need not be read explicitly because the EEPROM is read during SPC_init and the module type information and the adjust values are taken into account when the SPC module registers are loaded.

SPC_get_adjust_parameters.vi

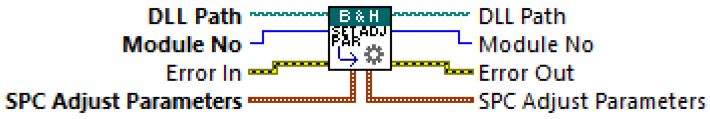


The structure 'adjpara' is filled with adjust parameters of the SPC module ‘mod_no’ that are currently in use. The parameters can either be previously loaded from the EEPROM by SPC_init or SPC_get_eeprom_data or - not recommended - set by SPC_set_adust_parameters.

The structure "SPC_Adjust_Para" is defined in the file spcm_def.h.

Normally, the adjust parameters need not be read explicitly because the EEPROM is read during SPC_init and the adjust values are taken into account when the SPC module registers are loaded.

SPC_set_adjust_parameters.vi

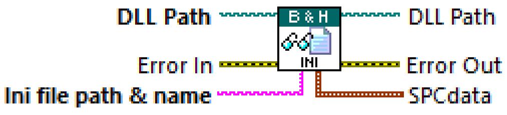


The adjust parameters in the internal DLL structures (not in the EEPROM) of the module 'mod_no' are set to values from the structure "adjpara". The function is used to set the module adjust parameters to values other than the values from the EEPROM.

The new adjust values will be used until the next call of SPC_init. The next call to SPC_init replaces the adjust parameters by the values from the EEPROM. We strongly discourage to use modified adjust parameters, because the module function can be seriously corrupted by wrong adjust values.

The structure "SPC_Adjust_Para" is defined in the file spcm_def.h.

SPC_read_parameters_from_inifile.vi



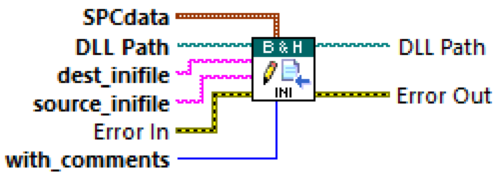
The procedure reads parameters from the file 'inifile' and transfers them to the 'SPCdata' structure 'data'.

The 'inifile' file is an ASCII file with a structure shown in SPC_init description. We recommend to use either the original .ini files or the files created using function SPC_save_parameters_to_inifile.

If a particular parameter is not present in .ini file or cannot be read, the appropriate field in SPCdata 'data' structure is set to the parameter's default value.

Use SPC_set_parameters to send result parameters set to the SPC module.

SPC_save_parameters_to_inifile.vi



The parameters set from the 'SPCdata' structure 'data' is saved to the section [spc_module] in dest_inifile file. [spc_base] section and initial comment lines are copied to dest_inifile from the source_inifile file.

If the parameter 'source_inifile' is equal NULL, ini_file used in SPC_init function call is used as the source file for dest_inifile.

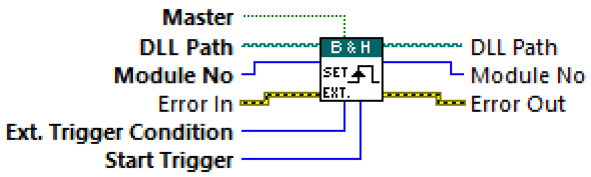
Additionally, when 'with_comments' parameter is equal 1, comment lines for the particular parameters are taken from the source file and saved to dest_inifile together with the parameter value.

The 'dest_inifile' and 'source_inifile' filea are ASCII files with a structure shown in SPC_init description.

Use SPC_read_parameters_from_inifile to read back the parameters set from the file and then SPC_set_parameters to send it to the SPC module.

Convenience Vis in \SPC_Wrappers\Setup

SetTrigger.vi

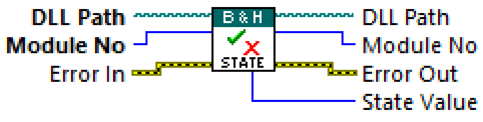


This VI is a wrapper for SPC_set_parameter.vi with the ‘SPC parameter ID’ set to ‘External Trigger Condition’. The flag values are calculated for convenience.

Status Functions:

\SPC_Wrappers\Action-Status

SPC_test_state.vi



SPC_test_state sets a state variable according to the current state of the measurement on SPC module ‘mod_no’. The function is used to control the measurement loop.

Note: SPC_read_state is clearing some flags directly. Take care when running in separate threats.

Note: This VI is set to execute as preallocated clone to improve performance when run in parallel loops for multi module setups.

The status bits delivered by the function are listed below (see also SPCM_DEF.H).

SPC_OVERFL	0x1	stopped on overflow
SPC_OVERFLOW	0x2	overflow occurred
SPC_TIME_OVER	0x4	stopped on expiration of collection timer
SPC_COLTIM_OVER	0x8	collection timer expired
SPC_CMD_STOP	0x10	stopped on user command
SPC_ARMED	0x80	measurement in progress (current bank)
SPC_REPTIM_OVER	0x20	repeat timer expired

SPC_COLTIM_2OVER	0x100	second overflow of collection timer
SPC_REPTIM_2OVER	0x200	second overflow of repeat timer

For SPC600(630) and SPC130 modules only:

SPC_SEQ_GAP	0x40	Sequencer is waiting for another bank to be armed

For SPC6x0(13x)(140)(830)(15x)(160) modules only :

SPC_FOVFL	0x400	Fifo overflow, data lost
SPC_FEMPTY	0x800	Fifo empty

For SPC7x0(140) (830) (15x) (131) (160) and SPC930 modules only:

SPC_SCRDY	0x400	Scan ready (data can be read)
SPC_FBRDY	0x800	Flow back of scan finished
SPC_MEASURE	0x40	Measurement active = no margin, no wait for trigger, armed
SPC_WAIT_TRG	0x1000	Wait for external trigger
SPC_HFILL_NRDY	0x8000	hardware fill not finished

For SPC140 (15x) (131) and SPC160 modules only:

SPC_ SEQ_STOP	0x4000	disarmed (measurement stopped) by sequencer

For SPC15x (131) (160) modules only:

SPC_ SEQ_GAP150	0x2000	Sequencer is waiting for another bank to be armed

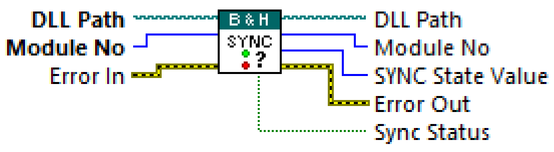
For SPC140 (15x) (160) and SPC830 modules in FIFO_32M mode

SPC_ WAIT_FR	0x2000	FIFO IMAGE measurement waits for the frame signal to stop

For collection times longer than 80 seconds SPC_test_state updates also DLL software counters (hardware timers count up to 80 sec). Therefore, during the measurement SPC_get_actual_coltime or SPC_get_time_from_start calls must be done after PC_test_state call.

LV users might want to use the convenience VI /SPC_Wrappers/Action-Status/**CheckState.vi** which transfers the State Value into a Boolean cluster.

SPC_get_sync_state.vi



The procedure sets "sync_state" according to the actual sync state on the SPC module 'mod_no'.

For SPC-130(140)(930) module possible values are:

0: SYNC NOT OK, sync input not triggered

1: SYNC OK, sync input triggers

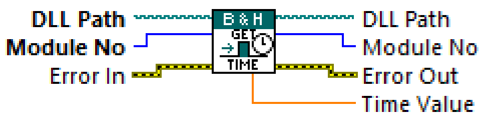
For other SPC modules possible values are:

0: NO SYNC, sync input not triggered

1: SYNC OK, sync input triggers

2, 3: SYNC OVERLOAD.

SPC_get_time_from_start.vi



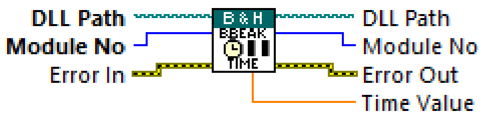
The procedure reads the SPC repeat timer and calculates the time from the start of the measurement for the SPC module 'mod_no'. It should be called during the measurement, because the timer starts to run after (re)starting the measurement.

For collection times longer than 80 seconds be sure that SPC_test_state is called in the loop during the measurement before the SPC_get_time_from_start call. SPC_test_state updates software counter which is needed for times longer than 80 sec.

The procedure can be used to test the progress of the measurement or to the start next measurement step in a multi-step measurements (such as f(t,T) in the standard software).

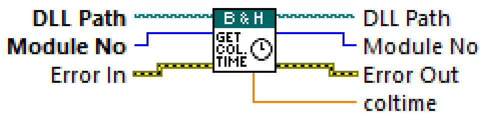
When the sequencer is running the repeat timer is not available. In this case SPC_get_time_from_start uses a software timer to measure the time from the start of the measurement.

SPC_get_break_time.vi



The procedure calculates for the SPC module 'mod_no' the time from the start of the measurement to the moment of a measurement interruption by a user break (SPC_stop_measurement or SPC_pause_measurement) or by a stop on overflow. The procedure can be used to find out the moment of measurement interrupt.

SPC_get_actual_coltime.vi



The procedure reads the timer for the collection time and calculates the actual collection time value for the SPC module 'mod_no'. During the measurement this value decreases from the specified collection time to 0.

For collection times longer than 80 seconds be sure that SPC_test_state is called in the loop during the measurement before the SPC_get_actual_coltime call. SPC_test_state updates software counter which is needed for times longer than 80 sec.

In comparison to the procedure SPC_get_time_from_start, which delivers the real time from start, the procedure returns the actual state of the dead time compensated collection time.

