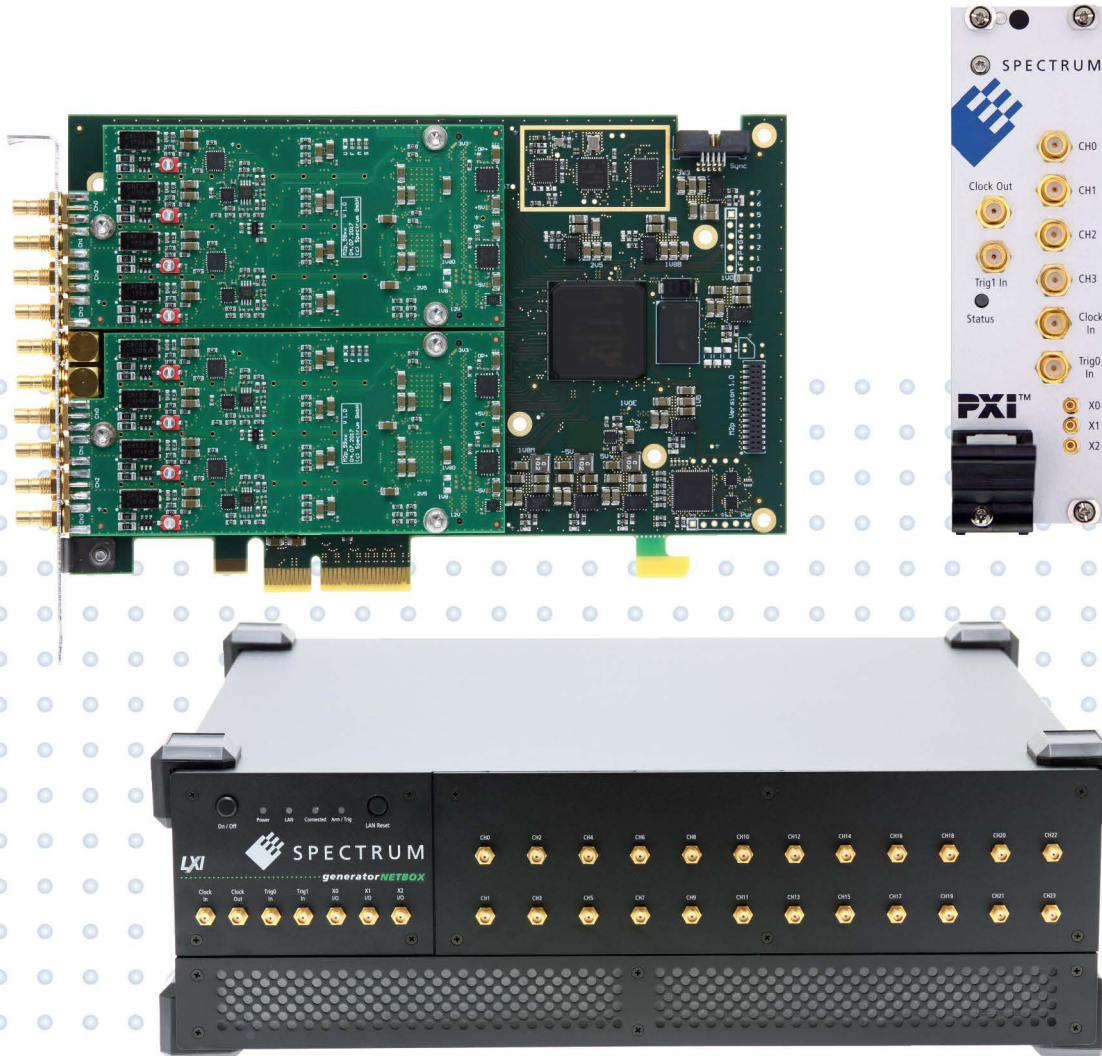




SPECTRUM
INSTRUMENTATION

Perfect fit – modular designed solutions



M4i.77xx-x8

**high-speed Digital Waveform Acquisition
for PCI Express bus**

valid for all versions

**Hardware Manual
Software Driver Manual**

Manual printed: 23. February 2024

Digitizers | Transient Recorders | Arbitrary Waveform Generators | Digital Waveform Acquisition Cards
for PCI Express, PXI Express and LXI / Ethernet

(c) SPECTRUM INSTRUMENTATION GMBH
AHRENSFELDER WEG 13-17, 22927 GROSSHANS DORF, GERMANY

SBench, digitizerNETBOX, generatorNETBOX and hybridNETBOX are registered trademarks of Spectrum Instrumentation GmbH.
Microsoft, Visual C++, Windows, Windows 98, Windows NT, Windows 2000, Windows XP, Windows Vista, Windows 7, Windows 8, Windows 10, Windows 11 and Windows Server are trademarks/registered trademarks of Microsoft Corporation.
LabVIEW, DASyLab, Diadem and LabWindows/CVI are trademarks/registered trademarks of National Instruments Corporation.
MATLAB is a trademark/registered trademark of The Mathworks, Inc.
Delphi and C++Builder are trademarks or registered trademarks of Embarcadero Technologies, Inc.
Keysight VEE, VEE Pro and VEE OneLab are trademarks/registered trademarks of Keysight Technologies, Inc.
FlexPro is a registered trademark of Weisang GmbH & Co. KG.
PCIe, PCI Express, PCI-X and PCI-SIG are trademarks of PCI-SIG.
PICMG and CompactPCI are trademarks of the PCI Industrial Computer Manufacturers Group.
PXI is a trademark of the PXI Systems Alliance.
LXI is a registered trademark of the LXI Consortium.
IVI is a registered trademark of the IVI Foundation.
Oracle and Java are registered trademarks of Oracle and/or its affiliates.
Python is a trademark/registered trademark of Python Software Foundation.
Julia is a trademark/registered trademark of Julia Computing, Inc.
Intel and Intel Core i3, Core i5, Core i7, Core i9 and Xeon are trademarks and/or registered trademarks of Intel Corporation.
AMD, Opteron, Sempron, Phenom, FX, Ryzen and EPYC are trademarks and/or registered trademarks of Advanced Micro Devices.
Arm is a trademark or registered trademark of Arm Limited (or its subsidiaries).
NVIDIA, CUDA, GeForce, Quadro, Tesla and Jetson are trademarks and/or registered trademarks of NVIDIA Corporation.

Table of Contents

Table of Contents	3
Safety Instructions	8
Symbols and Safety Labels	8
General safety information	8
Requirements for users and duties for operators.....	8
General safety at work.....	8
Bringing the product into service	9
Intended use	9
Application area of the product.....	9
Requirements for the technical state of the product	9
Requirements for operation	9
Electrical safety and power supply	9
Requirements for the location	9
Requirements on the ventilation	9
Maintenance.....	9
Repair/Service	10
Cleaning the module housing (NETBOX devices, cables, amplifiers, systems only)	10
Opening the module (NETBOX devices, amplifiers only).....	10
Dismounting parts of the card (instrument card only)	10
Markings and Labelling.....	10
Packing list.....	11
PCIe (PCI Express) cards M4i.xxxx	11
PXIe (PXI Express) cards M4x.xxxx.....	11
Introduction.....	12
Preface	12
Overview	12
M4i cards for PCIe Express (PCIe)	12
M4x cards for PXI Express (PXIe)	12
General Information	13
Connector location at the M4i.77xx series	14
Additional options.....	15
Star-Hub (M4i only)	15
Digital I/O with Dig-SMA (M4i.44xx only)	15
The Spectrum type plate	16
Hardware information.....	17
Block diagram differential input version	17
Block diagram single-ended input version	17
Technical Data	18
Order Information.....	21
Hardware Installation	22
ESD Precautions	22
Sources of noise.....	22
Cooling Precautions.....	22
Connector Handling Precautions	22
M4i PCIe Cards	23
System Requirements.....	23
Installing the M4i board in the system	23
Additional notes on PCIe x16 slot retention.....	24
Providing additional power to M4i.xxxx-x8 cards.....	24
Installing a M4i.44xx board with digital inputs/outputs mounted on an extra bracket	24
Installing multiple boards synchronized by star-hub option	25
Shipment of systems with Spectrum cards installed	26
M4x PXIe Cards.....	26
System Requirements.....	26
Installing the M4x board in the system	26

Software Driver Installation and Driver Update.....	27
Windows	27
Before initial installation	27
Running the driver Installer/Update.....	27
After installation	28
Linux.....	29
Overview	29
Driver Installation with Installation Script.....	29
Standard Driver Update	30
Compilation of kernel driver sources (optional and local cards only)	30
Update of a self compiled kernel driver	30
Installing the library only without a kernel (for remote devices)	31
Installation from Spectrum Repository	31
Control Center	32
Software	34
Software Overview.....	34
Card Control Center	34
Discovery of Remote Cards, digitizerNETBOX/generatorNETBOX/hybridNETBOX products	35
Wake On LAN of digitizerNETBOX/generatorNETBOX/hybridNETBOX.....	35
Netbox Monitor	36
Device identification	36
Hardware information.....	37
Firmware information	37
Software License information.....	38
Driver information.....	38
Installing and removing Demo cards	39
Feature upgrade.....	39
Software License upgrade.....	39
Performing card calibration (A/D only)	39
Performing memory test	40
Transfer speed test.....	40
Debug logging for support cases	40
Device mapping	41
Firmware upgrade	41
Accessing the hardware with SBench 6.....	42
C/C++ Driver Interface.....	42
Header files	42
General Information on Windows 64 bit drivers.....	43
Microsoft Visual C++ 6.0, 2005 and newer 32 Bit.....	43
Microsoft Visual C++ 2005 and newer 64 Bit.....	43
Linux Gnu C/C++ 32/64 Bit	43
C++ for .NET	44
Other Windows C/C++ compilers 32 Bit.....	44
Other Windows C/C++ compilers 64 Bit.....	44
Driver functions	44
Delphi (Pascal) Programming Interface	49
Driver interface	49
Examples.....	51
.NET programming languages	52
Library	52
Declaration.....	52
Using C#.....	52
Using Managed C++/CLI.....	54
Using VB.NET	54
Using J#	54
Python Programming Interface and Examples	55
Driver interface	55
Examples.....	55
Java Programming Interface and Examples	56
Driver interface	56
Examples.....	56
Julia Programming Interface and Examples	57
Driver interface	57
Examples.....	57
LabVIEW driver and examples	58
MATLAB driver and examples	58
SCAPP – CUDA GPU based data processing	59

Programming the Board	60
Overview	60
Register tables	60
Programming examples.....	60
Initialization.....	61
Initialization of Remote Products	61
Error handling.....	61
Gathering information from the card.....	62
Card type	63
Hardware and PCB version	63
Reading currently used PXI slot No. (M4x only)	63
Firmware versions.....	64
Production date	64
Last calibration date (analog cards only)	64
Serial number	65
Maximum possible sampling rate	65
Installed memory	65
Installed features and options	65
Miscellaneous Card Information	66
Function type of the card	67
Used type of driver	67
Custom modifications	68
Reset.....	68
Digital Inputs.....	69
Channel Selection	69
Important note on channel selection	69
Read out of input features	69
Setting up the input threshold and termination (single-ended models only).....	70
Acquisition modes	71
Overview	71
Setup of the mode	71
Commands.....	72
Card Status.....	73
Acquisition cards status overview	74
Generation card status overview	74
Data Transfer	74
Standard Single acquisition mode	77
Card mode	77
Memory, Pre- and Posttrigger	77
Example	77
.....	78
FIFO Single acquisition mode	78
Card mode	78
Length and Pretrigger	78
Difference to standard single acquisition mode.....	78
Example FIFO acquisition	79
Limits of pre trigger, post trigger, memory size	79
Buffer handling	80
Data organisation	83
Clock generation	84
Overview	84
Clock Mode Register.....	84
The different clock modes	84
Details on the different clock modes.....	85
Standard internal sampling clock (PLL)	85
Oversampling	85
External clock (reference clock)	86
PXI Reference Clock (M4x cards only)	87
Direct external clock (state clock)	87
Clock Delay (for state clock)	87
Clock Edge Selection	88
Setting up Ext0 clock threshold and termination (single-ended models only)	88

Trigger modes and appendant registers	89
General Description.....	89
Trigger Engine Overview.....	89
Trigger masks	90
Trigger OR mask	90
Trigger AND mask.....	91
Software trigger	93
Force- and Enable trigger	93
Trigger delay	94
External trigger	95
External Trigger Mode	95
Setting up the trigger Ext0 threshold and termination (single-ended models only)	95
Channel (pattern) trigger	96
Channel Trigger Mode.....	96
Edge and level triggers	97
Examples for pattern matching.....	98
Multi Purpose I/O Lines	100
On-board I/O lines (X0, X1, X2)	100
Using asynchronous I/O	100
Special behaviour of trigger output.....	101
Setting up the X0, X1 threshold and termination (single-ended models only)	101
Mode Multiple Recording	102
Recording modes	102
Standard Mode.....	102
FIFO Mode	102
Limits of pre trigger, post trigger, memory size	103
Multiple Recording and Timestamps	104
Trigger Modes	104
Programming examples.....	104
Mode Gated Sampling	106
Acquisition modes	106
Standard Mode.....	106
FIFO Mode	106
Limits of pre trigger, post trigger, memory size	107
Programming examples.....	107
Timestamps	109
General information	109
Example for setting timestamp mode:	110
Timestamp modes.....	110
Standard mode	110
StartReset mode.....	110
Refclock mode.....	111
Reading out the timestamps	112
General.....	112
Data Transfer using DMA	113
Data Transfer using Polling	114
Comparison of DMA and polling commands.....	115
Data format	115
Combination of Memory Segmentation Options with Timestamps	116
Multiple Recording and Timestamps	117
Option Star-Hub (M3i and M4i only)	118
Star-Hub introduction	118
Star-Hub trigger engine	118
Star-Hub clock engine	118
Software Interface	118
Star-Hub Initialization.....	118
Setup of Synchronization	120
Setup of Trigger	121
Run the synchronized cards	121
SH-Direct: using the Star-Hub clock directly without synchronization	122
Error Handling	123

Option Remote Server	124
Introduction	124
Installing and starting the Remote Server	124
Windows	124
Linux	124
Detecting the digitizerNETBOX/generatorNETBOX/hybridNETBOX	124
Discovery Function.....	124
Finding the digitizerNETBOX/generatorNETBOX/hybridNETBOX in the network	125
Troubleshooting.....	126
Accessing remote cards	126
Appendix	127
Error Codes	127
Spectrum Knowledge Base	128
Temperature sensors	129
Temperature read-out registers	129
Temperature hints	129
77xx temperatures and limits	129
Details on M4i/M4x cards status LED	130
Turning on card identification LED	130
Pin assignment of the digital VHDCI connector	131
Pin Assignment for 32 channel single-ended version	131
Pin Assignment for 32 channel differential (LVDS) version	132
Continuous memory for increased data transfer rate	133
Background	133
Setup on Linux systems	134
Setup on Windows systems.....	135
Usage of the buffer	135
Abbreviations.....	137
List of Figures	138
List of Tables	139









Safety Instructions

This chapter contains information about the following topics:

- General safety information
- Requirements for users and duties for operators
- Intended use
- Markings and Labelling

Symbols and Safety Labels

Table 1: Symbols and Safety Labels

Label	Where	Description
	Cards	ESD symbol Parts can be damaged by electrostatic discharge. Follow these precautions: Avoid touching pins, leads, or circuitry. Always be properly grounded when touching a static-sensitive component or assembly.
	NETBOX chassis	GND symbol To enhance the immunity of the equipment against conducted and radiated RF disturbance, sensitive electrical circuits are connected to the chassis.
		Protective Conductor Class I This movable devices of protection class I is equipped with a cable with additional protective conductor and a protective contact plug. The device may only be connected to the protective conductor system of the fixed electrical installation, which is at ground potential.
	Products	Labelling for CE conformity Spectrum confirms with the CE marking affixed to the product or its packaging that the product complies with the product-specific applicable European directives. The CE declaration of conformity for the product is available upon request.
	Products	Labelling for WEEE The WEEE symbol on the product or its packaging indicates that the product must not be disposed of with other waste. The user is obliged to collect the old devices separately and to make them available to the WEEE take-back system for recycling.
	NETBOX chassis	Labelling for battery disposal Batteries must not be disposed of with household waste. You are legally obliged to return old batteries so that proper disposal can be guaranteed. You can dispose of used batteries at a municipal collection point or in local stores
	Manual	Important part of the manual with safety related content
	Manual	Additional information inside the manual which helps to understand a topic in more detail

General safety information

Carefully read the documentation (Installation manual and hardware manual) that belongs to the product prior to the start-up. Please observe the product safety instructions and the following safety notices to avoid health issues or damage to the device.

The manufacturer does not assume any liability for damages resulting from improper handling, unintended use or non-observance of the safety precautions.

Applicable regulations and laws governing the location and use of the product must be observed and all accident prevention and occupational safety regulations must be complied with.

Requirements for users and duties for operators

The product may be assembled, operated and maintained only if you have the necessary qualification and experience for this product. Improper use or use by a user without sufficient qualification can lead to damages or injuries to one's health or damages to property. The assembler of the system is responsible for the safety of any system incorporating the equipment.

General safety at work

The existing regulations for safety at work and accident prevention must be followed. All applicable regulations and statutes regarding operation must be strictly followed when using this product.

Bringing the product into service

The following steps need to be done when first bringing the product into service:

- Please check the content of the delivery against the above stated packing list upon first opening of the package
- Check the products before connecting them to any power source for any damages. Do not connect a damaged product to any power source
- Be sure to have the correct knowledge to install this product
- Carefully read the installation manual and take the stated precautions
- Follow the installation process step by step as described in this manual
- The product relies on proper cooling as described in this manual. Make sure to avoid to restrict the airflow to any part. Do not cover or block any cooling fans or cooling vents

Intended use

Application area of the product

The device has been developed for indoor use in controlled laboratory and industrial environments not exceeding an operating height of 2000 m and for an ambient temperature of 0°C to +40°C with non-condensing humidity up to 10% to 90%.

Requirements for the technical state of the product

The product is designed in accordance with state-of-the-art technology and recognized safety rules. The product may be operated only in a technically flawless condition and according to the intended purpose and with regard to safety and dangers as stated in the respective product documentation. If the product is not used according to its intended purpose, the protection of the product may be impaired.

Requirements for operation

Use the product only according to the specifications in the corresponding User's Guide. With any deviating operation, the product safety is no longer ensured.

The use of the product is permitted only in accordance with the specifications and information of the respective user manual. Product safety is not guaranteed in the event of deviating use. Use in wet or humid environments or in potentially explosive areas is not permitted.

The installer is responsible for the safety of the system in which the device is installed.

Electrical safety and power supply

Observe the regulations applicable at the operating location concerning electrical safety as well as the laws and regulations concerning work safety! Connect only current circuits with safety extra-low voltage in accordance with EN 61140 (degree of protection III) to the connections of the module.

Ensure that the connection and setting values are being followed (see the information in the chapter "Technical data"). Do not apply any voltages to the connections of the module that do not correspond to the specifications of the respective connection. When setting up the appliance, care must be taken to ensure that the power plug of the chassis is easily accessible and the power cable can be unplugged in the event of an emergency shut-down.

Use only approved cables at the connections of the product. Adhere to the maximum permissible cable lengths! Do not use any damaged cables! Never apply force to insert a plug into a socket. Ensure that there is no contamination in and on the connection, that the plug fits the socket, and that you correctly aligned the plugs with the connection.

There is no danger from the device in case of power supply interruption or shut down.

Requirements for the location

The housing and the connectors of the module as well as the plug connectors of the cables meet the degree of protection IP20. Position the module on a smooth, level and solid underground. The module or the module stack must always be securely fastened.

The functionality and safety of the device is only guaranteed at operation conditions of IP20 and contamination class II up to a light contamination by non-conductive materials.

Requirements on the ventilation

Keep the module away from heat sources and protect it against direct exposure to the sun. The free space above and behind the module must be selected so that sufficient air circulation is ensured. During normal operation there are no hot surfaces that pose any danger to the operator.

Maintenance

The product is maintenance-free.

Repair/Service

In the event of a necessary repair, the product must be returned to the manufacturer. Before returning any good get in contact with the support group and obtain a RMA code. The support group can be reached by email: Support@spec.de

Please ensure suitable packaging to avoid damage during transport.

World-wide service address is:

Spectrum Instrumentation GmbH
Ahrensfelder Weg 13-17
22927 Grosshansdorf
Germany

Cleaning the module housing (NETBOX devices, cables, amplifiers, systems only)

Use a dry or lightly moistened, soft cloth for cleaning the module housing. Do not use any sprays, solvents or abrasive cleaners which could damage the housing. Ensure that no moisture enters the housing. Never spray cleaning agents directly onto the module.

Opening the module (NETBOX devices, amplifiers only)

Do not open or change the module housing! Work on the module housing may only be performed by the manufacturer.

Dismounting parts of the card (instrument card only)

Do not dismount any part of the card like modules, front plates or internal cable connections.

Markings and Labelling

The product complies with the current European directives on CE marking. A CE declaration of conformity is available on request.

The product complies with the current European Directives on the Use of Certain Hazardous Substances (RoHS-3) 2015/863/EU).

According to the European directive WEEE (Waste Electrical and Electronic Equipment), the user is obliged to return the product to the system for collection, treatment and recycling of waste electronic equipment. Disposal via residual waste is not permitted.

Up-to-date information on notifiable substances according to REACH regulation (EC) No 1907 /2006 can be quoted on request.

Packing list

The following items are containing in the packing. Some of these items need to be ordered separately as an option.

PCIe (PCI Express) cards M4i.xxxx

Table 2: Packing List M4i card (PCI Express)

Item	Contained	Description
Card	Yes	Ordered card type inside ESD safety bag
Star-Hub M4i.xxxx-SH8ex, -SH81m	Optional	Star-Hub mounted on card, containing 8 sync-cables
Digital Option M4i.44xx-DigSMB	Optional (M4i.44xx only)	Digital option, mounted on card
Manual	Yes	Printed Installation Manual
USB Stick	Yes	Containing drivers, software and programming manuals
Cables	Optional	Ordered cables, each packed in own bag

PXIe (PXI Express) cards M4x.xxxx

Table 3: Packing List M4x card (PXI Express)

Item	Contained	Description
Card	Yes	Ordered card type inside ESD safety bag
Manual	Yes	Printed Installation Manual
USB Stick	Yes	Containing drivers, software and programming manuals
Cables	Optional	Ordered cables, each packed in own bag

Introduction

Preface

This manual provides detailed information on the hardware features of your Spectrum board. This information includes technical data, specifications, block diagram and a connector description.

In addition, this guide takes you through the process of installing your board and also describes the installation of the delivered driver package for each operating system.

Finally this manual provides you with the complete software information of the board and the related driver. The reader of this manual will be able to integrate the board in any PC system with one of the supported bus and operating systems.

Please note that this manual provides no description for specific driver parts such as those for IVI, LabVIEW or MATLAB. These driver manuals are available on USB-Stick or on the Spectrum website.

For any new information on the board as well as new available options or memory upgrades please contact our website www.spectrum-instrumentation.com. You will also find the current driver package with the latest bug fixes and new features on our site.

Please read this manual carefully before you install any hardware or software. Spectrum is not responsible for any hardware failures resulting from incorrect usage.



Overview

M4i cards for PCIe Express (PCIe)



The M4i generation is the fast streaming and high performance platform from Spectrum. The 3/4 length cards are available in different speed grades and resolutions with best performance.

The cards have been optimized for extremely fast data transfer and allow to read data for online analysis or offline storage with more than 3 GB/s using the PCI Express x8 Gen 2 bus interface. Mechanically the card family needs x8 or x16 lane PCI Express connectors with any PCI Express generation. Electrically the card can handle smaller number of PCI Express lanes with reduced transfer speed.



When using high sampling rates the 4 GByte standard on-board memory (2 GSamples for cards with 12/14/16 bit resolution) is sufficient to acquire up to several seconds of high-speed data. The M4i cards are carefully designed and offer an optimized clock section, a wide range of trigger possibilities, new and improved features, easy usability and programming as well as an outstanding software support.

The PCI Express bus was first introduced in 2004. In today's standard PC there are usually two to six slots available for instrumentation boards. Special industrial PCs offer up to a maximum of 16 slots. The PCI Express Gen2 standard theoretically delivers up to 8 GByte/s data transfer rate per x16 slot. The Spectrum M4i boards are available as PCI Express x8 (eight lane) Gen2, 3/4 length card.

Within this document the name M4i or M4i.xxxx is used as a synonym for the PCI Express version with the full name of M4i.xxxx-x8 to enhance readability. The exact order information can be found in the related passage in this manual.



M4x cards for PXI Express (PXIe)



The M4x platform takes the features of the M4i series of PCIe cards to an industrial bus standard. The 3U two slot cards are available in different speed grades and resolutions with best performance.

Based on Spectrums proven M4i series of PCIe products the new M4x PXIe modules deliver the same advanced features and signal quality.

It also allows the new modules to share a common software interface and offer the same FPGA based averaging and statistics options.



Compared to PCIe, PXIe systems come with superior mechanical design, better connectors and a defined air flow for cooling. This makes it the ideal platform for many industrial and mobile applications.



The M4x cards have been optimized for extremely fast data transfer and allow to read data for online analysis or offline storage with more than 1.5 GB/s using the PCI Express x4 Gen 2 bus interface.

When using high sampling rates the 4 GByte standard on-board memory (2 GSamples for cards with 12/14/16 bit resolution) is sufficient to acquire up to several seconds of high-speed data. The M4x cards are carefully designed and offer an optimized clock section, a wide range of trigger possibilities, new and improved features, easy usability and programming as well as an outstanding software support.

Within this document the name M4x or M4x.xxxx is used as a synonym for the PXI Express version with the full name of M4x.xxxx-x4 to enhance readability. The exact order information can be found in the related passage in this manual.



General Information

The M4i.77xx-x8 series digital waveform acquisition card which is available with different front-ends and different speed grades. The digital waveform acquisition card features a PCI Express x8 Gen 2 interface that offers outstanding data streaming performance. The interface and Spectrum's optimized drivers enable data transfer rates in excess of 3 GB/s so that signals can be continuously recorded at even the highest sampling speed.

While the cards have been designed using the latest technology they are still software compatible with the drivers from earlier Spectrum digitizers. So, existing customers can use the same software they developed for a 10 year old Digital I/O card and for an M4i series 720 MS/s digital waveform acquisition card

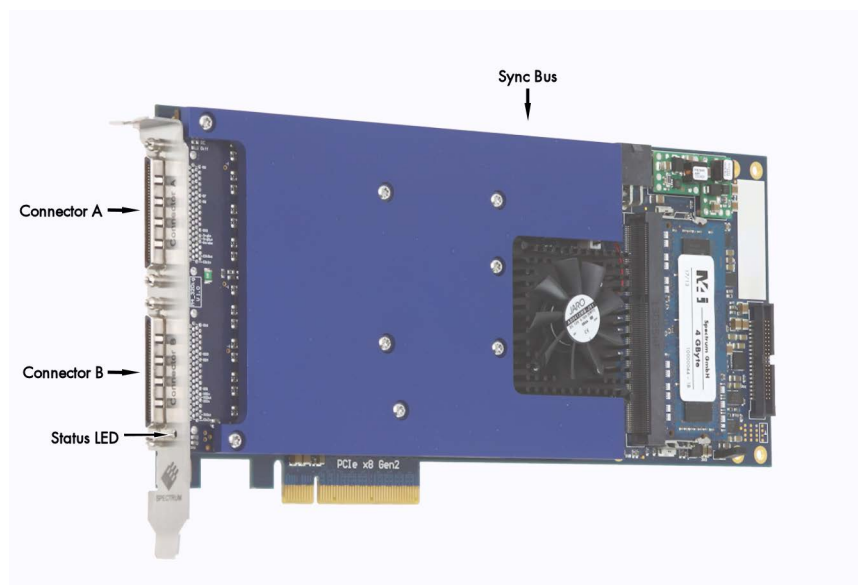
Data is written in the internal 4 GByte large memory. This memory can also be used as a FIFO buffer. In FIFO mode data will be transferred online into the PC RAM or to hard disk.

Application examples: Recording of digital data, chip test, system test, pattern recognition, laboratory, development.

Connector location at the M4i.77xx series

The following overview shows the location of the two connectors of the M4i.77xx series. The different models only differ in speed and interface setup and are all using the same connectors.

- **M4i.7710-x8**
- **M4i.7720-x8**
- **M4i.7730-x8**
- **M4i.7725-x8**
- **M4i.7735-x8**



Additional options

Star-Hub (M4i only)

The star hub module allows the synchronization of up to 8 M4i cards. It is possible to synchronize only cards of the same family with each other.

Two different versions of the star-hub module allowing the synchronization of up to 8 cards are available. A version that is mounted on top of the carrier card as a piggy-back module (option SH8tm) extending the width of the card to two slots.

The second version (option SH8ex) is mounted behind the card and extends the M4i base card to a full-length PCI Express card. Therefore it requires the availability of a full-length slot in the system but does not need the space of an additional slot.

The module acts as a star hub for clock and trigger signals. Each board is connected with a small cable of the same length, even the master board. That minimizes the clock skew between the different cards. The picture shows the piggy-back module mounted on the base board schematically without any cables to achieve a better visibility.

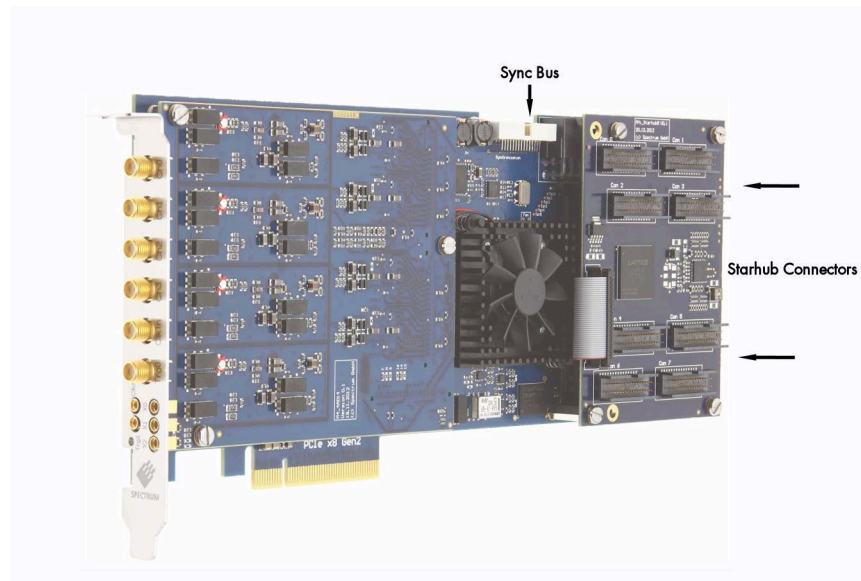


Image 1: M4i card showing mounted star-hub and sync bus connector

The carrier card acts as the clock master and the same or any other card can be the trigger master. All trigger modes that are available on the master card are also available if the synchronization star-hub is used.

The cable connection of the boards is automatically recognized and checked by the driver when initializing the star-hub module. So no care must be taken on how to cable the cards. The star-hub module itself is handled as an additional device just like any other card and the programming consists of only a few additional commands.

Digital I/O with Dig-SMA (M4i.44xx only)

The Digital I/O options „Dig-SMA“ adds eight additional Multi-Purpose I/O lines to the card.

All eight lines are provided via SMA miniature coaxial connectors, just like the analog channels or clock and trigger input.

All of these lines are mounted on the PCI bracket and are hence accessible from the outside of the PC.

These lines extend the already existing Multi-Purpose I/O lines that come standard with the main card (X0 .. X2) and are intended to expand the number of available digital lines, that can be recorded synchronously alongside the analog channels.

Although the capabilities of these additional lines are not identical to those on the main card, they are consistently named (X3 .. X10).

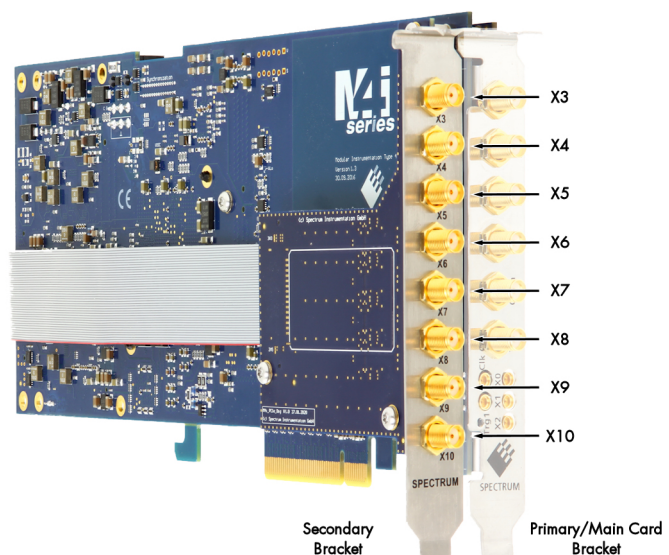


Image 2: M4i card showing mounted digital option

Either one of the two Star-Hub or the DigSMA option can be mounted on any board physically at any given time. It is not possible to mount a Star-Hub and a digital option DigSMA onto the same board.



The Spectrum type plate

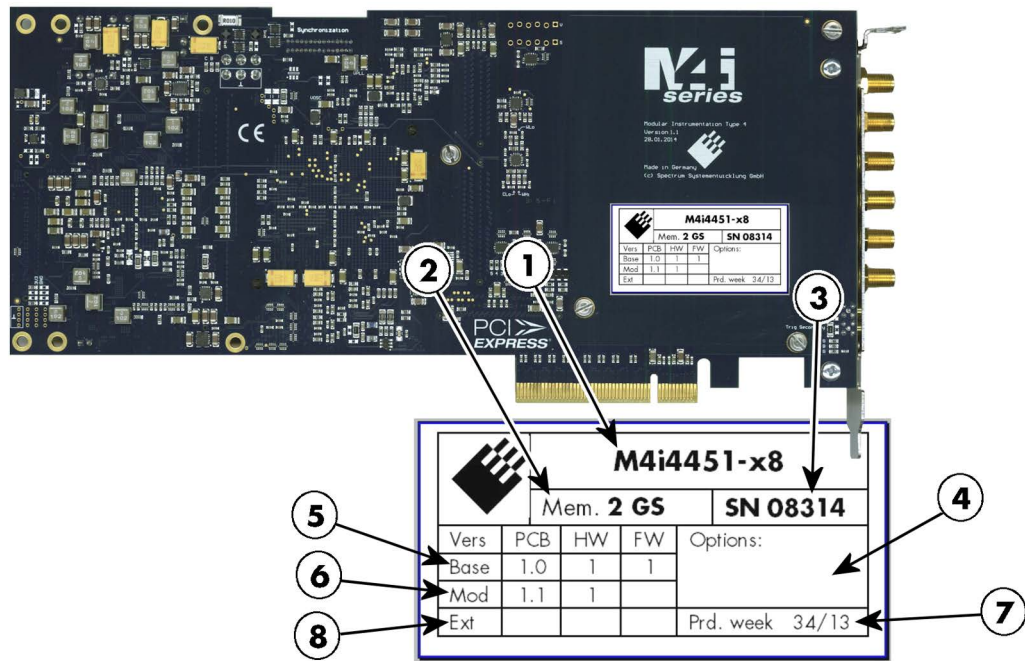


Image 3: M4i card backside showing type plate location and content

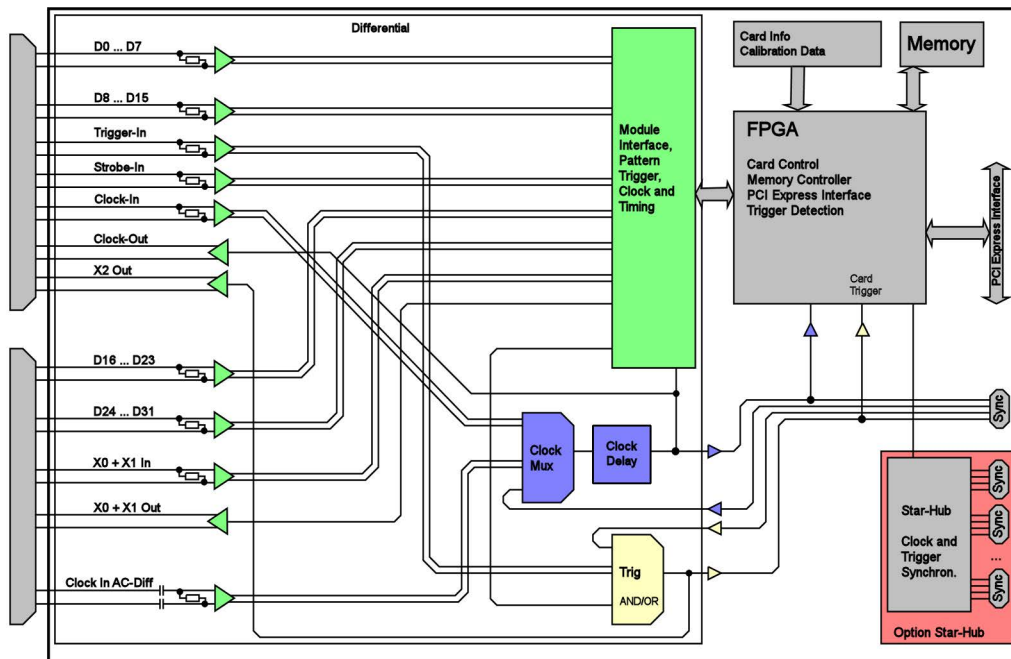
The Spectrum type plate, which consists of the following components, can be found on all of our boards. Please check whether the printed information is the same as the information on your delivery note. All this information can also be read out by software:

- ① The board type, consisting of the two letters describing the bus (in this case M4i for the PCI Express x8 bus) and the model number.
- ② The size of the on-board installed memory in MSample or GSample. In this example there are 2 GS = 2048 MSample (4 GByte = 4096 MByte) installed.
- ③ The serial number of your Spectrum board. Every board has a unique serial number.
- ④ A list of the installed options. A complete list of all available options is shown in the order information. In this example no additional options are installed.
- ⑤ The base card version, consisting of the printed circuit board (PCB) version, the hardware version and the firmware version.
- ⑥ The version of the analog/digital front-end module, consisting of the printed circuit board (PCB) version, the hardware version and the firmware version (if available). If no programmable device is located on the module, the firmware field is left empty.
- ⑦ The date of production, consisting of the calendar week and the year.
- ⑧ The version of the extension module (such as a Starhub) if one is installed, consisting of the printed circuit board (PCB) version, the hardware version and the firmware version. If no extension module is installed this part is left empty.

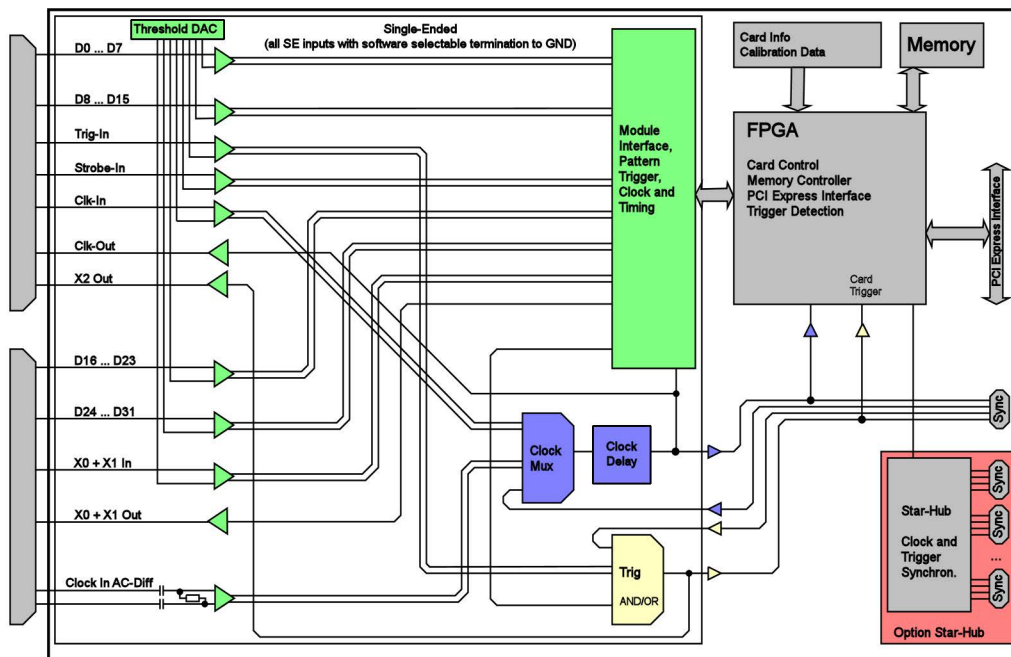
Please always supply us with the above information, especially the serial number in case of support request. That allows us to answer your questions as soon as possible. Thank you.

Hardware information

Block diagram differential input version



Block diagram single-ended input version



Technical Data



Only figures that are given with a maximum reading or with a tolerance reading are guaranteed specifications. All other figures are typical characteristics that are given for information purposes only. Figures are valid for products stored for at least 2 hours inside the specified operating temperature range, after a 30 minute warm-up, after running an on-board calibration and with proper cooled products. All figures have been measured in lab environment with an environmental temperature between 20°C and 25°C and an altitude of less than 100 m.