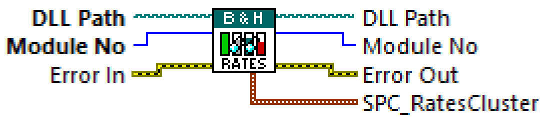
At high count rates the real time of collection can be considerably longer than the specified collection time value.

For SPC6x0,130 modules only:

- If the sequencer is running, the collection timer cannot be accessed.

- The dead time compensation can be switched off. In this case the collection timer runs with the same speed as the repeat timer, and the result is the same as that of the procedure SPC_get_time_from_start.

SPC_read_rates.vi



The procedure reads the rate counters for the SPC module 'mod_no', calculates the rate values and writes the results to the 'rates' structure.

The procedure can be called at any time after an initial call to the SPC_clear_rates function. If the rate values are ready (after 1sec of integration time), the procedure fills 'rates', starts a new integration cycle and returns 0, otherwise it returns -SPC_RATES_NOT_RDY.

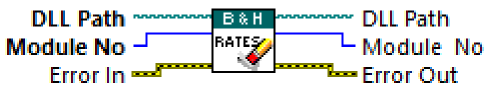
Integration time of rate values is equal 1sec, but for SPC-13x/830/930/15x/160 modules can have also other values according to the parameter RATE_COUNT_TIME (1.0s, 250ms, 100ms, 50ms are possible).

To get correct results the SPC_clear_rates function must be called before the first call of "SPC_read_rates".

As a default value the parameter "rate count time" is set to one second. To change that, please use the "Set Parameter" function with the keyword "rate count time".

Note: This VI is set to execute as preallocated clone to improve performance when run in parallel loops for multi module setups.

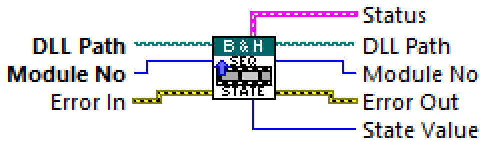
SPC_clear_rates.vi



The procedure clears all rate counters for the specified SPC module no.

To get correct rate values the procedure must be called once before the first call of the SPC_read_rates function. SPC_clear_rates start a new rate integration cycle.

SPC_get_sequencer_state.vi



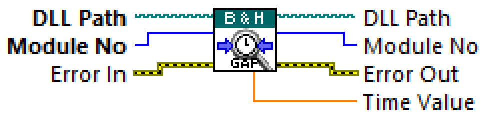
The procedure is used to get the current state of the sequencer status bits on SPC module ‘mod_no’. The sequencer status bits are defined in the spcm_def.h file:

SPC_SEQ_ENABLE	0x1 sequencer is enabled
SPC_SEQ_RUNNING	0x2 sequencer is running

only for SPC6x0/13x/15x/160 modules

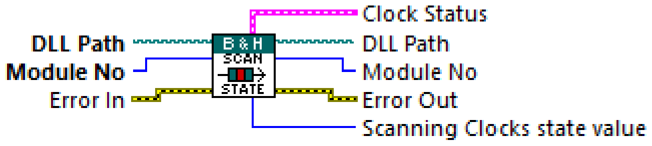
SPC_SEQ_GAP_BANK	0x4 sequencer is waiting for another bank to be armed

SPC_read_gap_time.vi



The procedure is used to read the gap time that can occur during a measurement with the sequencer of SPC6x0/130/150/131 modules. ‘time’ is set to the last gap time in ms on the SPC module ‘mod_no’.

SPC_get_scan_clk_state.vi



The procedure sets "scan_state" according to the actual state of the scanning clocks on the SPC module 'mod_no'.

Scan_state value is valid only when the module's measurement mode is set to 'Scan In' (by setting parameter MODE using SPC_set_parameter procedure). Otherwise the procedure returns error code -SPC_BAD_FUNC.

The procedure works only for SPC-830/140/15x/930/160 modules and SPC-7x0 modules with FPGA version greater than 300(hex). For other modules it returns error code -SPC_BAD_FUNC (FPGA version can be checked using SPC_get_version procedure).

Scan_state bits should be interpreted as follows:

Bit mask value (hex):

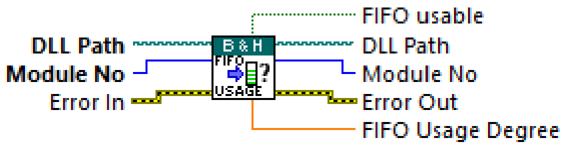
1 – External Pixel Clock present

2 – Line Clock present

4 – Frame Clock present

This VI interprets the bit mask as a boolean cluster.

SPC_get_fifo_usage.vi



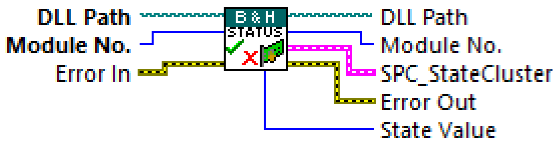
The procedure works only for SPC-13x/140/830/6x0/930/15x/160 modules in fifo mode.

The procedure sets "usage_degree" with the value in the range from 0 to 1, which tells how occupied is FIFO memory on the SPC module 'mod_no'.

FIFO memory usage is set to 0 at the start of FIFO measurement (in SPC_start_measurement) and after reading data from FIFO (SPC_read_fifo).

Convenience Vis in \SPC_Wrappers\Action-Status

CheckState.vi

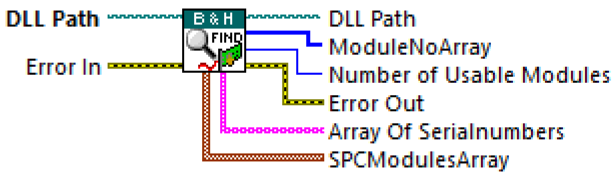


This VI is a wrapper for the **SPC_test_state.vi** and interprets the 'State Value'-flags as a Boolean cluster for convenience.

Note: SPC_read_state is clearing some flags directly. Take care when using in different threats.

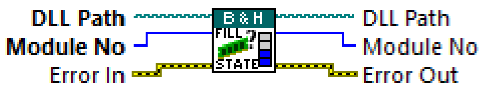
Note: This VI is set to execute as preallocated clone to improve performance when run in parallel loops for multi module setups.

FindModulesInUse.vi



This VI is a wrapper for the **SPC_get_module_info.vi** and **SPC_get_eeprom_data.vi** and provides the 'SPCModulesArray' and 'Array Of Serialnumbers' for all free SPC modules of possible 32 modules. The amount of currently free modules is counted.

TestFillState.vi



This VI is a wrapper for the 'SPC_test_state.vi' checking the flag 'SPC_HFILL_NRDY' in a loop till it becomes inactive or a timeout of 25sec occurs.

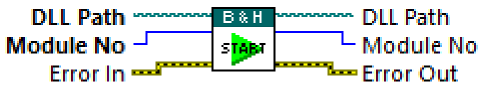
It is used to wait till the module has finished the filling process initiated by **SPC_fill_memory.vi**.

Note: This VI is set to execute as preallocated clone to improve performance when run in parallel loops for multi module setups.

Measurement Control Functions:

SPC_Wrappers\MeasurementControl

SPC_start_measurement.vi



The procedure is used to start the measurement on the SPC module 'mod_no'.

Before a measurement is started by SPC_start_measurement

- the SPC parameters must be set (SPC_init or SPC_set_parameter(s)),

- the SPC memory must be configured (SPC_configure_memory in normal modes),

- the measured blocks in SPC memory must be filled (cleared) (SPC_fill_memory),

- the measurement page must be set (SPC_set_page)

Because of hardware differences the procedure action is different for different SPC module types.

If the sequencer is not enabled (normal measurement):

- The repeat and collection timers are started with the specified collect_time

- The SPC is armed i.e. the photon collection is started.

If the sequencer is enabled ('Continuous Flow Mode'):

If the sequencer is not running:

- The sequencer is started

- The SPC is armed for next memory bank. The photon collection is not yet started! This action is not done when sequencer was enabled with 'enable' = 2 (see also SPC_enable_sequencer function) !

- SPC is armed for the current memory bank and the photon collection is started.

If the sequencer is already running:

- SPC is armed for the current memory bank

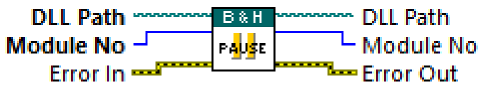
- The memory bank is reversed.

For SPC-6x0/13x/830/140/930/15x/160 in FIFO mode:

- Macro time and FIFO are cleared

- The SPC is armed i.e. the photon collection is started.

SPC_pause_measurement.vi



The procedure is used to pause a running measurement on the SPC module ‘mod_no’.

Because of hardware differences the procedure’s action varies for different SPC module types.

For all SPC module types (except SPC-6x0/13x/830/140/930/15x/160 in FIFO modes):

When the sequencer is not enabled (normal measurement):

- the repeat and collection timers are stopped,

- the SPC is disarmed (photon collection is stopped).

When the sequencer is enabled:

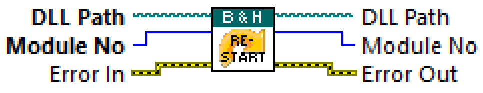
-an error is returned - this measurement can’t be paused

For SPC-6x0/13x/830/140/930/15x/160 in FIFO mode:

The procedure should not be used for FIFO modules.

The measurement can be restarted by the procedure ‘SPC_restart_measurement’.

SPC_restart_measurement.vi



The procedure is used to restart a measurement that was paused by SPC_pause_measurement on the SPC module ‘mod_no’.

Because of hardware differences the procedure's action varies for different SPC module types.

For all SPC module types (except SPC-6x0/13x/830/140/930/15x/160 in FIFO modes):

When the sequencer is not enabled (normal measurement):

- the repeat and collection timers are started,

- the SPC is armed (photon collection is started).

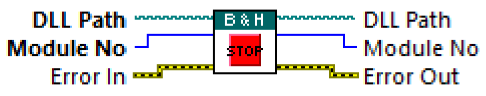
When the sequencer is enabled:

-an error is returned, this measurement can’t be restarted

For SPC-6x0/13x/830/140/930/15x/160 in FIFO mode:

The procedure should not be used for FIFO modules.

SPC_stop_measurement.vi



The procedure is used to terminate a running measurement on the SPC module 'mod_no'.

Because of hardware differences the procedure's action varies for different SPC module types.

For all SPC module types (except SPC-6x0/13x/830/140/930/15x/160 in FIFO mode):

If the sequencer is not enabled (normal measurement):

- The SPC is disarmed (i.e. the photon collection is stopped)

- The repeat and collection timers are read to get the break times

When the sequencer is enabled:

- The sequencer is stopped

- The SPC is disarmed (photon collection is stopped)

The procedure should be called after finished scan mode measurement to stop the sequencer and clear scan flags (SPC_FBRDY).

<u>For all SPC module types (except SPC-6x0/130/131) in SCAN_IN mode:</u>

If the measurement was started in SCAN_IN mode, 1st call to the function forces very short collection time to finish the current frame and returns error -21. The measurement will stop automatically after finishing current frame. 2nd call will stop the measurement without waiting for the end of frame.

<u>For SPC-6x0/13x/830/140/930/15x/160 in FIFO mode:</u>

- The SPC is disarmed (photon collection is stopped)

- The FIFO pipeline is cleared

<u>For SPC-830/140/15x/160 in FIFO_32M mode (Fifo Image):</u>

The functionality is added which enables higher level software to stop the measurement after collecting the whole frames.

Stopping the measurement requires two calls of SPC_stop_measurement.

1st call of SPC_stop_measurement:

- photon collection is not yet stopped

- current position of write pointer of FIFO is remembered as a stop position -

subsequent SPC_read_fifo calls will return 1 when this place in FIFO is reached - this is a sign for higher level software that all photons collected up to the stop moment were read. From this moment photons should be read until a frame marker will appear in photons stream -

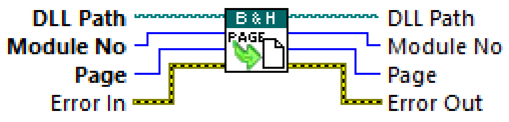
then higher-level software should call SPC_stop_measurement again to stop collecting photons.

2nd call of SPC_stop_measurement:

- The SPC is disarmed (photon collection is stopped)

- The FIFO pipeline is cleared

SPC_set_page.vi



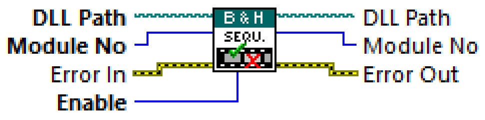
The procedure defines the page of memory (on SPC module ‘mod_no’) in which the data of a subsequent measurement will be recorded. SPC_set_page must be called before a measurement is started.

If ‘mod_no’ = -1, page will be changed for all SPC modules which are actually in use. Be sure that in such case all modules are configured in the same way (the best is to use SPC_configure_memory with ‘mod_no’ = -1).

The range of the parameters ‘block’ and ‘page’ depends on the actual configuration of the SPC memory (see SPC_configure_memory). To provide correct access to the SPC memory it is required that the function SPC_configure_memory be used before the first call of SPC_set_page (in normal modes). This function also delivers the required information about the block/page structure of the SPC memory:

long max_block_no	total number of blocks (=curves) in the memory (memory bank for the SPC-6x0/13x/15x/160)
long blocks_per_frame	Number of blocks (=curves) per frame
long frames_per_page	Number of frames per page
long maxpage	max number of pages to use in a measurement
long block_length	Number of 16-bits words per one block (curve)

SPC_enable_sequencer.vi



The procedure is used to enable or disable the sequencer of the SPC module ‘mod_no’.

If ‘mod_no’ = -1, sequencer will be enabled/disabled for all SPC modules which are actually in use.

If enable = 0:

If the sequencer is running:

- The sequencer is stopped and disabled,
- The SPC is disarmed (photon collection is stopped),

If the sequencer was enabled:

- The sequencer is disabled
- The dead time compensation of collection timer is switched to the state specified in the system parameters

When enable = 1 or 2:

If the sequencer was not enabled:

- The sequencer is enabled
- The dead time compensation of the collection timer is switched off

The way of enabling sequencer ('enable' = 1 or 2) changes slightly the action of SPC_start_measurement function for sequencer operation on SPC-130/6x0/150/131. When 'enable' =1, SPC_start_measurement arms SPC for both memory banks, while for 'enable' = 2, the function arms SPC only for current memory bank.

The 2nd case is used by the main software to program Continuous Flow measurements with accumulation.

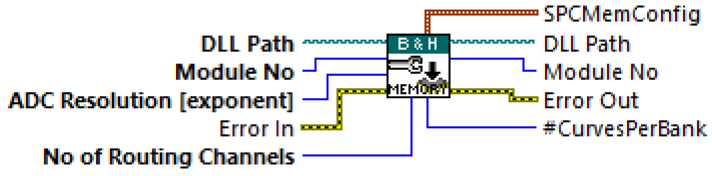
The sequencer must be enabled before starting the measurement in the following cases:

- Continuous flow measurements for SPC modules 13x/15x/160 and 6x0 - normal operation (routing in) with sequencer
- scanning modes (Scan In, Scan Out) for SPC modules 7x0, 830, 140, 930, 15x, 160)
- Continuous flow in Scan In mode for SPC-15x/160 module
- block address out mode for SPC modules 7x0

Memory Transfer Functions:

\SPC_Wrappers\MemoryTransfer

SPC_configure_memory.vi



adc_resolution:	ADC resolution (-1*,6,8,10,12 additionally 0,2,4 for SPC830/140/930/15x/131/160)
	* With adc_resolution = -1 the procedure 'mem_info' is filled with the current values disregarding 'no_of_routing_bits'.
no_of_routing_bits:	number of routing bits (0 - 3 for SPC-130) (0 - 7 for SPC6x0/140/15x/131/160) (0 – 14 for SPC-7xx(830) modules)

The procedure configures the memory of SPC module 'mod_no' depending on the specified ADC resolution, the module type and the number of detector channels (if a router is used). The action is done for normal operation modes. In FIFO modes the procedure sets hardware

defined fixed values to ADC and no_of_routing_bits. In Scan modes the procedure does not configure the memory and should be called with 'adc_resolution' = -1 to get the current state of the DLL SPCMemConfig structure (after setting scan parameters).

If 'mod_no' = -1, memory will be configured on all SPC modules which are actually in use.

The procedure has to be called before the first access to the SPC memory or before a measurement is started and always after setting ADC resolution parameter value. The memory configuration determines in which part of SPC memory the measurement data is recorded (see also SPC_set_page).

The SPC memory is interpreted as a set of 'pages'. One page contains a number of 'blocks'.

One block contains one decay curve. The number of points per block (per curve) is defined by the ADC resolution. The number of blocks per page depends on the number of points per block and on the number of detector channels (PointsX and PointsY)

In the scanning modes of the SPC-7/8/9/140/15x/160, a page contains a number of 'frames' (normally 1). Each frame contains a number of blocks. The number of blocks per page depends on the number of points per block and on the number of detector channels (PointsX and PointsY) and on the scanning parameters (pixels per line and lines per frame).

The differences between the modules are listed below.

<u>SPC-13x/6x0/15x/160 modules:</u>

The length of the recorded curves is determined by the ADC resolution and can range from 64 (0 for SPC-15x/131/160) to 4096. Therefore, the number of complete measurement data sets (or 'pages') depends on the ADC resolution and the number of routing bits used.

SPC-13x/6x0/15x/160 modules in the Histogram modes:

The length of the recorded curves is determined by the ADC resolution and can range from 64 (0 for SPC-15x/131/160) to 4096. Therefore, the number of complete measurement data sets (or 'pages') depends on the ADC resolution and the number of routing bits used.

SPC-13x/6x0/830/140/15x/160/930 modules in the Fifo modes:

The module memory is configured as a FIFO memory – there are no curves and pages. Instead, a stream of collected photons is written to the fifo. A SPC_configure_memory function call is not required.

SPC-7x0/830/140/15x/131/160 modules, Normal operation modes:

The length of the recorded curves is determined by the ADC resolution and can range from 0 (SPC-830 and other) or 64 (only SPC-7x0) to 4096. Therefore, the number of complete

measurement data sets (or 'pages') depends on the ADC resolution and the number of routing

bits used.

SPC-7x0/830/140/15x/160/930 modules, Scanning modes:

The Memory configuration is not done not by SPC_configure_memory. Instead, the memory is configured by setting the parameters:

ADC_RESOLUTION – defines block_length,

SCAN_SIZE_X, SCAN_SIZE_Y – defines blocks_per_frame

SCAN_ROUT_X, SCAN_ROUT_Y – defines frames_per_page

However, after setting these parameters SPC_configure_memory should be called with 'adc_resolution' = -1 to get the current state of the DLL SPCMemConfig structure.

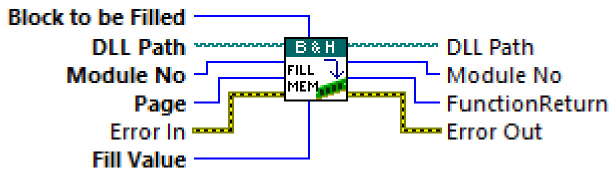
To ensure correct access to the curves in the memory by the memory read/write functions, the SPC_configure_memory function loads a structure of the type SPCMemConfig with the values listed below:

long max_block_no	total number of blocks (=curves) in the memory (per memory bank for the SPC-6x0(13x/15x/160))
long blocks_per_frame	Number of blocks (=curves) per frame
long frames_per_page	Number of frames per page
long maxpage	max number of pages to use in a measurement
short block_length	Number of curve points16-bits words per block (curve)

Possible operation modes for the SPC modules are defined in the spcm_def.h file.