Differential Interface

Available inputs		Data D0 to D31, Trigger (TrigIn), Strobe, Clock (ClkIn), X0, X1
Data Channel Selection	software programmable	32 channels, 16 channels, 8 channels
Data/Control Input Compatibility		LVDS, LVPECL, PECL, (N)ECL, universal differential inputs
Input Coupling		DC
Input Type		high-speed comparator
Input maximum voltage levels		-3.0 V to +5.0 V, max difference between inputs ± 8 V
Input voltage hysteresis		25 mV
Input termination		differential termination with 125 Ω
Open inputs		fail save -> defined and fixed input level with open inputs, no external termination necessary
Available outputs		Clock (ClkOut), Trigger (TrigOut), X0, X1
Output signal type		LVDS

Single-Ended Interface

Available inputs		Data D0..D31, Trigger (TrigIn), Strobe, Clock (ClkIn), Multi-Purpose In (X0,X1) + Out (X0,X1,X2)
Data Channel Selection	software programmable	32 channels, 16 channels, 8 channels
Data/Control Input Compatibility		compatible to 1.2V, 1.5V, 1.8V, 2.5V, 3.3V, 5.0V (LV)TTL and (LV)CMOS logic levels
Input Coupling		DC
Input Type		high-speed comparator
Input threshold level	software programmable	0.0 V up to 4.0 V in steps of 10 mV, separately programmable for (D0..D7), (D8..D15), (D16..D23), (D24..D31), TrigIn, ClkIn, Strobeln, (X0..X1)
Input maximum voltage levels		-3.0 V to +5.0 V
Input voltage hysteresis		25 mV
Input termination	software programmable	75 Ω (to GND) / 4.7 k Ω (to GND) separately programmable for (D0..D7), (D8..D15), (D16..D23), (D24..D31), TrigIn, ClkIn, Strobeln, X0, X1
Open inputs		fail save -> defined and fixed input level with open inputs, no external termination necessary
Available outputs		Clock (ClkOut), Multi-Purpose Out (X0, X1, X2)
Output signal type		3.3V LVTTTL compatible

Trigger

Available trigger sources	software programmable	External trigger, pattern trigger, software
Trigger edge	software programmable	Rising edge, falling edge or both edges
Trigger delay	software programmable	0 to (8GSamples - 32) = 8589934576 Samples in steps of 32 samples
Multi, Gate: re-arming time		40 samples (+ programmed pretrigger)
Pretrigger at Multi, Gate, FIFO	software programmable	32 up to 4096 samples in steps of 32
Posttrigger	software programmable	32 up to 8G samples in steps of 32 (defining pretrigger in standard scope mode)
Memory depth	software programmable	32 up to [installed memory / number of active channels] samples in steps of 32
Multiple Recording segment size	software programmable	32 up to [installed memory / 2 / active channels] samples in steps of 32
Internal/External trigger accuracy		1 sample
Timestamp modes	software programmable	Standard, Startreset, external reference clock on X1 (e.g. PPS from GPS, IRIG-B)
Data format		Std., Startreset: 64 bit counter, increments with sample clock (reset manually or on start) RefClock: 24 bit upper counter (increment with RefClock) 40 bit lower counter (increments with sample clock, reset with RefClock)
Extra data	software programmable	none, acquisition of X0/X1 inputs at trigger time
Size per stamp		128 bit = 16 bytes

Multi Purpose I/O lines (on VHDCI connector)

Number of multi purpose lines		three named X0 and X1, separate lines for input and output, X2 (output only)
Input: available signal types	software programmable	Asynchronous Digital-In, Timestamp Reference Clock
Output: available signal types	software programmable	Asynchronous Digital-Out, Run, Arm, Trigger, PLL RefClk
Multi Purpose input impedance (Diff.)		differential termination with 125 Ω
Multi Purpose input impedance (SE)	software programmable	75 Ω (to GND) / 4.7 k Ω (to GND), separately programmable for X0 and X1
Multi Purpose input type (Diff.)		LVDS, LVPECL, PECL, (N)ECL, universal differential inputs
Multi Purpose input type (SE)		compatible to 1.2V, 1.5V, 1.8V, 2.5V, 3.3V, 5.0V (LV)TTL and (LV)CMOS logic levels
Multi Purpose input threshold level (SE)	software programmable	0.0 V up to 4.0 V in steps of 10 mV (common programmable level for X0 and X1)
Multi Purpose output type (Diff.)		Differential LVDS
Multi Purpose output type (SE)		3.3V LVTTTL compatible

Power Source (on VHDCI connector)

Number of power pins	6
Voltage	3.3 V
Maximum current	500 mA combined on all pins
Fuse	Self resetting fuse (PTC)

Clock

Clock Modes	software programmable	internal PLL, external reference clock, state clock, sync
Internal clock accuracy		$\leq \pm 20$ ppm
Internal clock setup granularity		1 Hz
Clock setup range gaps		562 MHz to 574 MHz (no clock setup possible in that range)

Primary Clk-In (Ext0) as reference clock

External reference clock range	software programmable	≥ 10 MHz and ≤ 1 GHz
External reference clock input impedance (Diff.)		differential termination with $125\ \Omega$
External reference clock input type (Diff.)		LVDS, LVPECL, PECL, (N)ECL, universal differential inputs
External reference clock input type (SE)		compatible to 1.2V, 1.5V, 1.8V, 2.5V, 3.3V, 5.0V (LV)TTL and (LV)CMOS logic levels
External reference clock input impedance (SE)	software programmable	$75\ \Omega$ (to GND) / $4.7\ k\Omega$ (to GND)
External reference clock input threshold level (SE)	software programmable	0.0 V up to 4.0 V in steps of 10 mV
External reference clock input edge		Rising edge
External reference clock input duty cycle requirement		45% to 55%
External reference clock input requirements		no frequency changes, no gaps

Primary Clk-In (Ext0) as state clock

External state clock input coupling		DC
External state clock input impedance (Diff.)		differential termination with $125\ \Omega$
External state clock input impedance (SE)	software programmable	$75\ \Omega$ (to GND) / $4.7\ k\Omega$ (to GND)
External state clock input type (Diff.)		LVDS, LVPECL, PECL, (N)ECL, universal differential inputs
External state clock input type (SE)		compatible to 1.2V, 1.5V, 1.8V, 2.5V, 3.3V, 5.0V (LV)TTL and (LV)CMOS logic levels
External state clock input threshold level (SE)	software programmable	0.0 V up to 4.0 V in steps of 10 mV
External state clock input edge	software programmable	Rising edge or falling edge (SDR) or both edges (DDR)
External state clock input requirements		Any frequency within specification, changes allowed, gaps allowed, DC allowed
External state clock delay	software programmable	0 ps to 2000 ps with a step size of 40 ps

Secondary Clk-In-AC (Ext1) as state clock

External secondary clock input coupling		AC
External secondary clock input impedance		differential termination with $100\ \Omega$
External secondary clock input type		LVPECL
External secondary clock input voltage swing		(ClkIn-AC+ to ClkIn-AC-): ± 100 mV up to ± 1.7 V
Sampling clock output type (Diff.)		Differential LVDS
Sampling clock output type (SE)		3.3V LVTTTL compatible
Sampling clock output frequency	Internal or External reference	$\frac{1}{2}$ of internal sampling clock (disabled for frequencies above 125 MHz on SE models)
Sampling clock output frequency	External state clock	Copy of fed in state clock (not available for M4i.7730)
Star-Hub synchronization clock modes	software selectable	Internal clock, External reference clock (state clock is not available with synchronization)

Clock Limits

	M4i.7710-x8	M4i.7720-x8	M4i.7730-x8	M4i.7725-x8	M4i.7735-x8
Interface	Single-Ended	Single-Ended	Single-Ended	Differential	Differential
minimum internal clock	610 S/s	610 S/s	610 S/s	610 S/s	610 S/s
maximum internal clock	125 MS/s	250 MS/s	720 MS/s	250 MS/s	720 MS/s
minimum state clock (Ext 0)	DC	DC	DC	DC	DC
minimum state clock (Ext 1)	30 kHz	30 kHz	30 kHz	30 kHz	30 kHz
maximum state clock (single data rate)	125 MHz	250 MHz	350 MHz	250 MHz	350 MHz
maximum state clock (double data rate)	62.5 MHz	125 MHz	350 MHz	125 MHz	350 MHz
maximum state clock data rate (SDR)	125 MBit/s	250 MBit/s	350 MBit/s	250 MBit/s	350 MBit/s
maximum state clock data rate (DDR)	125 MBit/s	250 MBit/s	700 MBit/s	250 MBit/s	700 MBit/s

Timings

		Signal type	External reference clock Input Ext 0	State Clock Input Ext 0		State Clock Input Ext 1	
			No Clk Delay	No Clk Delay	Max. Clk Delay	No Clk Delay	Max. Clk Delay
t_{setup}	Setup time before clock edge	input signals	TBD	720 ps	-1280 ps	TBD	TBD
t_{hold}	Hold time after clock edge	input signals	TBD	200 ps	2200 ps	TBD	TBD
t_{delay}	Delay from clock input to clock output	input signals	TBD	TBD	TBD	TBD	TBD

Connectors

Number of connectors	2
Connector type	68 pin standard VHDCI
Connector impedance	125 Ω (differential), 75 Ω (single-ended),
Cable recommendations	compatible to SCSI ultra-320, double shielded, twisted pair, max length 1 m, cable twisting: Pin1/Pin35, Pin2/Pin36 ... Pin 34/68

Connection Cycles

All connectors have an expected lifetime as specified below. Please avoid to exceed the specified connection cycles or use connector savers.

VHDCI connector	2000 connection cycles
PCIe connector	50 connection cycles
PCIe power connector	30 connection cycles

Environmental and Physical Details

Dimension (Single Card)	L x H x W: 241 mm (¾ PCIe length) x 107 mm x 20 mm (single slot width)
Dimension (Card with option SH8tm installed)	241 mm (¾ PCIe length) x 107 mm x 40 mm (double slot width, extends W by 1 slot right of the main card's bracket, on „component side“ of the PCIe card.)
Dimension (Card with option SH8ex installed)	Extends L to 312 mm (full PCIe length) x 107 mm x 20 mm (single slot width)
Dimension (Card with option M4i.44xx-DigSMA installed)	241 mm (¾ PCIe length) x 107 mm x 40 mm (double slot width, extends W by 1 slot left of the main card's bracket, on „solder side“ of the PCIe card.)
Weight (M4i.44xx series)	maximum 290 g
Weight (M4i.22xx, M4i.23xx, M4i.66xx, M4i.77xx series)	maximum 420 g
Weight (Option star-hub -sh8ex, -sh8tm)	including 8 sync cables 130 g
Weight (Option M4i.44xx-DigSMA)	320 g
Warm up time	10 minutes
Operating temperature	0°C to 50°C
Storage temperature	-10°C to 70°C
Humidity	10% to 90%
Dimension of packing	1 or 2 cards 470 mm x 250 mm x 130 cm
Volume weight of packing	1 or 2 cards 4 kg

PCI Express specific details

PCIe slot type	x8 Generation 2 (Gen2)
PCIe slot compatibility (physical)	x8/x16
PCIe slot compatibility (electrical)	x1, x2, x4, x8, x16 with PCIe Gen1, Gen2, Gen3, Gen4 or Gen5
Sustained streaming mode (Card-to-System): M4i.22xx, M4i.23xx, M4i.44xx, M4i.77xx	> 3.4 GB/s (measured with a chipset supporting a TLP size of 256 bytes, using PCIe x8 Gen2)
Sustained streaming mode (System-to-Card): M4i.66xx	> 2.8 GB/s (measured with a chipset supporting a TLP size of 256 bytes, using PCIe x8 Gen2)

Certification, Compliance, Warranty

Conformity Declaration	EN 17050-1:2010	General Requirements
EU Directives	2014/30/EU 2014/35/EU 2011/65/EU 2006/1907/EC 2012/19/EU	EMC - Electromagnetic Compatibility LVD - Electrical equipment designed for use within certain voltage limits RoHS - Restriction of the use of certain hazardous substances in electrical and electronic equipment REACH - Registration, Evaluation, Authorisation and Restriction of Chemicals WEEE - Waste from Electrical and Electronic Equipment
Compliance Standards	EN 61010-1: 2010 EN 61187:1994 EN 61326-1:2021 EN 61326-2-1:2021 EN IEC 63000:2018	Safety regulations for electrical measuring, control, regulating and laboratory devices - Part 1: General requirement Electrical and electronic measuring equipment - Documentation Electrical equipment for measurement, control and laboratory use EMC requirements - Part 1: General requirements EMC requirements - Part 2-1: Particular requirements - Test configurations, operational conditions and performance criteria for sensitive test and measurement equipment for EMC unprotected applications Technical documentation for the assessment of electrical and electronic products with respect to the restriction of hazardous substances
Product warranty	5 years starting with the day of delivery	
Software and firmware updates	Life-time, free of charge	

Power Consumption

		PCI EXPRESS		
		3.3V	12 V	Total
M4i.771x-x8, M4i.772x-x8	Power output 0mA	0.2 A	2.9 A	36 W
M4i.773x-x8	Power output 0mA	0.2 A	3.1 A	38 W

MTBF

MTBF	100000 hours
------	--------------

Order Information

The card is delivered with 4 GByte on-board memory and supports standard acquisition (Scope), FIFO acquisition (streaming), Multiple Recording, Gated Sampling and Timestamps. Operating system drivers for Windows/Linux 32 bit and 64 bit, examples for C/C++, LabVIEW (Windows), MATLAB (Windows and Linux), .NET, Delphi, Java, Python, Julia and a Base license of the oscilloscope/logic-analyzer software SBench 6 are included.

Adapter cables are not included. Please order separately!

PCI Express x8

Order no.	Channels	Interface	Standard mem	Sampling Clock	State Clock
M4i.7710-x8	32	Single-Ended	4 GByte	125 MBit/s	125 MBit/s
M4i.7720-x8	32	Single-Ended	4 GByte	250 MBit/s	250 MBit/s
M4i.7730-x8	32	Single-Ended	4 GByte	720 MBit/s	700 MBit/s
M4i.7725-x8	32	Differential	4 GByte	250 MBit/s	250 MBit/s
M4i.7735-x8	32	Differential	4 GByte	720 MBit/s	700 MBit/s

Options

Order no.	Option
M4i.xxxx-SH8ex ⁽¹⁾	Synchronization Star-Hub for up to 8 cards (extension), only one slot width, extension of the card to full PCI Express length (312 mm). 8 synchronization cables included.
M4i.xxxx-SH8tm ⁽¹⁾	Synchronization Star-Hub for up to 8 cards (top mount), two slots width, top mounted on card. 8 synchronization cables included.
M4i-upgrade	Upgrade for M4i.xxxx: Later installation of option Star-Hub

VHDCI

	Option
Cab-v68-v68-100	Ω differentiell, 90 Ω single-ended
Cab-v68-v68-300	Ω differentiell, 90 Ω single-ended

Software SBench6

Order no.	
SBench6	Base version included in delivery. Supports standard mode for one card.
SBench6-Pro	Professional version for one card: FIFO mode, export/import, calculation functions
SBench6-Multi	Option multiple cards: Needs SBench6-Pro. Handles multiple synchronized cards in one system.
Volume Licenses	Please ask Spectrum for details.

Software Options

Order no.	
SPc-RServer	Remote Server Software Package - LAN remote access for M2i/M3i/M4i/M4x/M2p/M5i cards
SPc-SCAPP	Spectrum's CUDA Access for Parallel Processing - SDK for direct data transfer between Spectrum card and CUDA GPU. Includes RDMA activation and examples.

⁽¹⁾ : Just one of the options can be installed on a card at a time.

⁽²⁾ : Third party product with warranty differing from our export conditions. No volume rebate possible.

Hardware Installation

ESD Precautions

All Spectrum boards contain electronic components that can be damaged by electrostatic discharge (ESD).

Before installing the board in your system or protective conductive packaging, discharge yourself by touching a grounded bare metal surface or approved anti-static mat before picking up this ESD sensitive product.



Sources of noise

Noise sensitive analog devices, such as analog acquisition and generator boards should be placed physically as far away from any noise producing source (like e.g. the power supply) as possible. It should especially be avoided to place the board in the slot directly adjacent to another fast board like e.g. a graphics controller.

Cooling Precautions

The boards of the M4i.xxxx-x8 and M4x.xxxx-x4 series operate with components having very high power consumption at high speeds. For this reason it is absolutely required to cool the boards sufficiently.

For all M4i cards it is absolutely mandatory to have installed cooling fans specifically providing a stream of air across the board's surface.



- Make absolutely sure, that the on-board fan on the M4i card is not blocked by PC internal cabling or any other means.
- Ensure that there is plenty of space around the PC chassis fan's intake and exhaust vents, both inside and outside the chassis.
- If your chassis includes fan filters, make sure that these are regularly cleaned.
- Set the rotation speed for all chassis fans and especially those providing air for the PCIe/PXle cards to highest setting in the BIOS/UEFI.
- Whenever possible leave the slot adjacent to the M4i/M4x card empty. This allows for best possible air flow over the card's surface.
- If you do need to use any adjacent slots, preferably install cards, that grant the most clearance between the devices, such as low-profile adapters.
- If available install filler panels with ventilation holes for all unused PCI or PCI Express slots to allow for additional air flow for the M4i cards and serve as an additional outtake.

Connector Handling Precautions

The connectors used on this product are designed for high signal quality and good shielding. Due to the limited space on the front-panel they have to be as small as possible to fit the needed signal connections on the front panel. Therefore these connectors are vulnerable to mechanical damages when used not properly. Especially SMB and MMCX connectors may be broken when not operated correctly.

Always dismount the connections by operating the connector itself and not the cable. Always move the cable connector in a straight line from the board connector. Do not cant the connector when opening the connection. A broken connector can only be replaced in factory and is not covered by warranty.



M4i PCIe Cards

System Requirements

All Spectrum M4i.xxxx-x8 instrumentation cards are compliant to the PCI Express 2.0 standard and require in general one free 3/4 length PCI Express slot. This can mechanically either be a x8 or x16 slot, electrically all lane widths are supported, be it x1, x4, x8 or x16. Some x16 PCIe slots are for the use of graphic cards only and can not be used for other cards. Please consult your mainboard manual for details. Depending on the installed options additional free slots can be necessary.

Installing the M4i board in the system

Please be sure that the system is powered-down and all power cables are disconnected from the system before starting with the installation process.

Installing a single board without any options

Before installing the board you first need to unscrew and remove the dedicated blind-bracket usually mounted to cover unused slots of your PC. Please keep the screw in reach to fasten your Spectrum card afterwards. All Spectrum M4i cards mechanically require one PCI Express x8 or x16 slot (electrically either x1, x4, x8 or x16). Now insert the board slowly into your computer. This is done best with one hand each at both fronts of the board.

Please take especial care to not bend the card in any direction while inserting it into the system. Bending of the card may damage the PCB totally and is not covered by the standard warranty.



Please be very careful when inserting the board in the slot, as most of the mainboards are mounted with spacers and therefore might be damaged if they are exposed to high pressure.



After the insertion of the board fasten the screw of the bracket carefully, without overdoing.

Installing the M4i.xxxx-x8 PCI Express card in a PCIe x8 or x16 slot:

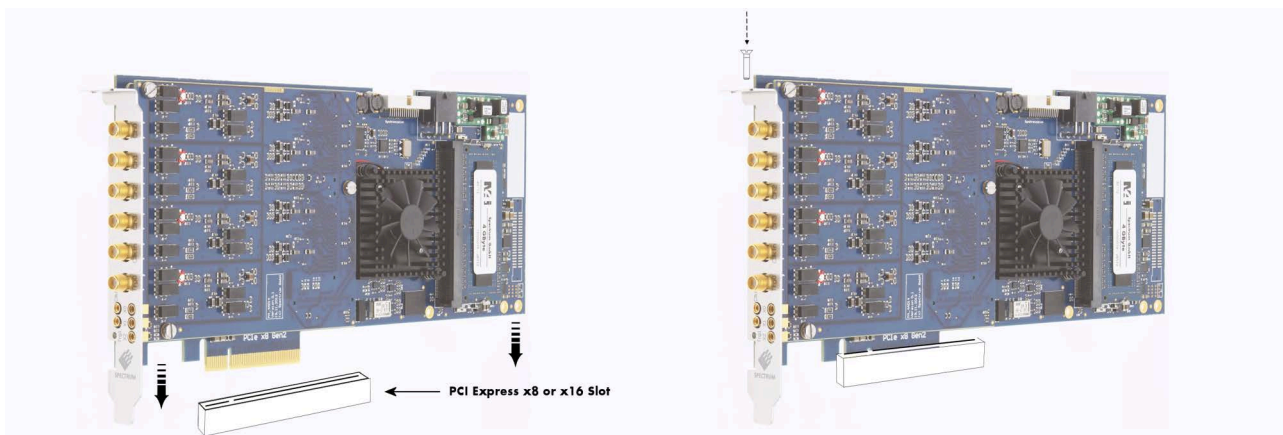


Image 4: Mounting M4i PCIe card into connector

Additional notes on PCIe x16 slot retention

M4i-xxx-x8 cards starting with hardware version V7 (which includes the new PCB revision V1.2) do have an additional PCIe retention hook (hockey stick) added to the PCB.

That allows the card to be additionally locked when being installed into a PCIe x16 slot.

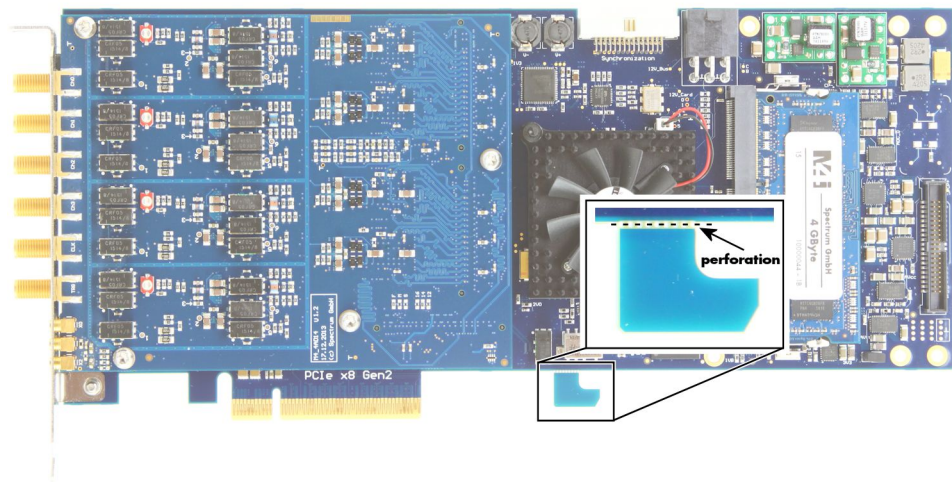


Image 5: M4i card slot retention with perforation

When installing the card in a x16 slot, make sure that the locking mechanism of the slots properly lock in place with the retention hook.



In the case that there are any components on the mainboard in the way of the retention hook when installing the card in an x8 slot, you can remove the hook by carefully breaking it off at its perforation line.



Providing additional power to M4i.xxxx-x8 cards

All PCI Express cards, with the exception of graphic adapters, are per specification only allowed to consume a maximum power of 25W per card. While some of the M4i PCIe cards are specified with a power consumption to meet these power limits, many do consume more than 25W of total power.

This is why all M4i cards can be optionally supplied with the required voltages via a dedicated PCIe 6-pin power connector directly from the system power supply.

As part of its power-on routine, the card will automatically detect, whether a cable is plugged or not and will give preference to the cable-supplied voltages.

Although it would be considered good practice to always provide the power via cable in case the card's rated power consumption is above the 25W limit, in typical system setups with one or at maximum two cards installed, not doing so and using just the slot power usually works out perfectly fine. Having more M4i cards in a system will definitely require a separate power cable per card.

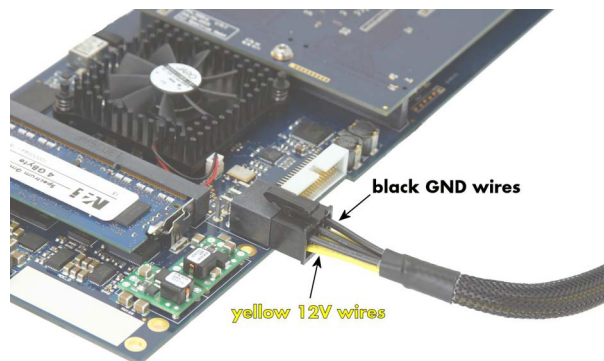


Image 6: M4i card additional power connection usage

Please only connect 6-pin PCIe power cables to the M4i cards power connector and make absolutely sure, that its three lower row wires are marked yellow (hence providing 12V) and the three upper row wires (the side of the connectors retention hook) are marked black providing a connection to system ground (GND), as shown on the picture.



Installing a M4i.44xx board with digital inputs/outputs mounted on an extra bracket

Before installing the board you first need to unscrew and remove the dedicated blind-bracket usually mounted to cover unused slots of your PC. Please keep the screw in reach to fasten your Spectrum card afterwards. All Spectrum M4i cards mechanically require one PCI Express x8 or x16 slot (electrically either x1, x4, x8 or x16). Now insert the board with it's attached extra bracket slowly into your computer. This is done best with one hand each at both fronts of the board.

Please take special care to not bend the card in any direction while inserting it into the system. A bending of the card may damage the PCB totally and is not covered by the standard warranty.



Please be very carefully when inserting the board in the PCI slot, as most of the mainboards are mounted with spacers and therefore might be damaged they are exposed to high pressure.



After the board's insertion fasten the screws of both brackets carefully, without overdoing. The figure shows a board with the option installed.



Image 7: M4i card with digital option mechanical installation and position of screws

Installing multiple boards synchronized by star-hub option

Precautions M4i cards with Star-Hub option (SH8ex)

Due to the length of the SMA connectors on the card's bracket in combination with the full-length of a card having the option SH8ex installed, it may be necessary with some computer case designs, to remove the black plastic retainer bracket from the end of the M4i card's main PCB to properly plug the card into the PCIe slot.

In case the retainer must be removed, an alternative to steadily holding the back of the PCB should then be implemented (if not already present in the case design).

This is especially critical, when the probability exists that the computer may be subject to sudden movement, shock or during shipment!

When fitting the card, please take care not to damage the motherboard with the lower edge of the metal front connector bracket.



Hooking up the boards

Before mounting several synchronized boards for a multi channel system into the PC you can hook up the cards with their synchronization cables first. If there is enough space in your computer's case (e.g. a big tower case) you can also mount the boards first and hook them up afterwards. Spectrum ships the card carrying the star-hub option together with the needed amount of synchronization cables. All of them are matched to the same length, to achieve a zero clock delay between the cards.

Only use the included flat ribbon cables.

All of the cards, **including the one that carries the star-hub piggy-back module**, must be wired to the star-hub.

It does not matter which of the available connectors on the star-hub module you use for which board. The software driver will detect the types and order of the synchronized boards automatically.

All of the synchronization cables are secured against wrong plugging, but nonetheless you should take care to have the pin 1 markers on the connector and on the cable on the same side, as the figure on the right is showing.

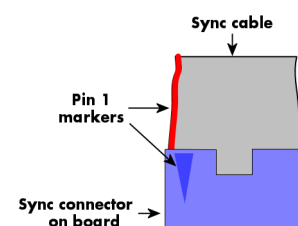
Mounting the wired boards

Before installing the cards you first need to unscrew and remove the dedicated blind-brackets usually mounted to cover unused slots of your PC. Please keep the screws in reach to fasten your Spectrum cards afterwards.

Spectrum M4i cards with the option „M4i.xxxx-SH8tm“ installed require two slots with $\frac{3}{4}$ PCIe length, whilst M4i cards with the option „M4i.xxxx-SH8ex“ installed require one single full length PCIe slot with a track at the backside to guide the card by its retainer.



Image 8: M4i cards with star-hub ex installed and connecting cables



Now insert the cards slowly into your computer. This is done best with one hand each at both fronts of the board.

While inserting the board take care not to tilt the retainer in the track. Please take especial care to not bend the card in any direction while inserting it in the system. A bending of the card may damage the PCB totally and is not covered by the standard warranty.



Please be very careful when inserting the cards in the slots, as most of the mainboards are mounted with spacers and therefore might be damaged if they are exposed to high pressure.



Shipment of systems with Spectrum cards installed

When shipping complete systems with Spectrum cards installed make sure that the cards are properly secured and cannot bend while being transported. When using freight forwarders, the transport and handling processes can be quite rough potentially subjecting the shipped PC system to quite large shocks. If the installed spectrum cards are not well mounted, secured correctly at the front and - if applicable for your model - back of the card, it's possible that they can bend when subjected to strong forces, such when a shipment container is dropped.

If damage occurs during a transport, we do not consider this to be covered by the warranty.

To avoid this we strongly recommend that when shipping these systems, customers either:

- Install the cards securely - with separate protection - so they cannot bend, or
- Remove the cards and ship them separately in their original shipping boxes (or similar packaging)

Please note that a sole fixing of the card at the front panel may not be sufficient to avoid damages in case of a mechanical shock!

M4x PXle Cards

System Requirements

The Spectrum M4x PXle 3U cards run in dedicated 3U PXle slots as well as 3U PXI/PXle hybrid slots. The M4x series of cards occupies two slots width, so up to eight cards can be installed in a large chassis providing 16 PXle slots for peripheral cards.

The M4x cards cannot be installed in either the CPU system slot nor in the dedicated system timing slot. Only a peripheral slot marked with the „circle“ symbol is suited for the cards.



Installing the M4x board in the system

Installing a single board without any options

The locks on the bottom side of PXle boards need to be unlocked and opened before installing the board into a free slot of the system. Therefore you need to press the little button on the inside of the fastener and move it outwards (see figure). Now slowly insert the card into the host system using the key ways until the lock snaps in with a „click“.

While inserting the board take care not to tilt it.



After the board's insertion fasten the four screws carefully, without overdoing.

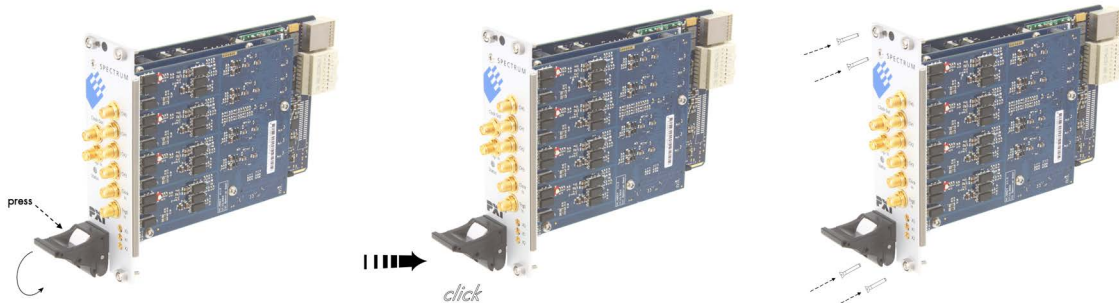


Image 9: Installation of PXle cards with connector handling and mounting screws

Software Driver Installation and Driver Update

Before using the board, a driver must be installed that matches the operating system. Later on the same principles for the initial installation also apply, when updating an existing driver on the system to a newer version.

Since driver V3.33 (released on install-disk V3.48 in August 2017) the installation is done via an installer executable rather than manually via the Windows Device Manager. The steps for manually installing a card has since been moved to a separate application note „AN008 - Legacy Windows Driver Installation“.



This new installer is common on all currently supported Windows platforms (Windows 7, Windows 8, Windows 10 and Windows 11) both 32bit and 64bit. The driver from the USB-Stick supports all cards of the M2i/M3i, M4i/M4x, M2p and M5i series, meaning that you can use the same driver for all cards of these families. This driver installer is also available from the Spectrum homepage under <https://spectrum-instrumentation.com/support/downloads.php>

Windows

Before initial installation

When you install a card for the very first time, Windows will discover the new hardware and might try to search the Microsoft Website for available matching driver modules (where no matching driver will be found).

Prior to running the Spectrum installer, the card will hence appear in the Windows device manager as a generalized card, shown here is the device manager of a Windows 10 as an example.

- M2i and M3i cards will be shown as „DPIO module“
- M5i, M4i, M4x and M2p cards will be shown as „PCI Data Acquisition and Signal Processing Controller“

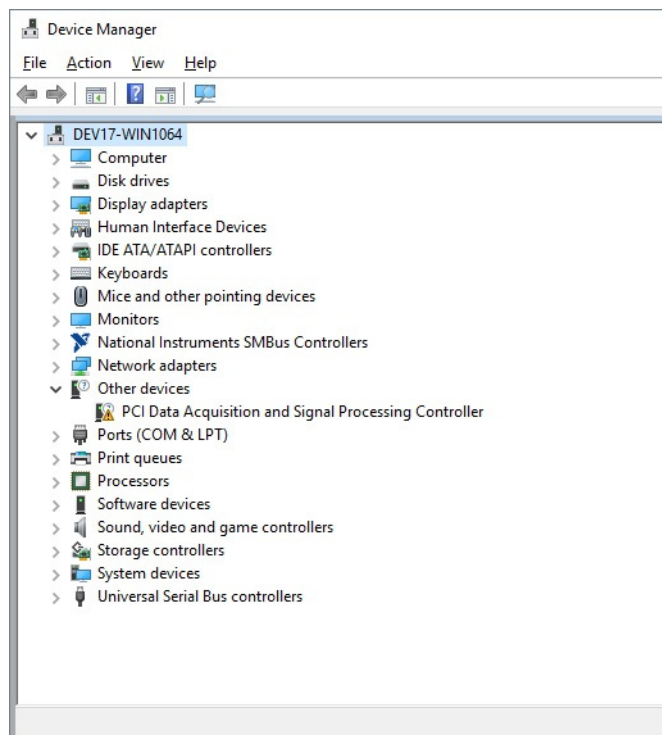


Image 10: Windows Device Manager showing a new Spectrum card

Running the driver Installer/Update

Simply run the installer supplied either on the USB-Stick „\Driver\windows“ folder or download it from our homepage and run it.

The installer can be run on a fresh system for the first install or also later on, when updating an already existing driver on the system.

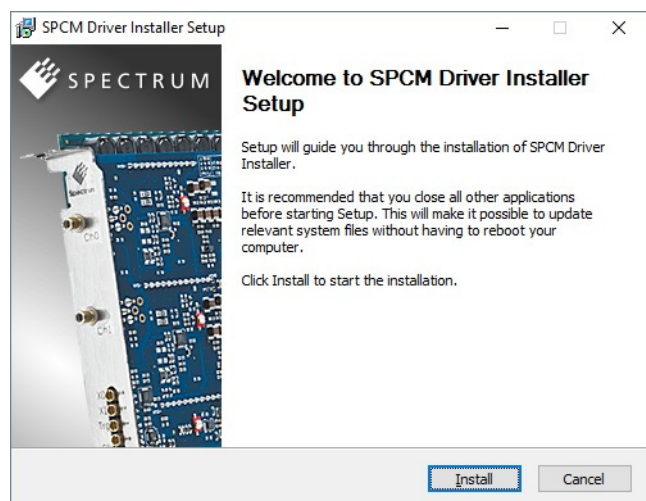


Image 11: Spectrum Driver Installer Welcome Screen

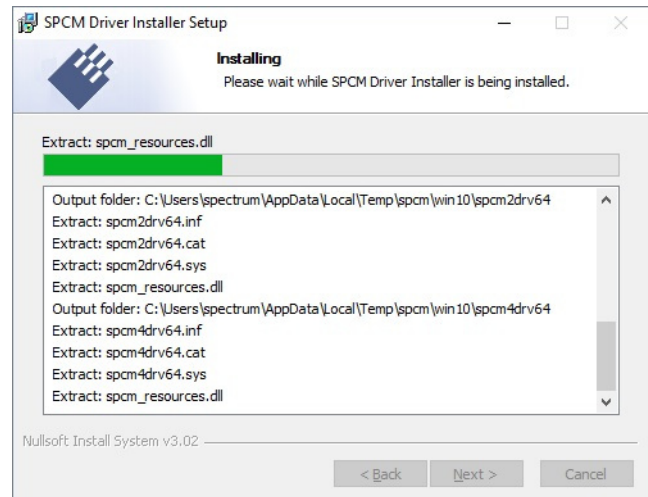


Image 12: Spectrum Driver Installer - Progress

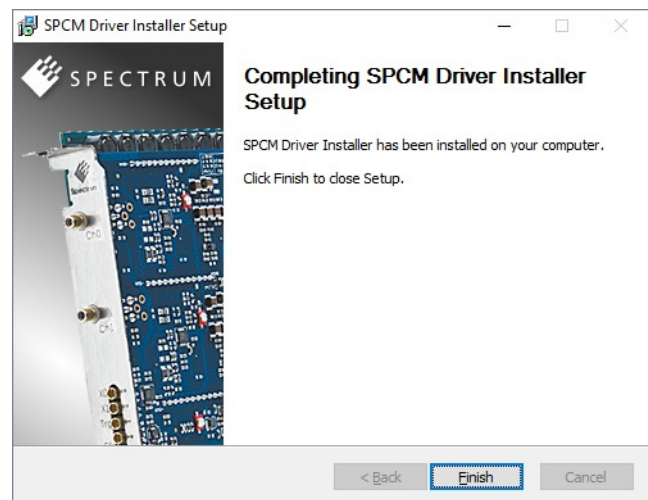


Image 13: Spectrum Driver Installer - finished

After installation

After running the Spectrum driver installer, the card will appear in the Windows device manager with its name matching the card series.

The card is now ready to be used with the new or updated driver.

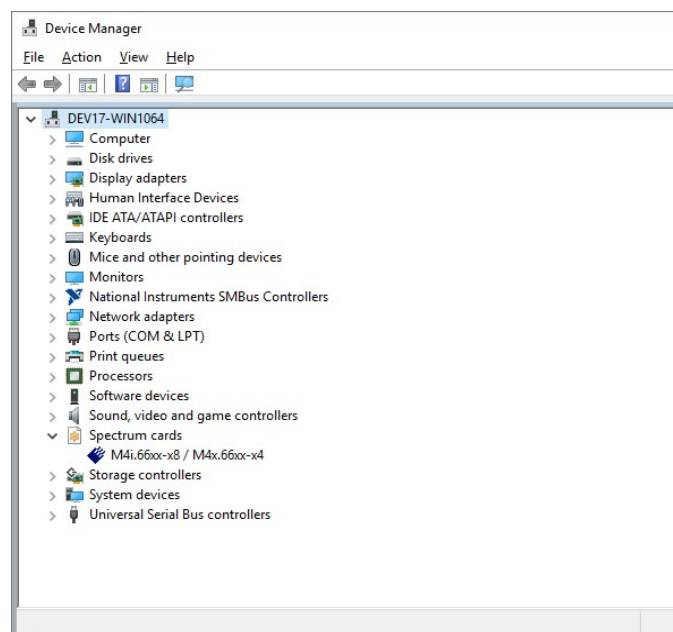


Image 14: Windows Device Manager showing properly installed Spectrum card

Linux

Overview

The Spectrum M2i/M3i/M4i/M4x/M2p/M5i cards and digitizerNETBOX/generatorNETBOX or hybridNETBOX products are delivered with Linux drivers suitable for Linux installations based on kernel 2.6, 3.x, 4.x or 5.x, single processor (non-SMP) and SMP systems, 32 bit and 64 bit systems. As each Linux distribution contains different kernel versions and different system setup it is in nearly every case necessary, to have a directly matching kernel driver for card level products to run it on a specific system. For digitizerNETBOX/generatorNETBOX or hybridNETBOX products the library is sufficient and no kernel driver has to be installed.

Spectrum delivers pre-compiled kernel driver modules for a number of common distributions with the cards. You may try to use one of these kernel modules for different distributions which have a similar kernel version. Unfortunately this won't work in most cases as most Linux system refuse to load a driver which is not exactly matching. In this case it is possible to get the kernel driver sources from Spectrum. Please contact your local sales representative to get more details on this procedure.

The Standard delivery contains the pre-compiled kernel driver modules for the most popular Linux distributions, like Suse, Debian, Fedora and Ubuntu. The list with all pre-compiled and readily supported distributions and their respective kernel version can be found under: https://spectrum-instrumentation.com/support/knowledgebase/software/Supported_Linux_Distributions.php or via the shown QR code.

The Linux drivers have been tested with all above mentioned distributions by Spectrum. Each of these distributions has been installed with the default setup using no kernel updates. A lot more different distributions are used by customers with self compiled kernel driver modules.



Driver Installation with Installation Script

The driver is delivered as installable kernel modules together with libraries to access the kernel driver. The installation script will help you with the installation of the kernel module and the library.

This installation is only needed if you are operating real locally installed cards. For software emulated demo cards, remotely installed cards or for digitizerNETBOX/generatorNETBOX/hybridNETBOX products it is only necessary to install the libraries without a kernel as explained further below.



Login as root

It is necessary to have the root rights for installing a driver.

Call the `install.sh <install_path> script`

This script will try to use the package management of the system to install the kernel module and user-space driver library packages:

- the kernel driver package is called „`spcm`“ (M2i, M3i) or „`spcm4`“ (M4i, M4x, M2p, M5i)
- the driver library package is called „`libspcm_linux`“

Udev support

Once the driver is loaded it automatically generates the device nodes under `/dev`. The cards are automatically named to `/dev/spcm0`, `/dev/spcm1`,...

You may use all the standard naming and rules that are available with udev.

Start the driver

The kernel driver should be loaded automatically when the system boots. If you need to load the kernel driver manually use the „`modprobe`“ command (as root or using `sudo`):

For M2i and M3i cards:

```
modprobe spcm
```

For M5i, M4i, M4x and M2p cards:

```
modprobe spcm4
```

Get first driver info

After the driver has been loaded successfully some information about the installed boards can be found in the matching `/proc/` file as shown below. Some basic information from the on-board EEPROM is listed for every card.

For M2i and M3i cards:

```
cat /proc/spcm_cards
```

For M5i, M4i, M4x and M2p cards:

```
cat /proc/spcm4_cards
```

Stop the driver

You can unload the kernel driver using the „modprobe -r“ command (as root or using sudo):

For M2i and M3i cards:

```
modprobe -r spcm
```

For M5i, M4i, M4x and M2p cards:

```
modprobe -r spcm4
```

Standard Driver Update

A driver update is done with the same commands as shown above. Please make sure that the driver has been stopped before updating it. To stop the driver you may use the proper “modprobe -r” command as shown above.

Compilation of kernel driver sources (optional and local cards only)

The driver sources are only available for existing customers upon special request. Please send an email to Support@spec.de to receive the kernel driver sources. The driver sources are not part of the standard delivery. The driver source package contains only the sources of the kernel module, not the sources of the library.

Please do the following steps for compilation and installation of the kernel driver module:

Login as root

It is necessary to have the root rights for installing a driver.

Call the compile script

The compile script depends on the type of card that you have installed:

- for M2i and M3i cards: make_spcm_linux_kerneldrv.sh
- for M5i, M4i, M4x and M2p cards: make_spcm4_linux_kerneldrv.sh

This script will examine the type of system you use and compile the kernel with the correct settings. The compilation of the kernel driver modules requires the kernel sources of the running kernel. These are normally available as a package with a name like kernel-devel, kernel-dev, kernel-source and need to match the running kernel.

The compiled driver module will be copied to the module directory of the kernel (/lib/modules/\$(uname -r)/kernel/drivers/), and will be loaded automatically at the next boot. To load or unload the kernel driver module manually use the modprobe command as explained above in “Start the driver” and “Stop the driver”.

Update of a self compiled kernel driver

If the kernel driver has changed, one simply has to perform the same steps as shown above and recompile the kernel driver module. However the kernel driver module isn't changed very often.

Normally an update only needs new libraries. To update the libraries only you can either download the full Linux driver (spcm_linux_drv_v123b4567) and only use the libraries out of this or one downloads the library package which is much smaller and doesn't contain the pre-compiled kernel driver module (spcm_linux_lib_v123b4567).

The update is done with a dedicated script which only updates the library file. This script is present in both driver archives:

```
sh install_libonly.sh
```

Installing the library only without a kernel (for remote devices)

The kernel driver module only contains the basic hardware functions that are necessary to access locally installed card level products. The main part of the driver is located inside a dynamically loadable library that is delivered with the driver. This library is available in two different versions:

- `spcm_linux_32bit_std++6.so` - supporting `libstdc++6` on 32 bit systems
- `spcm_linux_64bit_std++6.so` - supporting `libstdc++6` on 64 bit systems

The matching version is installed automatically in the `"/usr/lib"` or `"/usr/lib64/"` or `"/usr/lib/x86_64-linux-gnu"` directory (depending on your Linux distribution) by the kernel driver install script for card level products. The library is renamed for easy access to `libspcm_linux.so`.

For digitizerNETBOX/generatorNETBOX/hybridNETBOX products and also for evaluating or using only the software simulated demo cards the library is installed with a separate install script:

```
sh install_libonly.sh
```

To access the driver library one must include the library in the compilation:

```
gcc -o test_prgr -lspcm_linux test.cpp
```

To start programming the cards under Linux please use the standard C/C++ examples which are all running under Linux and Windows.

Installation from Spectrum Repository

The driver library, Spectrum Control Center and SBench6 can be easily installed and updated from our online repositories. Adding the repository to the system and installing software differs depending on the package format used by the Linux distribution.

DEB based distributions (like Debian, Ubuntu and derived distributions)

Execute the following commands to get the Spectrum repository key and convert it for local use:

```
wget http://spectrum-instrumentation.com/dl/repo-key.asc  
gpg --dearmor -o repo-key.gpg repo-key.asc  
cp repo-key.gpg /etc/apt/spectrum-instrumentation.gpg
```

To add the repository create a new file `/etc/apt/sources.list.d/spectrum-instrumentation.list` with this content. Please note that there is a mandatory blank between URL and `"/"`:

```
deb [signed-by=/etc/apt/spectrum-instrumentation.gpg] http://spectrum-instrumentation.com/dl/ ./
```

Alternatively this line can be added to `/etc/apt/sources.list`

Then run

```
sudo apt update
```

to update the repository information.

To install the software (e.g. SBench6) run

```
sudo apt install sbench6
```

An overview of DEB based distributions can be found here: https://en.wikipedia.org/wiki/Category:Debian-based_distributions

RPM based distributions

On distributions using Zypper (such as openSUSE, SLES, ...) to add the repository run:

```
sudo zypper ar --repo http://spectrum-instrumentation.com/dl/spectrum_instrumentation.repo
```

The repository information will be updated automatically.

To install the software (e.g. SBench6) run

```
sudo zypper install SBench6
```

On distributions using DNF (such as Fedora, CentOS Stream, RHEL, ...) to add the repository run

```
sudo dnf config-manager --add-repo http://spectrum-instrumentation.com/dl/spectrum_instrumentation.repo
```

The repository information will be updated automatically.

To install the software (e.g. SBench6) run

```
sudo dnf install SBench6
```

An overview of RPM based distributions can be found here: https://en.wikipedia.org/wiki/Category:RPM-based_Linux_distributions

Control Center

The Spectrum Control Center is also available for Linux and needs to be installed separately. The features of the Control Center are described in a later chapter in deeper detail. The Control Center has been tested under all Linux distributions for which Spectrum delivers pre-compiled kernel modules. The following packages need to be installed to run the Control Center:

- X-Server
- expat
- freetype
- fontconfig
- libpng
- libspcm_linux (the Spectrum Linux driver library)

Installation

Use the supplied packages in either *.deb or *.rpm format found in the driver section of the USB stick by double clicking the package file root rights from a X-Windows window.

The Control Center is installed under KDE, Gnome or Unity in the system/system tools section. It may be located directly in this menu or under a „More Programs“ menu. The final location depends on the used Linux distribution. The program itself is installed as /usr/bin/spcmcontrol and may be started directly from here.

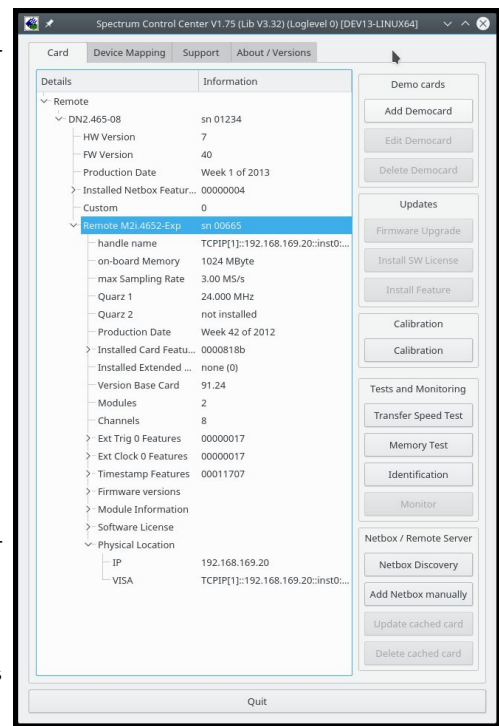


Image 15: Device Manager showing a new Spectrum card

Manual Installation

To manually install the Control Center, first extract the files from the rpm matching your distribution:

```
rpm2cpio spcmcontrol-{Version}.rpm > ~/spcmcontrol-{Version}.cpio
cd ~/
cpio -id < spcmcontrol-{Version}.cpio
```

You get the directory structure and the files contained in the rpm package. Copy the binary spcmcontrol to /usr/bin. Copy the .desktop file to /usr/share/applications. Run ldconfig to update your systems library cache. Finally you can run spcmcontrol.

Troubleshooting

If you get a message like the following after starting spcmcontrol:

```
spcm_control: error while loading shared libraries: libz.so.1: cannot open shared object file: No such file or directory
```


Run `ldd spcm_control` in the directory where `spcm_control` resides to see the dependencies of the program. The output may look like this:

```
libXext.so.6 => /usr/X11R6/lib/libXext.so.6 (0x4019e000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x401ad000)
libz.so.1 => not found
libdl.so.2 => /lib/libdl.so.2 (0x402ba000)
libpthread.so.0 => /lib/tls/libpthread.so.0 (0x402be000)
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x402d0000)
```

As seen in the output, one of the libraries isn't found inside the library cache of the system. Be sure that this library has been properly installed. You may then run `ldconfig`. If this still doesn't help please add the library path to `/etc/ld.so.conf` and run `ldconfig` again.

If the `libspcm_linux.so` is quoted as missing please make sure that you have installed the card driver properly before. If any other library is stated as missing please install the matching package of your distribution.

Software

This chapter gives you an overview about the structure of the drivers and the software, where to find and how to use the examples. It shows in detail, how the drivers are included using different programming languages and deals with the differences when calling the driver functions from them.

This manual only shows the use of the standard driver API. For further information on programming drivers for third-party software like LabVIEW, MATLAB, IVI or SCAPP an additional manual is required that is available on the USB stick or by download from our homepage.



Software Overview

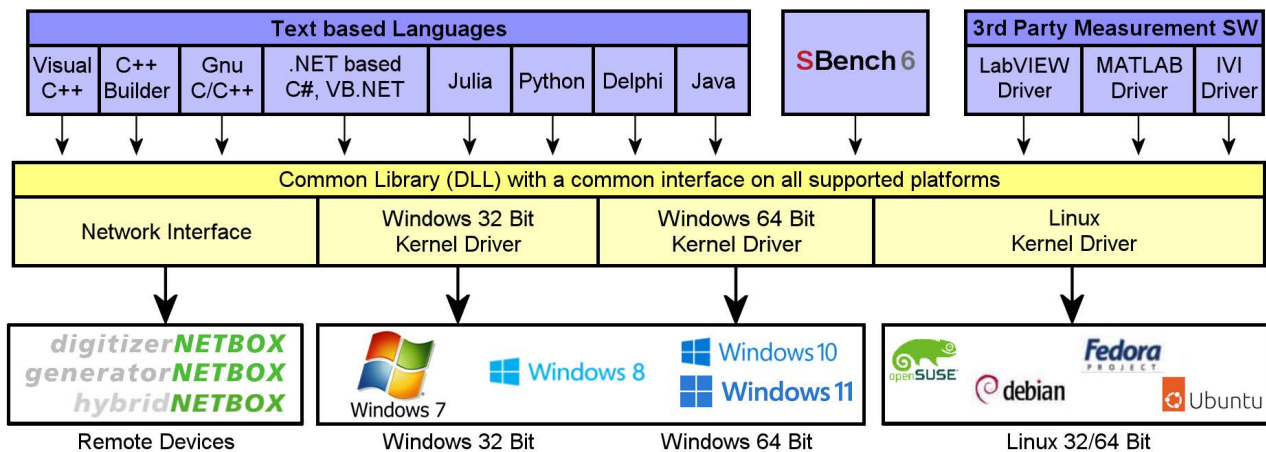


Image 16: Spectrum Kernel Driver, API Library and Software structure

The Spectrum drivers offer you a common and fast API for using all of the board hardware features. This API is the same on all supported operating systems. Based on this API one can write own programs using any programming language that can access the driver API. This manual describes in detail the driver API, providing you with the necessary information to write your own programs.

The drivers for third-party products like LabVIEW or MATLAB, IVI or SCAPP are also based on this API. The special functionality of these drivers is not subject of this document and is described with separate manuals available on the USB stick or on the website.

Card Control Center

A special Card Control Center is available on the USB stick and from the internet for all Spectrum M2i/M3i/M4i/M4x/M2p/M5i cards and for all digitizerNETBOX, generatorNETBOX or hybridNETBOX products. Windows users find the Control Center installer on the USB stick under „Install\win\spcmcontrol_install.exe“.

Linux users find the versions for the different stdc++ libraries under /Install/linux/spcm_control_center/ as RPM packages.

When using a digitizerNETBOX/generatorNETBOX/hybridNETBOX the Card Control Center installers for Windows and Linux are also directly available from the integrated webserver.

The Control Center under Windows and Linux is available as an executive program.

Under Windows it is also linked as a system control and can be accessed directly from the Windows control panel. Under Linux it is also available from the KDE System Settings, the Gnome or Unity Control Center. The different functions of the Spectrum Card Control Center are explained in detail in the following passages.

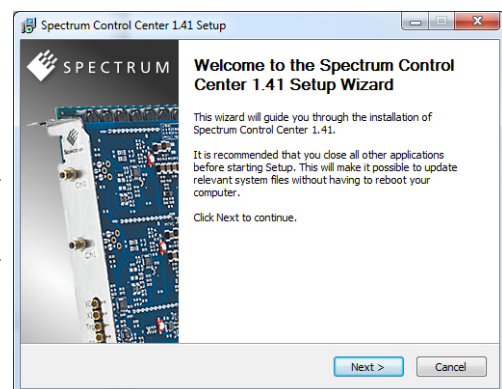


Image 17: Spectrum Control Center Installer



To install the Spectrum Control Center you will need to be logged in with administrator rights for your operating system. On all Windows versions, starting with Windows Vista, installations with enabled UAC will ask you to start the installer with administrative rights (run as administrator).

Discovery of Remote Cards, digitizerNETBOX/generatorNETBOX/hybridNETBOX products

The Discovery function helps you to find and identify the Spectrum LXI instruments like digitizerNETBOX, generatorNETBOX or hybridNETBOX available to your computer on the network. The Discovery function will also locate Spectrum card products handled by an installed Spectrum Remote Server somewhere on the network. The function is not needed if you only have locally installed cards.

Please note that only remote products are found that are currently not used by another program. Therefore in a bigger network the number of Spectrum products found may vary depending on the current usage of the products.

Execute the Discovery function by pressing the „Discovery“ button. There is no progress window shown. After the discovery function has been executed the remotely found Spectrum products are listed under the node Remote as separate card level products. Inhere you find all hardware information as shown in the next topic and also the needed VISA resource string to access the remote card.

Please note that these information is also stored on your system and allows Spectrum software like SBench 6 to access the cards directly once found with the Discovery function.

After closing the control center and re-opening it the previously found remote products are shown with the prefix cached, only showing the card type and the serial number. This is the stored information that allows other Spectrum products to access previously found cards. Using the „Update cached cards“ button will try to re-open these cards and gather information of it. Afterwards the remote cards may disappear if they're in use from somewhere else or the complete information of the remote products is shown again.

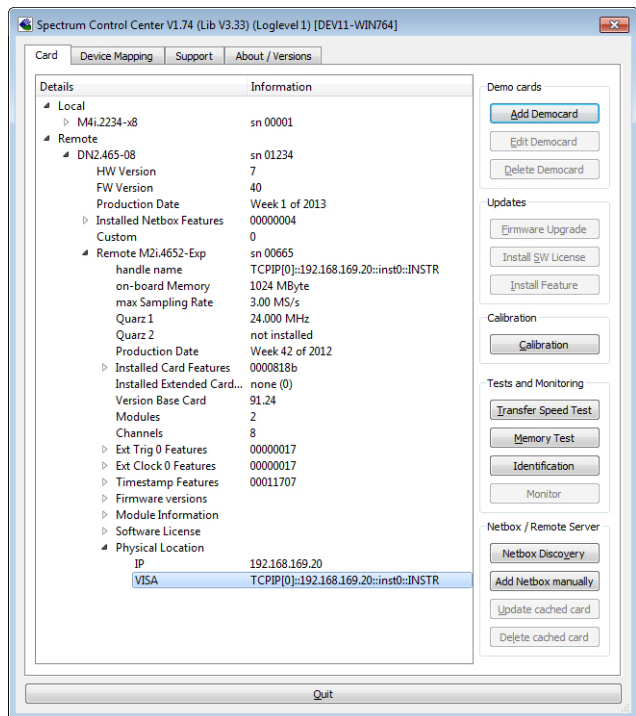


Image 18: Spectrum Control Center showing detail card information

Enter IP Address of digitizerNETBOX/generatorNETBOX/hybridNETBOX manually

If for some reason an automatic discovery is not suitable, such as the case where the remote device is located in a different subnet, it can also be manually accessed by its type and IP address.

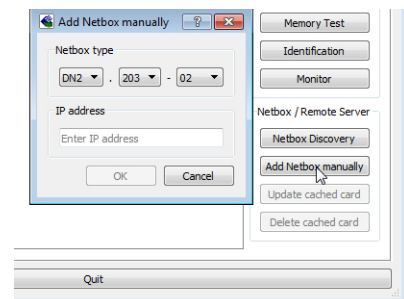


Image 19: Spectrum Control Center - entering an IP address for a NETBOX

Wake On LAN of digitizerNETBOX/generatorNETBOX/hybridNETBOX

Cached digitizerNETBOX/generatorNETBOX/hybridNETBOX products that are currently in standby mode can be woken up by using the „Wake remote device“ entry from the context menu.

The Control Center will broadcast a standard Wake On LAN „Magic Packet“, that is sent to the device's MAC address.

It is also possible to use any other Wake On LAN software to wake e.g. a digitizerNETBOX by sending such a „Magic Packet“ to the MAC address, which must be then entered manually.

It is also possible to wake a remote device from your own application software by using the SPC_NETBOX_WAKEONLAN register. To wake a digitizerNETBOX, generatorNETBOX or hybridNETBOX with the MAC address „00:03:2d:20:48“, the following command can be issued:

```
spcm_dwSetParam_i64 (NULL, SPC_NETBOX_WAKEONLAN, 0x00032d2048ec);
```

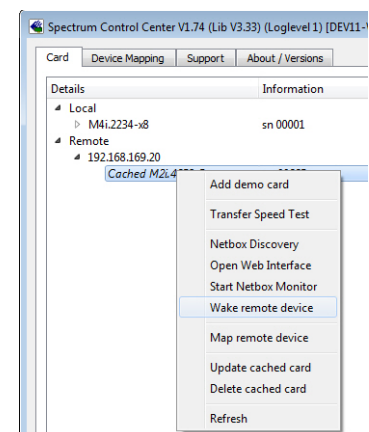


Image 20: Spectrum Control Center: wake on LAN for a cached card

Netbox Monitor

The Netbox Monitor permanently monitors whether the digitizerNETBOX/generatorNETBOX/hybridNETBOX is still available through LAN. This tool is helpful if e.g. the digitizerNETBOX is located somewhere in the company LAN or located remotely or directly mounted inside another device. Starting the Netbox Monitor can be done in two different ways:

- Starting manually from the Spectrum Control Center using the context menu as shown above
- Starting from command line. The Netbox Monitor program is automatically installed together with the Spectrum Control Center and is located in the selected install folder. Using the command line tool one can place a simple script into the autostart folder to have the Netbox Monitor running automatically after system boot. The command line tool needs the IP address of the digitizerNETBOX/generatorNETBOX/hybridNETBOX to monitor:

```
NetboxMonitor 192.168.169.22
```

The Netbox Monitor is shown as a small window with the type of digitizerNETBOX/generatorNETBOX in the title and the IP address under which it is accessed in the window itself. The Netbox Monitor runs completely independent of any other software and can be used in parallel to any application software. The background of the IP address is used to display the current status of the device. Pressing the Escape key or alt + F4 (Windows) terminates the Netbox Monitor permanently.



After starting the Netbox Monitor it is also displayed as a tray icon under Windows. The tray icon itself shows the status of the digitizerNETBOX/generatorNETBOX/hybridNETBOX as a color. Please note that the tray icon may be hidden as a Windows default and need to be set to visible using the Windows tray setup.

Left clicking on the tray icon will hide/show the small Netbox Monitor status window. Right clicking on the tray icon as shown in the picture on the right will open up a context menu. In here one can again select to hide/show the Netbox Monitor status window, one can directly open the web interface from here or quit the program (including the tray icon) completely.

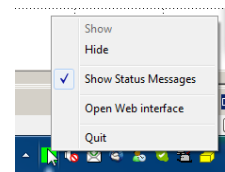


Image 21: Netbox Monitor activation

The checkbox „Show Status Message“ controls whether the tray icon should emerge a status message on status change. If enabled (which is default) one is notified with a status message if for example the LAN connection to the digitizerNETBOX/generatorNETBOX/hybridNETBOX is lost.

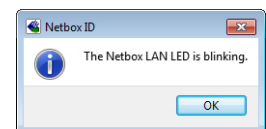
The status colors:

- Green: digitizerNETBOX/generatorNETBOX/hybridNETBOX available and accessible over LAN
- Cyan: digitizerNETBOX/generatorNETBOX/hybridNETBOX is used from my computer
- Yellow: digitizerNETBOX/generatorNETBOX/hybridNETBOX is used from a different computer
- Red: LAN connection failed, digitizerNETBOX/generatorNETBOX/hybridNETBOX is no longer accessible

Device identification

Pressing the *Identification* button helps to identify a certain device in either a remote location, such as inside a 19" rack where the back of the device with the type plate is not easily accessible, or a local device installed in a certain slot. Pressing the button starts flashing a visible LED on the device, until the dialog is closed, for:

- On a digitizerNETBOX/generatorNETBOX/hybridNETBOX: the LAN LED light on the front plate of the device
- On local or remote M5i, M4i, M4x or M2p card: the indicator LED on the card's bracket



This feature is not available for M2i/M3i cards, either local or remote, other than inside a digitizerNETBOX or generatorNETBOX.

Hardware information

Through the Control Center you can easily get the main information about all the installed Spectrum hardware. For each installed card there is a separate tree of information available. The picture shows the information for one installed card by example. This given information contains:

- Basic information as the type of card, the production date and its serial number, as well as the installed memory, the hardware revision of the base card, the number of available channels and installed acquisition modules.
- Information about the maximum sampling clock and the available quartz clock sources.
- The installed features/options in a sub-tree. The shown card is equipped for example with the option Multiple Recording, Gated Sampling, Timestamp and ABA-mode.
- Detailed Information concerning the installed acquisition modules. In case of the shown analog acquisition card the information consists of the module's hardware revision, of the converter resolution and the last calibration date as well as detailed information on the available analog input ranges, offset compensation capabilities and additional features of the inputs.

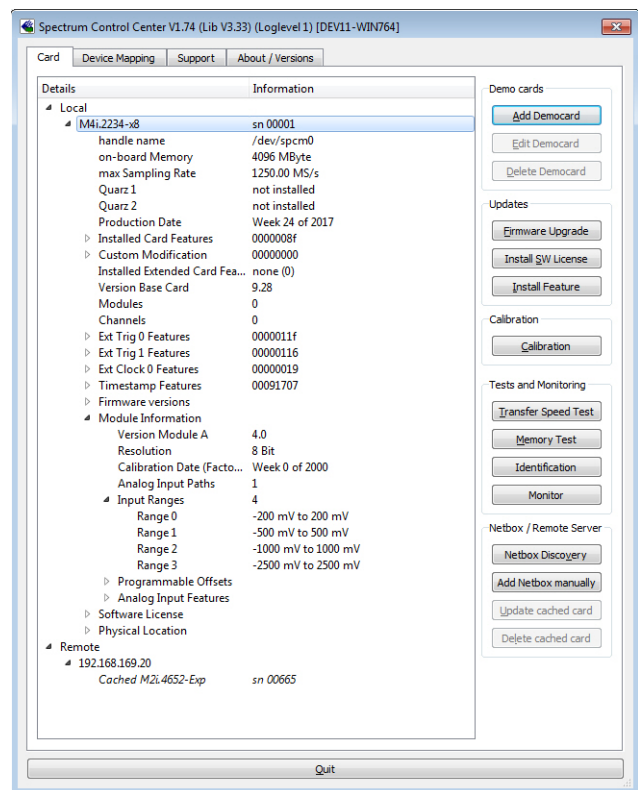


Image 22: Spectrum Control Center: detailed hardware information on installed card

Firmware information

Another sub-tree is informing about the cards firmware version. As all Spectrum cards consist of several programmable components, there is one firmware version per component.

Nearly all of the components firmware can be updated by software. The only exception is the configuration device, which only can receive a factory update.

The procedure on how to update the firmware of your Spectrum card with the help of the card control center is described in a dedicated section later on.

The procedure on how to update the firmware of your digitizerNETBOX/generatorNETBOX/hybridNETBOX with the help of the integrated Webserver is described in a dedicated chapter later on.

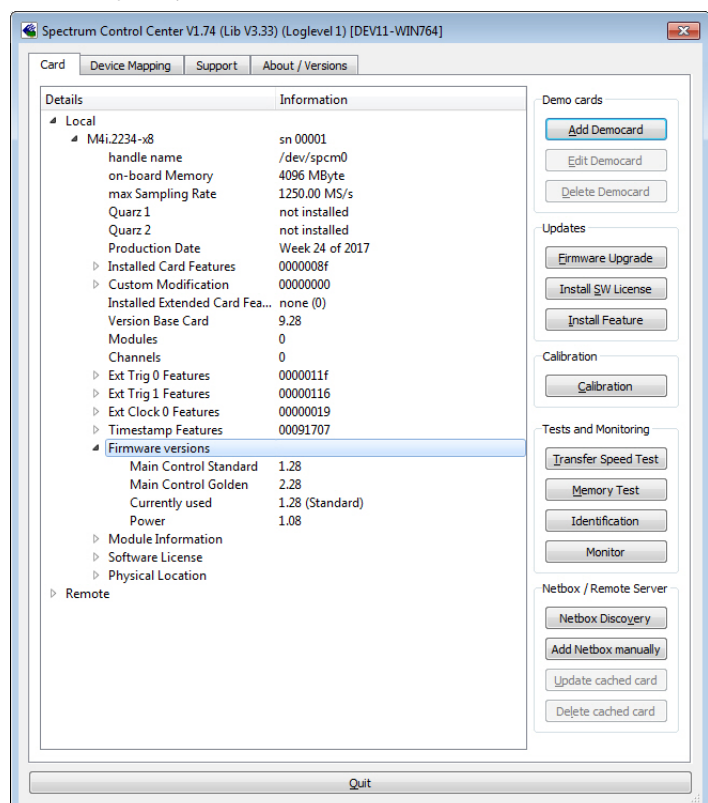


Image 23: Spectrum Control Center - showing firmware information of an installed card

Software License information

This sub-tree is informing about installed possible software licenses.

As a default all cards come with the demo professional license of SBench6, that is limited to 30 starts of the software with all professional features unlocked.

The number of demo starts left can be seen here.

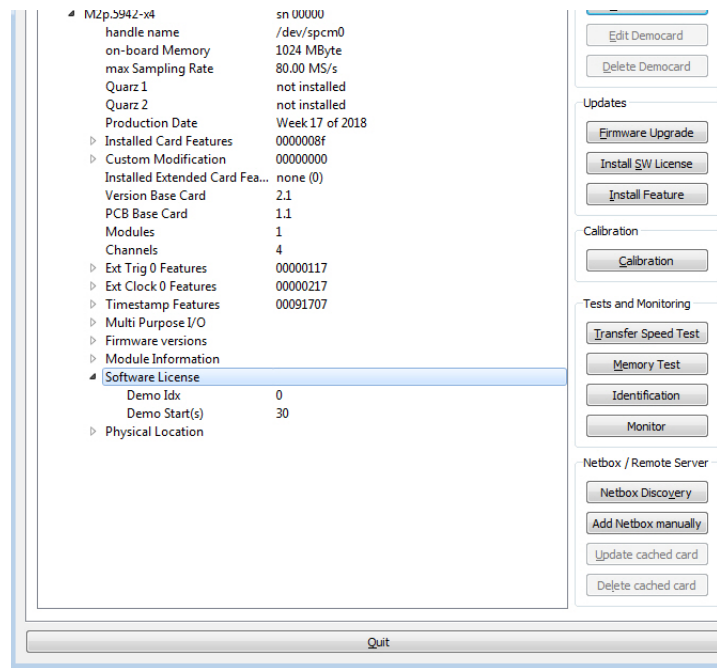


Image 24: Spectrum Control Center - showing firmware information of an installed card

Driver information

The Spectrum card control center also offers a way to gather information on the installed and used Spectrum driver.

The information on the driver is available through a dedicated tab, as the picture is showing in the example.

The provided information informs about the used type, distinguishing between Windows or Linux driver and the 32 bit or 64 bit type.

It also gives direct information about the version of the installed Spectrum kernel driver, separately for M2i/ M3i cards and M4i/M4x/M2p/M5i cards and the version of the library (which is the *.dll file under Windows).

The information given here can also be found under Windows using the device manager from the control panel. For details in driver details within the control panel please stick to the section on driver installation in your hardware manual.

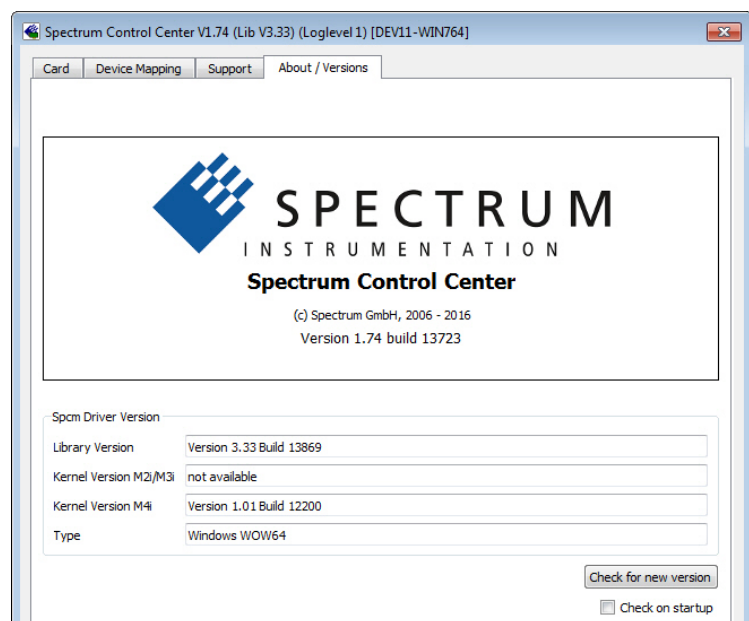


Image 25: Spectrum Control Center - showing driver information details

Installing and removing Demo cards

With the help of the card control center one can install demo cards in the system. A demo card is simulated by the Spectrum driver including data production for acquisition cards. As the demo card is simulated on the lowest driver level all software can be tested including SBench, own applications and drivers for third-party products like LabVIEW. The driver supports up to 64 demo cards at the same time. The simulated memory as well as the simulated software options can be defined when adding a demo card to the system.

Please keep in mind that these demo cards are only meant to test software and to show certain abilities of the software. They do not simulate the complete behavior of a card, especially not any timing concerning trigger, recording length or FIFO mode notification. The demo card will calculate data every time directly after been called and give it to the user application without any more delay. As the calculation routine isn't speed optimized, generating demo data may take more time than acquiring real data and transferring them to the host PC.

Installed demo cards are listed together with the real hardware in the main information tree as described above. Existing demo cards can be deleted by clicking the related button. The demo card details can be edited by using the edit button. It is for example possible to virtually install additional feature to one card or to change the type to test with a different number of channels.

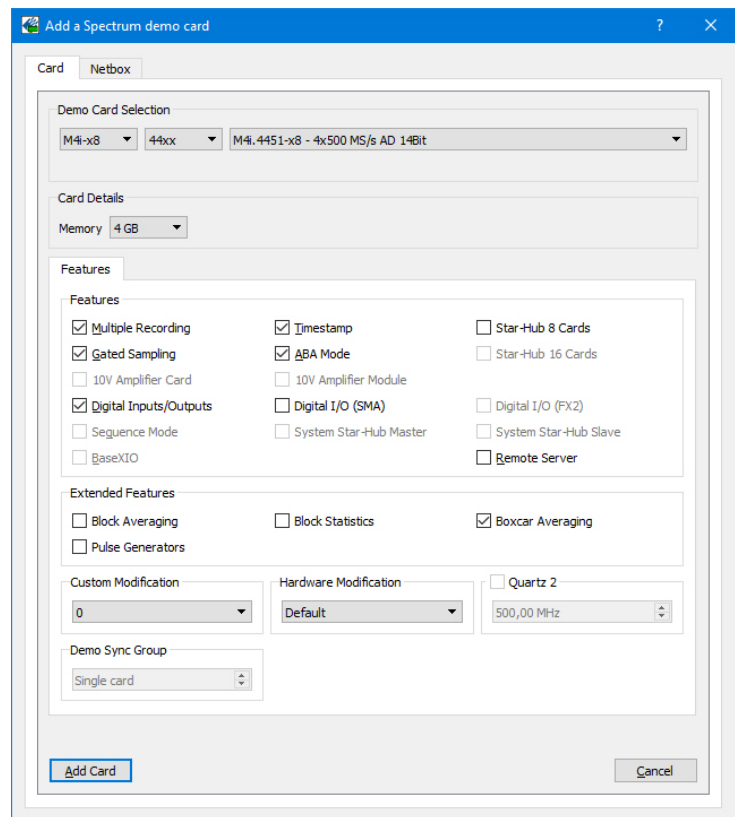


Image 26: Spectrum Control Center - adding a demo card to the system

For installing demo cards on a system without real hardware simply run the Control Center installer. If the installer is not detecting the necessary driver files normally residing on a system with real hardware, it will simply install the Spcm_driver.



Feature upgrade

All optional features of the M2i/M3i/M4i/M4x/M2p/M5i cards that do not require any hardware modifications can be installed on fielded cards. After Spectrum has received the order, the customer will get a personalized upgrade code. Just start the card control center, click on „install feature“ and enter that given code. After a short moment the feature will be installed and ready to use. No restart of the host system is required.

For details on the available options and prices please contact your local Spectrum distributor.

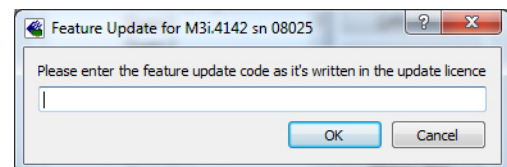


Image 27: Spectrum Control Center - feature update, code entry

Software License upgrade

The software license for SBench 6 Professional is installed on the hardware. If ordering a software license for a card that has already been delivered you will get an upgrade code to install that software license. The upgrade code will only match for that particular card with the serial number given in the license. To install the software license please click the „Install SW License“ button and type in the code exactly as given in the license.

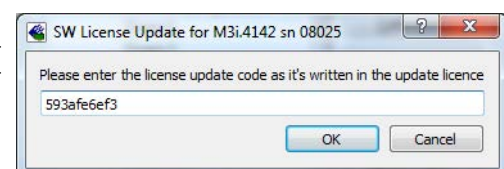


Image 28: Spectrum Control Center - software license install

Performing card calibration (A/D only)

The card control center also provides an easy way to access the automatic card calibration routines of the Spectrum A/D converter cards. Depending on the used card family this can affect offset calibration only or also might include gain calibration. Please refer to the dedicated chapter in your hardware manual for details.

This function is not available for D/A cards (AWG) or digital I/O cards

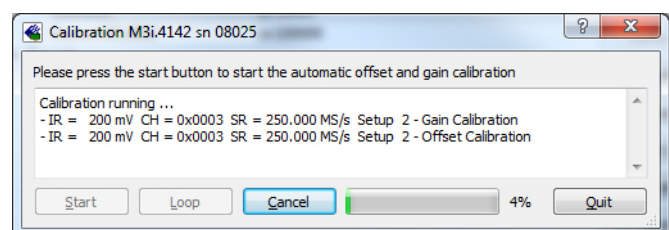


Image 29: Spectrum Control Center - running an on-board calibration

Performing memory test

The complete on-board memory of the Spectrum M2i/M3i/M4i/M4x/M2p/M5i cards can be tested by the memory test included with the card control center.

When starting the test, randomized data is generated and written to the on-board memory. After a complete write cycle all the data is read back and compared with the generated pattern.

Depending on the amount of installed on-board memory, and your computer's performance this operation might take a while.

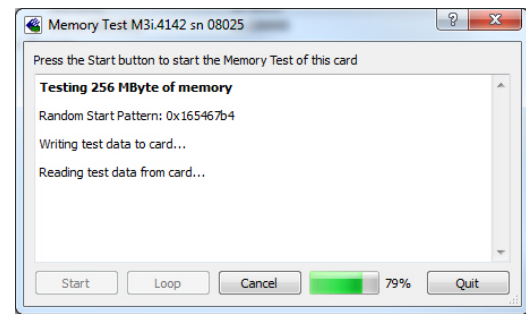


Image 30: Spectrum Control Center - performing memory test

Transfer speed test

The control center allows to measure the bus transfer speed of an installed Spectrum card. Therefore different setup is run multiple times and the overall bus transfer speed is measured. To get reliable results it is necessary that you disable debug logging as shown below. It is also highly recommended that no other software or time-consuming background threads are running on that system. The speed test program runs the following two tests:

- Repetitive Memory Transfers: single DMA data transfers are repeated and measured. This test simulates the measuring of pulse repetition frequency when doing multiple single-shots. The test is done using different block sizes. One can estimate the transfer in relation to the transferred data size on multiple single-shots.
- FIFO mode streaming: this test measures the streaming speed in FIFO mode. The test can only use the same direction of transfer the card has been designed for (card to PC=read for all DAQ cards, PC to card=write for all generator cards and both directions for I/O cards). The streaming speed is tested without using the front-end to measure the maximum bus speed that can be reached. The Speed in FIFO mode depends on the selected notify size which is explained later in this manual in greater detail.

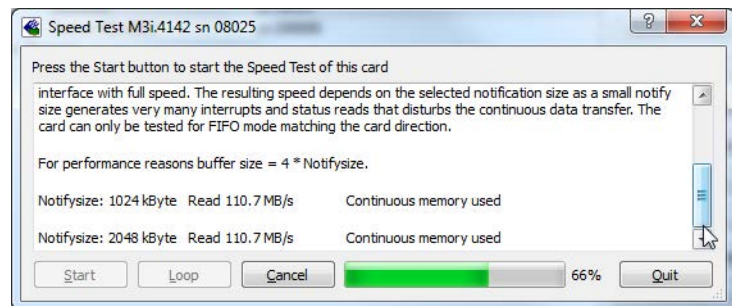


Image 31: Spectrum Control Center - running a transfer speed test of one card

The results are given in MB/s meaning MByte per second. To estimate whether a desired acquisition speed is possible to reach one has to calculate the transfer speed in bytes. There are a few things that have to be put into the calculation:

- 12, 14 and 16 bit analog cards need two bytes for each sample.
- 16 channel digital cards need 2 bytes per sample while 32 channel digital cards need 4 bytes and 64 channel digital cards need 8 bytes.
- The sum of analog channels must be used to calculate the total transfer rate.
- The figures in the Speed Test Utility are given as MBytes, meaning $1024 * 1024$ Bytes, 1 MByte = 1048576 Bytes

As an example running a card with 2 14 bit analog channels with 28 MHz produces a transfer rate of $[2 \text{ channels} * 2 \text{ Bytes/Sample} * 28000000] = 112000000 \text{ Bytes/second}$. Taking the above figures measured on a standard 33 MHz PCI slot the system is just capable of reaching this transfer speed: $108.0 \text{ MB/s} = 108 * 1024 * 1024 = 113246208 \text{ Bytes/second}$.

Unfortunately it is not possible to measure transfer speed on a system without having a Spectrum card installed.

Debug logging for support cases

For answering your support questions as fast as possible, the setup of the card, driver and firmware version and other information is very helpful.

Therefore the card control center provides an easy way to gather all that information automatically.

Different debug log levels are available through the graphical interface. By default the log level is set to „no logging“ for maximum performance.

The customer can select different log levels and the path of the generated ASCII text file. One can also decide to delete the previous log file first before creating a new one automatically or to append different logs to one single log file.

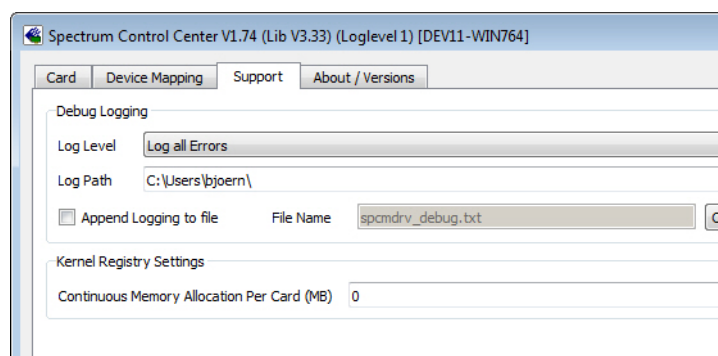


Image 32: Spectrum Control Center - activate debug logging for support cases



For maximum performance of your hardware, please make sure that the debug logging is set to „no logging“ for normal operation. Please keep in mind that a detailed logging in append mode can quickly generate huge log files.

Device mapping

Within the „Device mapping“ tab of the Spectrum Control Center, one can enable the re-mapping of Spectrum devices, be it either local cards, remote instruments such as a digitizerNETBOX, generatorNETBOX, hybridNETBOX or even cards in a remote PC and accessed via the Spectrum remote server option.

In the left column the re-mapped device name is visible that is given to the device in the right column with its original un-mapped device string.

In this example the two local cards „spcm0“ and „spcm1“ are re-mapped to „spcm1“ and „spcm0“ respectively, so that their names are simply swapped.

The remote digitizerNETBOX device is mapped to spcm2.

The application software can then use the re-mapped name for simplicity instead of the quite long VISA string.

Changing the order of devices within one group (either local cards or remote devices) can simply be accomplished by dragging&dropping the cards to their desired position in the same table.

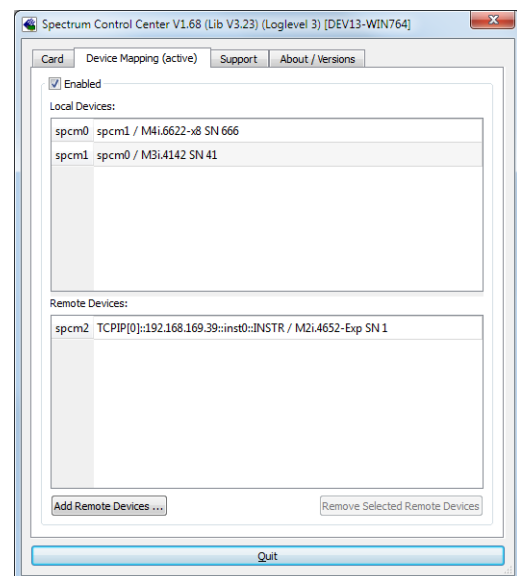


Image 33: Spectrum Control Center - using device mapping

Firmware upgrade

One of the major features of the card control center is the ability to update the card's firmware by an easy-to-use software. The latest firmware revisions can be found in the download section of our homepage under <http://www.spectrum-instrumentation.com>.

A new firmware version is provided there as an installer, that copies the latest firmware to your system. All files are located in a dedicated subfolder „FirmwareUpdate“ that will be created inside the Spectrum installation folder. Under Windows this folder by default has been created in the standard program installation directory.

Please do the following steps when wanting to update the firmware of your M2i/M3i/M4i/M4x/M2p/M5i card:

- Download the latest software driver for your operating system provided on the Spectrum homepage.
- Install the new driver as described in the driver install section of your hardware manual or install manual. All manuals can also be found on the Spectrum homepage in the literature download section.
- Download and run the latest Spectrum Control Center installer.
- Download the installer for the new firmware version.
- Start the installer and follow the instructions given there.
- Start the card control center, select the „card“ tab, select the card from the listbox and press the „firmware update“ button on the right side.

The dialog then will inform you about the currently installed firmware version for the different devices on the card and the new versions that are available. All devices that will be affected with the update are marked as „update needed“. Simply start the update or cancel the operation now, as a running update cannot be aborted.

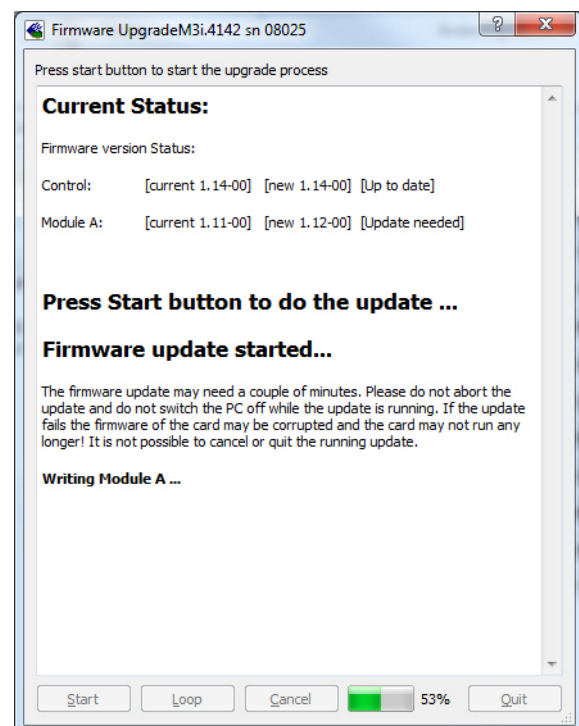


Image 34: Spectrum Control Center - doing a firmware update for one device



Please keep in mind that you have to start the update for each card installed in your system separately. Select one card after the other from the listbox and press the „firmware update“ button. The firmware installer on the other hand only needs to be started once prior to the update.



Do not abort or shut down the computer while the firmware update is in progress. After a successful update please shut down your PC completely (remove power). The re-powering is required to finally activate the new firmware version of your Spectrum card.

Accessing the hardware with SBench 6

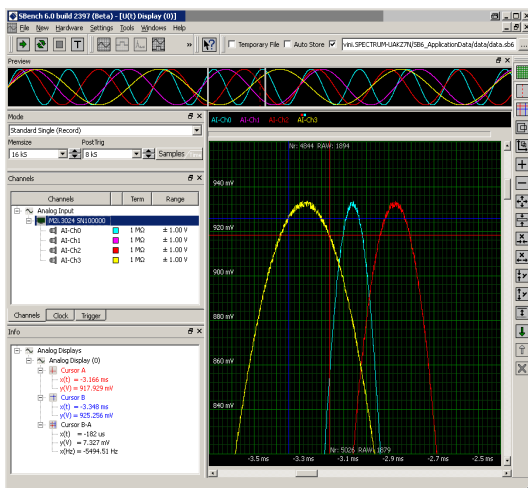


Image 35: SBench 6 overview of main functionality with demo data

After the installation of the cards and the drivers it can be useful to first test the card function with a ready to run software before starting with programming. If accessing a digitizerNETBOX/generatorNETBOX a full SBench 6 Professional license is installed on the system and can be used without any limitations. For plug-in card level products a base version of SBench 6 is delivered with the card on USB stick also including a 30 starts Professional demo version for plain card products. If you already have bought a card prior to the first SBench 6 release please contact your local dealer to get a SBench 6 Professional demo version. All digitizerNETBOX/generatorNETBOX products come with a pre-installed full SBench 6 Professional.

SBench 6 supports all current acquisition and generation cards and digitizerNETBOX/generatorNETBOX products from Spectrum. Depending on the used product and the software setup, one can use SBench as a digital storage oscilloscope, a spectrum analyzer, a signal generator, a pattern generator, a logic analyzer or simply as a data recording front end. Different export and import formats allow the use of SBench 6 together with a variety of other programs.

On the USB stick you'll find an install version of SBench 6 in the directory „/Install/SBench6“.

The current version of SBench 6 is available free of charge directly from the Spectrum website: www.spectrum-instrumentation.com. Please go to the download section and get the latest version there.

SBench 6 has been designed to run under Windows 7, 8, 10 and Windows 11 as well as Linux using KDE, Gnome or Unity Desktop.

C/C++ Driver Interface

C/C++ is the main programming language for which the drivers have been designed for. Therefore the interface to C/C++ is the best match. All the small examples of the manual showing different parts of the hardware programming are done with C. As the libraries offer a standard interface it is easy to access the libraries also with other programming languages like Delphi, Basic, Python or Java. Please read the following chapters for additional information on this.