The operation mode can be changed by setting the parameter MODE.

SPC_fill_memory.vi



The procedure is used to clear the measurement memory before a new measurement is started.

If ‘mod_no’ = -1 memory on all used SPC modules will be cleared, otherwise only on the module ‘mod_no’.

The procedure fills a specified part of the SPC memory with the value ‘fill_value’.

To provide correct memory access it is required that the function SPC_configure_memory be used (normal operation modes) before the first call of SPC_fill_memory and always after setting ADC resolution parameter value.

The parameter ‘block’ can range from 0 to blocks_per_page - 1.

If the value ‘-1’ is used all blocks on the specified page(s) are filled.

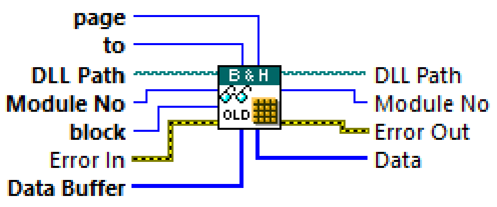
The parameter ‘page’ can vary from 0 to maxpage -1.

If the value ‘-1’ is used all pages in current memory bank are filled.

Attention: The procedure returns on success the number of the modules on which filling the memory was started but is still not finished. If this value is > 0, the function SPC_test_state must be called next to check whether the started filling process is already finished (if the bit SPC_HFILL_NRDY is set in state, filling is not finished, see spcm_def.h for bit definition).

Note: This VI is set to execute as preallocated clone to improve performance when run in parallel loops for multi module setups.

SPC_read_data_block.vi



B&H suggests using the VI **SPC_read_block.vi** instead.

The procedure reads data from a block of the SPC memory (on module ‘mod_no’) defined by the parameters ‘block’ and ‘page’ to the buffer ‘data’. The procedure is used to read measurement results from the SPC memory.

The function performs a data reduction by averaging a specified number of data points into one result value.

The parameter 'reduction_factor' defines the number of points that are averaged.

The value of 'reduction_factor' must be a power of 2.

The number of values stored in 'data'(named below no_of_points) is equal to block length divided by reduction_factor.

The parameters 'from' and 'to' define the address range inside the buffer 'data' i.e. refer to the (compressed) destination data. 'from' and 'to' must be in the range from 0 to no_of_points-1.

The parameter 'to' must be greater than or equal to the parameter 'from'.

The range of the parameters 'block' and 'page' depends on the actual configuration of the SPC memory (see SPC_configure_memory).

The assumption is done that frames_per_page is equal 1 (page = frame) (see 'Memory Configuration).

To provide correct access to the SPC memory it is required that the function SPC_configure_memory be used (in normal operation modes) before the first call of SPC_read_data_block.

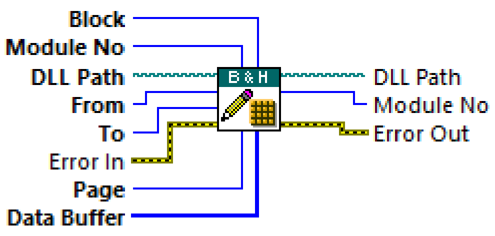
This function also delivers the required information about the block/page structure of the SPC memory:

long max_block_no	total number of blocks (=curves) in the memory (memory bank for the SPC-6x0/13x/15x/160)
long blocks_per_frame	Number of blocks (=curves) per frame
long frames_per_page	Number of frames per page
long maxpage	max number of pages to use in a measurement
long block_length	Number of 16-bits words per one block (curve)

Please make sure that the buffer 'data' be allocated with enough memory for no_of_points.

Note: This VI is set to execute as preallocated clone to improve performance when run in parallel loops for multi module setups.

SPC_write_data_block.vi



The procedure reads data from the buffer 'data' in the PC and writes it to a block of the memory defined by the parameters 'block' and 'page' on the SPC module 'mod_no'. The procedure is used to write data from the from PC memory to the memory of the SPC module.

Parameters ‘from’ and ‘to’ define the address range inside the buffer ‘data’ and the address range inside the SPC memory block to which the data will be written.

The range of the parameters ‘block’ and ‘page’ depends on the actual configuration of the SPC memory (see SPC_configure_memory).

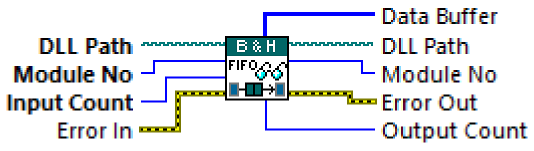
The assumption is done that frames_per_page is equal 1 (page = frame) (see 'Memory Configuration).

To provide correct access to the SPC memory it is required that the function SPC_configure_memory be called (in normal operation modes) before the first call of SPC_write_data_block. This function also delivers the required information about the block/page structure of the SPC memory:

long max_block_no	total number of blocks (=curves) in the memory (memory bank for the SPC-6x0/13x/15x/160)
long blocks_per_frame	Number of blocks (=curves) per frame
long frames_per_page	Number of frames per page
long maxpage	max number of pages to use in a measurement
long block_length	Number of 16-bits words per one block (curve)

Please make sure that the buffer ‘data’ be allocated with enough memory for no_of_points.

SPC_read_fifo.vi



The procedure reads data from the FIFO memory of SPC modules SPC6x0/13x/830/140/15x/160 and has no effect for other SPC module types. Because of hardware differences the procedure action is different for different SPC module types.

<u>For SPC600(630) modules</u>:

Before calling the function, FIFO mode must be set by calling function SPC_set_parameter to change parameter MODE to one of two possible FIFO modes: FIFO_48 (48 bits frame) or FIFO_32 (32 bits frame) (fifo mode values are defined in spcm_def.h file).

<u>For FIFO_48</u> mode the function reads 48-bits frames from the FIFO memory and writes them to the buffer ‘data’ until the FIFO is empty or ‘Count’ number of 16-bit words was already written. The ‘Count’ variable is filled on exit with the number of 16-bit words written to the buffer.

For FIFO_32 mode the function reads 32-bits frames from the FIFO memory and writes them to the buffer 'data' until the FIFO is empty or 'Count' number of 16-bit words was already written. The 'Count' variable is filled on exit with the number of 16-bit words written to the buffer. Subsequent frames which don't contain valid data but only macro time overflow information are compressed to one frame which contains the number of macro time overflows in the bits 29:0. It enables a correct macro time calculation and eliminates invalid data frames from the buffer.

For SPC13x/830/140/15x/160/930 modules:

Before calling the function, FIFO mode must be set by calling function SPC_set_parameter to change parameter MODE to FIFO mode (32 bits frame different than for SPC6x0 modules) (fifo mode values are defined in spcm_def.h file). After setting FIFO mode SPC module memory has FIFO structure. SPC_read_fifo function reads 32-bits frames from the FIFO memory and writes them to the buffer 'data' until the FIFO is empty or 'Count' number of 16-bit words was already written.

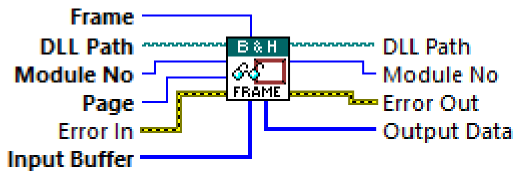
The 'Count' variable is filled on exit with the number of 16-bit words written to the buffer. Subsequent frames which don't contain valid data (photons or markers) but only macro time overflow information are compressed to one frame which contains the number of macro time overflows in the bits 29:0. It enables a correct macro time calculation and eliminates invalid data frames from the buffer.

For SPC830/140/15x/160 modules in FIFO_32M (FIFO IMAGE) mode:

works as above with a difference after calling SPC_stop_measurement - function will return 1 (instead of 0) when photons are read up to stop pointer in FIFO - see explanation in SPC_stop_measurement function description

Please make sure that the buffer 'data' be allocated with enough memory for the expected number of frames (at least 'Count' 16-bit words).

SPC_read_data_frame.vi



The procedure reads data from a frame of the SPC memory on module 'mod_no' defined by the parameters 'frame' and 'page' to the buffer 'data'. The procedure is used to read measurement results from the SPC memory when frames_per_page is greater than 1 (this can be the case for SPC7x0/830/140 modules in scanning modes).

The range of the parameters 'frame' and 'page' depends on the actual configuration of the SPC memory (see SPC_configure_memory).

If 'frame' is equal –1, all frames(frames_per_page) from the page 'page' are read.

To provide correct access to the SPC memory it is required that the function SPC_configure_memory be used (in normal operation modes) before the first call of SPC_read_data_frame. This function also delivers the required information about the block/page structure of the SPC memory:

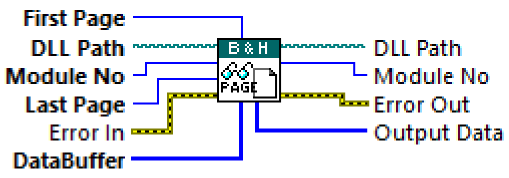
long max_block_no	total number of blocks (=curves) in the memory (memory bank for the SPC-6x0/13x/15x/160)
long blocks_per_frame	Number of blocks (=curves) per frame
long frames_per_page	Number of frames per page
long maxpage	max number of pages to use in a measurement
long block_length	Number of 16-bits words per one block (curve)

Please make sure that the buffer 'data' be allocated with enough memory for

block_length * blocks_per_frame 16-bit values, when one frame is read, or

block_length * blocks_per_frame * frames_per_page 16-bit values, when 'frame' = -1.

SPC_read_data_page.vi



The procedure reads data from the pages of the SPC memory on module 'mod_no' defined by the parameters 'first_page' and 'last_page to the buffer 'data'. The procedure is used to read measurement results from the SPC memory.