Header files

The basic task before using the driver is to include the header files that are delivered on USB stick together with the board. The header files are found in the directory /Driver/c_header. Please don't change them in any way because they are updated with each new driver version to include the new registers and new functionality.

Table 4: list of C/C++ header files in driver

dlltyp.h	Includes the platform specific definitions for data types and function declarations. All data types are based on these definitions. The use of this type definition file allows the use of examples and programs on different platforms without changes to the program source. The header file supports Microsoft Visual C++, Borland C++ Builder and GNU C/C++ directly. When using other compilers it might be necessary to make a copy of this file and change the data types according to this compiler.
regs.h	Defines all registers and commands which are used in the Spectrum driver for the different boards. The registers a board uses are described in the board specific part of the documentation. This header file is common for all cards. Therefore this file also contains a huge number of registers used on other card types than the one described in this manual. Please stick to the manual to see which registers are valid for your type of card.
spcm_drv.h	Defines the functions of the used SpcM driver. All definitions are taken from the file dlltyp.h. The functions themselves are described below.
spcerr.h	Contains all error codes used with the Spectrum driver. All error codes that can be given back by any of the driver functions are also described here briefly. The error codes and their meaning are described in detail in the appendix of this manual.

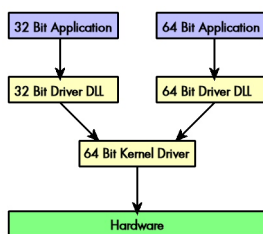
Example for including the header files:

```
// ----- driver includes -----
#include "dlltyp.h"      // 1st include
#include "regs.h"        // 2nd include
#include "spcerr.h"      // 3rd include
#include "spcm_drv.h"    // 4th include
```



Please always keep the order of including the four Spectrum header files. Otherwise some or all of the functions do not work properly or compiling your program will be impossible!

General Information on Windows 64 bit drivers



After installation of the Spectrum 64 bit driver there are two general ways to access the hardware and to develop applications. If you're going to develop a real 64 bit application it is necessary to access the 64 bit driver dll (spcm_win64.dll) as only this driver dll is supporting the full 64 bit address range.

But it is still possible to run 32 bit applications or to develop 32 bit applications even under Windows 64 bit. Therefore the 32 bit driver dll (spcm_win32.dll) is also installed in the system. The Spectrum SBench5 software is for example running under Windows 64 bit using this driver. The 32 bit dll of course only offers the 32 bit address range and is therefore limited to access only 4 GByte of memory. Beneath both drivers the 64 bit kernel driver is running.

Mixing of 64 bit application with 32 bit dll or vice versa is not possible.

Microsoft Visual C++ 6.0, 2005 and newer 32 Bit

Include Driver

The driver files can be directly included in Microsoft C++ by simply using the library file spcm_win32_msvcpp.lib that is delivered together with the drivers. The library file can be found on the CD in the path /examples/c_cpp/c_header. Please include the library file in your Visual C++ project as shown in the examples. All functions described below are now available in your program.

Examples

Examples can be found on CD in the path /examples/c_cpp. This directory includes a number of different examples that can be used with any card of the same type (e.g. A/D acquisition cards, D/A acquisition cards). You may use these examples as a base for own programming and modify them as you like. The example directories contain a running workspace file for Microsoft Visual C++ 6.0 (*.dsw) as well as project files for Microsoft Visual Studio 2005 and newer (*.vcproj) that can be directly loaded or imported and compiled. There are also some more board type independent examples in separate subdirectory. These examples show different aspects of the cards like programming options or synchronization and can be combined with one of the board type specific examples.

As the examples are build for a card class there are some checking routines and differentiation between cards families. Differentiation aspects can be number of channels, data width, maximum speed or other details. It is recommended to change the examples matching your card type to obtain maximum performance. Please be informed that the examples are made for easy understanding and simple showing of one aspect of programming. Most of the examples are not optimized for maximum throughput or repetition rates.

Microsoft Visual C++ 2005 and newer 64 Bit

Depending on your version of the Visual Studio suite it may be necessary to install some additional 64 bit components (SDK) on your system. Please follow the instructions found on the MSDN for further information.

Include Driver

The driver files can be directly included in Microsoft C++ by simply using the library file spcm_win64_msvcpp.lib that is delivered together with the drivers. The library file can be found on the CD in the path /examples/c_cpp/c_header. All functions described below are now available in your program.

Linux Gnu C/C++ 32/64 Bit

Include Driver

The interface of the linux drivers does not differ from the windows interface. Please include the "libspcm_linux.so" library in your makefile using the below shown "LIBS = -lspcm_linux" line, to have access to all driver functions. A makefile may look like this:

```

COMPILER = gcc
EXECUTABLE = test_prg
LIBS = -lspcm_linux

OBJECTS = test.o\
          test2.o

all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(COMPILER) $(CFLAGS) -o $(EXECUTABLE) $(LIBS) $(OBJECTS)

%.o: %.cpp
    $(COMPILER) $(CFLAGS) -o $*.o -c $*.cpp
  
```

Examples

The Gnu C/C++ examples share the source with the Visual C++ examples. Please see above chapter for a more detailed documentation of the examples. Each example directory contains a makefile for the Gnu C/C++ examples.

C++ for .NET

Please see the next chapter for more details on the .NET inclusion.

Other Windows C/C++ compilers 32 Bit

Include Driver

To access the driver using a compiler such as e.g. MinGW or Borland, the driver functions must be loaded from the 32 bit driver DLL. Most compilers offer special tools to generate a matching library (e.g. Borland offers the `implib` tool that generates a matching library out of the windows driver DLL). If such a tool is available it is recommended to use it. Otherwise the driver functions need to be loaded from the dll using standard Windows functions. There is one example in the example directory `/examples/c_cpp/dll_loading` that shows the process.

Example of function loading:

```
hDLL = LoadLibrary ("spcm_win32.dll"); // Load the 32 bit version of the Spcm driver
pfn_spcm_hOpen = (SPCM_HOPEN*) GetProcAddress (hDLL, "spcm_hOpen@4");
pfn_spcm_vClose = (SPCM_VCLOSE*) GetProcAddress (hDLL, "spcm_vClose@4");
```

Other Windows C/C++ compilers 64 Bit

Include Driver

To access the driver using a compiler such as e.g. MinGW or Borland, the driver functions must be loaded from the 64 bit the driver DLL. Most compilers offer special tools to generate a matching library (e.g. Borland offers the `implib` tool that generates a matching library out of the windows driver DLL). If such a tool is available it is recommended to use it. Otherwise the driver functions need to be loaded from the dll using standard Windows functions. There is one example in the example directory `/examples/c_cpp/dll_loading` that shows the process for 32 bit environments. The only line that needs to be modified is the one loading the DLL:

Example of function loading:

```
hDLL = LoadLibrary ("spcm_win64.dll"); // Modified: Load the 64 bit version of the Spcm driver here
pfn_spcm_hOpen = (SPCM_HOPEN*) GetProcAddress (hDLL, "spcm_hOpen");
pfn_spcm_vClose = (SPCM_VCLOSE*) GetProcAddress (hDLL, "spcm_vClose");
```

Driver functions

The driver contains seven main functions to access the hardware.

Own types used by our drivers

To simplify the use of the header files and our examples with different platforms and compilers and to avoid any implicit type conversions we decided to use our own type declarations. This allows us to use platform independent and universal examples and driver interfaces. If you do not stick to these declarations please be sure to use the same data type width. However it is strongly recommended that you use our defined type declarations to avoid any hard to find errors in your programs. If you're using the driver in an environment that is not natively supported by our examples and drivers please be sure to use a type declaration that represents a similar data width

Table 5: C/C++ type declarations for drivers and examples

Declaration	Type	Declaration	Type
int8	8 bit signed integer (range from -128 to +127)	uint8	8 bit unsigned integer (range from 0 to 255)
int16	16 bit signed integer (range from -32768 to 32767)	uint16	16 bit unsigned integer (range from 0 to 65535)
int32	32 bit signed integer (range from -2147483648 to 2147483647)	uint32	32 bit unsigned integer (range from 0 to 4294967295)
int64	64 bit signed integer (full range)	uint64	64 bit unsigned integer (full range)
drv_handle	handle to driver, implementation depends on operating system platform		

Notation of variables and functions

In our header files and examples we use a common and reliable form of notation for variables and functions. Each name also contains the type as a prefix. This notation form makes it easy to see implicit type conversions and minimizes programming errors that result from using incorrect types. Feel free to use this notation form for your programs also-

Table 6: C/C++ type naming convention throughout drivers and examples

Declaration	Notation	Declaration	Notation
int8	byName (byte)	uint8	cName (character)
int16	nName	uint16	wName (word)
int32	lName (long)	uint32	dwName (double word)
int64	llName (long long)	uint64	qwName (quad word)
int32*	plName (pointer to long)	char	szName (string with zero termination)

Function `spcm_hOpen`

This function initializes and opens an installed card supporting the new SPCm driver interface, which at the time of printing, are all cards of the M2i/M3i/M4i/M4x/M2p/M5i series and the related digitizerNETBOX/generatorNETBOX/hybridNETBOX devices. The function returns a handle that has to be used for driver access. If the card can't be found or the loading of the driver generated an error the function

returns a NULL. When calling this function all card specific installation parameters are read out from the hardware and stored within the driver. It is only possible to open one device by one software as concurrent hardware access may be very critical to system stability. As a result when trying to open the same device twice an error will be raised and the function returns NULL.

Function `spcm_hOpen (const char* szDeviceName):`

```
drv_handle _stdcall spcm_hOpen (        // tries to open the device and returns handle or error code
    const char* szDeviceName);          // name of the device to be opened
```

Under Linux the device name in the function call needs to be a valid device name. Please change the string according to the location of the device if you don't use the standard Linux device names. The driver is installed as default under `/dev/spcm0`, `/dev/spcm1` and so on. The kernel driver numbers the devices starting with 0.

Under Windows the only part of the device name that is used is the trailing number. The rest of the device name is ignored. Therefore to keep the examples simple we use the Linux notation in all our examples. The trailing number gives the index of the device to open. The Windows kernel driver numbers all devices that it finds on boot time starting with 0.

Example for local installed cards

```
drv_handle hDrv;                // returns the handle to the opened driver or NULL in case of error
hDrv = spcm_hOpen ("/dev/spcm0"); // open the first card (spcm0) and get a handle to this card
if (!hDrv)
    printf ("open of driver failed\n");
```

Example for digitizerNETBOX/generatorNETBOX and remote installed cards

```
drv_handle hDrv;                // returns the handle to the opened driver or NULL in case of error
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST0::INSTR");
if (!hDrv)
    printf ("open of driver failed\n");
```

If the function returns a NULL it is possible to read out the error description of the failed open function by simply passing this NULL to the error function. The error function is described in one of the next topics.

Function `spcm_vClose`

This function closes the driver and releases all allocated resources. After closing the driver handle it is not possible to access this driver any more. Be sure to close the driver if you don't need it any more to allow other programs to get access to this device.

Function `spcm_vClose:`

```
void _stdcall spcm_vClose (        // closes the device
    drv_handle hDevice);           // handle to an already opened device
```

Example:

```
spcm_vClose (hDrv);
```

Function `spcm_dwSetParam`

All hardware settings are based on software registers that can be set by one of the functions `spcm_dwSetParam`. These functions set a register to a defined value or execute a command. The board must first be initialized by the `spcm_hOpen` function. The parameter `IRegister` must have a valid software register constant as defined in `regs.h`. The available software registers for the driver are listed in the board specific part of the documentation below. The function returns a 32 bit error code if an error occurs. If no error occurs the function returns `ERR_OK`, what is zero.

Function `spcm_dwSetParam`

```

uint32 _stdcall spcm_dwSetParam_i32 ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be modified
    int32 lValue);                    // the value to be set

uint32 _stdcall spcm_dwSetParam_i64m ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be modified
    int32 lValueHigh,                 // upper 32 bit of the value. Containing the sign bit !
    uint32 dwValueLow);               // lower 32 bit of the value.

uint32 _stdcall spcm_dwSetParam_i64 ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be modified
    int64 llValue);                   // the value to be set

uint32 _stdcall spcm_dwSetParam_d64 ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be modified
    double dValue);                  // the value to be set

uint32 _stdcall spcm_dwSetParam_ptr ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be modified
    void* pvValue,                   // pointer for the return value
    unit64 qwLen);                   // length of the buffer behind the pvValue

```

The functions `spcm_dwSetParam_d64` and `spcm_dwSetParam_ptr` have been added with driver release V 7.00

Example:

```

if (spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, 16384) != ERR_OK)
    printf ("Error when setting memory size\n");

```

This example sets the memory size to 16 kSamples (16384). If an error occurred the example will show a short error message

Function `spcm_dwGetParam`

All hardware settings are based on software registers that can be read by one of the functions `spcm_dwGetParam`. These functions read an internal register or status information. The board must first be initialized by the `spcm_hOpen` function. The parameter `lRegister` must have a valid software register constant as defined in the `regs.h` file. The available software registers for the driver are listed in the board specific part of the documentation below. The function returns a 32 bit error code if an error occurs. If no error occurs the function returns `ERR_OK`, what is zero.

Function `spcm_dwGetParam`

```

uint32 _stdcall spcm_dwGetParam_i32 ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be read out
    int32* plValue);                  // pointer for the return value

uint32 _stdcall spcm_dwGetParam_i64m ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be read out
    int32* plValueHigh,               // pointer for the upper part of the return value
    uint32* pdwValueLow);             // pointer for the lower part of the return value

uint32 _stdcall spcm_dwGetParam_i64 ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be read out
    int64* pllValue);                 // pointer for the return value

uint32 _stdcall spcm_dwGetParam_d64 ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be modified
    double* dValue);                 // pointer for the return value

uint32 _stdcall spcm_dwGetParam_ptr ( // Return value is an error code
    drv_handle hDevice,               // handle to an already opened device
    int32 lRegister,                  // software register to be modified
    void* pvValue,                   // pointer for the return value
    unit64 qwLen);                   // length of the buffer behind the pvValue

```

The functions `spcm_dwGetParam_d64` and `spcm_dwGetParam_ptr` have been added with driver release V 7.00

Example:

```
int32 lSerialNumber;
spcm_dwGetParam_i32 (hDrv, SPC_PCISERIALNO, &lSerialNumber);
printf ("Your card has serial number: %05d\n", lSerialNumber);
```

The example reads out the serial number of the installed card and prints it. As the serial number is available under all circumstances there is no error checking when calling this function.

Different call types of spcm_dwSetParam and spcm_dwGetParam: i32, i64, i64m, d64

The four functions only differ in the type of the parameters that are used to call them. As some of the registers can exceed the 32 bit integer range (like memory size or post trigger) it is recommended to use the `_i64` function to access these registers. However as there are some programs or compilers that don't support 64 bit integer variables there are two functions that are limited to 32 bit integer variables. In case that you do not access registers that exceed 32 bit integer please use the `_i32` function. In case that you access a register which exceeds 64 bit value please use the `_i64m` calling convention. Inhere the 64 bit value is split into a low double word part and a high double word part. Please be sure to fill both parts with valid information.

As some registers need to be read/written in double precision and can't be read/written as integer values, two additional new functions for accessing double values have been added with the suffix `_d64`.

If accessing 64 bit registers with 32 bit functions the behaviour differs depending on the real value that is currently located in the register. Please have a look at this table to see the different reactions depending on the size of the register:

Table 7: Spectrum driver API functions overview and differentiation between 32 bit and 64 bit registers

Internal register	read/write	Function type	Behavior
32 bit register	read	<code>spcm_dwGetParam_i32</code>	value is returned as 32 bit integer in <code>pIValue</code>
32 bit register	read	<code>spcm_dwGetParam_i64</code>	value is returned as 64 bit integer in <code>pIValue</code>
32 bit register	read	<code>spcm_dwGetParam_i64m</code>	value is returned as 64 bit integer, the lower part in <code>pIValueLow</code> , the upper part in <code>pIValueHigh</code> . The upper part can be ignored as it's only a sign extension
32 bit register	read	<code>spcm_dwGetParam_d64</code>	value is returned as 64 bit double in <code>pDValue</code>
32 bit register	write	<code>spcm_dwSetParam_i32</code>	32 bit value can be directly written
32 bit register	write	<code>spcm_dwSetParam_i64</code>	64 bit value can be directly written, please be sure not to exceed the valid register value range
32 bit register	write	<code>spcm_dwSetParam_i64m</code>	32 bit value is written as <code>lIValueLow</code> , the value <code>lIValueHigh</code> needs to contain the sign extension of this value. In case of <code>lIValueLow</code> being a value ≥ 0 <code>lIValueHigh</code> can be 0, in case of <code>lIValueLow</code> being a value < 0 , <code>lIValueHigh</code> has to be -1.
32 bit register	write	<code>spcm_dwSetParam_d64</code>	32 bit value needs to be converted to double. Please make sure not to exceed the valid register range
64 bit register	read	<code>spcm_dwGetParam_i32</code>	If the internal register has a value that is inside the 32 bit integer range (-2G up to (2G - 1)) the value is returned normally. If the internal register exceeds this size an error code <code>ERR_EXCEEDSINT32</code> is returned. As an example: reading back the installed memory will work as long as this memory is < 2 GByte. If the installed memory is ≥ 2 GByte the function will return an error.
64 bit register	read	<code>spcm_dwGetParam_i64</code>	value is returned as 64 bit integer value in <code>pIValue</code> independent of the value of the internal register.
64 bit register	read	<code>spcm_dwGetParam_i64m</code>	the internal value is split into a low and a high part. As long as the internal value is within the 32 bit range, the low part <code>pIValueLow</code> contains the 32 bit value and the upper part <code>pIValueHigh</code> can be ignored. If the internal value exceeds the 32 bit range it is absolutely necessary to take both value parts into account.
64 bit register	read	<code>spcm_dwGetParam_d64</code>	value is returned as 64 bit double in <code>pDValue</code> . Please note that double values are limited to 2^{48} . Any larger value is not returned with full precision.
64 bit register	write	<code>spcm_dwSetParam_i32</code>	the value to be written is limited to 32 bit range. If a value higher than the 32 bit range should be written, one of the other function types need to be used.
64 bit register	write	<code>spcm_dwSetParam_i64</code>	the value has to be split into two parts. Be sure to fill the upper part <code>lIValueHigh</code> with the correct sign extension even if you only write a 32 bit value as the driver every time interprets both parts of the function call.
64 bit register	write	<code>spcm_dwSetParam_i64m</code>	the value can be written directly independent of the size.
64 bit register	write	<code>spcm_dwSetParam_d64</code>	the value need to be converted to double. Any value up to 2^{48} can be written directly. Larger values need to be written using the <code>_i64</code> function

Function spcm_dwGetContBuf

This function reads out the internal continuous memory buffer in bytes, in case one has been allocated. If no buffer has been allocated the function returns a size of zero and a NULL pointer. You may use this buffer for data transfers. As the buffer is continuously allocated in memory the data transfer will speed up by up to 15% - 25%, depending on your specific kind of card. Please see further details in the appendix of this manual.

```
uint32_stdcall spcm_dwGetContBuf_i64 ( // Return value is an error code
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType,                  // type of the buffer to read as listed above under SPCM_BUF_XXXX
    void** ppvDataBuffer,              // address of available data buffer
    uint64* pqwContBufLen);            // length of available continuous buffer

uint32_stdcall spcm_dwGetContBuf_i64m (// Return value is an error code
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType,                  // type of the buffer to read as listed above under SPCM_BUF_XXXX
    void** ppvDataBuffer,              // address of available data buffer
    uint32* pdwContBufLenH,            // high part of length of available continuous buffer
    uint32* pdwContBufLenL);           // low part of length of available continuous buffer
```



These functions have been added in driver version 1.36. The functions are not available in older driver versions.



These functions also only have effect on locally installed cards and are neither useful nor usable with any digitizerNETBOX or generatorNETBOX products, because no local kernel driver is involved in such a setup. For remote devices these functions will return a NULL pointer for the buffer and 0 Bytes in length.

Function `spcm_dwDefTransfer`

The `spcm_dwDefTransfer` function defines a buffer for a following data transfer. This function only defines the buffer, there is no data transfer performed when calling this function. Instead the data transfer is started with separate register commands that are documented in a later chapter. At this position there is also a detailed description of the function parameters.

Please make sure that all parameters of this function match. It is especially necessary that the buffer address is a valid address pointing to memory buffer that has at least the size that is defined in the function call. Please be informed that calling this function with non valid parameters may crash your system as these values are base for following DMA transfers.

The use of this function is described in greater detail in a later chapter.

Function `spcm_dwDefTransfer`

```
uint32_stdcall spcm_dwDefTransfer_i64m ( // Defines the transfer buffer by 2 x 32 bit unsigned integer
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType,                  // type of the buffer to define as listed above under SPCM_BUF_XXXX
    uint32 dwDirection,                // the transfer direction as defined above
    uint32 dwNotifySize,               // no. of bytes after which an event is sent (0=end of transfer)
    void* pvDataBuffer,               // pointer to the data buffer
    uint32 dwBrdOffsH,                // high part of offset in board memory (zero when using FIFO mode)
    uint32 dwBrdOffsL,                // low part of offset in board memory (zero when using FIFO mode)
    uint32 dwTransferLenH,             // high part of transfer buffer length
    uint32 dwTransferLenL);           // low part of transfer buffer length

uint32_stdcall spcm_dwDefTransfer_i64 ( // Defines the transfer buffer by using 64 bit unsigned integer values
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType,                  // type of the buffer to define as listed above under SPCM_BUF_XXXX
    uint32 dwDirection,                // the transfer direction as defined above
    uint32 dwNotifySize,               // no. of bytes after which an event is sent (0=end of transfer)
    void* pvDataBuffer,               // pointer to the data buffer
    uint64 qwBrdOffs,                 // offset for transfer in board memory (zero when using FIFO mode)
    uint64 qwTransferLen);            // buffer length
```

This function is available in two different formats as the `spcm_dwGetParam` and `spcm_dwSetParam` functions are. The background is the same. As long as you're using a compiler that supports 64 bit integer values please use the `_i64` function. Any other platform needs to use the `_i64m` function and split offset and length in two 32 bit words.

Example:

```
int16* pnBuffer = (int16*) pvAllocMemPageAligned (16384);
if (spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 0, (void*) pnBuffer, 0, 16384) != ERR_OK)
    printf ("DefTransfer failed\n");
```

The example defines a data buffer of 8 kSamples of 16 bit integer values = 16 kByte (16384 byte) for a transfer from card to PC memory. As notify size is set to 0 we only want to get an event when the transfer has finished.

Function `spcm_dwInvalidateBuf`

The invalidate buffer function is used to tell the driver that the buffer that has been set with `spcm_dwDefTransfer` call is no longer valid. It is necessary to use the same buffer type as the driver handles different buffers at the same time. Call this function if you want to delete the buffer memory after calling the `spcm_dwDefTransfer` function. If the buffer already has been transferred after calling `spcm_dwDefTransfer` it is not necessary to call this function. When calling `spcm_dwDefTransfer` any previously defined buffer of this type is automatically invalidated.

Function `spcm_dwInvalidateBuf`

```
uint32_stdcall spcm_dwInvalidateBuf ( // invalidate the transfer buffer
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType);                // type of the buffer to invalidate as
                                     // listed above under SPCM_BUF_XXXX
```

Function `spcm_dwGetErrorInfo`

The function returns complete error information on the last error that has occurred. The error handling itself is explained in a later chapter in greater detail. When calling this function please be sure to have a text buffer allocated that has at least `ERRORTXTLEN` length. The error text function returns a complete description of the error including the register/value combination that has raised the error and a short description of the error details. In addition it is possible to get back the error generating register/value for own error handling. If not needed the buffers for register/value can be left to NULL.

Note that the timeout event (`ERR_TIMEOUT`) is not counted as an error internally as it is not locking the driver but as a valid event. Therefore the `GetErrorInfo` function won't return the timeout event even if it had occurred in between. You can only recognize the `ERR_TIMEOUT` as a direct return value of the wait function that was called.



Function `spcm_dwGetErrorInfo`

```
// for reading errors that occur during hOpen(), leave the drv_handle parameter NULL

uint32_stdcall spcm_dwGetErrorInfo_i32 (
    drv_handle hDevice,           // handle to an already opened device
    uint32*    pdwErrorReg,       // address of the error register (can be NULL if not of interest)
    int32*     plErrorValue,      // address of the error value (can be NULL if not of interest)
    char       pszErrorTextBuffer[ERRORTEXTLEN]); // text buffer for text error

uint32_stdcall spcm_dwGetErrorInfo_i64 (
    drv_handle hDevice,           // handle to an already opened device
    uint32*    pdwErrorReg,       // address of the error register (can be NULL if not of interest)
    int64*     pllErrorValue,     // address of the error value (can be NULL if not of interest)
    char       pszErrorTextBuffer[ERRORTEXTLEN]); // text buffer for text error

uint32_stdcall spcm_dwGetErrorInfo_d64 (
    drv_handle hDevice,           // handle to an already opened device
    uint32*    pdwErrorReg,       // address of the error register (can be NULL if not of interest)
    double*    pdErrorValue,     // address of the error value (can be NULL if not of interest)
    char       pszErrorTextBuffer[ERRORTEXTLEN]); // text buffer for text error
```

The function `spcm_dwGetErrorInfo_i64` and `spcm_dwGetErrorInfo_d64` have been added with driver release V 7.00

Example:

```
char szErrorBuf[ERRORTEXTLEN];
if (spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, -1))
{
    spcm_dwGetErrorInfo_i64 (hDrv, NULL, NULL, szErrorBuf);
    printf ("Set of memsize failed with error message: %s\n", szErrorBuf);
}
```

Delphi (Pascal) Programming Interface

Driver interface

The driver interface is located in the sub-directory `d_header` and contains the following files. The files need to be included in the delphi project and have to be put into the „uses“ section of the source files that will access the driver. Please do not edit any of these files as they're regularly updated if new functions or registers have been included.

file spcm_win32.pas

The file contains the interface to the driver library and defines some needed constants and variable types. All functions of the delphi library are similar to the above explained standard driver functions:

```
// ----- device handling functions -----
function spcm_hOpen (strName: pchar): int32; stdcall; external 'spcm_win32.dll' name '_spcm_hOpen@4';
procedure spcm_vClose (hDevice: int32); stdcall; external 'spcm_win32.dll' name '_spcm_vClose@4';

function spcm_dwGetErrorInfo_i32 (hDevice: int32; var lErrorReg, lErrorValue: int32; strError: pchar): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwGetErrorInfo_i32@16';

function spcm_dwGetErrorInfo_i64 (hDevice: int32; var plErrorReg: int32; var pllErrorValue: int64; strError:
PAnsiChar): uint32; stdcall; external 'spcm_win32.dll' name '_spcm_dwGetErrorInfo_i64@16';

function spcm_dwGetErrorInfo_d64 (hDevice: int32; var plErrorReg: int32; var pdErrorValue: double; strError:
PAnsiChar): uint32; stdcall; external 'spcm_win32.dll' name '_spcm_dwGetErrorInfo_d64@16';

// ----- register access functions -----
function spcm_dwSetParam_i32 (hDevice, lRegister, lValue: int32): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwSetParam_i32@12';

function spcm_dwSetParam_i64 (hDevice, lRegister: int32; llValue: int64): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwSetParam_i64@16';

function spcm_dwSetParam_d64 (hDevice, lRegister: int32; dValue: double): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwSetParam_d64@16';

function spcm_dwGetParam_i32 (hDevice, lRegister: int32; var plValue: int32): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwGetParam_i32@12';

function spcm_dwGetParam_i64 (hDevice, lRegister: int32; var pllValue: int64): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwGetParam_i64@12';

function spcm_dwGetParam_d64 (hDevice, lRegister: int32; var pdValue: double): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwGetParam_d64@12';

// ----- data handling -----
function spcm_dwDefTransfer_i64 (hDevice, dwBufType, dwDirection, dwNotifySize: int32; pvDataBuffer: Pointer;
llBrdOffs, llTransferLen: int64): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwDefTransfer_i64@36';

function spcm_dwInvalidateBuf (hDevice, lBuffer: int32): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwInvalidateBuf@8';
```

The file also defines types used inside the driver and the examples. The types have similar names as used under C/C++ to keep the examples more simple to understand and allow a better comparison.

file spcm_win64.pas

The file contains the interface to the driver library and defines some needed constants and variable types. All functions of the delphi library are similar to the above explained standard driver functions:

```
// ----- device handling functions -----
function spcm_hOpen (strName: pchar): int32; stdcall; external 'spcm_win32.dll' name '_spcm_hOpen@4';
procedure spcm_vClose (hDevice: int32); stdcall; external 'spcm_win32.dll' name '_spcm_vClose@4';

function spcm_dwGetErrorInfo_i32 (hDevice: int32; var lErrorReg: lErrorValue: int32; strError: pchar): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwGetErrorInfo_i32@16';

function spcm_dwGetErrorInfo_i64 (hDevice: int32; var plErrorReg: int32; var pllErrorValue: int64; strError:
PAnsiChar): uint32; stdcall; external 'spcm_win32.dll' name '_spcm_dwGetErrorInfo_i64@16';

function spcm_dwGetErrorInfo_d64 (hDevice: int32; var plErrorReg: int32; var pdErrorValue: double; strError:
PAnsiChar): uint32; stdcall; external 'spcm_win32.dll' name '_spcm_dwGetErrorInfo_d64@16';

// ----- register access functions -----
function spcm_dwSetParam_i32 (hDevice, lRegister, lValue: int32): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwSetParam_i32@12';

function spcm_dwSetParam_i64 (hDevice, lRegister: int32; llValue: int64): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwSetParam_i64@16';

function spcm_dwSetParam_d64 (hDevice, lRegister: int32; dValue: double): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwSetParam_d64@16';

function spcm_dwGetParam_i32 (hDevice, lRegister: int32; var plValue: int32): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwGetParam_i32@12';

function spcm_dwGetParam_i64 (hDevice, lRegister: int32; var pllValue: int64): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwGetParam_i64@12';

function spcm_dwGetParam_d64 (hDevice, lRegister: int32; var pdValue: double): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwGetParam_d64@12';

// ----- data handling -----
function spcm_dwDefTransfer_i64 (hDevice, dwBufType, dwDirection, dwNotifySize: int32; pvDataBuffer: Pointer;
llBrdOffs, llTransferLen: int64): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwDefTransfer_i64@36';

function spcm_dwInvalidateBuf (hDevice, lBuffer: int32): uint32;
stdcall; external 'spcm_win32.dll' name '_spcm_dwInvalidateBuf@8';
```

file SpcRegs.pas

The SpcRegs.pas file defines all constants that are used for the driver. The constant names are the same names as used under the C/C++ examples. All constants names will be found throughout this hardware manual when certain aspects of the driver usage are explained. It is recommended to only use these constant names for better visibility of the programs:

```
const SPC_M2CMD                = 100;                { write a command }
const M2CMD_CARD_RESET        = $00000001;          { hardware reset   }
const M2CMD_CARD_WRITESETUP    = $00000002;          { write setup only }
const M2CMD_CARD_START         = $00000004;          { start of card (including writesetup) }
const M2CMD_CARD_ENABLETRIGGER = $00000008;          { enable trigger engine }
...
```

file SpcErr.pas

The SpcErr.pas file contains all error codes that may be returned by the driver.

Including the driver files

To use the driver function and all the defined constants it is necessary to include the files into the project as shown in the picture on the right. The project overview is taken from one of the examples delivered on the USB stick. Besides including the driver files in the project it is also necessary to include them in the uses section of the source files where functions or constants should be used:

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls,

  SpcRegs, SpcErr, spcm_win32;
```

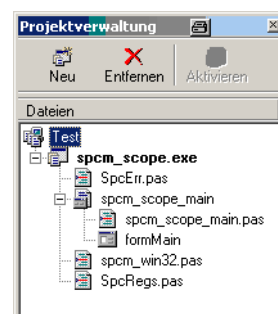


Image 36: Structure of the Delphi examples

Examples

Examples for Delphi can be found on the USB stick in the directory /examples/delphi. The directory contains the above mentioned delphi header files and a couple of universal examples, each of them working with a certain type of card. Please feel free to use these examples as a base for your programs and to modify them in any kind.

spcm_scope

The example implements a very simple scope program that makes single acquisitions on button pressing. A fixed setup is done inside the example. The spcm_scope example can be used with any analog data acquisition card from Spectrum. It covers cards with 1 byte per sample (8 bit resolution) as well as cards with 2 bytes per sample (12, 14 and 16 bit resolution)

The program shows the following steps:

- Initialization of a card and reading of card information like type, function and serial number
- Doing a simple card setup
- Performing the acquisition and waiting for the end interrupt
- Reading of data, re-scaling it and displaying waveform on screen

.NET programming languages

Library

For using the driver with a .NET based language Spectrum delivers a special library that encapsulates the driver in a .NET object. By adding this object to the project it is possible to access all driver functions and constants from within your .NET environment.

There is one small console based example for each supported .NET language that shows how to include the driver and how to access the cards. Please combine this example with the different standard examples to get the different card functionality.

Declaration

The driver access methods and also all the type, register and error declarations are combined in the object Spcm and are located in one of the two DLLs either SpcmDrv32.NET.dll or SpcmDrv64.NET.dll delivered with the .NET examples.



For simplicity, either file is simply called „SpcmDrv.NET.dll“ in the following passages and the actual file name must be replaced with either the 32bit or 64bit version according to your application.

Spectrum also delivers the source code of the DLLs as a C# project. These sources are located in the directory SpcmDrv.NET.

```
namespace Spcm
{
    public class Drv
    {
        [DllImport("spcm_win32.dll")]public static extern IntPtr spcm_hOpen (string szDeviceName);
        [DllImport("spcm_win32.dll")]public static extern void spcm_vClose (IntPtr hDevice);
        ...
        public class CardType
        {
            public const int TYP_M2I2020 = unchecked ((int)0x00032020);
            public const int TYP_M2I2021 = unchecked ((int)0x00032021);
            public const int TYP_M2I2025 = unchecked ((int)0x00032025);
            ...
        }
        public class Regs
        {
            public const int SPC_M2CMD = unchecked ((int)100);
            public const int M2CMD_CARD_RESET = unchecked ((int)0x00000001);
            public const int M2CMD_CARD_WRITESETUP = unchecked ((int)0x00000002);
            ...
        }
    }
}
```

Using C#

The SpcmDrv.NET.dll needs to be included within the Solution Explorer in the References section. Please use right mouse and select „AddReference“. After this all functions and constants of the driver object are available.

Please see the example in the directory CSharp as a start:

```
// ----- open card -----
hDevice = Drv.spcm_hOpen("/dev/spcm0");
if ((int)hDevice == 0)
{
    Console.WriteLine("Error: Could not open card\n");
    return 1;
}

// ----- get card type -----
dwErrorCode = Drv.spcm_dwGetParam_i32(hDevice, Regs.SPC_PCITYP, out lCardType);
dwErrorCode = Drv.spcm_dwGetParam_i32(hDevice, Regs.SPC_PCISERIALNR, out lSerialNumber);
```

Example for digitizerNETBOX/generatorNETBOX and remotely installed cards:

```
// ----- open remote card -----  
hDevice = Drv.spcm_hOpen("TCPIP::192.168.169.14::INST0::INSTR");
```

Using Managed C++/CLI

The SpcmDrv.NET.dll needs to be included within the project options. Please select „Project“ - „Properties“ - „References“ and finally „Add new Reference“. After this all functions and constants of the driver object are available.

Please see the example in the directory CppCLR as a start:

```
// ----- open card -----
hDevice = Drv::spcm_hOpen("/dev/spcm0");
if ((int)hDevice == 0)
{
    Console::WriteLine("Error: Could not open card\n");
    return 1;
}

// ----- get card type -----
dwErrorCode = Drv::spcm_dwGetParam_i32(hDevice, Regs::SPC_PCITYP, lCardType);
dwErrorCode = Drv::spcm_dwGetParam_i32(hDevice, Regs::SPC_PCISERIALNR, lSerialNumber);
```

Example for digitizerNETBOX/generatorNETBOX and remotely installed cards:

```
// ----- open remote card -----
hDevice = Drv::spcm_hOpen("TCPIP::192.168.169.14::INST0::INSTR");
```

Using VB.NET

The SpcmDrv.NET.dll needs to be included within the project options. Please select „Project“ - „Properties“ - „References“ and finally „Add new Reference“. After this all functions and constants of the driver object are available.

Please see the example in the directory VB.NET as a start:

```
' ----- open card -----
hDevice = Drv.spcm_hOpen("/dev/spcm0")

If (hDevice = 0) Then
    Console.WriteLine("Error: Could not open card\n")
Else

    ' ----- get card type -----
    dwError = Drv.spcm_dwGetParam_i32(hDevice, Regs.SPC_PCITYP, lCardType)
    dwError = Drv.spcm_dwGetParam_i32(hDevice, Regs.SPC_PCISERIALNR, lSerialNumber)
```

Example for digitizerNETBOX/generatorNETBOX and remotely installed cards:

```
' ----- open remote card -----
hDevice = Drv.spcm_hOpen("TCPIP::192.168.169.14::INST0::INSTR")
```

Using J#

The SpcmDrv.NET.dll needs to be included within the Solution Explorer in the References section. Please use right mouse and select „AddReference“. After this all functions and constants of the driver object are available.

Please see the example in the directory JSharp as a start:

```
// ----- open card -----
hDevice = Drv.spcm_hOpen("/dev/spcm0");

if (hDevice.ToInt32() == 0)
    System.out.println("Error: Could not open card\n");
else
{
    // ----- get card type -----
    dwErrorCode = Drv.spcm_dwGetParam_i32(hDevice, Regs.SPC_PCITYP, lCardType);
    dwErrorCode = Drv.spcm_dwGetParam_i32(hDevice, Regs.SPC_PCISERIALNR, lSerialNumber);
```

Example for digitizerNETBOX/generatorNETBOX and remotely installed cards:

```
' ----- open remote card -----
hDevice = Drv.spcm_hOpen("TCPIP::192.168.169.14::INST0::INSTR")
```

Python Programming Interface and Examples

Driver interface

The driver interface contains the following files. The files need to be included in the python project. Please do not edit any of these files as they are regularly updated if new functions or registers have been included. To use pyspcm you need either python 2 (2.4, 2.6 or 2.7) or python 3 (3.x) and ctypes, which is included in python 2.6 and newer and needs to be installed separately for Python 2.4.

file pyspcm.py

The file contains the interface to the driver library and defines some needed constants. All functions of the python library are similar to the above explained standard driver functions and use ctypes as input and return parameters:

```
# ----- Windows -----
# Load DLL into memory.

# use windll because all driver access functions use _stdcall calling convention under windows
if (bIs64Bit == 1):
    spcmDll = windll.LoadLibrary ("spcm_win64.dll")
else:
    spcmDll = windll.LoadLibrary ("spcm_win32.dll")

# load spcm_hOpen
if (bIs64Bit):
    spcm_hOpen = getattr(spcmDll, "spcm_hOpen")
else:
    spcm_hOpen = getattr(spcmDll, "_spcm_hOpen@4")
spcm_hOpen.argtype = [c_char_p]
spcm_hOpen.restype = drv_handle

# load spcm_vClose
if (bIs64Bit):
    spcm_vClose = getattr(spcmDll, "spcm_vClose")
else:
    spcm_vClose = getattr(spcmDll, "_spcm_vClose@4")
spcm_vClose.argtype = [drv_handle]
spcm_vClose.restype = None

# load spcm_dwGetErrorInfo_i32
if (bIs64Bit):
    spcm_dwGetErrorInfo_i32 = getattr(spcmDll, "spcm_dwGetErrorInfo_i32")
else:
    spcm_dwGetErrorInfo_i32 = getattr(spcmDll, "_spcm_dwGetErrorInfo_i32@16")
spcm_dwGetErrorInfo_i32.argtype = [drv_handle, uptr32, ptr32, c_char_p]
spcm_dwGetErrorInfo_i32.restype = uint32

...
```

file regs.py

The regs.py file defines all constants that are used for the driver. The constant names are the same names compared to the C/C++ examples. All constant names will be found throughout this hardware manual when certain aspects of the driver usage are explained. It is recommended to only use these constant names for better readability of the programs:

```
SPC_M2CMD = 1001                # write a command
M2CMD_CARD_RESET = 0x000000011  # hardware reset
M2CMD_CARD_WRITESETUP = 0x000000021 # write setup only
M2CMD_CARD_START = 0x000000041    # start of card (including writesetup)
M2CMD_CARD_ENABLETRIGGER = 0x000000081 # enable trigger engine
...
```

file spcerr.py

The spcerr.py file contains all error codes that may be returned by the driver.

Examples

Examples for Python can be found on the USB stick in the directory /examples/python. The directory contains the above mentioned header files and some examples, each of them working with a certain type of card. Please feel free to use these examples as a base for your programs and to modify them in any kind.

When allocating the buffer for DMA transfers, use the following function to get a mutable character buffer:
ctypes.create_string_buffer(init_or_size[, size])



Java Programming Interface and Examples

Driver interface

The driver interface contains the following Java files (classes). The files need to be included in your Java project. Please do not edit any of these files as they are regularly updated if new functions or registers have been included. The driver interface uses the Java Native Access (JNA) library.

This library is licensed under the LGPL (<https://www.gnu.org/licenses/lgpl-3.0.en.html>) and has also to be included to your Java project.

To download the latest jna.jar package and to get more information about the JNA project please check the projects GitHub page under: <https://github.com/java-native-access/jna>

The following files can be found in the „SpcmDrv“ folder of your Java examples install path.

SpcmDrv32.java / SpcmDrv64.java

The files contain the interface to the driver library and defines some needed constants. All functions of the driver interface are similar to the above explained standard driver functions. Use the SpcmDrv32.java for 32 bit and the SpcmDrv64.java for 64 bit projects:

```
...

public interface SpcmWin64 extends StdCallLibrary {

    SpcmWin64 INSTANCE = (SpcmWin64)Native.loadLibrary ("spcm_win64", SpcmWin64.class);

    long spcm_hOpen (String sDeviceName);
    void spcm_vClose (long hDevice);
    int spcm_dwSetParam_i64 (long hDevice, int lRegister, long llValue);
    int spcm_dwGetParam_i64 (long hDevice, int lRegister, LongByReference pllValue);
    int spcm_dwSetParam_ptr (long hDevice, int lRegister, Pointer pValue, long llLen);
    int spcm_dwGetParam_ptr (long hDevice, int lRegister, Pointer pValue, long llLen);
    int spcm_dwSetParam_d64 (int hDevice, int lRegister, double dValue);
    int spcm_dwGetParam_d64 (int hDevice, int lRegister, DoubleByReference pdValue);
    int spcm_dwDefTransfer_i64 (long hDevice, int lBufType, int lDirection, int lNotifySize, Pointer pDataBuffer,
    long llBrdOffs, long llTransferLen);

    int spcm_dwInvalidateBuf (long hDevice, int lBufType);

    int spcm_dwGetErrorInfo_i32 (long hDevice, IntByReference plErrorReg, IntByReference plErrorValue, Pointer sErrorTextBuffer);

    int spcm_dwGetErrorInfo_i64 (long hDevice, IntByReference plErrorReg, LongByReference pllErrorValue, Pointer sErrorTextBuffer);

    int spcm_dwGetErrorInfo_d64 (long hDevice, IntByReference plErrorReg, DoubleByReference pdErrorValue, Pointer sErrorTextBuffer);
}
...
```

SpcmRegs.java

The SpcmRegs class defines all constants that are used for the driver. The constants names are the same names compared to the C/C++ examples. All constant names will be found throughout this hardware manual when certain aspects of the driver usage are explained. It is recommended to only use these constant names for better readability of the programs:

```
...

public static final int SPC_M2CMD = 100;
public static final int M2CMD_CARD_RESET = 0x00000001;
public static final int M2CMD_CARD_WRITESETUP = 0x00000002;
public static final int M2CMD_CARD_START = 0x00000004;
public static final int M2CMD_CARD_ENABLETRIGGER = 0x00000008;
...
```

SpcmErrors.java

The SpcmErrors class contains all error codes that may be returned by the driver.

Examples

Examples for Java can be found on the USB stick in the directory /examples/java. The directory contains the above mentioned header files and some examples, each of them working with a certain type of card. Please feel free to use these examples as a base for your programs and to modify them in any kind.

Julia Programming Interface and Examples

Driver interface

The driver interface contains the following files. The files need to be included in the julia project. Please do not edit any of these files as they are regularly updated if new functions or registers have been included.

file `spcm_drv.jl`

The file contains the interface to the driver library and defines some needed constants. All functions of the Julia library are similar to the above explained standard driver functions.

```
hDevice::Int64 = spcm_hOpen(sDeviceName::String)
Cvoid spcm_vClose(hDevice::Int64)

dwErr::UInt32, lValue::Int32 = spcm_dwGetParam_i32(hDevice::Int64, lRegister::Int32)
dwErr::UInt32, llValue::Int64 = spcm_dwGetParam_i64(hDevice::Int64, lRegister::Int32)
dwErr::UInt32, dValue::Float64 = spcm_dwGetParam_d64(hDevice::Int64, lRegister::Int32)

dwErr::UInt32 = spcm_dwSetParam_i32(hDevice::Int64, lRegister::Int32, lValue::Int32)
dwErr::UInt32 = spcm_dwSetParam_i64(hDevice::Int64, lRegister::Int32, llValue::Int64)
dwErr::UInt32 = spcm_dwSetParam_d64(hDevice::Int64, lRegister::Int32, dValue::Float64)

dwErr::UInt32 = spcm_dwDefTransfer_i64(hDevice::Int64, lBufType::Int32, lDirection::Int32,
                                     dwNotifySize::UInt32, pDataBuffer::Array{Int16,1},
                                     qwBrdOffs::UInt64, qwTransferLen::UInt64)

dwErr::UInt32 = spcm_dwDefTransfer_i64(hDevice::Int64, lBufType::Int32, lDirection::Int32,
                                     dwNotifySize::UInt32, pDataBuffer::Array{Int8,1},
                                     qwBrdOffs::UInt64, qwTransferLen::UInt64)

dwErr::UInt32 = spcm_dwInvalidateBuf(hDevice::Int64, lBufType::Int32)

dwErr::UInt32, dwErrReg::UInt32, lErrVal::Int32, sErrText::String = spcm_dwGetErrorInfo_i32(hDevice::Int64)
dwErr::UInt32, dwErrReg::UInt32, llErrVal::Int64, sErrText::String = spcm_dwGetErrorInfo_i64(hDevice::Int64)
dwErr::UInt32, dwErrReg::UInt32, dErrVal::Float64, sErrText::String = spcm_dwGetErrorInfo_d64(hDevice::Int64)
```

file `regs.jl`

The `regs.jl` file defines all constants that are used for the driver. The constant names are the same names compared to the C/C++ examples. All constant names will be found throughout this hardware manual when certain aspects of the driver usage are explained. It is recommended to only use these constant names for better readability of the programs:

```
const SPC_M2CMD = Int32(100) # write a command
const M2CMD_CARD_RESET = Int32(1) # 0x00000001 # hardware reset
const M2CMD_CARD_WRITESETUP = Int32(2) # 0x00000002 # write setup only
const M2CMD_CARD_START = Int32(4) # 0x00000004 # start of card (including writesetup)
const M2CMD_CARD_ENABLETRIGGER = Int32(8) # 0x00000008 # enable trigger engine
# ...
```

file `spcerr.jl`

The `spcerr.jl` file contains all error codes that may be returned by the driver.

Examples

Examples for Julia can be found on USB-Stick in the directory `/examples/julia`. The directory contains the above mentioned include files and some examples, each of them working with a certain type of card. Please feel free to use these examples as a base for your programs and to modify them in any kind.

LabVIEW driver and examples

A full set of drivers and examples is available for LabVIEW for Windows. LabVIEW for Linux is currently not supported. The LabVIEW drivers have their own manual. The LabVIEW drivers, examples and the manual are found on the USB stick that has been included in the delivery. The latest version is also available on our webpage www.spectrum-instrumentation.com

Please follow the description in the LabVIEW manual for installation and useage of the LabVIEW drivers for this card.

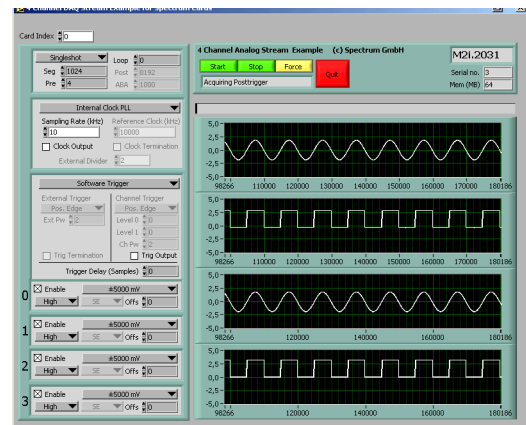


Image 37: LabVIEW driver oscilloscope example

MATLAB driver and examples

A full set of drivers and examples is available for Mathworks MATLAB for Windows (32 bit and 64 bit versions) and also for MATLAB for Linux (64 bit version). There is no additional toolbox needed to run the MATLAB examples and drivers.

The MATLAB drivers have their own manual. The MATLAB drivers, examples and the manual are found on the USB stick that has been included in the delivery. The latest version is also available on our webpage www.spectrum-instrumentation.com

Please follow the description in the MATLAB manual for installation and useage of the MATLAB drivers for this card.

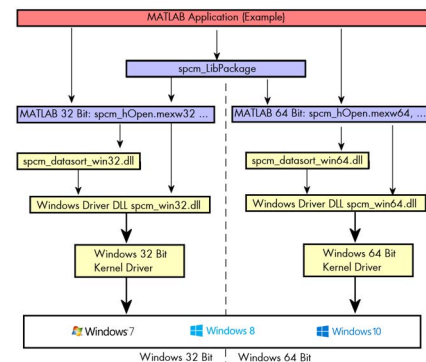


Image 38: Spectrum MATLAB driver structure

SCAPP – CUDA GPU based data processing

Spectrum's CUDA Access for Parallel Processing

Modern GPUs (Graphic Processing Units) are designed to handle a large number of parallel operations. While a CPU offers only a few cores for parallel calculations, a GPU can offer thousands of cores. This computing capabilities can be used for calculations using the Nvidia CUDA interface.

Since bus bandwidth and CPU power are often a bottleneck in calculations, CUDA Remote Direct

Memory Access (RDMA) can be used to directly transfer data from/to a Spectrum Digitizer/Generator to/from a GPU card for processing, thus avoiding the transfer of raw data to the host memory and benefiting from the computational power of the GPU.

For applications requiring high performance signal and data processing Spectrum offers SCAPP (Spectrum's CUDA Access for Parallel Processing).

The SCAPP SDK allows a direct link between Spectrum digitizers or generators and CUDA based GPU cards. Once data is available to the GPU, users can harness the processing power of the GPU's massive number of processing cores and large, ultra-high-speed GPU memory. SCAPP uses an RDMA (Linux only) process to send data at the digitizers full PCIe transfer speed to the GPU card. The SDK includes a set of examples for interaction between the digitizer or generator and the GPU card and another set of CUDA parallel processing examples with easy building blocks for basic functions like filtering, averaging, data de-multiplexing, data conversion or FFT. All the software is based on C/C++ and can easily be implemented, expanded and modified with normal programming skills.

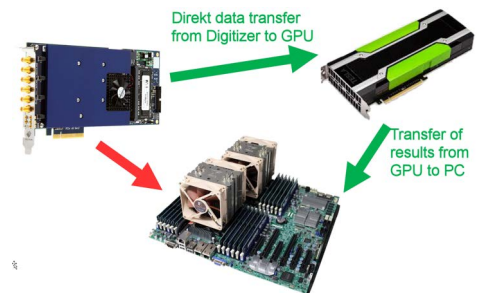


Image 39: GPU usage with SCAPP SDK: data transfer options

Please follow the description in the SCAPP manual for installation and usage of the SCAPP drivers for this card.

Programming the Board

Overview

The following chapters show you in detail how to program the different aspects of the board. For every topic there's a small example. For the examples we focused on Visual C++. However as shown in the last chapter the differences in programming the board under different programming languages are marginal. This manual describes the programming of the whole hardware family. Some of the topics are similar for all board versions. But some differ a little bit from type to type. Please check the given tables for these topics and examine carefully which settings are valid for your special kind of board.

Register tables

The programming of the boards is totally software register based. All software registers are described in the following form:

The name of the software register as found in the regs.h file. These Mnemonics should be used to increase readability.	The decimal value of the software register. Also found in the regs.h file. This value must be used with all programs or compilers that cannot use the header file directly.	Describes whether the register can be read (r) and/or written (w).	Short description of the functionality of the register. A more detailed description is found above or below the register tables.

Table 8: Spectrum API: Command register and basic commands

Register	Value	Direction	Description
SPC_M2CMD	100	w	Command register of the board.
M2CMD_CARD_START	4h		Starts the board with the current register settings.
M2CMD_CARD_STOP	40h		Stops the board manually.

Any constants that can be used to program the register directly are shown inserted beneath the register table.	The decimal or hexadecimal value of the constant, also found in the regs.h file. Hexadecimal values are indicated with an „h“ at the end. This value must be used with all programs or compilers that cannot use the header file directly.	Short description of the use of this constant.
--	--	--

If no constants are given below the register table, the dedicated register is used as a switch. All such registers are activated if written with a "1" and deactivated if written with a "0".



Programming examples

In this manual a lot of programming examples are used to give you an impression on how the actual mentioned registers can be set within your own program. All of the examples are located in a separated colored box to indicate the example and to make it easier to differ it from the describing text.

All of the examples mentioned throughout the manual are written in C/C++ and can be used with any C/C++ compiler for Windows or Linux.

Complete C/C++ Example

```
#include "../c_header/dlltyp.h"
#include "../c_header/regs.h"
#include "../c_header/spcm_drv.h"

#include <stdio.h>

int main()
{
    drv_handle hDrv; // the handle of the device
    int32 lCardType; // a place to store card information

    hDrv = spcm_hOpen ("/dev/spcm0"); // Opens the board and gets a handle
    if (!hDrv) // check whether we can access the card
        return -1;

    spcm_dwGetParam_i32 (hDrv, SPC_PCITYP, &lCardType); // simple command, read out of card type
    printf ("Found card M2i/M3i/M4i/M4x/M2p/M5i.%04x in the system\n", lCardType & TYP_VERSIONMASK);
    spcm_vClose (hDrv);

    return 0;
}
```

Initialization

Before using the card it is necessary to open the kernel device to access the hardware. It is only possible to use every device exclusively using the handle that is obtained when opening the device. Opening the same device twice will only generate an error code. After ending the driver use the device has to be closed again to allow later re-opening. Open and close of driver is done using the `spcm_hOpen` and `spcm_vClose` function as described in the "Driver Functions" chapter before.

Open/Close Example

```
drv_handle hDrv; // the handle of the device

hDrv = spcm_hOpen ("/dev/spcm0"); // Opens the board and gets a handle
if (!hDrv) // check whether we can access the card
{
    printf "Open failed\n";
    return -1;
}

... do any work with the driver

spcm_vClose (hDrv);
return 0;
```

Initialization of Remote Products

The only step that is different when accessing remotely controlled cards or digitizerNETBOXes is the initialization of the driver. Instead of the local handle one has to open the VISA string that is returned by the discovery function. Alternatively it is also possible to access the card directly without discovery function if the IP address of the device is known.

```
drv_handle hDrv; // the handle of the device

hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INSTR"); // Opens the remote board and gets a handle
if (!hDrv) // check whether we can access the card
{
    printf "Open of remote card failed\n";
    return -1;
}

...
```

Multiple cards are opened by indexing the remote card number:

```
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INSTR"); // Opens the remote board #0
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST0::INSTR"); // or alternatively
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST0::INSTR"); // Opens the remote board #0
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST1::INSTR"); // all other boards require an index:
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST2::INSTR"); // Opens the remote board #1
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST2::INSTR"); // Opens the remote board #2
```

Error handling

If one action caused an error in the driver this error and the register and value where it occurs will be saved.

The driver is then locked until the error is read out using the error function `spcm_dwGetErrorInfo_i32`. Any calls to other functions will just return the error code `ERR_LASTERR` showing that there is an error to be read out.



This error locking functionality will prevent the generation of unseen false commands and settings that may lead to totally unexpected behavior. For sure there are only errors locked that result on false commands or settings. Any error code that is generated to report a condition to the user won't lock the driver. As example the error code `ERR_TIMEOUT` showing that the a timeout in a wait function has occurred won't lock the driver and the user can simply react to this error code without reading the complete error function.

As a benefit from this error locking it is not necessary to check the error return of each function call but just checking the error function once at the end of all calls to see where an error occurred. The enhanced error function returns a complete error description that will lead to the call that produces the error.

Example for error checking at end using the error text from the driver:

```
char szErrorText[ERRORTXTLEN];

spcm_dwSetParam_i64 (hDrv, SPC_SAMPLERATE, 1000000);           // correct command
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, -345);                // faulty command
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, 1024);            // correct command
if (spcm_dwGetErrorInfo_i32 (hDrv, NULL, NULL, szErrorText) != ERR_OK) // check for an error
{
    printf (szErrorText);                                       // print the error text
    spcm_vClose (hDrv);                                         // close the driver
    exit (0);                                                    // and leave the program
}
```

This short program then would generate a printout as:

```
Error occurred at register SPC_MEMSIZE with value -345: value not allowed
```

All error codes are described in detail in the appendix. Please refer to this error description and the description of the software register to examine the cause for the error message.



Any of the parameters of the `spcm_dwGetErrorInfo_i32` function can be used to obtain detailed information on the error. If one is not interested in parts of this information it is possible to just pass a `NULL` (zero) to this variable like shown in the example. If one is not interested in the error text but wants to install its own error handler it may be interesting to just read out the error generating register and value.

Example for error checking with own (simple) error handler:

```
uint32 dwErrorReg;
int32 lErrorValue;
uint32 dwErrorCode;

spcm_dwSetParam_i64 (hDrv, SPC_SAMPLERATE, 1000000);           // correct command
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, -345);                // faulty command
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, 1024);            // correct command
dwErrorCode = spcm_dwGetErrorInfo_i32 (hDrv, &dwErrorReg, &lErrorValue, NULL); // check for an error
if (dwErrorCode)
{
    printf ("Errorcode: %d in register %d at value %d\n", lErrorCode, dwErrorReg, lErrorValue);
    spcm_vClose (hDrv);                                         // close the driver
    exit (0);                                                    // and leave the program
}
```

Gathering information from the card

When opening the card the driver library internally reads out a lot of information from the on-board eeprom. The driver also offers additional information on hardware details. All of this information can be read out and used for programming and documentation. This chapter will show all general information that is offered by the driver. There is also some more information on certain parts of the card, like clock machine or trigger machine, that is described in detail in the documentation of that part of the card.

All information can be read out using one of the `spcm_dwGetParam` functions. Please stick to the "Driver Functions" chapter for more details on this function.

Card type

The card type information returns the specific card type that is found under this device. When using multiple cards in one system it is highly recommended to read out this register first to examine the ordering of cards. Please don't rely on the card ordering as this is based on the BIOS, the bus connections and the operating system.

Table 9: Spectrum API: Card Type Register

Register	Value	Direction	Description
SPC_PCITYP	2000	read	Type of board as listed in the table below.

The SPC_PCITYP register can be used to read the numeric card type as well as a full name of the card using the `spcm_dwGetParam_ptr` function:

```
// read out the numeric card type as shown in the list below
spcm_dwGetParam_i32 (hDrv, SPC_PCITYP, &lCardType);

// read out the official name of the card
char acCardType[20] = {};
spcm_dwGetParam_ptr (hCard, SPC_PCITYP, acCardType, sizeof (acCardType));

// printout both information:
printf ("Found: %s (decimal: %d)\n", acCardType, lCardType);
```

One of the following values is returned, when reading this register. Each card has its own card type constant defined in `regs.h`. Please note that when reading the card information as a hex value, the lower word shows the digits of the card name while the upper word is a indication for the used bus type.

Table 10: Spectrum API: list of card type codes for M4i.77xx series

Card type	Card type as defined in regs.h	Value hexadecimal	Value decimal	Card type	Card type as defined in regs.h	Value hexadecimal	Value decimal
M4i.7710-x8	TYP_M4I7710_X8	77710h	489232				
M4i.7720-x8	TYP_M4I7720_X8	77720h	489248	M4i.7725-x8	TYP_M4I7725_X8	77725h	489253
M4i.7730-x8	TYP_M4I7730_X8	77730h	489264	M4i.7735-x8	TYP_M4I7735_X8	77735h	489269

Hardware and PCB version

Since all of the boards from Spectrum are modular boards, they consist of one base board and one piggy-back front-end module and eventually of an extension module like the star-hub. Each of these three kinds of hardware has its own version register. Normally you do not need this information but if you have a support question, please provide the revision together with it.

Table 11: Spectrum API: hardware and PCB version register overview

Register	Value	Direction	Description
SPC_PCIVERSION	2010	read	Base card version: the upper 16 bit show the hardware version, the lower 16 bit show the firmware version.
SPC_BASEPCBVERSION	2014	read	Base card PCB version: the lower 16 bit are divided into two 8 bit values containing pre/post decimal point version information. For example a lower 16 bit value of 0106h represents a PCB version V1.6. The upper 16 bit are always zero.
SPC_PCIMODULEVERSION	2012	read	Module version: the upper 16 bit show the hardware version, the lower 16 bit show the firmware version.
SPC_MODULEPCBVERSION	2015	read	Module PCB version: the lower 16 bit are divided into two 8 bit values containing pre/post decimal point version information. For example a lower 16 bit value of 0106h represents a PCB version V1.6. The upper 16 bit are always zero.

If your board has an additional piggy-back extension module mounted you can get the hardware version with the following register.

Table 12: Spectrum API: extension module hardware and PCB version register

Register	Value	Direction	Description
SPC_PCIEXTVERSION	2011	read	Extension module version: the upper 16 bit show the hardware version, the lower 16 bit show the firmware version.
SPC_EXTPCBVERSION	2017	read	Extension module PCB version: the lower 16 bit are divided into two 8 bit values containing pre/post decimal point version information. For example a lower 16 bit value of 0106h represents a PCB version V1.6. The upper 16 bit are always zero.

Reading currently used PXI slot No. (M4x only)

For the PXIe cards of the M4x.xxxx series it is possible to read out the current slot number, in which the card is installed within the chassis:

Table 13: Spectrum API: register for reading back the PXIe card slot number

Register	Value	Direction	Description
SPC_PXIHW_SLOTNO	2055	read	Returns the currently used slot number of the chassis.

Firmware versions

All the cards from Spectrum typically contain multiple programmable devices such as FPGAs, CPLDs and the like. Each of these have their own dedicated firmware version. This version information is readable for each device through the various version registers. Normally you do not need this information but if you have a support question, please provide us with this information. Please note that number of devices and hence the readable firmware information is card series dependent:

Table 14: Spectrum API: Register overview of firmware versions

Register	Value	Direction	Description	Available for					
				M2i	M3i	M4i	M4x	M2p	M5i
SPCM_FW_CTRL	210000	read	Main control FPGA version: the upper 16 bit show the firmware type, the lower 16 bit show the firmware version. For the standard release firmware, the type has always a value of 1.	X	X	X	X	X	X
SPCM_FW_CTRL_GOLDEN	210001	read	Main control FPGA golden version: the upper 16 bit show the firmware type, the lower 16 bit show the firmware version. For the golden (recovery) firmware, the type has always a value of 2.	—	—	X	X	X	X
SPCM_FW_CLOCK	210010	read	Clock distribution version: the upper 16 bit show the firmware type, the lower 16 bit show the firmware version. For the standard release firmware, the type has always a value of 1.	X	—	—	—	—	—
SPCM_FW_CONFIG	210020	read	Configuration controller version: the upper 16 bit show the firmware type, the lower 16 bit show the firmware version. For the standard release firmware, the type has always a value of 1.	X	X	—	—	—	—
SPCM_FW_MODULEA	210030	read	Front-end module A version: the upper 16 bit show the firmware type, the lower 16 bit show the firmware version. For the standard release firmware, the type has always a value of 1.	X	X	X	X	X	—
SPCM_FW_MODULEB	210031	read	Front-end module B version: the upper 16 bit show the firmware type, the lower 16 bit show the firmware version. For the standard release firmware, the type has always a value of 1. The version is zero if no second front-end module is installed on the card.	X	—	—	—	X	—
SPCM_FW_MODEXTRA	210050	read	Extension module (Star-Hub) version: the upper 16 bit show the firmware type, the lower 16 bit show the firmware version. For the standard release firmware, the type has always a value of 1. The version is zero if no extension module is installed on the card.	X	X	X	—	X	X
SPCM_FW_POWER	210060	read	Power controller version: the upper 16 bit show the firmware type, the lower 16 bit show the firmware version. For the standard release firmware, the type has always a value of 1.	—	—	X	X	X	X

Cards that do provide a golden recovery image for the main control FPGA, the currently booted firmware can additionally read out:

Table 15: Spectrum API: Register overview of reading current firmware

Register	Value	Direction	Description	Available for					
				M2i	M3i	M4i	M4x	M2p	M5i
SPCM_FW_CTRL_ACTIVE	210002	read	Cards that do provide a golden (recovery) firmware additionally have a register to read out the version information of the currently loaded firmware version string, to determine if it is standard or golden. The hexadecimal 32bit format is: TVVVUUUUh T: the currently booted type (1: standard, 2: golden) V: the version U: unused, in production versions always zero	—	—	X	X	X	X

Production date

This register informs you about the production date, which is returned as one 32 bit long word. The lower word is holding the information about the year, while the upper word informs about the week of the year.

Table 16: Spectrum API: production date register

Register	Value	Direction	Description
SPC_PCIDATE	2020	read	Production date: week in bits 31 to 16, year in bits 15 to 0

The following example shows how to read out a date and how to interpret the value:

```
spcm_dwGetParam_i32 (hDrv, SPC_PCIDATE, &lProdDate);
printf ("Production: week %d of year %d\n", (lProdDate >> 16) & 0xffff, lProdDate & 0xffff);
```

Last calibration date (analog cards only)

This register informs you about the date of the last factory calibration. When receiving a new card this date is similar to the delivery date when the production calibration is done. When returning the card to calibration this information is updated. This date is not updated when

the user does an on-board calibration. The date is returned as one 32 bit long word. The lower word is holding the information about the year, while the upper word informs about the week of the year.

Table 17: Spectrum API: calibration date register

Register	Value	Direction	Description
SPC_CALIBDATE	2025	read	Last calibration date: week in bit 31 to 16, year in bit 15 to 0

Serial number

This register holds the information about the serial number of the board. This number is unique and should always be sent together with a support question. Normally you use this information together with the register SPC_PCITYP to verify that multiple measurements are done with the exact same board.

Table 18: Spectrum API: hardware serial number register

Register	Value	Direction	Description
SPC_PCISERIALNO	2030	read	Serial number of the board

Maximum possible sampling rate

This register gives you the maximum possible sampling rate the board can run. The information provided here does not consider any restrictions in the maximum speed caused by special channel settings. For detailed information about the correlation between the maximum sampling rate and the number of activated channels please refer to the according chapter.

Table 19: Spectrum API: maximum sampling rate register

Register	Value	Direction	Description
SPC_PCISAMPLERATE	2100	read	Maximum sampling rate in Hz as a 64 bit integer value

Installed memory

This register returns the size of the installed on-board memory in bytes as a 64 bit integer value. If you want to know the amount of samples you can store, you must regard the size of one sample of your card. All 7 bit and 8 bit A/D and D/A cards use only one byte per sample, while all other A/D and D/A cards with 12, 14 and 16 bit resolution use two bytes to store one sample. All digital cards need one byte to store 8 data bits.

Table 20: Spectrum API: installed memory registers. 32 bit read is limited to a maximum of 1 GByte

Register	Value	Direction	Description
SPC_PCIMEMSIZE	2110	read_i32	Installed memory in bytes as a 32 bit integer value. Maximum return value will 1 GByte. If more memory is installed this function will return the error code ERR_EXCEEDINT32.
SPC_PCIMEMSIZE	2110	read_i64	Installed memory in bytes as a 64 bit integer value

The following example is written for a „two bytes“ per sample card (12, 14 or 16 bit board), on any 8 bit card memory in MSamples is similar to memory in MBytes.

```
spcm_dwGetParam_i64 (hDrv, SPC_PCIMEMSIZE, &llInstMemsize);
printf ("Memory on card: %d MBytes\n", (int32) (llInstMemsize /1024/1024));
printf ("          : %d MSamples\n", (int32) (llInstMemsize /1024/1024/2));
```

Installed features and options

The SPC_PCIFEATURES register informs you about the features, that are installed on the board. If you want to know about one option being installed or not, you need to read out the 32 bit value and mask the interesting bit. In the table below you will find every feature that may be installed on a M2i/M3i/M4i/M4x/M2p/M5i card. Please refer to the ordering information to see which of these features are available for your card series.

Table 21: Spectrum API: Feature Register and available feature flags

Register	Value	Direction	Description
SPC_PCIFEATURES	2120	read	PCI feature register. Holds the installed features and options as a bitfield. The read value must be masked out with one of the masks below to get information about one certain feature.
SPCM_FEAT_MULTI	1h		Is set if the feature Multiple Recording / Multiple Replay is available.
SPCM_FEAT_GATE	2h		Is set if the feature Gated Sampling / Gated Replay is available.
SPCM_FEAT_DIGITAL	4h		Is set if the feature Digital Inputs / Digital Outputs is available.
SPCM_FEAT_TIMESTAMP	8h		Is set if the feature Timestamp is available.
SPCM_FEAT_STARHUB6_EXTM	20h		Is set on the card, that carries the star-hub extension or piggy-back module for synchronizing up to 6 cards (M2p).
SPCM_FEAT_STARHUB8_EXTM	20h		Is set on the card, that carries the star-hub extension or piggy-back module for synchronizing up to 8 cards (M4i).
SPCM_FEAT_STARHUB4	20h		Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 4 cards (M3i).
SPCM_FEAT_STARHUB5	20h		Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 5 cards (M2i).
SPCM_FEAT_STARHUB16_EXTM	40h		Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 16 cards (M2p).
SPCM_FEAT_STARHUB8	40h		Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 8 cards (M3i and M5i).
SPCM_FEAT_STARHUB16	40h		Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 16 cards (M2i).
SPCM_FEAT_ABA	80h		Is set if the feature ABA mode is available.

SPCM_FEAT_BASEXIO	100h	Is set if the extra BaseXIO option is installed. The lines can be used for asynchronous digital I/O, extra trigger or timestamp reference signal input.
SPCM_FEAT_AMPLIFIER_10V	200h	Arbitrary Waveform Generators only: card has additional set of calibration values for amplifier card.
SPCM_FEAT_STARHUBSYSMASTER	400h	Is set in the card that carries a System Star-Hub Master card to connect multiple systems (M2i).
SPCM_FEAT_DIFFMODE	800h	M2i.30xx series only: card has option -diff installed for combining two SE channels to one differential channel.
SPCM_FEAT_SEQUENCE	1000h	Only available for output cards or I/O cards: Replay sequence mode available.
SPCM_FEAT_AMPMODULE_10V	2000h	Is set on the card that has a special amplifier module for mounted (M2i.60xx/61xx only).
SPCM_FEAT_STARHUBSYS_SLAVE	4000h	Is set in the card that carries a System Star-Hub Slave module to connect with System Star-Hub master systems (M2i).
SPCM_FEAT_NETBOX	8000h	The card is physically mounted within a digitizerNETBOX, generatorNETBOX or hybridNETBOX.
SPCM_FEAT_REMOTESERVER	10000h	Support for the Spectrum Remote Server option is installed on this card.
SPCM_FEAT_SCAPP	20000h	Support for the SCAPP option allowing CUDA RDMA access to supported graphics cards for GPU calculations (M5i, M4i and M2p).
SPCM_FEAT_DIG16_SMB	40000h	M2p: Set if option M2p.xxxx-DigSMB is installed, adding 16 additional digital I/Os via SMB connectors.
SPCM_FEAT_DIG16_FX2	80000h	M2p: Set if option M2p.xxxx-DigFX2 is installed, adding 16 additional digital I/Os via FX2 multipin connectors.
SPCM_FEAT_DIGITALBWFILTER	100000h	A digital (boxcar) bandwidth filter is available that can be globally enabled/disabled for all channels.
SPCM_FEAT_CUSTOMMOD_MASK	F0000000h	The upper 4 bit of the feature register is used to mark special custom modifications. This is only used if the card has been specially customized. Please refer to the extra documentation for the meaning of the custom modifications. (M2i/M3i). For M5i, M4i, M4x and M2p cards see „Custom modifications“ chapter instead.

The following example demonstrates how to read out the information about one feature.

```
spcm_dwGetParam_i32 (hDrv, SPC_PCIFEATURES, &lFeatures);
if (lFeatures & SPCM_FEAT_DIGITAL)
    printf("Option digital inputs/outputs is installed on your card");
```

The following example demonstrates how to read out the custom modification code.

```
spcm_dwGetParam_i32 (hDrv, SPC_PCIFEATURES, &lFeatures);
lCustomMod = (lFeatures >> 28) & 0xF;
if (lCustomMod != 0)
    printf("Custom modification no. %d is installed.", lCustomMod);
```

Installed extended Options and Features

Some cards (such as M5i/M4i/M4x/M2p cards) can have advanced features and options installed. This can be read out with the following register:

Table 22: Spectrum API: Extended feature register and available extended feature flags

Register	Value	Direction	Description
SPC_PCIXFEATURES	2121	read	PCI extended feature register. Holds the installed extended features and options as a bitfield. The read value must be masked out with one of the masks below to get information about one certain feature.
SPCM_FEAT_EXTFW_SEGSTAT	1h		Is set if the firmware option „Block Statistics“ is installed on the board, which allows certain statistics to be on-board calculated for data being recorded in segmented memory modes, such as Multiple Recording or ABA.
SPCM_FEAT_EXTFW_SEGAVERAGE	2h		Is set if the firmware option „Block Average“ is installed on the board, which allows on-board hardware averaging of data being recorded in segmented memory modes, such as Multiple Recording or ABA.
SPCM_FEAT_EXTFW_BOXCAR	4h		Is set if the firmware mode „Boxcar Average“ is supported in the installed firmware version.
SPCM_FEAT_EXTFW_PULSEGEN	8h		Is set if the firmware mode „Pulse Generator“ is installed on the board, which allows generation of pulses for output on the card's multi-purpose I/O lines (XIO).

Miscellaneous Card Information

Some more detailed card information, that might be useful for the application to know, can be read out with the following registers:

Table 23: Spectrum API: register overview of miscellaneous cards information

Register	Value	Direction	Description
SPC_MIINST_MODULES	1100	read	Number of the installed front-end modules on the card.
SPC_MIINST_CHPERMODULE	1110	read	Number of channels installed on one front-end module.
SPC_MIINST_BYTESPERSAMPLE	1120	read	Number of bytes used in memory by one sample.
SPC_MIINST_BITSPERSAMPLE	1125	read	Resolution of the samples in bits.
SPC_MIINST_MAXADCVALUE	1126	read	Decimal code of the full scale value.
SPC_MIINST_MINEXTCLOCK	1145	read	Minimum external clock that can be fed in for direct external clock (if available for card model).
SPC_MIINST_MAXEXTCLOCK	1146	read	Maximum external clock that can be fed in for direct external clock (if available for card model).
SPC_MIINST_MINEXTREFCLOCK	1148	read	Minimum external clock that can be fed in as a reference clock.
SPC_MIINST_MAXEXTREFCLOCK	1149	read	Maximum external clock that can be fed in as a reference clock.
SPC_MIINST_ISDEMOCARD	1175	read	Returns a value other than zero, if the card is a demo card.

Function type of the card

This register returns the basic type of the card:

Table 24: Spectrum API: register card function type and possible types

Register	Value	Direction	Description
SPC_FNCTYPE	2001	read	Gives information about what type of card it is.
SPCM_TYPE_AI	1h		Analog input card (analog acquisition; the M2i.4028 and M2i.4038 also return this value)
SPCM_TYPE_AO	2h		Analog output card (arbitrary waveform generators)
SPCM_TYPE_DI	4h		Digital input card (logic analyzer card)
SPCM_TYPE_DO	8h		Digital output card (pattern generators)
SPCM_TYPE_DIO	10h		Digital I/O (input/output) card, where the direction is software selectable.

Used type of driver

This register holds the information about the driver that is actually used to access the board. Although the driver interface doesn't differ between Windows and Linux systems it may be of interest for a universal program to know on which platform it is working.

Table 25: Spectrum API: register driver type information and possible driver types

Register	Value	Direction	Description
SPC_GETDRVTYPE	1220	read	Gives information about what type of driver is actually used
DRVTYP_UNIX32	1		Linux 32bit driver is used
DRVTYP_WDM32	4		Windows WDM 32bit driver is used (XP/Vista/Windows 7/8/10/11).
DRVTYP_WDM64	5		Windows WDM 64bit driver is used by 64bit application (XP64/Vista/Windows 7/8/10/11).
DRVTYP_WOW64	6		Windows WDM 64bit driver is used by 32bit application (XP64/Vista/Windows 7/8/10/11).
DRVTYP_UNIX64	7		Linux 64bit driver is used

Driver version

This register holds information about the currently installed driver library. As the drivers are permanently improved and maintained and new features are added user programs that rely on a new feature are requested to check the driver version whether this feature is installed.

Table 26: Spectrum API: driver version read register

Register	Value	Direction	Description
SPC_GETDRVVERSION	1200	read	Gives information about the driver library version

The resulting 32 bit value for the driver version consists of the three version number parts shown in the table below:

Driver Major Version	Driver Minor Version	Driver Build
8 Bit wide: bit 24 to bit 31	8 Bit wide: bit 16 to bit 23	16 Bit wide: bit 0 to bit 15

Kernel Driver version

This register informs about the actually used kernel driver. Windows users can also get this information from the device manager. Please refer to the „Driver Installation“ chapter. On Linux systems this information is also shown in the kernel message log at driver start time.

Table 27: Spectrum API: kernel driver version read register

Register	Value	Direction	Description
SPC_GETKERNELVERSION	1210	read	Gives information about the kernel driver version.

The resulting 32 bit value for the driver version consists of the three version number parts shown in the table below:

Driver Major Version	Driver Minor Version	Driver Build
8 Bit wide: bit 24 to bit 31	8 Bit wide: bit 16 to bit 23	16 Bit wide: bit 0 to bit 15

The following example demonstrates how to read out the kernel and library version and how to print them.

```
spcm_dwGetParam_i32 (hDrv, SPC_GETDRVVERSION, &lLibVersion);
spcm_dwGetParam_i32 (hDrv, SPC_GETKERNELVERSION, &lKernelVersion);
printf("Kernel V %d.%d build %d\n", lKernelVersion >> 24, (lKernelVersion >> 16) & 0xff, lKernelVersion & 0xffff);
printf("Library V %d.%d build %d\n", lLibVersion >> 24, (lLibVersion >> 16) & 0xff, lLibVersion & 0xffff);
```

This small program will generate an output like this:

```
Kernel V 1.11 build 817
Library V 1.1 build 854
```

Custom modifications

Since all of the boards from Spectrum are modular boards, they consist of one base board and one piggy-back front-end module and eventually of an extension module like the Star-Hub. Each of these three kinds of hardware has its own version register. Normally you do not need this information but if you have a support question, please provide the revision together with it.

Table 28: Spectrum API: custom modification register and different bitmasks to split the register in various hardware parts

Register	Value	Direction	Description
SPCM_CUSTOMMOD	3130	read	Dedicated feature register used to mark special custom modifications of the base card and/or the front-end module and/or the Star-Hub module. This is only used if the card has been specially customized. Please refer to the extra documentation for the meaning of the custom modifications. This register is supported for all M5i, M4i, M4x, M2p cards and all digitizerNETBOX, generatorNETBOX or hybridNETBOX based upon these series of cards.
SPCM_CUSTOMMOD_BASE_MASK	00000FFh		Mask for the custom modification of the base card.
SPCM_CUSTOMMOD_MODULE_MASK	0000FF00h		Mask for the custom modification of the front-end module(s).
SPCM_CUSTOMMOD_STARHUB_MASK	00FF0000h		Mask out custom modification of the Star-Hub module.

Reset

Every Spectrum card can be reset by software. Concerning the hardware, this reset is the same as the power-on reset when starting the host computer. In addition to the power-on reset, the reset command also brings all internal driver settings to a defined default state. A software reset is automatically performed, when the driver is first loaded after starting the host system.

Performing a board reset can be easily done by the related board command mentioned in the following table.

Table 29: Spectrum API: command register and reset command

Register	Value	Direction	Description
SPC_M2CMD	100	w	Command register of the board.
M2CMD_CARD_RESET	1h		A software and hardware reset is done for the board. All settings are set to the default values. The data in the board's on-board memory will be no longer valid. Any output signals like trigger or clock output will be disabled.

Digital Inputs

Channel Selection

One key setting that influences nearly all other possible settings is the channel enable register. An unique feature of the Spectrum boards is the possibility to program the data width. The complete on-board memory can then be used by samples with the actual data width.

This description shows you the channel enable register for the complete board family. However your specific board may have less input/output bits depending on the board type you purchased does not allow you to set the maximum number of bits shown here. The channel enable register is set as a 64 bit wide bitfield coping all possible channel enable combination.

Table 30: Spectrum API: channel enable register

Register			
SPC_CHENABLE	11000	r/w	Sets/reads out the channel enable mask for the up to 32 channels as a bitfield.

The following tables shows all allowed settings for the channel enable register for:

Number of activated digital channels			Value as hex	Activated channels
32	16	8		
X			00000000FFFFFFFFh	D31..D0
	X		000000000000FFFFh	D15..D0
		X	00000000000000FFh	D7..D0

Any channel activation mask that is not shown here is not valid. If programming an other channel activation, the driver will return with an error code ERR_VALUE.



Reading out the channel enable information can be done directly after setting it or later like this:

```
spcm_dwSetParam_i64 (hDrv, SPC_CHENABLE, 0x00000000FFFFFFFF);
spcm_dwGetParam_i64 (hDrv, SPC_CHENABLE, &llActivatedChannels);
spcm_dwGetParam_i32 (hDrv, SPC_CHCOUNT, &lChCount);

printf ("Activated channels bitmask is: 0x%16x\n", llActivatedChannels);
printf ("Number of activated channels with this bitmask: %d\n", lChCount);
```

Assuming that the 32 channels are available on your card the program will have the following output:

```
Activated channels bitmask is: 0x00000000FFFFFFFF
Number of activated channels with this bitmask: 32
```

Important note on channel selection

As some of the manuals passages are used in more than one hardware manual most of the registers and channel settings throughout this handbook are described for the maximum number of possible channels that are available on one card of the current series. There can be less channels on your actual type of board or bus-system. Please refer to the technical data section to get the actual number of available channels.



Read out of input features

The digital inputs of the different cards do have different features implemented, that can be read out to make the software more general. If you only operate one single card type in your software it is not necessary to read out these features.

Please note that the following table shows all input features settings that are available throughout all Spectrum digital I/O and logic analyzer cards. Some of these features are not installed on your specific hardware.

Table 31: Spectrum API: digital input features read register and register details

Register			
SPC_READFEATURES	3103	read	Returns a bit map with the available features of the digital input path. The possible return values are listed below.
SPCM_DI_TERM	00000001h	Programmable input termination available	
SPCM_DI_SE	00000002h	Input is single-ended. If available together with SPC_DI_DIFF: input type is software selectable.	
SPCM_DI_DIFF	00000004h	Input is differential. If available together with SPC_DI_SE: input type is software selectable.	
SPCM_DI_PROGTHRESHOLD	00000008h	Input offset programmable in per cent of input range	
SPCM_DI_HIGHIMP	00000010h	High impedance input is available. If available together with SPC_DI_TERM: input impedance is software selectable.	
SPCM_DI_LOWIMP	00000020h	Low impedance input is available. If available together with SPC_DI_TERM: input impedance is software selectable.	
SPCM_DI_INDIVPULSEWIDTH	00010000h	Trigger pulsewidth is individually per channel programmable	
SPCM_DI_IOCHANNEL	00200000h	The input channel is connected with a digital output (DO) channel (digital I/O cards only).	

Setting up the input threshold and termination (single-ended models only)

Programmable input threshold

For the single-ended inputs the threshold can be defined per data byte, to accomodate for different logic level standards:

Table 32: Spectrum API: threshold registers and register settings

Register	Value	Direction	Description	Offset range
SPC_THRESHOLD0	42400	read/write	Defines the common input threshold level for data lines D7 .. D0.	0 .. 4000 mV in steps of 10 mV
SPC_THRESHOLD1	42401	read/write	Defines the common input threshold level for data lines D15 .. D8.	0 .. 4000 mV in steps of 10 mV
SPC_THRESHOLD2	42402	read/write	Defines the common input threshold level for data lines D23 .. D16.	0 .. 4000 mV in steps of 10 mV
SPC_THRESHOLD3	42403	read/write	Defines the common input threshold level for data lines D31 .. D24.	0 .. 4000 mV in steps of 10 mV

As an example define two different levels for the two data bytes:

```
spcm_dwSetParam_i64 (hDrv, SPC_THRESHOLD0, 1500); // set D0..D7 threshold to 1500 mV
spcm_dwSetParam_i64 (hDrv, SPC_THRESHOLD1, 600); // set D8..D15 threshold to 600 mV
```

Programmable input termination

For the single-ended inputs the termination can also be defined per data byte:

Table 33: Spectrum API: input termination registers

Register	Value	Direction	Description
SPC_75OHM0	30060	read/write	A „1“ sets the 75 Ω termination to GND for D7 .. D0. A „0“ sets the termination to 4.7 k Ω to GND.
SPC_75OHM1	30160	read/write	A „1“ sets the 75 Ω termination to GND for D15 .. D8. A „0“ sets the termination to 4.7 k Ω to GND.
SPC_75OHM2	30260	read/write	A „1“ sets the 75 Ω termination to GND for D23 .. D16. A „0“ sets the termination to 4.7 k Ω to GND.
SPC_75OHM3	30360	read/write	A „1“ sets the 75 Ω termination to GND for D31 .. D24. A „0“ sets the termination to 4.7 k Ω to GND.

As an example enable termination for byte0 and set byte1 to high-impedance:

```
spcm_dwSetParam_i64 (hDrv, SPC_75OHM0, 1); // set to 75 Ohm
spcm_dwSetParam_i64 (hDrv, SPC_75OHM1, 0); // set to 4.7 kOhm
```

Acquisition modes

Your card is able to run in different modes. Depending on the selected mode there are different registers that each define an aspect of this mode. The single modes are explained in this chapter. Any further modes that are only available if an option is installed on the card is documented in a later chapter.

Overview

This chapter gives you a general overview on the related registers for the different modes. The use of these registers throughout the different modes is described in the following chapters.

Setup of the mode

The mode register is organized as a bitmap. Each mode corresponds to one bit of this bitmap. When defining the mode to use, please be sure just to set one of the bits. All other settings will return an error code.

The main difference between all standard and all FIFO modes is that the standard modes are limited to on-board memory and therefore can run with full sampling rate. The FIFO modes are designed to transfer data continuously over the bus to PC memory or to hard disk and can therefore run much longer. The FIFO modes are limited by the maximum bus transfer speed the PC can use. The FIFO mode uses the complete installed on-board memory as a FIFO buffer.

However as you'll see throughout the detailed documentation of the modes the standard and the FIFO mode are similar in programming and behavior and there are only a very few differences between them.

Table 34: Spectrum API: card mode and read out of available card mode software registers

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode, a read command will return the currently used mode.
SPC_AVAILCARDMODES	9501	read	Returns a bitmap with all available modes on your card. The modes are listed below.

Acquisition modes

Table 35: Spectrum API: possible values for the card mode register. Description of the different card modes

Mode	Value	Available on	Description
SPC_REC_STD_SINGLE	1h	all cards	Data acquisition to on-board memory for one single trigger event.
SPC_REC_STD_MULTI	2h	all cards	Data acquisition to on-board memory for multiple trigger events. Each recorded segment has the same size. This mode is described in greater detail in a special chapter about the Multiple Recording option.
SPC_REC_STD_GATE	4h	all M2p and M4i digitizers and NETBOXes	Data acquisition to on-board memory using an external Gate signal. Acquisition is only done as long as the gate signal has a programmed level. The mode is described in greater detail in a special chapter about the Gated Sampling option.
SPC_REC_STD_ABA	8h	all M2p and M4i digitizers and NETBOXes	Data acquisition to on-board memory for multiple trigger events. While the multiple trigger events are stored with programmed sampling rate the inputs are sampled continuously with a slower sampling speed. The mode is described in a special chapter about ABA mode option.
SPC_REC_STD_SEGSTATS	10000h	M4i/M4x.2xxx M4i/M4x.44xx DN2/DN6.2xx DN2/DN6.44x digitizers only	Data acquisition to on-board memory for multiple trigger events, using Block/Segment Statistic Module (FPGA firmware Option).
SPC_REC_STD_AVERAGE	20000h	M4i/M4x.2xxx M4i/M4x.44xx M5i.33xx DN2/DN6.2xx DN2/DN6.44x digitizers only	Data acquisition to on-board memory for multiple trigger events, using Block Average Module (FPGA firmware Option).
SPC_REC_STD_BOXCAR	800000h	M4i/M4x.44xx DN2/DN6.44x digitizers only	Enables Boxcar Averaging for standard acquisition. Requires digitizer module with firmware version V29 or newer.
SPC_REC_FIFO_SINGLE	10h	all cards	Continuous data acquisition for one single trigger event. The on-board memory is used completely as FIFO buffer.
SPC_REC_FIFO_MULTI	20h	all cards	Continuous data acquisition for multiple trigger events.
SPC_REC_FIFO_GATE	40h	all M2p and M4i digitizers and NETBOXes	Continuous data acquisition using an external gate signal.
SPC_REC_FIFO_ABA	80h	all M2p and M4i digitizers and NETBOXes	Continuous data acquisition for multiple trigger events together with continuous data acquisition with a slower sampling clock.
SPC_REC_FIFO_SEGSTATS	100000h	M4i/M4x.2xxx M4i/M4x.44xx DN2/DN6.2xx DN2/DN6.44x digitizers only	Enables Block/Segment Statistic for FIFO acquisition (FPGA firmware Option).
SPC_REC_FIFO_AVERAGE	200000h	M4i/M4x.2xxx M4i/M4x.44xx M5i.33xx DN2/DN6.2xx DN2/DN6.44x digitizers only	Enables Block Averaging for FIFO acquisition (FPGA firmware Option).
SPC_REC_FIFO_BOXCAR	1000000h	M4i/M4x.44xx DN2/DN6.44x digitizers only	Enables Boxcar Averaging for FIFO acquisition. Requires digitizer module firmware version V29 or newer.
SPC_REC_FIFO_SINGLE_MONITOR	2000000h	all M2p and M4i digitizers and NETBOXes	Combination of SPC_REC_FIFO_SINGLE mode with additional slower sampling clock data stream for monitoring purposes (same as A-data of SPC_REC_FIFO_ABA mode).

Commands

The data acquisition/data replay is controlled by the command register. The command register controls the state of the card in general and also the state of the different data transfers. Data transfers are explained in an extra chapter later on.

The commands are split up into two types of commands: execution commands that fulfill a job and wait commands that will wait for the occurrence of an interrupt. Again the commands register is organized as a bitmap allowing you to set several commands together with one call. As not all of the command combinations make sense (like the combination of reset and start at the same time) the driver will check the given command and return an error code ERR_SEQUENCE if one of the given commands is not allowed in the current state.

Table 36: Spectrum API: card command register and different commands with descriptions

Register	Value	Direction	Description
SPC_M2CMD	100	write only	Executes a command for the card or data transfer.

Card execution commands

M2CMD_CARD_RESET	1h	Performs a hard and software reset of the card as explained further above.
M2CMD_CARD_WRITESSETUP	2h	Writes the current setup to the card without starting the hardware. This command may be useful if changing some internal settings like clock frequency and enabling outputs.
M2CMD_CARD_START	4h	Starts the card with all selected settings. This command automatically writes all settings to the card if any of the settings has been changed since the last one was written. After card has been started, only some of the settings might be changed while the card is running, such as e.g. output level and offset for D/A replay cards.
M2CMD_CARD_ENABLETRIGGER	8h	The trigger detection is enabled. This command can be either sent together with the start command to enable trigger immediately or in a second call after some external hardware has been started.
M2CMD_CARD_FORCETRIGGER	10h	This command forces a trigger even if none has been detected so far. Sending this command together with the start command is similar to using the software trigger.
M2CMD_CARD_DISABLETRIGGER	20h	The trigger detection is disabled. All further trigger events are ignored until the trigger detection is again enabled. When starting the card the trigger detection is started disabled.
M2CMD_CARD_STOP	40h	Stops the current run of the card. If the card is not running this command has no effect.

Card wait commands

These commands do not return until either the defined state has been reached which is signaled by an interrupt from the card or the timeout counter has expired. If the state has been reached the command returns with an ERR_OK. If a timeout occurs the command returns with ERR_TIMEOUT. If the card has been stopped from a second thread with a stop or reset command, the wait function returns with ERR_ABORT.

M2CMD_CARD_WAITPREFULL	1000h	Acquisition modes only: the command waits until the pretrigger area has once been filled with data. After pretrigger area has been filled the internal trigger engine starts to look for trigger events if the trigger detection has been enabled.
M2CMD_CARD_WAITTRIGGER	2000h	Waits until the first trigger event has been detected by the card. If using a mode with multiple trigger events like Multiple Recording or Gated Sampling there only the first trigger detection will generate an interrupt for this wait command.
M2CMD_CARD_WAITREADY	4000h	Waits until the card has completed the current run. In an acquisition mode receiving this command means that all data has been acquired. In a generation mode receiving this command means that the output has stopped.

Wait command timeout

If the state for which one of the wait commands is waiting isn't reached any of the wait commands will either wait forever if no timeout is defined or it will return automatically with an ERR_TIMEOUT if the specified timeout has expired.

Table 37: Spectrum API: timeout definition register

Register	Value	Direction	Description
SPC_TIMEOUT	295130	read/write	Defines the timeout for any following wait command in a millisecond resolution. Writing a zero to this register disables the timeout.

As a default the timeout is disabled. After defining a timeout this is valid for all following wait commands until the timeout is disabled again by writing a zero to this register.

A timeout occurring should not be considered as an error. It did not change anything on the board status. The board is still running and will complete normally. You may use the timeout to abort the run after a certain time if no trigger has occurred. In that case a stop command is necessary after receiving the timeout. It is also possible to use the timeout to update the user interface frequently and simply call the wait function afterwards again.

Example for card control:

```
// card is started and trigger detection is enabled immediately
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);

// we wait a maximum of 1 second for a trigger detection. In case of timeout we force the trigger
spcm_dwSetParam_i32 (hDrv, SPC_TIMEOUT, 1000);
if (spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_WAITTRIGGER) == ERR_TIMEOUT)
{
    printf ("No trigger detected so far, we force a trigger now!\n");
    spcm_dwSetParam (hDrv, SPC_M2CMD, M2CMD_CARD_FORCETRIGGER);
}

// we disable the timeout and wait for the end of the run
spcm_dwSetParam_i32 (hDrv, SPC_TIMEOUT, 0);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_WAITREADY);
printf ("Card has stopped now!\n");
```

Card Status

In addition to the wait for an interrupt mechanism or completely instead of it one may also read out the current card status by reading the SPC_M2STATUS register. The status register is organized as a bitmap, so that multiple bits can be set, showing the status of the card and also of the different data transfers.

Table 38: Spectrum API: card status register and possible status values with descriptions of the status

Register	Value	Direction	Description
SPC_M2STATUS	110	read only	Reads out the current status information

M2STAT_CARD_PRETRIGGER	1h	Acquisition modes only: the first pretrigger area has been filled. In Multi/ABA/Gated acquisition this status is set only for the first segment and will be cleared at the end of the acquisition.
M2STAT_CARD_TRIGGER	2h	The first trigger has been detected.
M2STAT_CARD_READY	4h	The card has finished its run and is ready.
M2STAT_CARD_SEGMENT_PRETRG	8h	This flag will be set for each completed pretrigger area including the first one of a Single acquisition. Additionally for a Multi/ABA/Gated acquisition of M4i/M4x/M2p only, this flag will be set when the pretrigger area of a segment has been filled and will be cleared after the trigger for a segment has been detected.

Acquisition cards status overview

The following drawing gives you an overview of the card commands and card status information. After start of card with M2CMD_CARD_START the card is acquiring pretrigger data until one time complete pretrigger data has been acquired. Then the status bit M2STAT_CARD_PRETRIGGER is set. Either the trigger has been enabled together with the start command or the card now waits for trigger enable command M2CMD_CARD_ENABLETRIGGER. After receiving this command the trigger engine is enabled and card checks for a trigger event. As soon as the trigger event is received the status bit M2STAT_CARD_TRIGGER is set and the card acquires the programmed posttrigger data. After all post trigger data has been acquired the status bit M2STAT_CARD_READY is set and data can be read out:

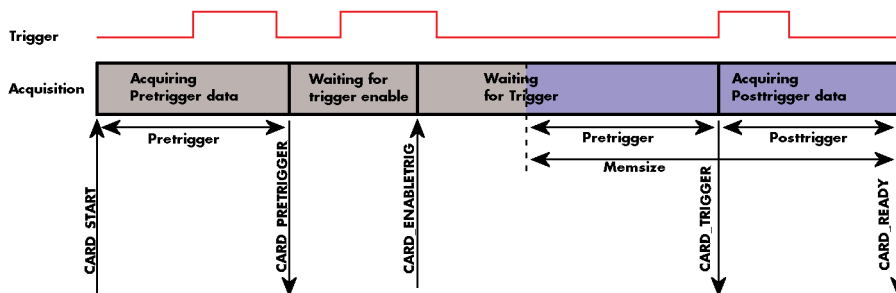


Image 40: Acquisition cards: graphical overview of acquisition status and card command interaction

Generation card status overview

This drawing gives an overview of the card commands and status information for a simple generation mode. After start of card with the M2CMD_CARD_START the card is armed and waiting. Either the trigger has been enabled together with the start command or the card now waits for trigger enable command M2CMD_CARD_ENABLETRIGGER. After receiving this command the trigger engine is enabled and card checks for a trigger event. As soon as the trigger event is received the status bit M2STAT_CARD_TRIGGER is set and the card starts with the data replay. After replay has been finished - depending on the programmed mode - the status bit M2STAT_CARD_READY is set and the card stops.

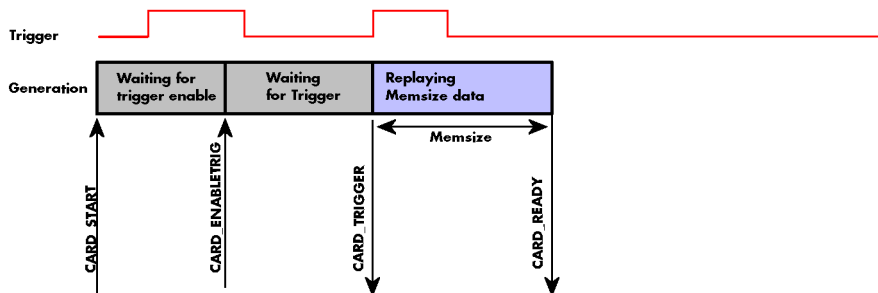


Image 41: Generation cards: graphical overview of generation status and card command interaction

Data Transfer

Data transfer consists of two parts: the buffer definition and the commands/status information that controls the transfer itself. Data transfer shares the command and status register with the card control commands and status information. In general the following details on the data transfer are valid for any data transfer in any direction:

- The memory size register (SPC_MEMSIZE) must be programmed before starting the data transfer.
- When the hardware buffer is adjusted from its default (see „Output latency“ section later in this manual), this must be done before defining the transfer buffers in the next step via the spcm_dwDefTransfer function.
- Before starting a data transfer the buffer must be defined using the spcm_dwDefTransfer function.
- Each defined buffer is only used once. After transfer has ended the buffer is automatically invalidated.
- If a buffer has to be deleted although the data transfer is in progress or the buffer has at least been defined it is necessary to call the spcm_dwInvalidateBuf function.

Definition of the transfer buffer

Before any data transfer can start it is necessary to define the transfer buffer with all its details. The definition of the buffer is done with the `spcm_dwDefTransfer` function as explained in an earlier chapter.

```
uint32 _stdcall spcm_dwDefTransfer_i64 (// Defines the transfer buffer by using 64 bit unsigned integer values
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType,                  // type of the buffer to define as listed below under SPCM_BUF_XXXX
    uint32 dwDirection,                // the transfer direction as defined below
    uint32 dwNotifySize,               // number of bytes after which an event is sent (0=end of transfer)
    void* pvDataBuffer,                // pointer to the data buffer
    uint64 qwBrdOffs,                  // offset for transfer in board memory
    uint64 qwTransferLen);              // buffer length
```

This function is used to define buffers for standard sample data transfer as well as for extra data transfer for additional ABA or timestamp information. Therefore the `dwBufType` parameter can be one of the following:

SPCM_BUF_DATA	1000	Buffer is used for transfer of standard sample data
SPCM_BUF_ABA	2000	Buffer is used to read out slow ABA data. Details on this mode are described in the chapter about the ABA mode option
SPCM_BUF_TIMESTAMP	3000	Buffer is used to read out timestamp information. Details on this mode are described in the chapter about the timestamp option.

The `dwDirection` parameter defines the direction of the following data transfer:

SPCM_DIR_PCTOCARD	0	Transfer is done from PC memory to on-board memory of card
SPCM_DIR_CARDTOPC	1	Transfer is done from card on-board memory to PC memory.
SPCM_DIR_CARDTOGPU	2	RDMA transfer from card memory to GPU memory, SCAPP option needed, Linux only
SPCM_DIR_GPUTCARD	3	RDMA transfer from GPU memory to card memory, SCAPP option needed, Linux only

The direction information used here must match the currently used mode. While an acquisition mode is used there's no transfer from PC to card allowed and vice versa. It is possible to use a special memory test mode to come beyond this limit. Set the `SPC_MEMTEST` register as defined further below.



The `dwNotifySize` parameter defines the amount of bytes after which an interrupt should be generated. If leaving this parameter zero, the transfer will run until all data is transferred and then generate an interrupt. Filling in notify size > zero will allow you to use the amount of data that has been transferred so far. The notify size is used on FIFO mode to implement a buffer handshake with the driver or when transferring large amount of data where it may be of interest to start data processing while data transfer is still running. Please see the chapter on handling positions further below for details.

M2i, M3i, M4i, M4x and M2p cards:

The Notify size sticks to the page size which is defined by the PC hardware and the operating system. Therefore the notify size must be a multiple of 4 kByte. For main data transfer it may also be a fraction of 4k in the range of 16, 32, 64, 128, 256, 512, 1k or 2k. No other values are allowed. For ABA and timestamp the notify size can be 2k as a minimum. If you need to work with ABA or timestamp data in smaller chunks please use the polling mode as described later.



M5i:

The Notify size sticks to the page size which is defined by the PC hardware and the operating system. Therefore the notify size must be a multiple of 4 kByte. For main data transfer it may also be a fraction of 4k in the range of 64, 128, 256, 512, 1k or 2k. No other values are allowed. For timestamp the notify size can be 2k as a minimum. If you need to work with timestamp data in smaller chunks please use the polling mode as described later.



The `pvDataBuffer` must point to an allocated data buffer for the transfer. Please be sure to have at least the amount of memory allocated that you program to be transferred. If the transfer is going from card to PC this data is overwritten with the current content of the card on-board memory.

The `pvDataBuffer` needs to be aligned to a page size (4096 bytes). Please use appropriate software commands when allocating the data buffer. Using a non-aligned buffer may result in data corruption.



When not doing FIFO mode one can also use the `qwBrdOffs` parameter. This parameter defines the starting position for the data transfer as byte value in relation to the beginning of the card memory. Using this parameter allows it to split up data transfer in smaller chunks if one has acquired a very large on-board memory.

The `qwTransferLen` parameter defines the number of bytes that has to be transferred with this buffer. Please be sure that the allocated memory has at least the size that is defined in this parameter. In standard mode this parameter cannot be larger than the amount of data defined with memory size.

M5i cards only:

On M5i cards the `qwTransferLen` parameter needs to be an integer multiple of 64 bytes.



Memory test mode

In some cases it might be of interest to transfer data in the opposite direction. Therefore a special memory test mode is available which allows random read and write access of the complete on-board memory. While memory test mode is activated no normal card commands are processed:

Table 39: Spectrum API: memory test register

Register	Value	Direction	Description
SPC_MEMTEST	200700	read/write	Writing a 1 activates the memory test mode, no commands are then processed. Writing a 0 deactivates the memory test mode again.

Invalidation of the transfer buffer

The command can be used to invalidate an already defined buffer if the buffer is about to be deleted by user. This function is automatically called if a new buffer is defined or if the transfer of a buffer has completed

```
uint32 _stdcall spcm_dwInvalidateBuf ( // invalidate the transfer buffer
    drv_handle hDevice,                // handle to an already opened device
    uint32     dwBufType);             // type of the buffer to invalidate as listed above under SPCM_BUF_XXXX
```

The `dwBufType` parameter need to be the same parameter for which the buffer has been defined:

SPCM_BUF_DATA	1000	Buffer is used for transfer of standard sample data
SPCM_BUF_ABA	2000	Buffer is used to read out slow ABA data. Details on this mode are described in the chapter about the ABA mode option. The ABA mode is only available on analog acquisition cards.
SPCM_BUF_TIMESTAMP	3000	Buffer is used to read out timestamp information. Details on this mode are described in the chapter about the timestamp option. The timestamp mode is only available on analog or digital acquisition cards.

Commands and Status information for data transfer buffers.

As explained above the data transfer is performed with the same command and status registers like the card control. It is possible to send commands for card control and data transfer at the same time as shown in the examples further below.

Table 40: Spectrum API: Command register and commands for DMA transfers

Register	Value	Direction	Description
SPC_M2CMD	100	write only	Executes a command for the card or data transfer
M2CMD_DATA_STARTDMA	10000h		Starts the DMA transfer for an already defined buffer. In acquisition mode it may be that the card hasn't received a trigger yet, in that case the transfer start is delayed until the card receives the trigger event
M2CMD_DATA_WAITDMA	20000h		Waits until the data transfer has ended or until at least the amount of bytes defined by notify size are available. This wait function also takes the timeout parameter described above into account.
M2CMD_DATA_STOPDMA	40000h		Stops a running DMA transfer. Data is invalid afterwards.

The data transfer can generate one of the following status information:

Table 41: Spectrum API: status register and status codes for DMA data transfer

Register	Value	Direction	Description
SPC_M2STATUS	110	read only	Reads out the current status information
M2STAT_DATA_BLOCKREADY	100h		The next data block as defined in the notify size is available. It is at least the amount of data available but it also can be more data.
M2STAT_DATA_END	200h		The data transfer has completed. This status information will only occur if the notify size is set to zero.
M2STAT_DATA_OVERRUN	400h		The data transfer had on overrun (acquisition) or underrun (replay) while doing FIFO transfer.
M2STAT_DATA_ERROR	800h		An internal error occurred while doing data transfer.

Example of data transfer

```
void* pvData = pvAllocMemPageAligned (1024);

// transfer data from PC memory to card memory (on replay cards) ...
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_PCTOCARD, 0, pvData, 0, 1024);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

// ... or transfer data from card memory to PC memory (acquisition cards)
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 0, pvData, 0, 1024);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

// explicitly stop DMA transfer prior to invalidating buffer
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STOPDMA);
spcm_dwInvalidateBuf (hDrv, SPCM_BUF_DATA);
vFreeMemPageAligned (pvData, 1024);
```

To keep the example simple it does no error checking. Please be sure to check for errors if using these command in real world programs!

Users should take care to explicitly send the `M2CMD_DATA_STOPDMA` command prior to invalidating the buffer, to avoid crashes due to race conditions when using higher-latency data transportation layers, such as to remote Ethernet devices.



Standard Single acquisition mode

The standard single mode is the easiest and mostly used mode to acquire analog data with a Spectrum acquisition card. In standard single recording mode the card is working totally independent from the PC, after the card setup is done. The advantage of the Spectrum boards is that regardless to the system usage the card will sample with equidistant time intervals.

The sampled and converted data is stored in the on-board memory and is held there for being read out after the acquisition. This mode allows sampling at very high conversion rates without the need to transfer the data into the memory of the host system at high speed.

After the recording is done, the data can be read out by the user and is transferred via the bus into PC memory.

This standard recording mode is the most common mode for all analog and digital acquisition and oscilloscope boards. The data is written to a programmed amount of the on-board memory (memsize). That part of memory is used as a ring buffer, and recording is done continuously until a trigger event is detected. After the trigger event, a certain programmable amount of data is recorded (post trigger) and then the recording finishes. Due to the continuous ring buffer recording, there are also samples prior to the trigger event in the memory (pretrigger).

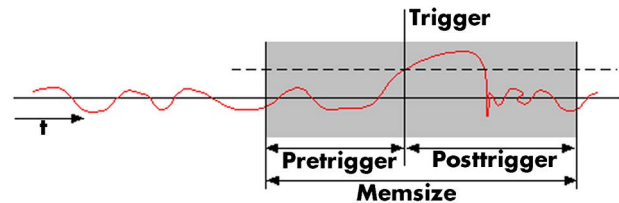


Image 42: standard acquisition mode and pretrigger/posttrigger/trigger relation

When the card is started the pre trigger area is filled up with data first. While doing this the board's trigger detection is not armed. If you use a huge pre trigger size and a slow sample rate it can take some time after starting the board before a trigger event will be detected.



Card mode

The card mode has to be set to the correct mode SPC_REC_STD_SINGLE.

Table 42: Spectrum API: card mode register and standard single mode setup

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode, a read command will return the currently used mode.
SPC_REC_STD_SINGLE	1h		Data acquisition to on-board memory for one single trigger event.

Memory, Pre- and Posttrigger

At first you have to define, how many samples are to be recorded at all and how many of them should be acquired after the trigger event has been detected.

Table 43: Spectrum API: memory size and posttrigger registers for standard single mode

Register	Value	Direction	Description
SPC_MEMSIZE	10000	read/write	Sets the memory size in samples per channel.
SPC_POSTTRIGGER	10100	read/write	Sets the number of samples to be recorded per channel after the trigger event has been detected.

You can access these settings by the register SPC_MEMSIZE, which sets the total amount of data that is recorded, and the register SPC_POSTTRIGGER, that defines the number of samples to be recorded after the trigger event has been detected. The size of the pretrigger results on the simple formula:

pretrigger = memsize - posttrigger

The maximum memsize that can be used for recording is of course limited by the installed amount of memory and by the number of channels to be recorded. Please have a look at the topic "Limits of pre, post memsize, loops" later in this chapter.

Example

The following example shows a simple standard single mode data acquisition setup with the read out of data afterwards. To keep this example simple there is no error checking implemented.

```
int32 lMemsize = 16384; // recording length is set to 16 kSamples

spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0); // only one channel activated
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_SINGLE); // set the standard single recording mode
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, lMemsize); // recording length
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, 8192); // samples to acquire after trigger = 8k

// now we start the acquisition and wait for the interrupt that signalizes the end
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_CARD_WAITREADY);

void* pvData = pvAllocMemPageAligned (2 * lMemsize); // assuming 2 bytes per sample

// read out the data
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 0, pvData, 0, 2 * lMemsize);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);
```

FIFO Single acquisition mode

The FIFO single mode does a continuous data acquisition using the on-board memory as a FIFO buffer and transferring data continuously to PC memory. One can make on-line calculations with the acquired data, store the data continuously to disk for later use or even have a data logger functionality with on-line data display.

Card mode

The card mode has to be set to the correct mode SPC_REC_FIFO_SINGLE.

Table 44: Spectrum API: card mode register and standard FIFO mode setup

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode, a read command will return the currently used mode.
SPC_REC_FIFO_SINGLE	10h		Continuous data acquisition to PC memory. Complete on-board memory is used as FIFO buffer.

Length and Pretrigger

Even in FIFO mode it is possible to program a pretrigger area. In general FIFO mode can run forever until it is stopped by an explicit user command or one can program the total length of the transfer by two counters Loop and Segment size

Table 45: Spectrum API: setup registers for standard FIFO mode

Register	Value	Direction	Description
SPC_PRETRIGGER	10030	read/write	Programs the number of samples to be acquired before the trigger event detection
SPC_SEGMENTSIZE	10010	read/write	Length of segments to acquire.
SPC_LOOPS	10020	read/write	Number of segments to acquire in total. If set to zero the FIFO mode will run continuously until it is stopped by the user.

The total amount of samples per channel that is acquired can be calculated by $[SPC_LOOPS * SPC_SEGMENTSIZ]$. Please stick to the below mentioned limitations of the registers.

Difference to standard single acquisition mode

The standard modes and the FIFO modes differ not very much from the programming side. In fact one can even use the FIFO mode to get the same behavior like the standard mode. The buffer handling that is shown in the next chapter is the same for both modes.

Pretrigger

When doing standard single acquisition memory is used as a circular buffer and the pre trigger can be up to the [installed memory] - [minimum post trigger]. Compared to this the pre trigger in FIFO mode is limited by a special pre trigger FIFO and hence considerably shorter.

Length of acquisition.

In standard mode the acquisition length is defined before the start and is limited to the installed on-board memory whilst in FIFO mode the acquisition length can either be defined or it can run continuously until user stops it.

Example FIFO acquisition

The following example shows a simple FIFO single mode data acquisition setup with the read out of data afterwards. To keep this example simple there is no error checking implemented.

```

spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0);           // only one channel activated
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_FIFO_SINGLE); // set the FIFO single recording mode
spcm_dwSetParam_i64 (hDrv, SPC_PRETRIGGER, 1024);             // 1 kSample of data before trigger

// in FIFO mode we need to define the buffer before starting the transfer
void* pvData = pvAllocMemPageAligned (llBufsizeInSamples * 2); // 2 bytes per sample
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 4096,
                        pvData, 0, 2 * llBufsizeInSamples);

// now we start the acquisition and wait for the first block
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

// we acquire data in a loop. As we defined a notify size of 4k we'll get the data in >=4k chunks
llTotalBytes = 0;
while (!dwError)
{
    spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_LEN, &llAvailBytes); // read out the available bytes
    llTotalBytes += llAvailBytes;

    // here is the right position to do something with the data (printf is limited to 32 bit variables)
    printf ("Currently Available: %lld, total: %lld\n", llAvailBytes, llTotalBytes);

    // now we free the number of bytes and wait for the next buffer
    spcm_dwSetParam_i64 (hDrv, SPC_DATA_AVAIL_CARD_LEN, llAvailBytes);
    dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_WAITDMA);
}

```

Limits of pre trigger, post trigger, memory size

The maximum memory size parameter is only limited by the number of activated channels and by the amount of installed memory. Minimum memory size as well as minimum and maximum post trigger limits are independent of the activated channels or the installed memory.

Due to the internal organization of the card memory there is a certain stepsize when setting these values that has to be taken into account. The following table gives you an overview of all limits concerning pre trigger, post trigger, memory size, segment size and loops. The table shows all values in relation to the installed memory size in samples. If more memory is installed the maximum memory size figures will increase according to the complete installed memory

Table 46: Limits of pre trigger, post trigger, memory size depending on selected mode and activated channels

Activated Channels	Used Mode	Memory size SPC_MEMSIZE			Pre trigger SPC_PRETRIGGER			Post trigger SPC_POSTTRIGGER			Segment size SPC_SEGMENTSIZE			Loops SPC_LOOPS		
		Min	Max	Step	Min	Max	Step	Min	Max	Step	Min	Max	Step	Min	Max	Step
8 Ch	Standard Single	64	Mem	32	32	Mem - 32	32	32	8G - 32	32	not used			not used		
	Standard Multi/ABA	64	Mem	32	32	4k	32	32	Mem/2-32	32	64	Mem/2	32	not used		
	Standard Gate	64	Mem	32	32	8k	32	32	Mem-32	32	not used			not used		
	FIFO Single	not used			32	4k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	4k	32	32	8G - 32	32	64	pre+post	32	0 (∞)	4G - 1	1
	FIFO Gate	not used			32	4k	32	32	8G - 32	32	not used			0 (∞)	4G - 1	1
16 Ch	Standard Single	64	Mem/2	32	32	Mem/2 - 32	32	32	8G - 32	32	not used			not used		
	Standard Multi/ABA	64	Mem/2	32	32	4k	32	32	Mem/4-32	32	64	Mem/4	32	not used		
	Standard Gate	64	Mem/2	32	32	4k	32	32	Mem/2-32	32	not used			not used		
	FIFO Single	not used			32	4k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	4k	32	32	8G - 32	32	64	pre+post	32	0 (∞)	4G - 1	1
	FIFO Gate	not used			32	4k	32	32	8G - 32	32	not used			0 (∞)	4G - 1	1
32 Ch	Standard Single	64	Mem/4	32	32	Mem/4 - 32	32	32	8G - 16	32	not used			not used		
	Standard Multi/ABA	64	Mem/4	32	32	4k	32	32	Mem/8-32	32	64	Mem/8	32	not used		
	Standard Gate	64	Mem/4	32	32	4k	32	32	Mem/4-16	32	not used			not used		
	FIFO Single	not used			32	4k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	4k	32	32	8G - 32	32	64	pre+post	32	0 (∞)	4G - 1	1
	FIFO Gate	not used			32	4k	32	32	8G - 32	32	not used			0 (∞)	4G - 1	1

All figures listed here are given in samples. An entry of [8G - 32] means [8 GSamples - 32] = 8,589,934,560 samples.

The given memory and memory / divider figures depend on the installed on-board memory as listed below:

	Installed Memory
	4 GSample
Mem	4 GSample
Mem / 2	2 GSample
Mem / 4	1 GSample
Mem / 8	512 MSample

Please keep in mind that this table shows all values at once. Only the absolute maximum and minimum values are shown. There might be additional limitations. Which of these values is programmed depends on the used mode. Please read the detailed documentation of the mode.

Buffer handling

To handle the huge amount of data that can possibly be acquired with the M5i/M4i/M4x/M2p series cards, there is a very reliable two step buffer strategy set up. The on-board memory of the card can be completely used as a real FIFO buffer. In addition a part of the PC memory can be used as an additional software buffer. Transfer between hardware FIFO and software buffer is performed interrupt driven and automatically by the driver to get best performance. The following drawing will give you an overview of the structure of the data transfer handling:

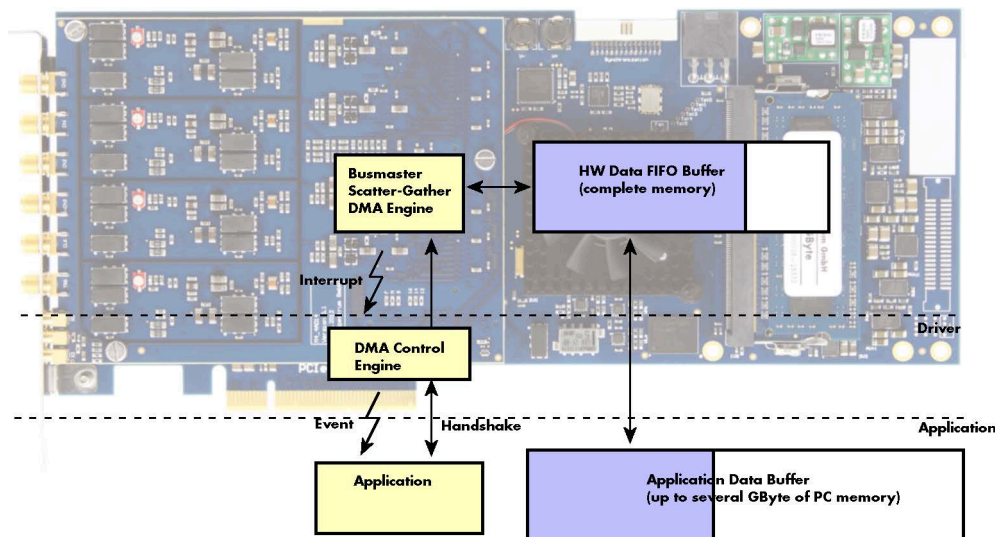


Image 43: Overview of buffer handling for DMA transfers showing and the interaction with the DMA engine

Although an M4i is shown here, this applies to M5i, M4x and M2p cards as well. A data buffer handshake is implemented in the driver which allows to run the card in different data transfer modes. The software transfer buffer is handled as one large buffer which is on the one side controlled by the driver and filled automatically by busmaster DMA from/to the hardware FIFO buffer and on the other hand it is handled by the user who set's parts of this software buffer available for the driver for further transfer. The handshake is fulfilled with the following 3 software registers:

Table 47: Spectrum API: registers for DMA buffer handling

Register	Value	Direction	Description
SPC_DATA_AVAIL_USER_LEN	200	read	Returns the number of currently to the user available bytes inside a sample data transfer.
SPC_DATA_AVAIL_USER_POS	201	read	Returns the position as byte index where the currently available data samples start.
SPC_DATA_AVAIL_CARD_LEN	202	write	Writes the number of bytes that the card can now use for sample data transfer again

Internally the card handles two counters, a user counter and a card counter. Depending on the transfer direction the software registers have slightly different meanings:

Table 48: Spectrum API: content of DMA buffer handling registers for different use cases

Transfer direction	Register	Direction	Description
Write to card	SPC_DATA_AVAIL_USER_LEN	read	This register contains the currently available number of bytes that are free to write new data to the card. The user can now fill this amount of bytes with new data to be transferred.
	SPC_DATA_AVAIL_CARD_LEN	write	After filling an amount of the buffer with new data to transfer to card, the user tells the driver with this register that the amount of data is now ready to transfer.
Read from card	SPC_DATA_AVAIL_USER_LEN	read	This register contains the currently available number of bytes that are filled with newly transferred data. The user can now use this data for own purposes, copy it, write it to disk or start calculations with this data.
	SPC_DATA_AVAIL_CARD_LEN	write	After finishing the job with the new available data the user needs to tell the driver that this amount of bytes is again free for new data to be transferred.

Table 48: Spectrum API: content of DMA buffer handling registers for different use cases

Transfer direction	Register	Direction	Description
Any direction	SPC_DATA_AVAIL_USER_POS	read	The register holds the current byte index position where the available bytes start. The register is just intended to help you and to avoid own position calculation
Any direction	SPC_FILLSIZEPROMILLE	read	The register holds the current fill size of the on-board memory (FIFO buffer) in promille (1/1000) of the full on-board memory. Please note that the hardware reports the fill size only in 1/16 parts of the full memory. The reported fill size is therefore only shown in $1000/16 = 63$ promille steps.

Directly after start of transfer the SPC_DATA_AVAIL_USER_LEN is every time zero as no data is available for the user and the SPC_DATA_AVAIL_CARD_LEN is every time identical to the length of the defined buffer as the complete buffer is available for the card for transfer.

The counter that is holding the user buffer available bytes (SPC_DATA_AVAIL_USER_LEN) is related to the notify size at the DefTransfer call. Even when less bytes already have been transferred you won't get notice of it in case the notify size is programmed to a higher value.



Remarks

- The transfer between hardware FIFO buffer and application buffer is done with scatter-gather DMA using a busmaster DMA controller located on the card. Even if the PC is busy with other jobs data is still transferred until the application data buffer is completely used.
- Even if application data buffer is completely used there's still the hardware FIFO buffer that can hold data until the complete on-board memory is used. Therefore a larger on-board memory will make the transfer more reliable against any PC dead times.
- As you see in the above picture data is directly transferred between application data buffer and on-board memory. Therefore it is absolutely critical to delete the application data buffer without stopping any DMA transfers that are running actually. It is also absolutely critical to define the application data buffer with an unmatched length as DMA can then try to access memory outside the application data area.
- As shown in the drawing above the DMA control will announce new data to the application by sending an event. Waiting for an event is done internally inside the driver if the application calls one of the wait functions. Waiting for an event does not consume any CPU time and is therefore highly desirable if other threads do a lot of calculation work. However it is not necessary to use the wait functions and one can simply request the current status whenever the program has time to do so. When using this polling mode the announced available bytes still stick to the defined notify size!
- If the on-board FIFO buffer has an overrun (card to PC) or an underrun (PC to card) data transfer is stopped. However in case of transfer from card to PC there is still a lot of data in the on-board memory. Therefore the data transfer will continue until all data has been transferred although the status information already shows an overrun.
- For very small notify sizes, getting best bus transfer performance could be improved by using a „continuous buffer“. This mode is explained in the appendix in greater detail.

M2i, M3i, M4i, M4x and M2p cards:

The Notify size sticks to the page size which is defined by the PC hardware and the operating system. Therefore the notify size must be a multiple of 4 kByte. For main data transfer it may also be a fraction of 4k in the range of 16, 32, 64, 128, 256, 512, 1k or 2k. No other values are allowed. For ABA and timestamp the notify size can be 2k as a minimum. If you need to work with ABA or timestamp data in smaller chunks please use the polling mode as described later.



M5i:

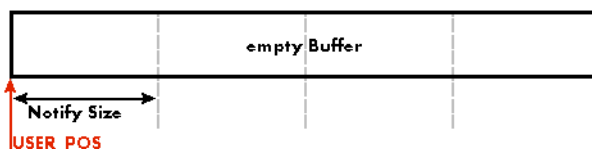
The Notify size sticks to the page size which is defined by the PC hardware and the operating system. Therefore the notify size must be a multiple of 4 kByte. For main data transfer it may also be a fraction of 4k in the range of 64, 128, 256, 512, 1k or 2k. No other values are allowed. For timestamp the notify size can be 2k as a minimum. If you need to work with timestamp data in smaller chunks please use the polling mode as described later.



The following graphs will show the current buffer positions in different states of the transfer. The drawings have been made for the transfer from card to PC. However all the block handling is similar for the opposite direction, just the empty and the filled parts of the buffer are inverted.

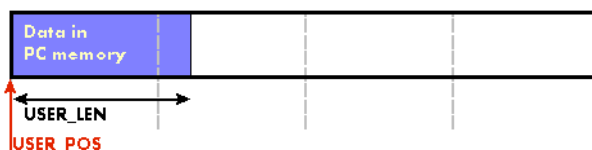
Step 1: Buffer definition

Directly after buffer definition the complete buffer is empty (card to PC) or completely filled (PC to card). In our example we have a notify size which is 1/4 of complete buffer memory to keep the example simple. In real world use it is recommended to set the notify size to a smaller stepsize.



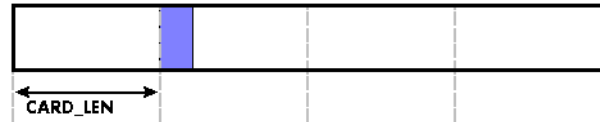
Step 2: Start and first data available

In between we have started the transfer and have waited for the first data to be available for the user. When there is at least one block of notify size in the memory we get an interrupt and can proceed with the data. Any data that already was transferred is announced. The USER_POS is still zero as we are right at the beginning of the complete transfer.

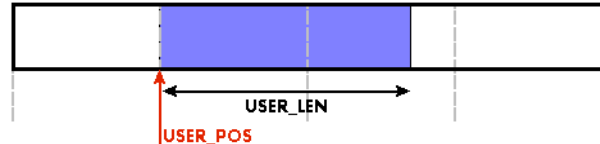


Step 3: set the first data available for card

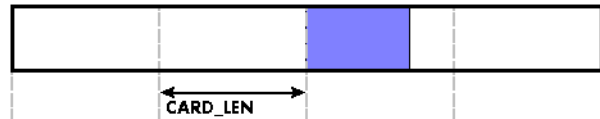
Now the data can be processed. If transfer is going from card to PC that may be storing to hard disk or calculation of any figures. If transfer is going from PC to card that means we have to fill the available buffer again with data. After the amount of data that has been processed by the user application we set it available for the card and for the next step.

**Step 4: next data available**

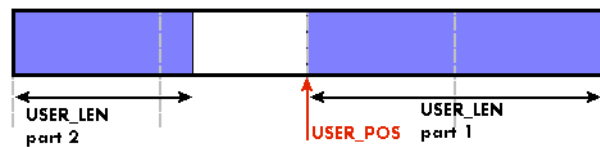
After reaching the next border of the notify size we get the next part of the data buffer to be available. In our example at the time when reading the USER_LEN even some more data is already available. The user position will now be at the position of the previous set CARD_LEN.

**Step 5: set data available again**

Again after processing the data we set it free for the card use. In our example we now make something else and don't react to the interrupt for a longer time. In the background the buffer is filled with more data.

**Step 6: roll over the end of buffer**

Now nearly the complete buffer is filled. Please keep in mind that our current user position is still at the end of the data part that we processed and marked in step 4 and step 5. Therefore the data to process now is split in two parts. Part 1 is at the end of the buffer while part 2 is starting with address 0.

**Step 7: set the rest of the buffer available**

Feel free to process the complete data or just the part 1 until the end of the buffer as we do in this example. If you decide to process complete buffer please keep in mind the roll over at the end of the buffer.



This buffer handling can now continue endless as long as we manage to set the data available for the card fast enough. The USER_POS and USER_LEN for step 8 would now look exactly as the buffer shown in step 2.

Buffer handling example for transfer from card to PC (Data acquisition)

```
int8* pcData = (int8*) pvAllocMemPageAligned (llBufferSizeInBytes);

// we now define the transfer buffer with the minimum notify size of one page = 4 kByte
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 4096, (void*) pcData, 0, llBufferSizeInBytes);

// we start the DMA transfer
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA);

do
{
    if (!dwError)
    {
        // we wait for the next data to be available. After this call we get at least 4k of data to proceed
        dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_WAITDMA);

        // if there was no error we can proceed and read out the available bytes that are free again
        spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_LEN, &llAvailBytes);
        spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_POS, &llBytePos);

        printf ("We now have %lld new bytes available\n", llAvailBytes);
        printf ("The available data starts at position %lld\n", llBytePos);

        // we take care not to go across the end of the buffer, handling the wrap-around
        if ((llBytePos + llAvailBytes) >= llBufferSizeInBytes)
            llAvailBytes = llBufferSizeInBytes - llBytePos;

        // our do function gets a pointer to the start of the available data section and the length
        vDoSomething (&pcData[llBytePos], llAvailBytes);

        // the buffer section is now immediately set available for the card
        spcm_dwSetParam_i64 (hDrv, SPC_DATA_AVAIL_CARD_LEN, llAvailBytes);
    }
}
while (!dwError); // we loop forever if no error occurs
```

Buffer handling example for transfer from PC to card (Data generation)