The procedure is recommended when big amounts of SPC memory must be read as fast as possible, because it works much faster than calling in the loop the function SPC_read_data_block. Even the whole memory bank can be read in one call, when 'first_page' = 0 and 'last_page' = maxpage – 1.

The range of the parameters 'first_page' and 'last_page' depends on the actual configuration of the SPC memory (see SPC_configure_memory).

To provide correct access to the SPC memory it is required that the function SPC_configure_memory be used (in normal operation modes) before the first call of SPC_read_data_page. This function also delivers the required information about the block/page structure of the SPC memory:

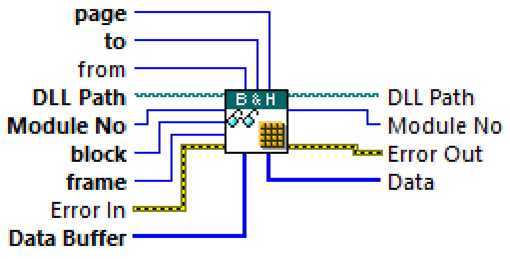
long max_block_no	total number of blocks (=curves) in the memory (memory bank for the SPC-6x0/13x/15x/160)
long blocks_per_frame	Number of blocks (=curves) per frame
long frames_per_page	Number of frames per page
long maxpage	max number of pages to use in a measurement

long block_length	Number of 16-bits words per one block (curve)

Please make sure that the buffer ‘data’ be allocated with enough memory for

block_length * blocks_per_frame * frames_per_page * (last_page – first_page +1) 16-bit values.

SPC_read_block.vi



The procedure reads data from a block of the SPC memory (on module ‘mod_no’) defined by the parameters ‘block’, ‘frame’ and ‘page’ to the buffer ‘data’. The procedure is used to read measurement results from the SPC memory especially for SPC7x0/830/140/15x/160 modules in scanning modes (when frames_per_page is greater than 1).

The parameters ‘from’ and ‘to’ define the address range inside the buffer ‘data’ i.e. refer to the destination data. ‘from’ and ‘to’ must be in the range from 0 to block_length -1.

The parameter ‘to’ must be greater than or equal to the parameter ‘from’.

The range of the parameters ‘block’, ‘frame’ and ‘page’ depends on the actual configuration of the SPC memory (see SPC_configure_memory).

To provide correct access to the SPC memory it is required that the function SPC_configure_memory be used (in normal operation modes) before the first call of SPC_read_block.

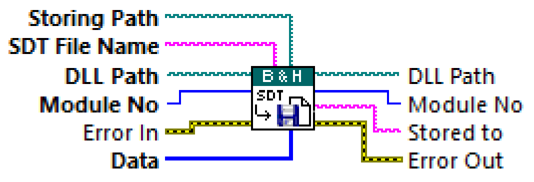
This function also delivers the required information about the block/page structure of the SPC memory:

long max_block_no	total number of blocks (=curves) in the memory (memory bank for the SPC-6x0/13x/15x/160)
long blocks_per_frame	Number of blocks (=curves) per frame
long frames_per_page	Number of frames per page
long maxpage	max number of pages to use in a measurement
long block_length	Number of 16-bits words per one block (curve)

Please make sure that the buffer ‘data’ be allocated with enough memory for block_length.

Note: This VI is set to execute as preallocated clone to improve performance when run in parallel loops for multi module setups.

SPC_save_data_to_sdtfile.vi



This procedure saves measurement data from the buffer 'data_buf' into the 'sdt_file' file in the .sdt format using current parameters of the SPC module 'mod_no'.

The assumption is done, that before using this function user made a measurement, read the results from SPC memory and that no DLL parameters were changed in between.

Because during the measurements always whole page(s) is affected, the best choice to read results is SPC_read_data_page function.

The created .sdt file can then be loaded into the SPC standard measurement software.

It contains a file header, an INFO section, measurement description block(s) and data set(s). It does not contain SETUP section, therefore Display, Trace, Window, Print settings will not change when the file will be loaded to SPC software.

(for .sdt format details see SPC hardware manual and spc_minfo.h file)

One measurement description block and data set is created per one module (measurement description block fields are created from current 'mod_no' module's parameters).

In case of multi-module configuration - if 'mod_no' = -1 (all used modules), measurement description blocks and data sets are created for all modules of the same type which are 'in_use'

(call SPC_get_module_info to know which modules are 'in_use', SPC_set_mode to change used modules configuration)

The 'data_buf' buffer should contain measurement data which were read earlier from SPC memory using one of memory transfer functions (SPC_read_data_page function is recommended).

When 'mod_no' = -1 (all used modules), the buffer should contain the data of all used modules in contiguous way (one after another – after last byte of 1st module's data the first byte of 2nd module should appear).

Please make sure that the buffer is allocated with minimum 'bytes_no' bytes, otherwise it can cause a crash.

The procedure checks at the beginning whether 'bytes_no' parameter fits to the current memory configuration (use SPC_configure_memory with parameter 'adc_resolution' = -1 to get it).

'bytes_no' must be equal to 2 * page_size * no_of_pages * no_of_modules, where

page_size = blocks_per_frame * frames_per_page * block_length, no_of_pages = always 1,

except 'Continuos Flow' mode (for modules SPC6x0/13x/15x/160

with mode = Normal and with sequencer enabled),

where no_of_pages = maxpage,

no_of_modules = 1, when 'mod_no'parameter >= 0,

= number of active modules, when 'mod_no'parameter = -1

Different measurement modes are used in .sdt file depending on the module type and current DLL parameters:

- mode SINGLE: all module types, when DLL parameter MODE = NORMAL

  and sequencer is disabled

- mode 'Continuos Flow': SPC-6x0 & SPC-13x/15x/160 module types, when DLL

  parameter MODE = NORMAL and sequencer is enabled

- mode 'Scan Sync In': SPC-7x0/830/140/930/15x/160 module types, when

  DLL parameter MODE = SCAN_IN and sequencer is enabled

- mode 'Scan Sync Out': SPC-7x0/830/140/930/15x/160 module types, when

  DLL parameter MODE = SCAN_OUT and sequencer is enabled

- mode 'Scan XY Out': SPC-7x0 module type, when DLL parameter MODE =

  ROUT_OUT and sequencer is enabled

- mode 'Camera': SPC-930 module type, when DLL parameter MODE =

  CAMERA

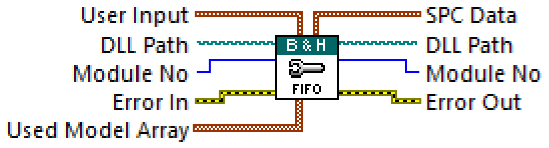
Other combinations of MODE value and module type are not supported and will return error.

Especially the procedure does not create the file (returns error) when DLL parameter MODE is set to one of FIFO modes (for modules SPC-6x0/830/13x/140/930/15x/160).

Currently B&H is working on an extension of this procedure to also cover the FIFO modes. Until than please use the pure LabVIEW VI \FileIO\SDT\SDT_WriteFile.vi in these cases.

Convenience Vis in \SPC_Wrappers\MemoryTransfer

ConfigFIFO.vi



This VI is a wrapper for the **SPC_get_parameters.vi** and **SPC_set_parameters.vi**.

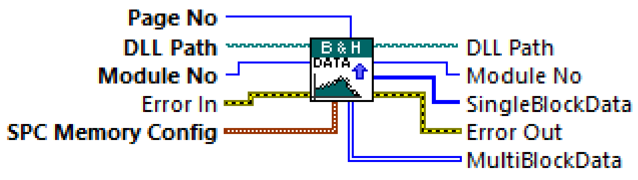
The 'User Input' as well as the correct mode code for FIFO operations at the current module type are set for convenience.

In FIFO modes 12 is the only correct value for adc_resolution in SPCData.

You might want to try different values for dither_range.

stop_on_time and stop_on_ovl might need different values in your setup.

AcqData.vi



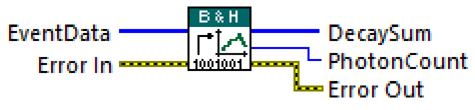
This VI is a wrapper for the **SPC_read_block.vi** calculating the necessary parameters from the 'SPC Memory Config' cluster for convenience.

The memory for 'Data Buffer' gets allocated.

Use the 1D or 2D BlockData output according to your blocks per frame value.

Note: This VI is set to execute as preallocated clone to improve performance when run in parallel loops for multi module setups.

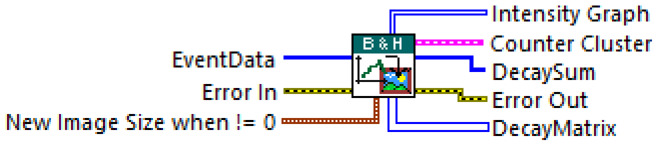
EventStreamToDecayCurve.vi



This VI goes through an EventData-Array received from **SPC_read_fifo.vi** and interprets the events as markers or photons and populates a single decay curve according to their microTime value.

Note: Events are always 2 word wide, and received in reversed byte and word order.

PopulateSingleFrame.vi



This VI goes through an EventData-Array received from **SPC_read_fifo.vi** and interprets the events as markers or photons and populates a DecayMatrix for a single frame.

The 4 dimensions of DecayMatrix are the decay curves according to the microTime values of photons, in their image X- and Y-axis positions and routing channel.

Note the two possible ways of creating an IntensityGraph: With every photon event the intensity could increase per X-Y-coordinate, or the really counted photon number during a pixel event is used (usually higher).

A functional global variable concept is used to initialize the control cluster and output arrays.