```
int8* pcData = (int8*) pvAllocMemPageAligned (llBufferSizeInBytes);

// before starting transfer we first need to fill complete buffer memory with meaningful data
vDoGenerateData (&pcData[0], llBufferSizeInBytes);

// we now define the transfer buffer with the minimum notify size of one page = 4 kByte
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_PCTOCARD, 4096, (void*) pcData, 0, llBufferSizeInBytes);

// and transfer some data to the hardware buffer before the start of the card
spcm_dwSetParam_i32 (hDrv, SPC_DATA_AVAIL_CARD_LEN, llBufferSizeInBytes);
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

do
{
    if (!dwError)
    {
        // if there was no error we can proceed and read out the current amount of available data
        spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_LEN, &llAvailBytes);
        spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_POS, &llBytePos);

        printf ("We now have %lld free bytes available\n", llAvailBytes);
        printf ("The available data starts at position %lld\n", llBytePos);

        // we take care not to go across the end of the buffer, handling the wrap-around
        if ((llBytePos + llAvailBytes) >= llBufferSizeInBytes)
            llAvailBytes = llBufferSizeInBytes - llBytePos;

        // our do function gets a pointer to the start of the available data section and the length
        vDoGenerateData (&pcData[llBytePos], llAvailBytes);

        // now we mark the number of bytes that we just generated for replay
        // and wait for the next free buffer
        spcm_dwSetParam_i64 (hDrv, SPC_DATA_AVAIL_CARD_LEN, llAvailBytes);
        dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_WAITDMA);
    }
}
while (!dwError); // we loop forever if no error occurs
```

Please keep in mind that you are using a continuous buffer writing/reading that will start again at the zero position if the buffer length is reached. However the DATA_AVAIL_USER_LEN register will give you the complete amount of available bytes even if one part of the free area is at the end of the buffer and the second half at the beginning of the buffer.



Data organisation

Data is organized in little-endian format in the transfer buffer, so that when having 16 channels or 32 channels active, the bytes will be ordered as mentioned below:

Sample width	Samples ordering in buffer memory starting with data offset zero																			
8 bit	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19
16 bit	A0	B0	A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	A8	B8	A9	B9
32 bit	A0	B0	C0	D0	A1	B1	C1	D1	A2	B2	C2	D2	A3	B3	C3	D3	A4	B4	C4	D4

The samples are re-named for better readability:

- A0: bits (D7..D0) of sample 0
- B0: bits (D15..D8) of sample 0
- C2: bits (D23..D16) of sample 2
- D4: bits (D31..D24) of sample 4

Clock generation

Overview

The digital Spectrum M4i PCI Express (PCIe) and M4x PXI Express (PXIe) cards offer a wide variety of different clock modes to match all the customers' needs.

Special focus has been given on achieving proper data capture when using external state clock, incorporating a programmable clock delay.

All of the clock modes are described in detail with programming examples in this chapter.

The figure is showing an overview of the complete engine used for clock generation on the digital M4i or M4x cards.

The purpose of this chapter is to give you a guide to the best matching clock settings for your specific application and needs.

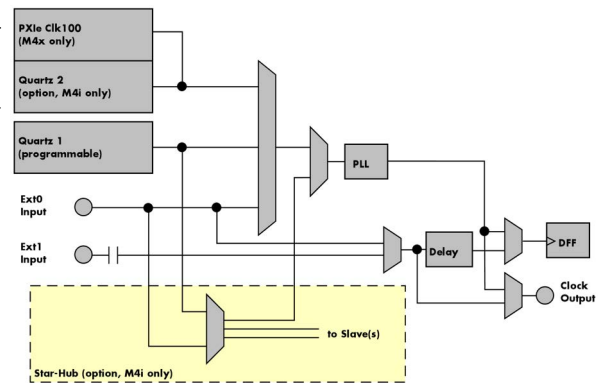


Image 44: M4i/M4x clock section overview

Clock Mode Register

The selection of the different clock modes has to be done by the SPC_CLOCKMODE register. All available modes, can be read out by the help of the SPC_AVAILCLOCKMODES register.

Table 49: Spectrum API: clock mode register and available clock modes

Register	Value	Direction	Description
SPC_AVAILCLOCKMODES	20201	read	Bitmask, in which all bits of the below mentioned clock modes are set, if available.
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode or reads out the actual selected one.
SPC_CM_INTPLL	1		Enables internal programmable high precision Quartz 1 for sample clock generation
SPC_CM_QUARTZ2	4		Enables optional Quartz 2 as reference for sample clock generation
SPC_CM_EXTREFCLOCK	32		Enables internal PLL with external reference for sample clock generation
SPC_CM_PXIREFCLOCK	64		M4x cards only: Enables internal PLL with PXIe backplane clock as reference for sample clock generation
SPC_CM_EXTERNAL0	8		Use external Ext0 clock DC-coupled input (either differential or single-ended, depending on card type) for direct clocking (state clock), that can contain gaps.
SPC_CM_EXTERNAL1	1024		Use external Ext1 clock AC-coupled differential input for direct clocking (state clock).

The different clock modes and all other related or required register settings are described on the following pages.

The different clock modes

Standard internal sample rate (programmable reference quartz 1)

This is the easiest and most common way to generate a sample rate with no need for additional external clock signals. The sample rate has a very fine resolution, low jitter and a high accuracy. The Quartz 1 is a high quality software programmable clock device acting as a reference to the internal PLL. The specification is found in the technical data section of this manual.

Quartz2 with PLL (option, M4i cards only)

This optional second Quartz 2 is for special customer needs, either for a special direct sampling clock or as a very precise reference for the PLL. Please feel free to contact Spectrum for your special needs. The Quartz 2 clock footprint can be equipped with a wide variety of clock sources that are available on the market.

External Reference Clock (Ext0)

Any clock can be fed in that matches the specification of the board. The external clock signal can be used to synchronize the board on a system clock or to feed in an exact matching sample rate. The external clock is divided/multiplied using a PLL allowing a wide range of external clock modes.

Direct external DC coupled clock (Ext0, state clock)

Any clock can be fed in that matches the specification of the board. The external clock signal can be used to synchronize the board on a system clock or to feed in an data clock (state clock) to synchronously sample data.

Direct external AC coupled clock (Ext1, state clock)

Any clock can be fed in that matches the specification of the board. The external clock signal can be used to synchronize the board on a system clock or to feed in an data clock (state clock) to synchronously sample data.

PXIe Reference Clock (M4x cards only)

The PXIe reference clock is a 100 MHz high-quality differential clock signal with an accuracy of ± 100 ppm or better. This reference clock is located on the PXIe backplane and is routed to every PXIe slot with the same trace length on the mainboard's PCB. PXIe cards from Spectrum

are able to use the PXIe reference clock for sampling clock generation. One big advantage of using the reference clock is the fact that all cards that are synchronized to the reference clock are running with the same clock frequency.

Synchronization Clock (option Star-Hub, M4i cards only)

The star-hub option allows the synchronization of up to 8 cards of the M4i series from Spectrum with a minimal phase delay between the different cards. The clock is distributed from the master card to all connected cards. As a source it is possible to either use the programmable Quartz 1 clock or the external Ext0 reference clock input of the master card. For details on the synchronization option please take a look at the dedicated chapter later in this manual.

Details on the different clock modes

Standard internal sampling clock (PLL)

The internal sampling clock is generated in default mode by a programmable high precision quartz. You need to select the clock mode by the dedicated register shown in the table below:

Table 50: Spectrum API: clock mode register and internal clock mode

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_INTPLL	1		Enables internal programmable high precision Quartz 1 for sample clock generation

The user does not have to care about how the desired sampling rate is generated by multiplying and dividing internally. You simply write the desired sample rate to the according register shown in the table below and the driver makes all the necessary calculations. If you want to make sure the sample rate has been set correctly you can also read out the register and the driver will give you back the sampling rate that is matching your desired one best.

Table 51: Spectrum API: samplerate register

Register	Value	Direction	Description
SPC_SAMPLERATE	20000	write	Defines the sample rate in Hz for internal sample rate generation.
		read	Read out the internal sample rate that is nearest matching to the desired one.

Independent of the used clock source it is possible to enable the clock output. The clock will be available on the external clock output connector and can be used to synchronize external equipment with the board.

Table 52: Spectrum API: clock output and clock output frequency register

Register	Value	Direction	Description
SPC_CLOCKOUT	20110	read/write	Writing a „1“ enables clock output on external clock output connector. Writing a „0“ disables the clock output (tristate)
SPC_CLOCKOUTFREQUENCY	20111	read	Allows to read out the frequency of an internally synthesized clock present at the clock output.

Example on writing and reading internal sampling rate

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_INTPLL); // Enables internal programmable quartz 1
spcm_dwSetParam_i64 (hDrv, SPC_SAMPLERATE, 62500000); // Set internal sampling rate to 62.5 MHz
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKOUT, 1); // enable the clock output of the card
spcm_dwGetParam_i64 (hDrv, SPC_SAMPLERATE, &lSamplerate); // Read back the programmed sample rate and print
printf („Sample rate = %d\n“, lSamplerate); // it. Output should be „Sample rate = 62500000“
```

Minimum internal sampling rate

The minimum and the maximum internal sampling rates depend on the specific type of board. Both values can be found in the technical data section of this manual.

Oversampling

All fast instruments have a minimum clock frequency that is limited by either the manufacturer limit of the used A/D converter or by limiting factors of the clock design. You find this minimum sampling rate specified in the technical data section as minimum native ADC converter clock.

When using one of the above mentioned internal clock modes the driver allows you to program sampling clocks that lie far beneath this minimum sampling clock. To run the instrument properly we use a special oversampling mode where the A/D converter/clock section is within its specification and only the digital part of the card is running with the slower clock. This is completely defined inside the driver and cannot be modified by the user. The following register allows to read out the oversampling factor for further calculation

Table 53: Spectrum API: clock oversampling readout register

Register	Value	Direction	Description
SPC_OVERSAMPLINGFACTOR	200123	read only	Returns the oversampling factor for further calculations. If oversampling isn't active a 1 is returned.

When using clock output the sampling clock at the output connector is the real instrument sampling clock and not the programmed slower sampling rate. To calculate the output clock, please just multiply the programmed sampling clock with the oversampling factor read with the above mentioned register.



External clock (reference clock)

The external clock input is fed through a PLL to the clock system. Therefore the input will act as a reference clock input thus allowing to either use a copy of the external clock or to generate any sampling clock within the allowed range from the reference clock. Please note the limited setup granularity in comparison to the internal sampling clock generation. Details are found in the technical data section.

Table 54: Spectrum API: clock mode register and external reference clock setup

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_EXTREFCLOCK	32		Enables internal PLL with external reference for sample clock generation

Due to the fact that the driver needs to know the external fed in frequency for an exact calculation of the sampling rate you must set the SPC_REFERENCECLOCK register accordingly as shown in the table below. The driver then automatically sets the PLL to achieve the desired sampling rate. Please be aware that the PLL has some internal limits and not all desired sampling rates may be reached with every reference clock.

Table 55: Spectrum API: reference clock register and available settings

Register	Value	Direction	Description
SPC_REFERENCECLOCK	20140	read/write	Programs the external reference clock in the range stated in the technical data section.
	External sampling rate in Hz as an integer value		You need to set up this register exactly to the frequency of the external fed in clock.

Example of reference clock:

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_EXTREFCLOCK); // Set to reference clock mode
spcm_dwSetParam_i32 (hDrv, SPC_REFERENCECLOCK, 10000000); // Reference clock that is fed in is 10 MHz
spcm_dwSetParam_i64 (hDrv, SPC_SAMPLERATE, 65200000); // We want to have 62.5 MHz as sampling rate
```

It is recommended that the sampling clock is always a multiple of the reference clock. If the sampling clock is a division of the reference clock, the clock starting phase is undetermined and may change between resets or clock configuration changes.



PLL Locking Error

The external clock signal is routed to a PLL to generate any sampling clock from this external clock. Due to the internal structure of the card the PLL is even used if a copy of the clock fed in externally is used for sampling (SPC_REFERENCECLOCK = SPC_SAMPLERATE). The PLL needs a stable and defined external clock with no gaps and no variation in the frequency. The external clock must be present when issuing the start command. It is not possible to start the card with external clock activated and no external clock available.

When starting the card all settings are written to hardware and the PLL is programmed to generate the desired sampling clock. If there has been any change to the clock setting the PLL then tries to lock on the external clock signal to generate the sampling clock. This locking will normally need 10 to 20 ms until the sampling clock is stable. Some clock settings may also need 200 ms to lock the PLL. This waiting time is automatically added at card start.

However if the PLL can not lock on the external clock either because there is no clock available or it hasn't sufficient signal levels or the clock is not stable the driver will return with an error code ERR_CLOCKNOTLOCKED. In that case it is necessary to check the external clock connection. Please see the example below:

```
// settings done to external clock like shown above.
if (spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER) == ERR_CLOCKNOTLOCKED)
{
    printf („External clock not locked. Please check connection\n");
    return -1;
}
```

Independent of the used clock source it is possible to enable the clock output. The clock will be available on the external clock output connector and can be used to synchronize external equipment with the board.

Table 56: Spectrum API: clock output and clock output frequency register

Register	Value	Direction	Description
SPC_CLOCKOUT	20110	read/write	Writing a „1“ enables clock output on external clock output connector. Writing a „0“ disables the clock output (tristate)
SPC_CLOCKOUTFREQUENCY	20111	read	Allows to read out the frequency of an internally synthesized clock present at the clock output.

PXI Reference Clock (M4x cards only)

Table 57: Spectrum API: clock mode register and PXI reference clock usage

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_PXIREFCLOCK	64		Enables internal PLL with PXI reference for sample clock generation

The 100 MHz PXIe system reference clock can be used as a reference clock for internal sample rate generation on all M4x PXIe cards from Spectrum. With the above mentioned software command the PXIe reference clock is routed to the internal PLL. Afterwards you only have to program the sample rate register to the desired sampling rate. The remaining internal calculations will be automatically done by the driver.

Example of PXI reference clock:

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_PXIREFCLOCK); // Set to PXI reference clock mode
spcm_dwSetParam_i64 (hDrv, SPC_SAMPLERATE, 65200000); // We want to have 62.5 MHz as sampling rate
```

PLL Locking Error

The PXI reference signal is routed to a PLL to generate any sampling clock from this external clock. The PLL needs a stable and defined external clock with no gaps and no variation in the frequency. Some backplanes might allow to turn off the reference clock. The PXI clock must be present when issuing the start command. It is not possible to start the card with external clock activated and no external clock available.

When starting the card all settings are written to hardware and the PLL is programmed to generate the desired sampling clock. If there has been any change to the clock setting the PLL then tries to lock on the external clock signal to generate the sampling clock. This locking will normally need 10 to 20 ms until the sampling clock is stable. Some clock settings may also need 200 ms to lock the PLL. This waiting time is automatically added at card start.

However if the PLL can not lock on the PXI clock because there is no clock available (if however disabled on the backplane), the driver will return with an error code ERR_CLOCKNOTLOCKED. In that case it is necessary to check the external clock connection. Please see the example below:

```
// settings done to PXI clock like shown above.
if (spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER) == ERR_CLOCKNOTLOCKED)
{
    printf („External clock not locked. Please check connection\n");
    return -1;
}
```

Direct external clock (state clock)

An external clock can be fed in on one of the external clock connectors of the board. This can be any clock, that matches the specification of the card. The external clock signal can be used to synchronize the board on a system clock or to feed in an data clock (state clock) to synchronously sample data.

Table 58: Spectrum API: direct external clock mode register settings

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_EXTERNAL0	8		Use external Ext0 clock DC-coupled input (either differential or single-ended, depending on card type) for direct clocking (state clock), that can contain gaps.
SPC_CM_EXTERNAL1	1024		Use external Ext1 clock AC-coupled differential input for direct clocking (state clock).

The maximum values for the external clock is board dependant and stated in the technical data section.

When using direct external clock in as a state-clock for synchronously sampling data, synchronization via Star-Hub is not possible. For using external clock with Star-Hub reference clock must be used instead.



Clock Delay (for state clock)

The state clock that is used to clock in the data to the capture flip flops can be delayed in very fine steps to ease synchronous data capture. The clock delay can be defined using the following register:

Table 59: Spectrum API: programmable clock delay registers

Register	Value	Direction	Description
SPC_CLOCK_AVAIDELAY_MIN	20220	read	Read out the minimum additional delay available in ps (pico seconds).
SPC_CLOCK_AVAIDELAY_MAX	20221	read	Read out the maximum additional delay available in ps (pico seconds).
SPC_CLOCK_AVAIDELAY_STEP	20222	read	Read out the step width in which the delay can be set.
SPC_CLOCK_DELAY	20223	read/write	Sets the additional external clock delay.

Value in ps (pico seconds).

Limits and interval steps can be read out by the above registers.

Example of clock delay:

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_EXTERNAL0); // Set to external clock mode (state clock)
spcm_dwSetParam_i64 (hDrv, SPC_CLOCK_DELAY, 600);           // additionally delay clock by 600 ps
```

Clock Edge Selection

The clock that is used to clock in the data to the capture flip flops can be chosen to capture the data only on the rising edge, the falling edge or both edges:

Table 60: Spectrum API: clock edge selection registers

Register			
SPC_AVAILOCKEDGES	20224	read	Bitmask, in which all bits of the below mentioned clock edges are set, if available.
SPC_CLOCK_EDGE	20225	read/write	Defines the used clock edge or reads out the actual selected one.
SPCM_EDGE_FALLING	1		Sample data on falling edge of external clock.
SPCM_EDGE_RISING	2		Sample data on rising edge of external clock.
SPCM_EDGES_BOTH	4		Sample data on both edges (rising and falling) of external clock signal. (DDR = Double Data Rate) clocking.

Example of clock edge selection:

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_EXTERNAL0); // Set to external clock mode (state clock)
spcm_dwSetParam_i32 (hDrv, SPC_CLOCK_EDGE, SPCM_EDGE_FALLING); // capture data on falling edge
```

Setting up Ext0 clock threshold and termination (single-ended models only)

Programmable threshold

For the single-ended Ext0 clock input the threshold can be defined, to accomodate for different logic level standards:

Table 61: Spectrum API: clock threshold register

Register	Value	Direction	Description	Offset range
SPC_CLOCK_THRESHOLD	42410	read/write	Defines the input threshold level for Ext0 clock input	0.. 4000 mV in steps of 10 mV

Programmable input termination

For the single-ended Ext0 clock input the termination can also be defined:

Table 62: Spectrum API: clock programmable termination register

Register	Value	Direction	Description
SPC_CLOCK75OHM	20120	read/write	A „1“ sets the 75 Ω termination. A „0“ sets the termination to 4.7 k Ω .

As an example define threshold for LVTTTL compatibility and enable 75 Ohm termination:

```
spcm_dwSetParam_i64 (hDrv, SPC_CLOCK_THRESHOLD, 1500); // set Ext0 clock threshold to 1500 mV
spcm_dwSetParam_i64 (hDrv, SPC_CLOCK75OHM, 1); // set to 75 Ohm
```


Trigger modes and appendant registers

General Description

The trigger modes of the Spectrum M4i/M4x series digital input cards are very extensive and give you the possibility to detect nearly any trigger event you can think of.

You can choose between external trigger modes and internal trigger modes including software and pattern trigger. This chapter is about to explain all of the different trigger modes and setting up the card's registers for the desired mode.

Trigger Engine Overview

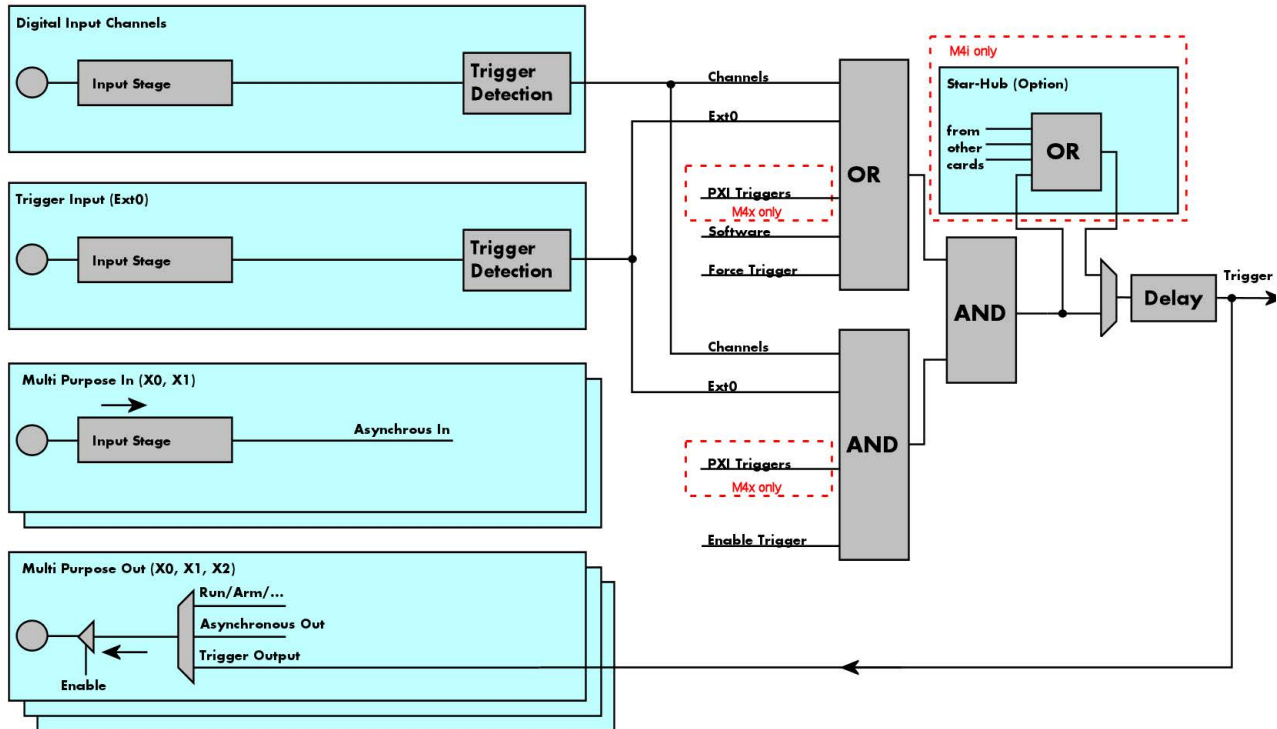


Image 45: Trigger Engine Overview. Red marked parts not available on all card types

The trigger engine of the M4i/M4x digital input card series allows to combine several different trigger sources with OR and AND combination, with a trigger delay or even with an OR combination across several cards when using the Star-Hub option. The above drawing gives a complete overview of the trigger engine and shows all possible features that are available.

The type of the „Input Stage“ is depending upon the card's input type, either differential or single-ended. This trigger chapter does make a difference here for simplicity and treats all inputs, be it the external trigger, the data lines or the multi-purpose inputs identically as „general logic inputs“. For details on the different input features, please see the dedicated chapter in this manual. All card types have a total of 33 different additional trigger sources: 32 data lines as well as one external trigger input. Additionally there are two multi purpose inputs and three multi-purpose outputs that can be software programmed to either asynchronous inputs or to output some extended status signals.

The Enable trigger allows the user to enable or disable all trigger sources (including channel/pattern trigger and external trigger) with a single software command. The enable trigger command will not work on force trigger.

When the card is waiting for a trigger event, either a channel trigger or an external trigger the force trigger command allows to force a trigger event with a single software command. The force trigger overrides the enable trigger command.

Before the trigger event is finally generated, it is wired through a programmable trigger delay. This trigger delay will also work when used in a synchronized system thus allowing each card to individually delay its trigger recognition.

Trigger masks

Trigger OR mask

The purpose of this passage is to explain the trigger OR mask (see left figure) and all the appendant software registers in detail.

The OR mask shown in the overview before as one object, is separated into two parts: a general OR mask for external trigger, the various PXI triggers (available on M4x PXIe cards only) and software trigger and a channel OR mask.

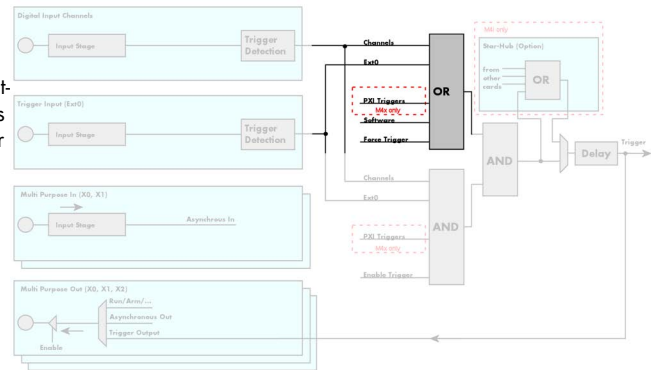


Image 46: trigger engine overview with trigger OR mask shown

Every trigger source of the M4i/M4x series cards is wired to one of the above mentioned OR masks. The user then can program which trigger source will be recognized, and which one won't.

This selection for the general mask is realized with the SPC_TRIG_ORMASK register in combination with constants for every possible trigger source.

This selection for the channel mask (pattern trigger) is realized with the SPC_TRIG_CH_ORMASK0 register in combination with constants for every possible channel trigger source.

In either case the sources are coded as a bitfield, so that they can be combined by one access to the driver with the help of a bitwise OR.

If no input is enabled, the output will be a logic "true", to not block the following static AND mask.

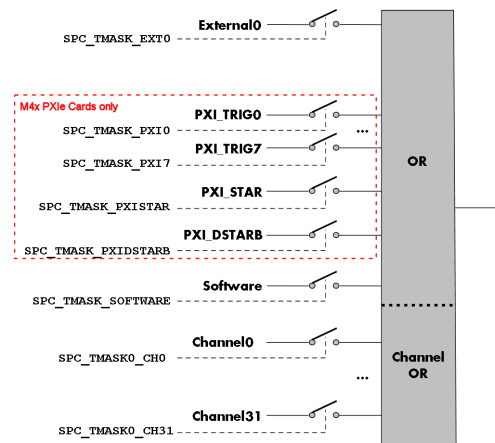


Image 47: trigger engine OR mask details

The table below shows the relating register for the general OR mask and the possible constants that can be written to it.

Table 63: Spectrum API: general trigger OR mask register and available settings

Register	Value	Direction	Description
SPC_TRIG_AVAILORMASK	40400	read	Bitmask, in which all bits of the below mentioned sources for the OR mask are set, if available.
SPC_TRIG_ORMASK	40410	read/write	Defines the events included within the trigger OR mask of the card.
SPC_TMASK_NONE	0		No trigger source selected
SPC_TMASK_SOFTWARE	1h		Enables the software trigger for the OR mask. The card will trigger immediately after start.
SPC_TMASK_EXT0	2h		Enables the external (analog window) trigger 0 for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI0	100000h		Enables the PXI_TRIG0 for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI1	200000h		Enables the PXI_TRIG1 for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI2	400000h		Enables the PXI_TRIG2 for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI3	800000h		Enables the PXI_TRIG3 for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI4	1000000h		Enables the PXI_TRIG4 for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI5	2000000h		Enables the PXI_TRIG5 for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI6	4000000h		Enables the PXI_TRIG6 for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI7	8000000h		Enables the PXI_TRIG7 for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXISTAR	10000000h		Enables the PXISTAR line for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXIDSTARB	20000000h		Enables the PXI_DSTARB for the OR mask. The card will trigger when the programmed condition for this input is valid.



Please note that as default the SPC_TRIG_ORMASK is set to SPC_TMASK_SOFTWARE. When not using any trigger mode requiring values in the SPC_TRIG_ORMASK register, this mask should explicitly cleared, as otherwise the software trigger will override other modes.

The following example shows, how to setup the OR mask for the external trigger input. As an example an edge detection has been chosen for Ext0 input. The explanation and a detailed description of the different trigger modes for the external trigger input will be shown in the dedicated passage within this chapter.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_POS); // Setting up external trigger for rising edges

// Enable external trigger within the OR mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0);
```

The table below is showing the registers for the channel OR mask (for pattern matching) and the possible constants that can be written to it.

Table 64: Spectrum API: channel trigger OR mask registers and available settings

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILORMASK0	40450	read	Bitmask, in which all bits of the below mentioned sources/channels (0...31) for the channel OR mask are set, if available.
SPC_TRIG_CH_ORMASK0	40460	read/write	Includes the digital data channels (0...31) within the channel trigger OR mask of the card.
SPC_TMASK0_CH0	00000001h		Enables channel0 for recognition within the channel OR mask.
SPC_TMASK0_CH1	00000002h		Enables channel1 for recognition within the channel OR mask.
SPC_TMASK0_CH2	00000004h		Enables channel2 for recognition within the channel OR mask.
SPC_TMASK0_CH3	00000008h		Enables channel3 for recognition within the channel OR mask.
...
SPC_TMASK0_CH28	10000000h		Enables channel28 for recognition within the channel OR mask.
SPC_TMASK0_CH29	20000000h		Enables channel29 for recognition within the channel OR mask.
SPC_TMASK0_CH30	40000000h		Enables channel30 for recognition within the channel OR mask.
SPC_TMASK0_CH31	80000000h		Enables channel31 for recognition within the channel OR mask.

The following example shows, how to setup the OR mask for channel trigger. As an example a simple edge detection has been chosen. The explanation and a detailed description of the different trigger modes for the channel trigger modes will be shown in the dedicated passage within this chapter.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_NONE); // disable default software trigger
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ORMASK0, SPC_TMASK_CH0); // Enable channel0 trigger within the OR mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE, SPC_TM_POS); // Setting up channel0 trigger for rising edges
```

Trigger AND mask

The purpose of this passage is to explain the trigger AND mask (see left figure) and all the appendant software registers in detail.

The AND mask shown in the overview before as one object, is separated into two parts: a general AND mask for external trigger and software trigger and a channel AND mask.

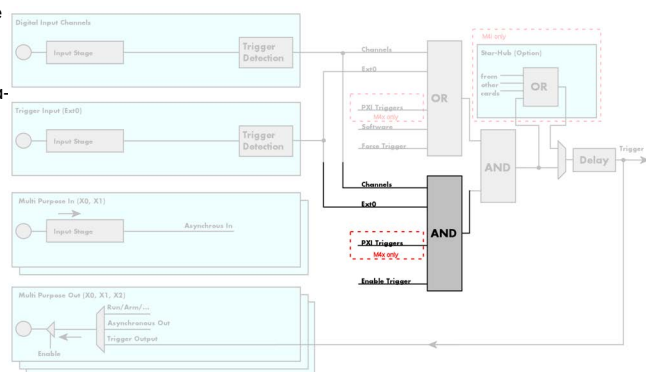


Image 48: trigger engine overview with trigger AND mask shown

Every trigger source of the M4i/M4x series cards except the software trigger is wired to one of the above mentioned AND masks. The user then can program which trigger source will be recognized, and which one won't.

This selection for the general mask is realized with the SPC_TRIG_ANDMASK register in combination with constants for every possible trigger source.

This selection for the channel mask (for pattern matching) is realized with the SPC_TRIG_CH_ANDMASK0 register in combination with constants for every possible channel trigger source.

In either case the sources are coded as a bitfield, so that they can be combined by one access to the driver with the help of a bitwise OR.

If no input is enabled, the output will be a logic "true", to not block the following static AND mask.

The table below shows the relating register for the general AND mask and the possible constants that can be written to it.

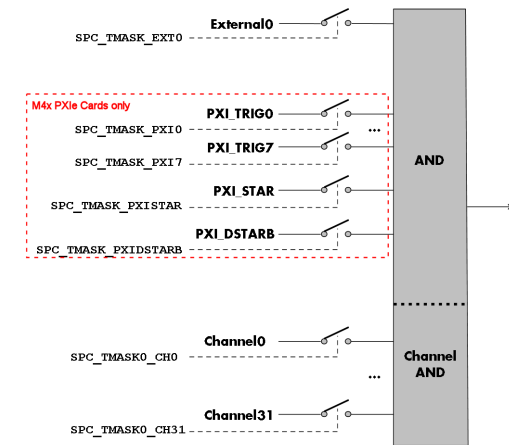


Image 49: trigger engine AND mask details

Table 65: Spectrum API: general trigger AND mask registers and available settings

Register	Value	Direction	Description
SPC_TRIG_AVAILANDMASK	40420	read	Bitmask, in which all bits of the below mentioned sources for the AND mask are set, if available.
SPC_TRIG_ANDMASK	40430	read/write	Defines the events included within the trigger AND mask of the card.
SPC_TMASK_NONE	0		No trigger source selected
SPC_TMASK_EXT0	2h		Enables the external (analog window) trigger 0 for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI0	100000h		Enables the PXI_TRIG0 for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI1	200000h		Enables the PXI_TRIG1 for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI2	400000h		Enables the PXI_TRIG2 for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI3	800000h		Enables the PXI_TRIG3 for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI4	1000000h		Enables the PXI_TRIG4 for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI5	2000000h		Enables the PXI_TRIG5 for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI6	4000000h		Enables the PXI_TRIG6 for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI7	8000000h		Enables the PXI_TRIG7 for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXISTAR	10000000h		Enables the PXISTAR line for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXIDSTARB	20000000h		Enables the PXI_DSTARB for the AND mask. The card will trigger when the programmed condition for this input is valid.

The following example shows, how to setup the AND mask, for an external trigger. As an example a simple high level detection has been chosen. When multiple external triggers shall be combined by AND, both of the external sources must be included in the AND mask register, similar to the OR mask example shown before. The explanation and a detailed description of the different trigger modes for the external trigger inputs will be shown in the dedicated passage within this chapter.

```

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_NONE); // disable default software trigger
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ANDMASK, SPC_TMASK_EXT0); // Enable external trigger within the AND mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_HIGH); // Setting up external trigger for HIGH level

```

The table below is showing the constants for the channel AND mask (for pattern matching) and all the constants for the different channels.

Table 66: Spectrum API: channel trigger AND mask registers and available settings

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILANDMASK0	40470	read	Bitmask, in which all bits of the below mentioned sources/channels (0...31) for the channel AND mask are set, if available.
SPC_TRIG_CH_ANDMASK0	40480	read/write	Includes the digital data channels (0...31) within the channel trigger AND mask of the card.
SPC_TMASK0_CH0	00000001h		Enables channel0 for recognition within the channel OR mask.
SPC_TMASK0_CH1	00000002h		Enables channel1 for recognition within the channel OR mask.
SPC_TMASK0_CH2	00000004h		Enables channel2 for recognition within the channel OR mask.
SPC_TMASK0_CH3	00000008h		Enables channel3 for recognition within the channel OR mask.
...
SPC_TMASK0_CH28	10000000h		Enables channel28 for recognition within the channel OR mask.
SPC_TMASK0_CH29	20000000h		Enables channel29 for recognition within the channel OR mask.
SPC_TMASK0_CH30	40000000h		Enables channel30 for recognition within the channel OR mask.
SPC_TMASK0_CH31	80000000h		Enables channel31 for recognition within the channel OR mask.

The following example shows, how to setup the AND mask for a pattern matching trigger. The explanation and a detailed description of the different trigger modes for the channel trigger modes will be shown in the dedicated passage within this chapter.

```

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_NONE); // disable default software trigger

// Include channels (3..0) into AND mask: All of the programmed conditions (aka Pattern) must be met
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ANDMASK0, SPC_TMASK_CH0); // Enable channel0 trigger within AND mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ANDMASK0, SPC_TMASK_CH1); // Enable channel1 trigger within AND mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ANDMASK0, SPC_TMASK_CH2); // Enable channel2 trigger within AND mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ANDMASK0, SPC_TMASK_CH3); // Enable channel3 trigger within AND mask

// Define Pattern to find a „1010“ = A (hexadecimal) on lowest nibble
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE, SPC_TM_LOW ); // Setting up ch0 trigger for LOW levels
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH1_MODE, SPC_TM_HIGH); // Setting up ch1 trigger for HIGH levels
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH2_MODE, SPC_TM_LOW ); // Setting up ch2 trigger for LOW levels
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH3_MODE, SPC_TM_HIGH); // Setting up ch3 trigger for HIGH levels

```

Software trigger

The software trigger is the easiest way of triggering any Spectrum board. The acquisition or replay of data will start immediately after the card is started and the trigger engine is armed. The resulting delay upon start includes the time the board needs for its setup and the time for recording the pre-trigger area (for acquisition cards).

For enabling the software trigger one simply has to include the software event within the trigger OR mask, as the following table is showing:



Table 67: Spectrum API: software register and register setting for software trigger

Register	Value	Direction	Description
SPC_TRIG_ORMASK	40410	read/write	Defines the events included within the trigger OR mask of the card.
SPC_TMASK_SOFTWARE	1h		Sets the trigger mode to software, so that the recording/replay starts immediately.

Example for setting up the software trigger:

```

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_SOFTWARE); // Internal software trigger mode is used

```

Force- and Enable trigger

In addition to the software trigger (free run) it is also possible to force a trigger event by software while the board is waiting for a real physical trigger event. The forcetrigger command will only have any effect, when the board is waiting for a trigger event. The command for forcing a trigger event is shown in the table below.

Issuing the forcetrigger command will every time only generate one trigger event. If for example using Multiple Recording that will result in only one segment being acquired by forcetrigger. After execution of the forcetrigger command the trigger engine will fall back to the trigger mode that was originally programmed and will again wait for a trigger event.

Table 68: Spectrum API: command register and force trigger command

Register	Value	Direction	Description
SPC_M2CMD	100	write	Command register of the M2i/M3i/M4i/M4x/M2p/M5i series cards.
M2CMD_CARD_FORCETRIGGER	10h		Forces a trigger event if the hardware is still waiting for a trigger event.

The example shows, how to use the forcetrigger command:

```
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_FORCETRIGGER); // Force trigger is used.
```

It is also possible to enable (arm) or disable (disarm) the card's whole triggerengine by software. By default the trigger engine is disabled.

Table 69: Spectrum API: command register and trigger enable/disable command

Register	Value	Direction	Description
SPC_M2CMD	100	write	Command register of the M2i/M3i/M4i/M4x/M2p/M5i series cards.
M2CMD_CARD_ENABLETRIGGER	8h		Enables the trigger engine. Any trigger event will now be recognized.
M2CMD_CARD_DISABLETRIGGER	20h		Disables the trigger engine. No trigger events will be recognized, except force trigger.

The example shows, how to arm and disarm the card's trigger engine properly:

```
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_ENABLETRIGGER); // Trigger engine is armed.
...
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_DISABLETRIGGER); // Trigger engine is disarmed.
```

Trigger delay

All of the Spectrum M4i/M4x series cards allow the user to program an additional trigger delay. As shown in the trigger overview section, this delay is the last element in the trigger chain. Therefore the user does not have to care for the sources when programming the trigger delay.

As shown in the overview the trigger delay is located after the star-hub connection meaning that every M4i card being synchronized can still have its own trigger delay programmed. The Star-Hub will combine the original trigger events before the result is being delayed.

The delay is programmed in samples. The resulting time delay will therefore be [Programmed Delay] / [Sampling Rate].

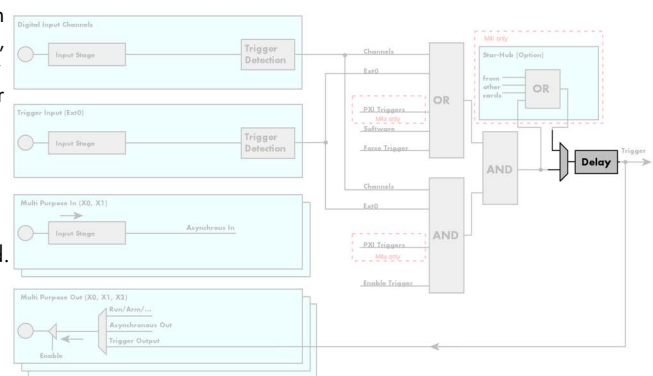


Image 50: trigger engine overview with marked trigger delay stage

The following table shows the related register and the possible values. A value of 0 disables the trigger delay.

Table 70: Spectrum API: trigger delay registers and available settings

Register	Value	Direction	Description
SPC_TRIG_AVAILDELAY	40800	read	Contains the maximum available delay as a decimal integer value.
SPC_TRIG_DELAY	40810	read/write	Defines the delay for the detected trigger events.
	0		No additional delay will be added. The resulting internal delay is mentioned in the technical data section.
	32...[8G-32] in steps of 32		Defines the additional trigger delay in number of sample clocks. The trigger delay can be programmed up to (8GSamples - 32) = 8589934560. Stepsize is 32 samples.

The example shows, how to use the trigger delay command:

```
spcm_dwSetParam_i64 (hDrv, SPC_TRIG_DELAY, 1984); // A detected trigger event will be
// delayed for 1984 sample clocks.
```

Using the delay trigger does not affect the ratio between pre trigger and post trigger recorded number of samples, but only shifts the trigger event itself. For changing these values, please take a look in the relating chapter about „Acquisition Modes“.



External trigger

The M4i/M4x card 77xx series has one external trigger input, that can be combined with the channel trigger (pattern trigger) modes explained later in this manual.

The external trigger can be easily combined with channel trigger (pattern trigger).

The programming of the masks is shown in the chapters above.

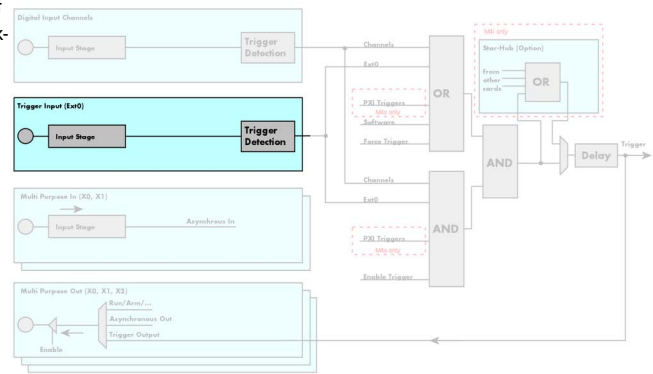


Image 51: trigger engine overview with marked main external trigger Ext0/Trg0

External Trigger Mode

Please find the external trigger input modes below. A detailed description of the modes follows in the next chapters..

Table 71: Spectrum API: external trigger Ext0 registers and register settings

Register	Value	Direction	Description
SPC_TRIG_EXT0_AVAILMODES	40500	read	Bitmask showing all available trigger modes for external 0 (Ext0).
SPC_TRIG_EXT0_MODE	40510	read/write	Defines the external trigger mode for the external trigger input. The trigger need to be added to either OR or AND mask input to be activated.
SPC_TM_NONE	0000000h		Channel is not used for trigger detection. This is as with the trigger masks another possibility for disabling channels.
SPC_TM_POS	00000001h		Trigger detection for positive edges
SPC_TM_NEG	00000002h		Trigger detection for negative edges
SPC_TM_BOTH	00000004h		Trigger detection for positive and negative edges
SPC_TM_HIGH	00000008h		Trigger detection for HIGH levels
SPC_TM_LOW	00000010h		Trigger detection for LOW levels

For all external edge and level trigger modes, the OR mask must contain the corresponding input, as the following table shows:

Table 72: Spectrum API: external trigger Ext0 OR mask settings

Register	Value	Direction	Description
SPC_TRIG_ORMASK	40410	read/write	Defines the OR mask for the different trigger sources.
SPC_TMASK_EXT0	2h		Enable external trigger input for the OR mask

Setting up the trigger Ext0 threshold and termination (single-ended models only)

Programmable threshold

For the single-ended Ext0 trigger input the threshold can be defined, to accomodate for different logic level standards:

Register	Value	Direction	Description	Offset range
SPC_TRIG_THRESHOLD	42411	read/write	Defines the input threshold level for Ext0 trigger input	0 to 4000 mV in steps of 10 mV

Programmable input termination

For the single-ended Ext0 trigger input the termination can also be defined:

Register	Value	Direction	Description
SPC_TRIGGER75OHM0	40110	read/write	A „1“ sets the 75 Ω termination. A „0“ sets the termination to 4.7 kΩ.