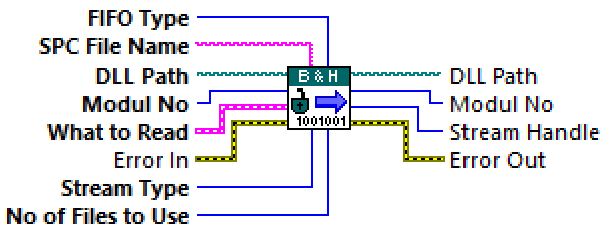
This VI accumulates only events for one frame.

A similar VI is used at the executable example 'ImagingEventStream-FifoImage.exe' that also collects multiple frames with improved memory management and is available at B&H.

Functions to Manage Photons Streams:

\SPC_Wrappers\PhotoStream

SPC_init_phot_stream.vi



fifo_type: 2 (FIFO_48), 3 (FIFO_32), 4 (FIFO_130), 5 (FIFO_830), 6 (FIFO_140),

7 (FIFO_150), 9 (FIFO_IMG)

* spc_file file: path of the 1st spc file

files_to_use: number of subsequent spc files to use, 1..999 or –1 for all files

stream_type: bit 0 = 1- stream of BH .spc files (1st entry in each file contains MT clock, flags & rout.chan info)

bit 0 = 0 - no special meaning of the first entry in the file

bit 9 = 1 - stream contains marker entries (FIFO_IMG mode)

bit 10 = 1 - stream contains raw data (diagnostics only) what_to_read: defines which entries will be extracted

bit 0 = 1 read valid photons, bit 1 = 1 read invalid photons,

bit 2 = 1 read markers 0 (pixel), bit 3 = 1 read markers 1 (line),

bit 4 = 1 read markers 2 (frame), bit 5 = 1 read markers 3

The procedure is needed to initiate the process of extracting photons from a stream of .spc files created during FIFO measurement.

If the files were created using BH measurement software, 1st entry in each file contains Macro Time clock resolution and used routing channels information. In such case bit 0 of ‘stream_type’ parameter should be set to 1, otherwise if the 1st entry have no special meaning (just photon frame) set it to 0. Set bit 9 of ‘stream_type’ if the file contains markers (was created in FIFO_IMG mode or FIFO mode with enabled markers).

Subsequent files created in BH software during one measurement have 3 digits file number in file name part of the path (for example xxx000.spc, xxx001.spc and so on).

Such files can be treated together during extracting photons (they contain the same measurement) as a stream of files.

When all photons from the 1st file will be extracted, the 2nd one will be opened during extraction and so on. The first file in the stream is given by ‘spc_file’ parameter and ‘files_to_use’ parameter tells how many files belong to the stream (-1 means the procedure.will evaluate number of files in the stream and use it as ‘files_to_use’ value).

‘fifo_type parameter defines the format of the photons data in the stream. It depends mainly on the SPC module type which was used during the measurement.

Possible ‘fifo_type’ values are defined in the spcm_def.h file.

‘what_to_read’ parameter defines which entries will be extracted from the stream.

In most cases only bit 0 will be set (valid photons). For files containing also markers –

markers 0-3 (pixel, line, frame) can also be extracted (bits 2-5).

If the stream is successfully initialised, the procedure creates internal DLL PhotStreamInfo structure and returns the handle to the stream (positive value). Use this handle as an input

parameter to the other extraction functions (SPC_close_phot_stream, SPC_get_phot_stream_info, SPC_get_photon).

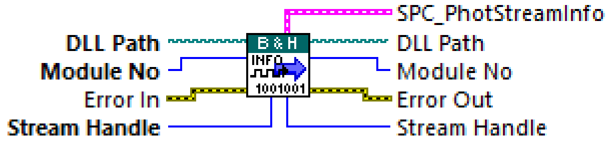
Max 8 streams can be initialised by the SPCM DLL.

Use SPC_get_phot_stream_info to get the current state of the stream and SPC_get_photon to extract subsequent photons from the stream.

After extracting photons stream should be closed using SPC_close_phot_stream function.

See use_spcm.c file for the extraction example.

SPC_get_phot_stream_info.vi

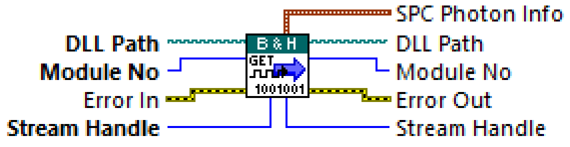


The procedure fills 'stream_info' structure with the contents of DLL internal structure of the stream defined by handle 'stream_hndl'.

Procedure returns error, if 'stream_hndl' is not the handle of the stream opened using SPC_init_phot_stream function.

PhotStreamInfo structure is defined in the spcm_def.h file.

SPC_get_photon.vi



The procedure can be used in a loop to extract subsequent photons from the opened photons stream defined by handle 'stream_hndl'. It accepts only streams of spc files.

To extract photons from buffered photons streams SPC_get_photons_from_stream function can be used.

The procedure fills 'phot_info' structure with the information of the photon extracted from the current stream position. After extracting procedure updates internal stream structures.

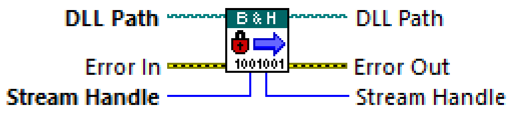
If needed, it opens and read data from the next stream file.

Use SPC_get_phot_stream_info function to get current stream state.

Procedure returns error, if 'stream_hndl' is not the handle of the stream opened using SPC_init_phot_stream function.

PhotInfo structure is defined in the spcm_def.h file.

SPC_close_phot_stream.vi



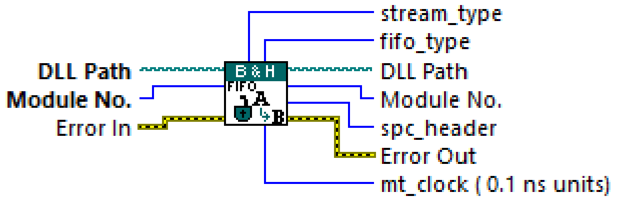
The procedure is used to close the opened photons stream defined by handle 'stream_hndl' after extraction of the photons.

The procedure frees all stream's memory and finally invalidates the handle 'stream_hndl'.

Procedure returns error, if ‘stream_hndl’ is not the handle of the stream opened using

SPC_init_phot_stream function.

SPC_get_fifo_init_vars.vi



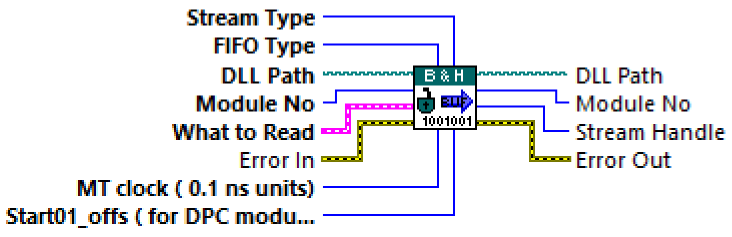
This procedure sets variables to be used as input parameters for SPC_init_buf_stream function.

It can also prepare .spc file header (1st word of .spc file), if saving .spc files is required.

The procedure is intended to be used directly before starting FIFO measurement to initialize a ‘buffered’ stream.

If photons are added to the stream not from running FIFO measurement, but e.g. .spc files, then input parameters for SPC_init_buf_stream must be taken from .spc file header.

SPC_init_buf_stream.vi



fifo_type: 2 (FIFO_48), 3 (FIFO_32), 4 (FIFO_130), 5 (FIFO_830), 6 (FIFO_140),

7 (FIFO_150), 9 (FIFO_IMG)

stream_type: bit 0 has no special meaning for buffered streams, is set to 1

bits 1,2,8 = 0, used only for DPC-230 FIFO data

bit 9 = 1 - stream contains marker entries (FIFO_IMG mode)

bit 10 = 1 - stream contains raw data (diagnostics only)

bit 12 = 1 – indicates buffered stream type

bit13 = 1 – stream buffer freed automatically after extracting photons from it

= 0. stream buffer freed using SPC_close_phot_stream

what_to_read: defines which entries will be extracted

bit 0 = 1 read valid photons, bit 1 = 1 read invalid photons, bit 2 = 1 read markers 0 (pixel), bit 3 = 1 read markers 1 (line),

bit 4 = 1 read markers 2 (frame), bit 5 = 1 read markers 3

The procedure is needed to initiate the process of extracting photons from a stream of photons placed into PC memory buffers previously 'buffered' stream.

To get input parameters needed to call SPC_init_buf_stream (fifo_type, stream_type, mt_clock) call SPC_get_fifo_init_vars function, when you are ready to start the FIFO measurement.

If the photons are taken from .spc files, input parameters should be taken from .spc file header (1st word).

Set bit 9 of 'stream_type' if the file contains markers (was created in FIFO_IMG mode or FIFO mode with enabled markers).

Buffers are allocated/reallocated automatically while adding photons to the stream.

The buffers can be freed in two ways depending on an option FREE_BUF_STREAM (bit 13 in 'stream_type' parameter).

If bit 13 is not set, the buffers are freed when the stream is closed (SPCM_close_phot_stream).

If bit 13 is set, the buffer will be freed, when, during SPC_get_photons_from_stream call, all photons from the current buffer are extracted.

After this it will be not possible to use the buffer again, for example to get data from it using SPC_get_buffer_from_stream function.

FREE_BUF_STREAM option is recommended for long measurements with lots of data read from FIFO, which could make buffers allocated space very (too) big.

If the photons rate or measurement time is not very big/long, buffers can stay allocated and another extract action can be started (with different start/stop condition) or buffers contents can be stored in .spc file.

User can add photons to the stream buffers by using function:

SPC_add_data_to_stream – photons are taken from the buffer (input buffer can be filled from .spc file)

SPC_read_fifo_to_stream - photons are read from FIFO during running FIFO measurement.

Adding photons to the stream or extracting photons can be done also during running measurement. During extracting or adding photons the stream is locked. Only one thread can access the thread safe stream at a time. If a thread requests the access to stream resources while another has it, the second thread waits in this function until the first thread releases the

stream.

Use function SPC_get_photons_from_stream to extract photons information from the stream buffers.

'fifo_type parameter defines the format of the photons data in the stream. It depends mainly on the SPC module type which was used during the measurement. Possible 'fifo_type' values are defined in the spcm_def.h file.

'what_to_read' parameter defines which entries will be extracted from the stream.

In most cases only bit 0 will be set (valid photons). For streams containing also markers –

markers 0-3 (pixel, line, frame) can also be extracted (bits 2-5).

If the stream is successfully initialised, the procedure creates internal DLL PhotStreamInfo structure and returns the handle to the stream (positive value). Use this handle as an input parameter to the other extraction functions (SPC_close_phot_stream, SPC_get_phot_stream_info, SPC_get_photons_from_stream and so on).

Max 8 streams can be initialised by the SPCM DLL.

Use SPC_get_phot_stream_info to get the current state of the stream and SPC_get_photons_from_stream to extract subsequent photons from the stream.

Using SPC_stream_start_condition and SPC_stream_stop_condition user can define start/stop condition of extracting photons, which can be specific macro time and/or occurrence of markers or routing channels.

As long as stream buffers are not freed user can call SPC_reset_stream and then extract photons again with another start/stop condition.

After extracting photons stream should be closed using SPC_close_phot_stream function.

See use_spcm.c file for the extraction example.

stream_type

bit 0 has no special meaning for buffered streams, is set to 1

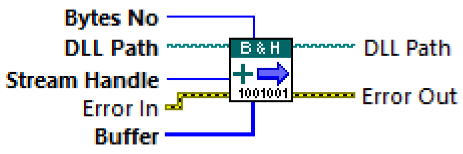
bits 1,2,8 = 0, used only for DPC-230 FIFO data

bit 9	= 1 - stream contains marker entries (FIFO_IMG mode)
bit 10	= 1 - stream contains raw data (diagnostics only)
bit 12	= 1 – indicates buffered stream type
bit 13	= 1 – stream buffer freed automatically after extracting photons from it
	= 0. stream buffer freed using SPC_close_phot_stream

what_to_read defines which entries will be extracted

bit 0	= 1 read valid photons,
bit 1	= 1 read invalid photons,
bit 2	= 1 read markers 0 (pixel),
bit 3	= 1 read markers 1 (line),
bit 4	= 1 read markers 2 (frame),
bit 5	= 1 read markers 3

SPC_add_data_to_stream.vi



The procedure can be used to add photons data to the opened 'buffered' photons stream defined by handle 'stream_hndl'.

Photons data in the input buffer can be taken from .spc files or read from module's FIFO.

Data are added to DLL internal buffers which have size between STREAM_MIN_BUF_SIZE and STREAM_MAX_BUF_SIZE.

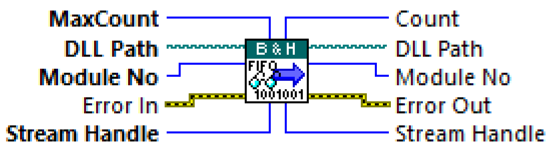
Internal buffers are allocated by DLL when required.

Maximum size of allocated stream buffers is equal STREAM_MAX_SIZE32 for 32-bit DLL and STREAM_MAX_SIZE64 for 64-bit DLL.

The buffers are freed, when the stream is closed (SPCM_close_phot_stream) or when, during SPC_get_photons_from_stream call, all photons from the current buffer are extracted (if an option FREE_BUF_STREAM (bit 13 in 'stream_type') is set).

Procedure returns error, if 'stream_hndl' is not the handle of the stream opened using SPC_init_buf_stream function.

SPC_read_fifo_to_stream.vi



The procedure can be used to add photons data to the opened 'buffered' photons stream defined by handle 'stream_hndl'.

Photons are read from the FIFO on SPC module 'mod_no' during running FIFO measurement.

The 'Count' variable is filled on exit with the number of 16-bit words added to the stream.

See also the description of SPC_read_fifo procedure because it is called internally.

Photons data read from FIFO are added to DLL internal buffers which have size between STREAM_MIN_BUF_SIZE and STREAM_MAX_BUF_SIZE.

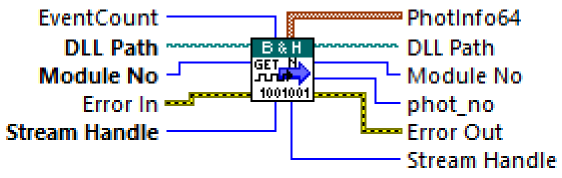
Internal buffers are allocated by DLL when required.

Maximum size of allocated stream buffers is equal STREAM_MAX_SIZE32 for 32-bit DLL and STREAM_MAX_SIZE64 for 64-bit DLL.

The buffers are freed, when the stream is closed (SPCM_close_phot_stream) or when, during SPC_get_photons_from_stream call, all photons from the current buffer are extracted (if an option FREE_BUF_STREAM (bit 13 in 'stream_type') is set).

Procedure returns error, if 'stream_hndl' is not the handle of the stream opened using SPC_init_buf_stream function.

SPC_get_photons_from_stream.vi



The procedure is used to extract photons data from the opened 'buffered' photons streamdefined by handle 'stream_hndl'.

Photons data are packed to the structures PhotInfo64 (defined in spcm_def.h file) in 'phot_info' buffer.

EventCount is the upper limit number of expected photons; used for array allocation.

The 'phot_no' variable is filled on exit with the number of photons actually extracted from the stream.

Extracting photons can be done also during running measurement. During extracting or adding photons the stream is locked. Only one thread can access the thread safe stream at a time. If a thread requests the access to stream resources while another has it, the second thread waits in this function until the first thread releases the stream.

Using SPC_stream_start_condition and SPC_stream_stop_condition user can define start/stop condition of extracting photons, which can be specific macro time and/or occurrence of markers or routing channels.

User can define start and stop condition for extracting photons using functions SPC_stream_start(stop)_condition.

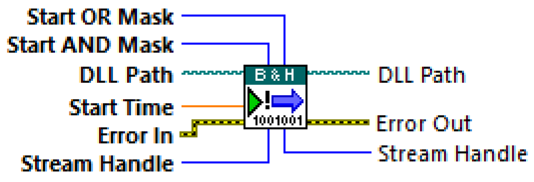
The condition can be a specified macro time value and/or occurrence of specific markers and/or routing channels. See description of SPC_stream_start(stop)_condition functions.

Photons extracted to 'phot_info' buffer can be saved to .ph file.

First 4 bytes of .ph file it is a header (the same as for .spc files). Call SPC_get_fifo_init_vars to get the header value. After the header subsequent PhotInfo64 photons structures are stored.

Such file can be used in SPCM software as an input file in 'Convert FIFO Files' panel.

SPC_stream_start_condition.vi



-start_time: macro time (in sec) at which extracting photons will start (during call to SPC_get_photons_from_stream)

-start_OR_mask: bitwise OR mask used to define start of extracting photons from the stream (in addition to 'start_time'), bits 31-28 – markers M3-M0, bits 27-0 - routing channels 27-0

-start_AND_mask: bitwise AND mask used to define start of extracting photons from the stream (in addition to 'start_time'), bits 31-28 – markers M3-M0, bits 27-0 - routing channels 27-0.

The procedure is used to define start of extracting photons from buffered stream 'stream_hndl' during SPC_get_photons_from_stream call.

Start_time and OR/AND masks can be used all together.

All photons (markers) are ignored until the macro time in the stream reaches 'start_time'

value. From this moment appearance of specified markers/channels is tested according to start_OR(AND)_mask.

Start condition is found when minimum one of markers/channels defined in start_OR_mask appears in the stream (since 'start_time').

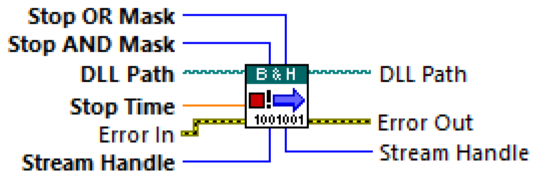
Start condition is also found when all of markers/channels defined in start_AND_mask

appeared in the stream (since 'start_time').

SPC_get_photons_from_stream returns an error, when start condition cannot be found in the stream.

While extracting photons during running measurement, start condition (if defined) can be found later after reading new portion of photons data from FIFO to the stream (using SPC_read_fifo_to_stream).

SPC_stream_stop_condition.vi



-stop_time: macro time (in sec) at which extracting photons will stop (during call to SPC_get_photons_from_stream) (if stop masks are not defined)

-stop_OR_mask: bitwise OR mask used to define stop of extracting photons from the stream (in addition to 'stop_time'), bits 31-28 – markers M3-M0, bits 27-0 - routing channels 27-0

-stop_AND_mask bitwise AND mask used to define stop of extracting photons from the stream (in addition to 'stop_time'), bits 31-28 – markers M3-M0, bits 27-0 - routing channels 27-0.

The procedure is used to define stop of extracting photons from buffered stream 'stream_hndl' during SPC_get_photons_from_stream call.

Stop_time and OR/AND masks can be used all together.

All photons (markers) are extracted until the macro time in the stream reaches 'stop_time' value.

From this moment appearance of specified markers/channels is tested according to stop_OR(AND)_mask.

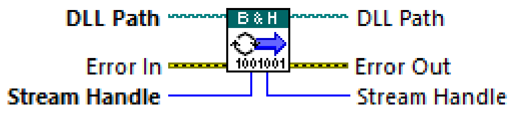
Stop condition is found when minimum one of markers/channels defined in stop_OR_mask appears in the stream (since 'stop_time').

Stop condition is also found when all of markers/channels defined in stop_AND_mask appeared in the stream (since 'stop_time').

SPC_get_photons_from_stream returns an error, when stop condition cannot be found in the stream.

While extracting photons during running measurement, stop condition (if defined) can be found later after reading new portion of photons data from FIFO to the stream (using SPC_read_fifo_to_stream).

SPC_stream_reset.vi



The procedure resets buffered stream 'stream_hndl' to the state before extracting the photons without affecting stream's internal data buffers. After this user can define new stream's tart/stop condition and extract photons once more from the beginning of the stream using new conditions.

But, attention, this is possible only when stream's data buffers are not freed after extracting photons.

An option FREE_BUF_STREAM (bit 13 in 'stream_type' parameter while initializing the stream using SPC_init_buf_stream function) defines a way in which stream's buffers are freed.

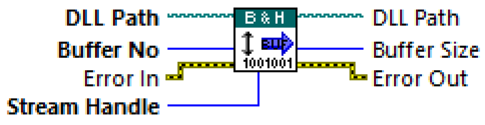
If it is set, the buffer is freed, when, during SPC_get_photons_from_stream call, all photons

from the current buffer are extracted.

After this it will be not possible to use the buffer again, for example to get data from it using SPC_get_buffer_from_stream function.

Therefore SPC_reset_stream can be used when FREE_BUF_STREAM is not used – buffers are not freed and extracting photons can be repeated.

SPC_get_stream_buffer_size.vi



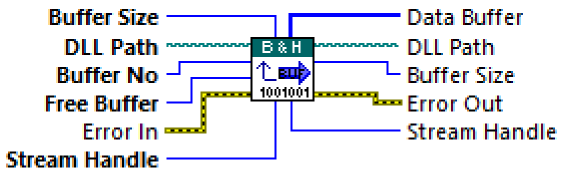
Before calling SPC_get_buffer_from_stream user must know the buffer size and should allocate big enough data buffer.

The SPC_get_stream_buffer_size procedure is used to get size of stream's internal buffer number 'buf_no'.

Stream is defined by handle 'stream_hndl'. The procedure works only with 'buffered' streams.

Call SPC_get_phot_stream_info to get info about current number of stream buffers (field no_of_buf of PhotStreamInfo structure).

SPC_get_buffer_from_stream.vi



The procedure is used to get contents of stream's internal buffers with photons data to the buffer 'data_buf'. Stream is defined by handle 'stream_hndl'.

The procedure works only with 'buffered' streams.

The procedure fills 'data_buf' with contents of stream's buffer 'buf_no'.

‘data_buf’ must be allocated with minimum ‘buf_size’ bytes.

Use SPC_get_stream_buffer_size procedure to get size of buffer ‘buf_no’, which means the required ‘buf_size’ value.

On exit ‘buf_size’ is set to real number of bytes copied to ‘data_buf’.

On demand, the buffer ‘buf_no’ can be freed on procedure exit, when ‘free_buf’ parameter equals 1.

Call SPC_get_phot_stream_info to get info about current number of stream buffers (field no_of_buf of PhotStreamInfo structure).

Be aware of option FREE_BUF_STREAM (bit 13 in ‘stream_type’ parameter while initializing the stream using SPC_init_buf_stream function) It defines a way in which stream’s buffers are freed. If it is set, the buffer is freed, when, during SPC_get_photons_from_stream call, all photons from the current buffer are extracted.

After this it will be not possible to get data from it using SPC_get_buffer_from_stream function.

Data taken from stream’ buffers can be stored in .spc file for future use (for example in SPCM application panel ‘Convert FIFO files’).

First 4 bytes of .spc file it is a header - call SPC_get_fifo_init_vars to get the header value.

After the header, subsequent stream’s buffers should be stored.

Advanced Package VIs

When purchasing the advanced package “BH-LabVIEW-SPC-FullVersion” you will find next to all the above-mentioned VIs another folder called “Advanced”.
In this folder you will find VIs based on the above-mentioned VIs in the Example folder, but with an extension on performance speed and user operationability. These Vis compile to fully operational standalone application covering all the standard use scenarios of SPC modules.

Should you be interested in extending a fully operational measurement and data collection application to your specific needs, these VIs would be your convenient starting point.

Advanced Application Vis

ImagingEventStream-FifoImage.vi

This VI is an extension of the “ImagingEventStream-SimpleFifoImage.vi” from the Examples folder.

Improved memory management allows for Fifo-Imaging recording of images up to 4096x4096pixel with ADC-resolution of 64 or alternatively 512x512pixel with ADC-resolution of 4096. This is close to the LV limit for array sizes of 2^31.

This VI is properly LV-Event driven and the GUI experience is optimized. It can be remotely controlled.

BH_EventStream-GetPhotonInfo.vi

This VI is an extension of the "EventStream-GetPhotonInfo.vi" from the Examples folder.

- Improved memory management will allow for longer data acquisition with higher count rates.
- Routing Channels are properly separated.
- Several types of measurement end conditions are selectable.
- Several types of data saving formats are selectable.
- A PeakHeight and FWHM calculation on the Decay graph is included
- A FCS calculation is included.
- Adjustable cursors on the MCS TimeLine graph allow for "zoom"-function on the Decay graph
- The GUI experience is optimized.
- It can be remotely controlled.

BH_HWhist-SingleCurve.vi

This VI is an extension of the "HWhist-SingleCurve.vi" from the Examples folder.

The GUI experience is optimized and it can be remotely controlled.

PopulateDecayMatrix.vi

This Vi is used by ImagingEventStream-FifoImage.vi to analyse the photon event stream and populate the 4D decay matrix with it.

ScanCFDLimitLow.vi

This Vi is an example for how several compiled VIs can be remotely controlled to achieve an automated adjustment process.
Here "BH_EventStream-GetPhotonInfo.exe" and "BH-DCC.exe" are controlled to scan through several values of "cfd_limit_low" and plot the resulting graph.

DecayOverTimeShift.vi

This Vi is an example for how several graphs can visualize the data from a *.sdt-file.

Ini2SPCDataCluster.vi

This is a convenience vi for the advanced application vis, reading the SPC parameters from a file and populating the LV-controls.

SPCDataCluster2UserInput_WithCFG.vi

This is a convenience vi for the advanced application vis, saving the SPC-parameters from LV-controls to a file.

About.vi and BHAboutLogo.ctl

This Vi displays the About box and copyright from Becker&Hickl and National Instruments' LabVIEW. To comply with copyright laws, this Vi needs to be included in a compiled application using B&H LV software components.

Functions Folder

This folder contains several support VIs for the advanced application vis. Mainly concerning the GUI experience, data flow and remote controlling.

PieChart Folder

This folder contains support VIs for the ImagingEventStream-FifoImage.vi to display the PeakHightHistogram-PieChart.

DataAnalysis Folder

Arbitary analysis routines

Ini-Files Folder

This folder contains *.ini text file being used as templates for the application builder.

Builds Folder

Ready compiled applications of the advanced application vis.

Appendix

End-User License Agreement (EULA) of SPC-LVD

This End-User License Agreement ("EULA") is a legal agreement between you and Becker & Hickl GmbH.

This EULA agreement governs your acquisition and use of our SPC-LVD software ("Software") directly from Becker & Hickl GmbH or indirectly through a Becker & Hickl GmbH authorized reseller or distributor (a "Reseller").

Please read this EULA agreement carefully before completing the installation process and using the SPC-LVD software. It provides a license to use the SPC-LVD software and contains warranty information and liability disclaimers.

If you register for a free trial of the SPC-LVD software, this EULA agreement will also govern that trial. By clicking "accept" or installing and/or using the SPC-LVD software, you are confirming your acceptance of the Software and agreeing to become bound by the terms of this EULA agreement.

If you are entering into this EULA agreement on behalf of a company or other legal entity, you represent that you have the authority to bind such entity and its affiliates to these terms and conditions. If you do not have such authority or if you do not agree with the terms and conditions of this EULA agreement, do not install or use the Software, and you must not accept this EULA agreement.

This EULA agreement shall apply only to the Software supplied by Becker & Hickl GmbH herewith regardless of whether other software is referred to or described herein. The terms also apply to any Becker & Hickl GmbH updates, supplements, Internet-based services, and support services for the Software, unless other terms accompany those items on delivery. If so, those terms apply.

License Grant

Becker & Hickl GmbH hereby grants you a personal, non-transferable, non-exclusive license to use the SPC-LVD software on your devices in accordance with the terms of this EULA agreement.

You are permitted to load the SPC-LVD software (for example on a PC, laptop, mobile or tablet) under your control. You are responsible for ensuring your device meets the minimum requirements of the SPC-LVD software.

You are not permitted to:

- Reproduce, copy, distribute, resell the Software
- Allow any third party to use the Software on behalf of or for the benefit of any third party
- Use the Software in any way which breaches any applicable local, national or international law
- use the Software for any purpose that Becker & Hickl GmbH considers is a breach of this EULA agreement

Intellectual Property and Ownership

Becker & Hickl GmbH shall at all times retain ownership of the Software as originally downloaded by you and all subsequent downloads of the Software by you. The Software (and the copyright, and other intellectual property rights of whatever nature in the Software, including any modifications made thereto) are and shall remain the property of Becker & Hickl GmbH.

Becker & Hickl GmbH reserves the right to grant licenses to use the Software to third parties.

Termination

This EULA agreement is effective from the date you first use the Software and shall continue until terminated. You may terminate it at any time upon written notice to Becker & Hickl GmbH.

It will also terminate immediately if you fail to comply with any term of this EULA agreement. Upon such termination, the licenses granted by this EULA agreement will immediately terminate and you agree to stop all access and use of the Software. The provisions that by their nature continue and survive will survive any termination of this EULA agreement.

Governing Law

This EULA agreement, and any dispute arising out of or in connection with this EULA agreement, shall be governed by and construed in accordance with the laws of Germany.