As an example define threshold for LVTTL compatibility and enable 4.7 kOhm termination:

```
spcm_dwSetParam_i64 (hDrv, SPC_TRIG_THRESHOLD, 1500); // set Ext0 trigger threshold to 1500 mV
spcm_dwSetParam_i64 (hDrv, SPC_TRIGGER75OHM0, 0); // set to 4.7 kOhm
```


Channel (pattern) trigger

The M4i/M4x card 77xx series can use all 32 data channels as a trigger source that additionally also can be combined with the external trigger explained earlier in this manual.

The channel or pattern trigger can use the same trigger modes as the external trigger input, separately selectable for every single data input.

The programming of the masks is shown in the chapters above.

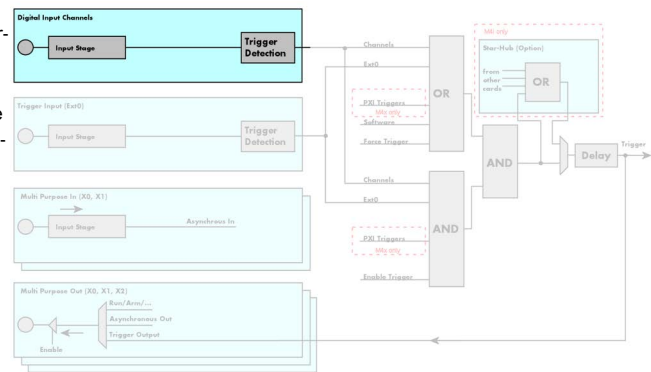


Image 52: trigger engine overview with pattern trigger marked

Channel Trigger Mode

Please find the channel trigger input modes below. A detailed description of the modes follows in the next chapters..

Table 73: Spectrum API: channel trigger mode registers and available register settings

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILMODES	40600	read	Bitmask, in which all bits of the below mentioned modes for the channel trigger are set.
SPC_TRIG_CH0_MODE	40610	read/write	Sets the trigger mode for channel 0, which also must be enabled in the channel OR or AND mask.
SPC_TRIG_CH1_MODE	40611	read/write	Sets the trigger mode for channel 1, which also must be enabled in the channel OR or AND mask.
SPC_TRIG_CH2_MODE	40612	read/write	Sets the trigger mode for channel 2, which also must be enabled in the channel OR or AND mask.
SPC_TRIG_CH3_MODE	40613	read/write	Sets the trigger mode for channel 3, which also must be enabled in the channel OR or AND mask.
...
SPC_TRIG_CH28_MODE	40638	read/write	Sets the trigger mode for channel 28, which also must be enabled in the channel OR or AND mask.
SPC_TRIG_CH29_MODE	40639	read/write	Sets the trigger mode for channel 29, which also must be enabled in the channel OR or AND mask.
SPC_TRIG_CH30_MODE	40640	read/write	Sets the trigger mode for channel 30, which also must be enabled in the channel OR or AND mask.
SPC_TRIG_CH31_MODE	40641	read/write	Sets the trigger mode for channel 31, which also must be enabled in the channel OR or AND mask.
SPC_TM_NONE	0000000h		Channel is not used for trigger detection. This is as with the trigger masks another possibility for disabling channels.
SPC_TM_POS	00000001h		Trigger detection for positive edges
SPC_TM_NEG	00000002h		Trigger detection for negative edges
SPC_TM_BOTH	00000004h		Trigger detection for positive and negative edges
SPC_TM_HIGH	00000008h		Trigger detection for HIGH levels
SPC_TM_LOW	00000010h		Trigger detection for LOW levels

The table below is showing the constants for the channel OR mask and the channel AND mask (for pattern matching) and all the constants for the different channels:

Table 74: Spectrum API: channel OR and AND mask registers and available register settings

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILORMASK0	40450	read	Bitmask, in which all bits of the below mentioned sources/channels (0...31) for the channel OR mask are set, if available.
SPC_TRIG_CH_AVAILANDMASK0	40470	read	Bitmask, in which all bits of the below mentioned sources/channels (0...31) for the channel AND mask are set, if available.
SPC_TRIG_CH_ORMASK0	40460	read/write	Includes the digital data channels (0...31) within the channel trigger OR mask of the card.
SPC_TRIG_CH_ANDMASK0	40480	read/write	Includes the digital data channels (0...31) within the channel trigger AND mask of the card.
SPC_TMASK0_CH0	00000001h		Enables channel0 for recognition within the respective channel mask.
SPC_TMASK0_CH1	00000002h		Enables channel1 for recognition within the respective channel mask.
SPC_TMASK0_CH2	00000004h		Enables channel2 for recognition within the respective channel mask.
SPC_TMASK0_CH3	00000008h		Enables channel3 for recognition within the respective channel mask.
...
SPC_TMASK0_CH28	10000000h		Enables channel28 for recognition within the respective channel mask.
SPC_TMASK0_CH29	20000000h		Enables channel29 for recognition within the respective channel mask.
SPC_TMASK0_CH30	40000000h		Enables channel30 for recognition within the respective channel mask.
SPC_TMASK0_CH31	80000000h		Enables channel31 for recognition within the respective channel mask.

Edge and level triggers

Rising edge trigger

This mode is for detecting the rising edges of a trigger source. The board will trigger on the first rising edge that is detected after starting the board. The next trigger event will then be detected, if the current recording has finished and the board is armed and waiting for a trigger again.

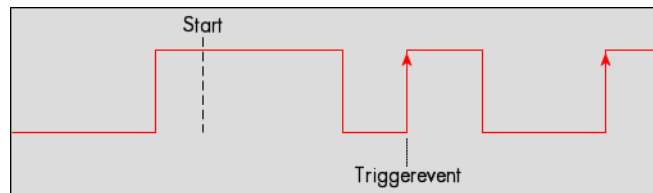


Table 75: Spectrum API: trigger mode registers and positive edge settings

Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the trigger mode for external trigger input.
SPC_TRIG_CH0_MODE	40610	read/write	Sets the trigger mode for channel 0.
...
SPC_TRIG_CH31_MODE	40641	read/write	Sets the trigger mode for channel 31.
SPC_TM_POS	1h		Sets the trigger mode to detect positive edges.

Example on how to set up the board for detecting a positive edge on the external trigger input:

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_POS); // Set up ext. trigger to detect positive edges
```

Example on how to set up the board for detecting a positive edge on data channel0 input:

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE, SPC_TM_POS); // Set up channel0 to detect positive edges
```

HIGH level trigger

This trigger mode detects the HIGH level of a trigger source. This can be used for level detection (pattern matching) or to generate an internal gate signal that can be very good used together with a second trigger mode to gate the trigger.

If using this mode as a single trigger source the card will detect a trigger event at the time when entering the HIGH level (acting like positive edge trigger) or if the trigger signal is already at high level at the start it will immediately detect a trigger event.

The trigger input is continuously sampled with the selected sample rate. The trigger event will be detected if the trigger input is at external HIGH level.

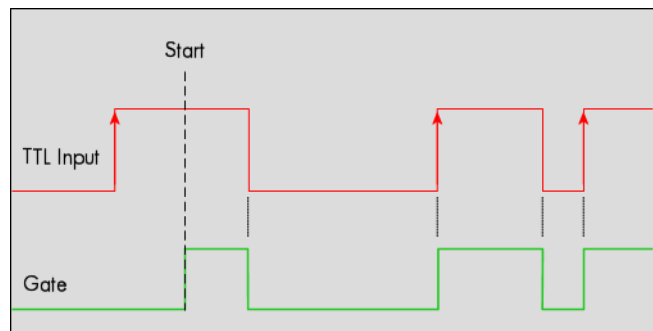


Table 76: Spectrum API: trigger mode registers and high level settings

Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the trigger mode for external trigger input.
SPC_TRIG_CH0_MODE	40610	read/write	Sets the trigger mode for channel 0.
...
SPC_TRIG_CH31_MODE	40641	read/write	Sets the trigger mode for channel 31.
SPC_TM_HIGH	8h		Sets the trigger mode to detect HIGH levels.

Negative edge trigger

This mode is for detecting the falling edges of a trigger source. The board will trigger on the first rising edge that is detected after starting the board. The next trigger event will then be detected, if the current recording has finished and the board is armed and waiting for a trigger again.

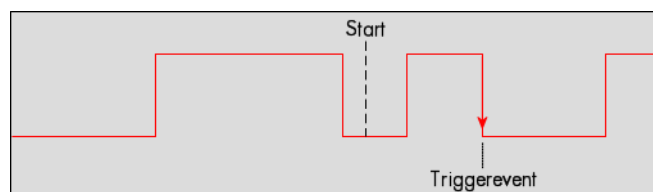


Table 77: Spectrum API: trigger mode registers and negative edge settings

Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the trigger mode for external trigger input.
SPC_TRIG_CH0_MODE	40610	read/write	Sets the trigger mode for channel 0.

Table 77: Spectrum API: trigger mode registers and negative edge settings

Register	Value	Direction	Description
...
SPC_TRIG_CH31_MODE	40641	read/write	Sets the trigger mode for channel 31.
SPC_TM_NEG	2h		Sets the trigger mode to detect negative edges.

LOW level trigger

This trigger mode detects the logic LOW level of a trigger source. This can be used for level detection (pattern matching) or to generate an internal gate signal that can be very good used together with a second trigger mode to gate the trigger.

If using this mode as a single trigger source the card will detect a trigger event at the time when entering the LOW level (acting like negative edge trigger) or if the trigger signal is already at LOW level at the start it will immediately detect a trigger event.

The trigger input is continuously sampled with the selected sample rate. The trigger event will be detected if the trigger input is at logic LOW level.

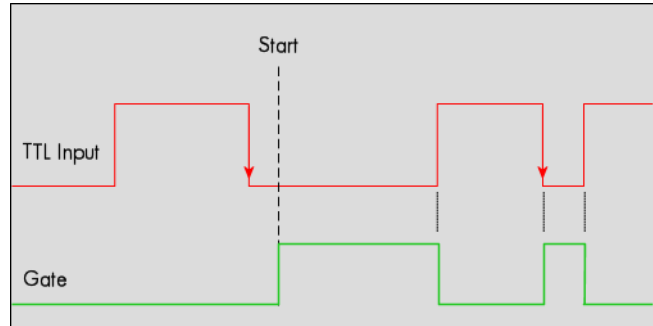


Table 78: Spectrum API: trigger mode registers and low level settings

Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the trigger mode for external trigger input.
SPC_TRIG_CH0_MODE	40610	read/write	Sets the trigger mode for channel 0.
...
SPC_TRIG_CH31_MODE	40641	read/write	Sets the trigger mode for channel 31.
SPC_TM_LOW	10h		Sets the trigger mode to detect LOW levels.

Positive and negative edge trigger (both edges)

This mode is for detecting the rising and falling edges of a trigger source. The board will trigger on the first rising or falling edge that is detected after starting the board. The next trigger event will then be detected, if the actual recording has finished and the board is armed and waiting for a trigger again.

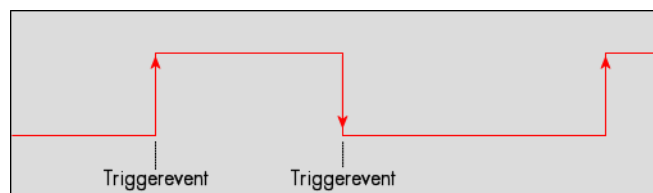


Table 79: Spectrum API: trigger mode registers and both edges settings

Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the trigger mode for external trigger input.
SPC_TRIG_CH0_MODE	40610	read/write	Sets the trigger mode for channel 0.
...
SPC_TRIG_CH31_MODE	40641	read/write	Sets the trigger mode for channel 31.
SPC_TM_BOTH	4h		Sets the trigger mode to detect positive and negative edges.

This mode for detecting both edges cannot be included in the AND masks, because it would be either always constantly valid (or always constantly false) and hence useless in either way being included within an AND condition.

**Examples for pattern matching****Simple pattern matching**

This is the most common trigger mode for digital signals used by logic analyzers.

You can define a pattern for a programmable number of bits and if this pattern occurs a trigger event will be detected.

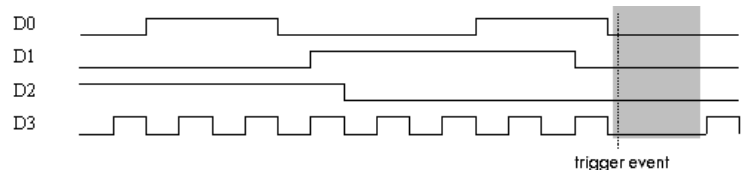


Image 53: single pattern matching timing diagram

The pattern trigger mode can be seen as the logical AND combination of several bits, that are set either to detect LOW or HIGH levels. Therefore the bits that should be used for pattern trigger detection must be enabled within the corresponding AND mask. In addition the trigger mode for each digital channel must be either set to detect a LOW or a HIGH level.

The following table shows the necessary registers:

Table 80: Spectrum API: example of trigger settings registers for pattern matching

Register	Value	Direction	set to	Value
SPC_TRIG_CH_ANDMASK0	40480	read/write	Bitfield that includes the digital channels (31...0) within the channel trigger AND mask of the card. Set channels to be included to 1 others 0.	Users choice
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_LOW, SPC_TM_HIGH according to desired pattern	Users choice
...
SPC_TRIG_CH31_MODE	40631	read/write		Users choice

The setup used in the software programming example is corresponding with the pattern shown in the figure.

```
for (i = 0; i < 4; i++)
    spcm_dwSetParem_i32 (hDrv, SPC_TRIG_CH0_MODE+i, SPC_TM_LOW); // defines zero pattern for lower 4 bits

spcm_dwSetParem_i32 (hDrv, SPC_TRIG_ANDMASK, 0x00000000); // disable all other sources like ext or sw
spcm_dwSetParem_i32 (hDrv, SPC_TRIG_ORMASK, 0x00000000); // ...
spcm_dwSetParem_i32 (hDrv, SPC_TRIG_CH_ANDMASK0, 0x0000000F); // Include lower 4 channels in AND mask
spcm_dwSetParem_i32 (hDrv, SPC_TRIG_CH_ORMASK0, 0x00000000); // Exclude all channels from the OR mask
```

Pattern and edge trigger

This trigger mode is similar to the simple pattern trigger mode, but with the addition of an edge detection.

You can define a pattern for a programmable number of bits and if this pattern occurs and then the programmed edge occurs on the one programmed bit, a trigger event is detected.

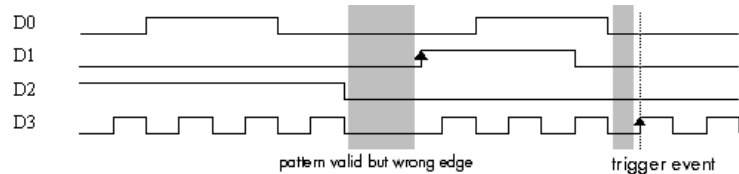


Image 54: pattern and edge matching timing diagram showing the edge detection

If the pattern is wrong, no trigger event will be detected.

Even if the pattern is right, but the edge is occurring either on the wrong bit with the right edge or on the right bit with the wrong edge no trigger event will be detected.

The pattern and edge trigger mode can be seen as the logical AND combination of several bits for the pattern, that are set either to detect LOW or HIGH levels. Therefore the bits that should be used for the pattern matching must be enabled within the corresponding AND mask. In addition the trigger mode for each digital channel must be either set to detect a LOW or a HIGH level.

The bits for the edge detection are enabled via the OR mask. Also the edge has to be programmed by setting the channel trigger mode of the dedicated bit to a proper edge detection mode.

The following table shows the necessary registers:

Table 81: Spectrum API: example of trigger settings registers for pattern and edge matching

Register	Value	Direction	set to	Value
SPC_TRIG_CH_ANDMASK0	40480	read/write	Bitfield that includes the digital channels (31...0) within the channel trigger AND mask of the card. Set channels to be included set to 1 others 0.	Users choice
SPC_TRIG_CH_ORMASK0	40460	read/write	Bitfield that includes the digital channels (31...0) within the channel trigger OR mask of the card. Set channel to be included to 1 others 0.	Users choice
SPC_TRIG_CH0_MODE	40610	read/write	Set channels for pattern detection to SPC_TM_LOW, SPC_TM_HIGH according to desired pattern and channels for edge detection to SPC_TM_POS, SPC_TM_NEG, SPC_TM_BOTH according to the desired edge.	Users choice
...
SPC_TRIG_CH31_MODE	40641	read/write		Users choice

The setup used in the software programming example is corresponding with the pattern shown in the figure.

```
for (i = 0; i < 3; i++)
    spcm_dwSetParem_i32 (hDrv, SPC_TRIG_CH0_MODE+i, SPC_TM_LOW); // defines zero pattern for channels D2..D0
    spcm_dwSetParem_i32 (hDrv, SPC_TRIG_CH3_MODE, SPC_TM_POS); // sets D3 to detect a rising edge

spcm_dwSetParem_i32 (hDrv, SPC_TRIG_ANDMASK, 0x00000000); // disable all other sources like ext or sw
spcm_dwSetParem_i32 (hDrv, SPC_TRIG_ORMASK, 0x00000000); // ...
spcm_dwSetParem_i32 (hDrv, SPC_TRIG_CH_ANDMASK0, 0x00000007); // Include lower 3 channels in AND mask
spcm_dwSetParem_i32 (hDrv, SPC_TRIG_CH_ORMASK0, 0x00000008); // Include channel 3 in the OR mask
```

Multi Purpose I/O Lines

On-board I/O lines (X0, X1, X2)

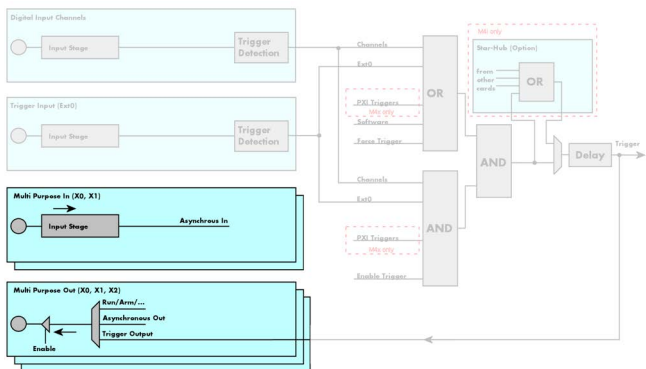


Image 55: trigger engine overview with multi-purpose trigger lines highlighted

Please be careful when wiring up external sources to these lines to not confuse the dedicated inputs and outputs on the connector. Connecting external sources to the card's outputs may damage components either on the external equipment or on the card itself.



Each multi purpose line can be individually programmed. Please check the available modes by reading the SPCM_X0_AVALIEMODES, SPCM_X1_AVALIEMODES and SPCM_X2_AVALIEMODES register first. The available modes may differ from card to card and may be enhanced with new driver/firmware versions to come.

Table 82: Spectrum API: multi-purpose registers and available register settings

Register	Value	Direction	Description
SPCM_X0_AVALIEMODES	47210	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X0)
SPCM_X1_AVALIEMODES	47211	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X1)
SPCM_X2_AVALIEMODES	47212	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X2)
SPCM_X0_MODE	47200	read/write	Defines the mode for (X0). Only one mode selection is possible to be set at a time
SPCM_X1_MODE	47201	read/write	Defines the mode for (X1). Only one mode selection is possible to be set at a time
SPCM_X2_MODE	47202	read/write	Defines the mode for (X2). Only one mode selection is possible to be set at a time
SPCM_XMODE_DISABLE	0000000h		No mode selected. Output is tristate (default setup)
SPCM_XMODE_ASYNCIN	00000001h		Connector is programmed for asynchronous input. Use SPCM_XX_ASYNCIO to read data asynchronous as shown in next chapter.
SPCM_XMODE_ASYNCOUT	00000002h		Connector is programmed for asynchronous output. Use SPCM_XX_ASYNCIO to write data asynchronous as shown in next chapter.
SPCM_XMODE_TRIGOUT	00000020h		Connector is programmed as trigger output and shows the trigger detection. The trigger output goes HIGH as soon as the trigger is recognized. After end of acquisition it is LOW again. In Multiple Recording/Gated Sampling/ABA mode it goes LOW after the acquisition of the current segment stops. In standard FIFO mode the trigger output is HIGH until FIFO mode is stopped.
SPCM_XMODE_RUNSTATE	00000100h		Connector shows the current run state of the card. If acquisition/output is running the signal is HIGH. If card has stopped the signal is LOW.
SPCM_XMODE_ARMSTATE	00000200h		Connector shows the current ARM state of the card. If the card is armed and ready to receive a trigger the signal is HIGH. If the card isn't running or the card is still acquiring pretrigger data or the trigger has been detected the signal is LOW.
SPCM_XMODE_REFCLKOUT	00001000h		Connector reflects the internal generated PLL reference clock in the range of 10 to 62.5 MHz.
SPCM_XMODE_CONTOUTMARK	00002000h		Generator Cards only: outputs a HIGH pulse as continuous marker signal for continuous replay mode. The marker signal length is 1/2 of the programmed memory size.



Please note that a change to the SPCM_X0_MODE, SPCM_X1_MODE or SPCM_X2_MODE will only be updated with the next call to either the M2CMD_CARD_START or M2CMD_CARD_WRITESETUP register. For further details please see the relating chapter on the M2CMD_CARD registers.

Using asynchronous I/O

To use asynchronous I/O on the multi purpose I/O lines it is first necessary to switch these lines to the desired asynchronous mode by programming the above explained mode registers. As a special feature asynchronous input can also be read if the mode is set to trigger input or digital input.

Table 83: Spectrum API: registers for setting up asynchronous I/O of multi-purpose lines

Register	Value	Direction	Description
SPCM_XX_ASYNCIO	47220	read/write	Connector X0 is linked to bit 0 of the register, connector X1 is linked to bit 1 while connector X2 is linked to bit 2 of this register. Data is written/read immediately without any relation to the currently used sampling rate or mode. If a line is programmed to output, reading this line asynchronously will return the current output level.

Example of asynchronous write and read. We write a high pulse on output X1 and wait for a high level answer on input X0:

```

spcm_dwSetParam_i32 (hDrv, SPCM_X0_MODE, SPCM_XMODE_ASYNCIN); // X0 set to asynchronous input
spcm_dwSetParam_i32 (hDrv, SPCM_X1_MODE, SPCM_XMODE_ASYNCOUT); // X1 set to asynchronous output
spcm_dwSetParam_i32 (hDrv, SPCM_X2_MODE, SPCM_XMODE_TRIGOUT); // X2 set to trigger output

spcm_dwSetParam_i32 (hDrv, SPCM_XX_ASYNCIO, 0); // programming a high pulse on output
spcm_dwSetParam_i32 (hDrv, SPCM_XX_ASYNCIO, 2);
spcm_dwSetParam_i32 (hDrv, SPCM_XX_ASYNCIO, 0);

do {
    spcm_dwGetParam_i32 (hDrv, SPCM_XX_ASYNCIO, &lAsyncIn); // read input in a loop
} while ((lAsyncIn & 1) == 0) // until X0 is going to high level

```

Special behaviour of trigger output

As the driver of the M4i/M4x series is the same as the driver for the M2i/M3i series and some old software may rely on register structure of the M2i/M3i card series there is a special compatible trigger output register that will work according to the M2i/M3i series style. It is not recommended to use this register unless you're writing software for multiple card series:

Table 84: Spectrum API: trigger output compatibility register

Register	Value	Direction	Description
SPC_TRIG_OUTPUT	40100	read/write	<p>M2i style trigger output programming. Write a „1“ to enable:</p> <ul style="list-style-type: none"> - X2 trigger output (SPCM_X2_MODE = SPCM_XMODE_TRIGOUT) - X1 arm state (SPCM_X1_MODE = SPCM_XMODE_ARMSTATE). - X0 run state (SPCM_X0_MODE = SPCM_XMODE_RUNSTATE). <p>Write a „0“ to disable both outputs:</p> <ul style="list-style-type: none"> - SPCM_X0_MODE = SPCM_X1_MODE = SPCM_X2_MODE = SPCM_XMODE_DISABLE

The SPC_TRIG_OUTPUT register overrides the multi purpose I/O settings done by SPCM_X0_MODE, SPCM_X1_MODE and SPCM_X2_MODE and vice versa. Please do not use both methods in one program.

Setting up the X0, X1 threshold and termination (single-ended models only)

Programmable threshold

For the single-ended X0 and X1 lines a common threshold can be defined, to accomodate for different logic level standards:

Table 85: Spectrum API: trigger threshold register and register settings

Register	Value	Direction	Description	Offset range
SPC_X0X1_THRESHOLD	42412	read/write	Defines the common input threshold level for X0 and X1 input	0 to 4000 mV in steps of 10 mV

Programmable input termination

For each of the single-ended X0 or X1 input the termination can also be defined:

Table 86: Spectrum API: multi-purpose I/O programmable termination registers

Register	Value	Direction	Description
SPC_X0_TERM	600100	read/write	A „1“ sets the 75 Ω termination for the X0 input. A „0“ sets the termination to 4.7 k Ω .
SPC_X1_TERM	600101	read/write	A „1“ sets the 75 Ω termination for the X1 input. A „0“ sets the termination to 4.7 k Ω .

As an example define threshold for LVTTL compatibility and enable 75 Ohm termination on X0 and 4.7 kOhm on X1:

```

spcm_dwSetParam_i64 (hDrv, SPC_X0X1_THRESHOLD, 1500); // set Ext0 clock threshold to 1500 mV
spcm_dwSetParam_i64 (hDrv, SPC_X0_TERM, 1); // set X0 to 75 Ohm
spcm_dwSetParam_i64 (hDrv, SPC_X1_TERM, 0); // set X0 to 4.7 kOhm

```

Mode Multiple Recording

The Multiple Recording mode allows the acquisition of data blocks with multiple trigger events without restarting the hardware.

The on-board memory will be divided into several segments of the same size. Each segment will be filled with data when a trigger event occurs (acquisition mode).

As this mode is totally controlled in hardware there is a very small re-arm time from end of one segment until the trigger detection is enabled again. You'll find that re-arm time in the technical data section of this manual.

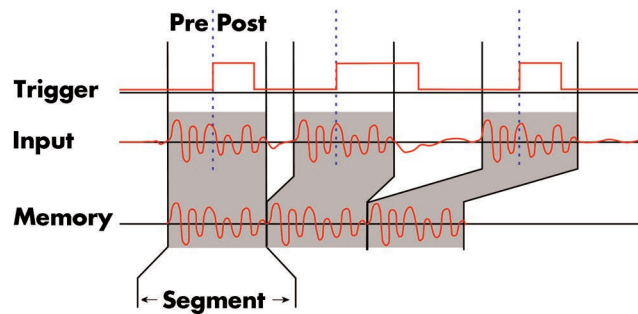


Image 56: Drawing of Multiple Recording acquisition

The following table shows the register for defining the structure of the segments to be recorded with each trigger event.

Table 87: Spectrum API: software registers for Multiple Recording mode setup

Register	Value	Direction	Description
SPC_POSTTRIGGER	10100	read/write	Acquisition only: defines the number of samples to be recorded per channel after the trigger event.
SPC_SEGMENTSIZE	10010	read/write	Size of one Multiple Recording segment: the total number of samples to be recorded per channel after detection of one trigger event including the time recorded before the trigger (pre trigger).

Each segment in acquisition mode can consist of pretrigger and/or posttrigger samples. The user always has to set the total segment size and the posttrigger, while the pretrigger is calculated within the driver with the formula: $[\text{pretrigger}] = [\text{segment size}] - [\text{posttrigger}]$.

When using Multiple Recording the maximum pretrigger is limited depending on the number of active channels. When the calculated value exceeds that limit, the driver will return the error ERR_PRETRIGGERLEN. Please have a look at the table further below to see the maximum pretrigger length that is possible.



Recording modes

Standard Mode

With every detected trigger event one data block is filled with data. The length of one multiple recording segment is set by the value of the segment size register SPC_SEGMENTSIZE. The total amount of samples to be recorded is defined by the memsize register. Memsize must be set to a multiple of the segment size. The table below shows the register for enabling Multiple Recording. For detailed information on how to setup and start the standard acquisition mode please refer to the according chapter earlier in this manual.

Table 88: Spectrum API: card mode register and multiple recording settings

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode
SPC_REC_STD_MULTI	2		Enables Multiple Recording for standard acquisition.

The total number of samples to be recorded to the on-board memory in Standard Mode is defined by the SPC_MEMSIZE register.

Table 89: Spectrum API: memory and loop registers with related multiple recording settings

Register	Value	Direction	Description
SPC_MEMSIZE	10000	read/write	Defines the total number of samples to be recorded per channel.

FIFO Mode

The Multiple Recording in FIFO Mode is similar to the Multiple Recording in Standard Mode. In contrast to the standard mode it is not necessary to program the number of samples to be recorded. The acquisition is running until the user stops it. The data is read block by block by the driver as described under FIFO single mode example earlier in this manual. These blocks are online available for further data processing by the user program. This mode significantly reduces the amount of data to be transferred on the PCI bus as gaps of no interest do not have to be transferred. This enables you to use faster sample rates than you would be able to in FIFO mode without Multiple Recording. The advantage of Multiple Recording in FIFO mode is that you can stream data online to the host system. You can make real-time data processing or store a huge amount of data to the hard disk. The table below shows the dedicated register for enabling Multiple Recording. For detailed information how to setup and start the board in FIFO mode please refer to the according chapter earlier in this manual.

Table 90: Spectrum API: card mode register and multiple replay FIFO mode settings

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode
SPC_REC_FIFO_MULTI	32		Enables Multiple Recording for FIFO acquisition.

The number of segments to be recorded must be set separately with the register shown in the following table:

Table 91: Spectrum API: loops register settings when using Multiple Replay FIFO mode

Register	Value	Direction	Description
SPC_LOOPS	10020	read/write	Defines the number of segments to be recorded
0			Recording will be infinite until the user stops it.
1 ... [4G - 1]			Defines the total segments to be recorded.

Limits of pre trigger, post trigger, memory size

The maximum memory size parameter is only limited by the number of activated channels and by the amount of installed memory. Minimum memory size as well as minimum and maximum post trigger limits are independent of the activated channels or the installed memory.

Due to the internal organization of the card memory there is a certain stepsize when setting these values that has to be taken into account. The following table gives you an overview of all limits concerning pre trigger, post trigger, memory size, segment size and loops. The table shows all values in relation to the installed memory size in samples. If more memory is installed the maximum memory size figures will increase according to the complete installed memory

Table 92: Limits of pre trigger, post trigger, memory size depending on selected mode and activated channels

Activated Channels	Used Mode	Memory size SPC_MEMSIZE			Pre trigger SPC_PRETRIGGER			Post trigger SPC_POSTTRIGGER			Segment size SPC_SEGMENTSIZE			Loops SPC_LOOPS		
		Min	Max	Step	Min	Max	Step	Min	Max	Step	Min	Max	Step	Min	Max	Step
8 Ch	Standard Single	64	Mem	32	32	Mem - 32 (defined by mem and post)	32	32	8G - 32	32	not used			not used		
	Standard Multi/ABA	64	Mem	32	32	4k (defined by segment and post)	32	32	Mem/2-32	32	64	Mem/2	32	not used		
	Standard Gate	64	Mem	32	32	8k	32	32	Mem-32	32	not used			not used		
	FIFO Single	not used			32	4k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	4k (defined by segment and post)	32	32	8G - 32	32	64	pre+post	32	0 (∞)	4G - 1	1
	FIFO Gate	not used			32	4k	32	32	8G - 32	32	not used			0 (∞)	4G - 1	1
16 Ch	Standard Single	64	Mem/2	32	32	Mem/2 - 32 (defined by mem and post)	32	32	8G - 32	32	not used			not used		
	Standard Multi/ABA	64	Mem/2	32	32	4k (defined by segment and post)	32	32	Mem/4-32	32	64	Mem/4	32	not used		
	Standard Gate	64	Mem/2	32	32	4k	32	32	Mem/2-32	32	not used			not used		
	FIFO Single	not used			32	4k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	4k (defined by segment and post)	32	32	8G - 32	32	64	pre+post	32	0 (∞)	4G - 1	1
	FIFO Gate	not used			32	4k	32	32	8G - 32	32	not used			0 (∞)	4G - 1	1
32 Ch	Standard Single	64	Mem/4	32	32	Mem/4 - 32 (defined by mem and post)	32	32	8G - 16	32	not used			not used		
	Standard Multi/ABA	64	Mem/4	32	32	4k (defined by segment and post)	32	32	Mem/8-32	32	64	Mem/8	32	not used		
	Standard Gate	64	Mem/4	32	32	4k	32	32	Mem/4-16	32	not used			not used		
	FIFO Single	not used			32	4k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	4k (defined by segment and post)	32	32	8G - 32	32	64	pre+post	32	0 (∞)	4G - 1	1
	FIFO Gate	not used			32	4k	32	32	8G - 32	32	not used			0 (∞)	4G - 1	1

All figures listed here are given in samples. An entry of [8G - 32] means $[8 \text{ GSamples} - 32] = 8,589,934,560$ samples.

The given memory and memory / divider figures depend on the installed on-board memory as listed below:

	Installed Memory
	4 GSample
Mem	4 GSample
Mem / 2	2 GSample
Mem / 4	1 GSample
Mem / 8	512 MSample

Please keep in mind that this table shows all values at once. Only the absolute maximum and minimum values are shown. There might be additional limitations. Which of these values is programmed depends on the used mode. Please read the detailed documentation of the mode.

Multiple Recording and Timestamps

Multiple Recording is well matching with the timestamp option. If timestamp recording is activated each trigger event and therefore each Multiple Recording segment will get timestamped as shown in the drawing on the right.

Please keep in mind that the trigger events are timestamped, not the beginning of the acquisition. The first sample that is available is at the time position of [Timestamp - Pretrigger].

The programming details of the timestamp option is explained in an extra chapter.

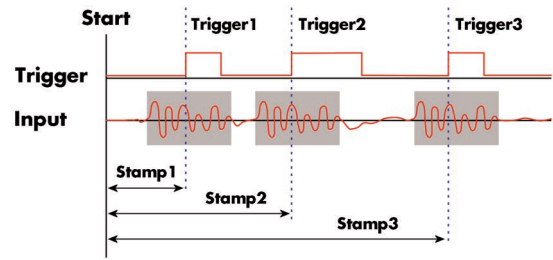


Image 57: drawing of Multiple Recording Acquisition with Timestamps

Trigger Modes

When using Multiple Recording all of the card's trigger modes can be used including software trigger. For detailed information on the available trigger modes, please take a look at the relating chapter earlier in this manual.

Programming examples

The following example shows how to set up the card for Multiple Recording in standard mode.

```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_MULTI); // Enables Standard Multiple Recording

spcm_dwSetParam_i64 (hDrv, SPC_SEGMENTSIZE, 1024); // Set the segment size to 1024 samples
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, 768); // Set the posttrigger to 768 samples and therefore
// the pretrigger will be 256 samples
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, 4096); // Set the total memsize for recording to 4096 samples
// so that actually four segments will be recorded

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_POS); // Set trigmode to ext. TTL mode (rising edge)
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0); // and enable it within the trigger OR-mask
```

The following example shows how to set up the card for Multiple Recording in FIFO mode.

```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_FIFO_MULTI); // Enables FIFO Multiple Recording

spcm_dwSetParam_i64 (hDrv, SPC_SEGMENTSIZE, 2048); // Set the segment size to 2048 samples
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, 1920); // Set the posttrigger to 1920 samples and therefore
// the pretrigger will be 128 samples
spcm_dwSetParam_i64 (hDrv, SPC_LOOPS, 256); // 256 segments will be recorded

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_NEG); // Set trigmode to ext. TTL mode (falling edge)
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0); // and enable it within the trigger OR-mask
```


Mode Gated Sampling

The Gated Sampling mode allows the data acquisition controlled by an external or an internal gate signal. Data will only be recorded if the programmed gate condition is true. When using the Gated Sampling acquisition mode it is in addition also possible to program a pre- and/or posttrigger for recording samples prior to and/or after the valid gate.

This chapter will explain all the necessary software register to set up the card for Gated Sampling properly.

The section on the allowed trigger modes deals with detailed description on the different trigger events and the resulting gates.

When using Gated Sampling the maximum pretrigger is limited as shown in the technical data section. When the programmed value exceeds that limit, the driver will return the error ERR_PRETRIGGERLEN.

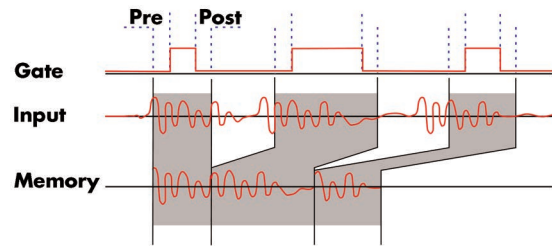


Image 58: Drawing of Gated Sampling mode

Table 93: Spectrum API: registers and settings for Gated Sampling mode

Register	Value	Direction	Description
SPC_PRETRIGGER	10030	read/write	Defines the number of samples to be recorded per channel prior to the gate start.
SPC_POSTTRIGGER	10100	read/write	Defines the number of samples to be recorded per channel after the gate end.

Acquisition modes

Standard Mode

Data will be recorded as long as the gate signal fulfils the programmed gate condition. At the end of the gate interval the recording will be stopped and the card will pause until another gates signal appears. If the total amount of data to acquire has been reached, the card stops immediately. For that reason the last gate segment is ended by the expiring memory size counter and not by the gate end signal. The total amount of samples to be recorded can be defined by the memsize register. The table below shows the register for enabling Gated Sampling. For detailed information on how to setup and start the standard acquisition mode please refer to the according chapter earlier in this manual.

Table 94: Spectrum API: card mode register and settings for Gated Sampling standard mode

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode
SPC_REC_STD_GATE	4		Enables Gated Sampling for standard acquisition.

The total number of samples to be recorded to the on-board memory in Standard Mode is defined by the SPC_MEMSIZE register.

Table 95: Spectrum API: memsize and loops register and register settings for Gated Replay mode

Register	Value	Direction	Description
SPC_MEMSIZE	10000	read/write	Defines the total number of samples to be recorded per channel.

FIFO Mode

The Gated Sampling in FIFO Mode is similar to the Gated Sampling in Standard Mode. In contrast to the Standard Mode you cannot program a certain total amount of samples to be recorded, but two other end conditions can be set instead. The acquisition can either run until the user stops it by software (infinite recording), or until a programmed number of gates has been recorded. The data is read continuously by the driver. This data is online available for further data processing by the user program. The advantage of Gated Sampling in FIFO mode is that you can stream data online to the host system with a lower average data rate than in conventional FIFO mode without Gated Sampling. You can make real-time data processing or store a huge amount of data to the hard disk. The table below shows the dedicated register for enabling Gated Sampling in FIFO mode. For detailed information how to setup and start the card in FIFO mode please refer to the according chapter earlier in this manual.

Table 96: Spectrum API: card mode register and Gated Sampling FIFO mode settings

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode
SPC_REC_FIFO_GATE	64		Enables Gated Sampling for FIFO acquisition.

The number of gates to be recorded must be set separately with the register shown in the following table:

Table 97: Spectrum API: Gated Sampling FIFO mode loops register settings

Register	Value	Direction	Description
SPC_LOOPS	10020	read/write	Defines the number of gates to be recorded
0			Recording will be infinite until the user stops it.
1 ... [4G - 1]			Defines the total number of gates to be recorded.

Limits of pre trigger, post trigger, memory size

The maximum memory size parameter is only limited by the number of activated channels and by the amount of installed memory. Minimum memory size as well as minimum and maximum post trigger limits are independent of the activated channels or the installed memory.

Due to the internal organization of the card memory there is a certain stepsize when setting these values that has to be taken into account. The following table gives you an overview of all limits concerning pre trigger, post trigger, memory size, segment size and loops. The table shows all values in relation to the installed memory size in samples. If more memory is installed the maximum memory size figures will increase according to the complete installed memory

Table 98: Limits of pre trigger, post trigger, memory size depending on selected mode and activated channels

Activated Channels	Used Mode	Memory size SPC_MEMSIZE			Pre trigger SPC_PRETRIGGER			Post trigger SPC_POSTTRIGGER			Segment size SPC_SEGMENTSIZE			Loops SPC_LOOPS		
		Min	Max	Step	Min	Max	Step	Min	Max	Step	Min	Max	Step	Min	Max	Step
8 Ch	Standard Single	64	Mem	32	32	Mem - 32 (defined by mem and post)	32	32	8G - 32	32	not used			not used		
	Standard Multi/ABA	64	Mem	32	32	4k (defined by segment and post)	32	32	Mem/2-32	32	64	Mem/2	32	not used		
	Standard Gate	64	Mem	32	32	8k	32	32	Mem-32	32	not used			not used		
	FIFO Single	not used			32	4k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	4k (defined by segment and post)	32	32	8G - 32	32	64	pre+post	32	0 (∞)	4G - 1	1
	FIFO Gate	not used			32	4k	32	32	8G - 32	32	not used			0 (∞)	4G - 1	1
16 Ch	Standard Single	64	Mem/2	32	32	Mem/2 - 32 (defined by mem and post)	32	32	8G - 32	32	not used			not used		
	Standard Multi/ABA	64	Mem/2	32	32	4k (defined by segment and post)	32	32	Mem/4-32	32	64	Mem/4	32	not used		
	Standard Gate	64	Mem/2	32	32	4k	32	32	Mem/2-32	32	not used			not used		
	FIFO Single	not used			32	4k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	4k (defined by segment and post)	32	32	8G - 32	32	64	pre+post	32	0 (∞)	4G - 1	1
	FIFO Gate	not used			32	4k	32	32	8G - 32	32	not used			0 (∞)	4G - 1	1
32 Ch	Standard Single	64	Mem/4	32	32	Mem/4 - 32 (defined by mem and post)	32	32	8G - 16	32	not used			not used		
	Standard Multi/ABA	64	Mem/4	32	32	4k (defined by segment and post)	32	32	Mem/8-32	32	64	Mem/8	32	not used		
	Standard Gate	64	Mem/4	32	32	4k	32	32	Mem/4-16	32	not used			not used		
	FIFO Single	not used			32	4k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	4k (defined by segment and post)	32	32	8G - 32	32	64	pre+post	32	0 (∞)	4G - 1	1
	FIFO Gate	not used			32	4k	32	32	8G - 32	32	not used			0 (∞)	4G - 1	1

All figures listed here are given in samples. An entry of [8G - 32] means [8 GSamples - 32] = 8,589,934,560 samples.

The given memory and memory / divider figures depend on the installed on-board memory as listed below:

	Installed Memory
	4 GSample
Mem	4 GSample
Mem / 2	2 GSample
Mem / 4	1 GSample
Mem / 8	512 MSample

Please keep in mind that this table shows all values at once. Only the absolute maximum and minimum values are shown. There might be additional limitations. Which of these values is programmed depends on the used mode. Please read the detailed documentation of the mode.

Programming examples

The following examples shows how to set up the card for Gated Sampling in standard mode for Gated Sampling in FIFO mode.

```

spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_GATE); // Enables Standard Gated Sampling

spcm_dwSetParam_i64 (hDrv, PRETRIGGER, 256); // Set the pretrigger to 256 samples
spcm_dwSetParam_i64 (hDrv, POSTTRIGGER, 2048); // Set the posttrigger to 2048 samples
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, 8192); // Set the total memsize for recording to 8192 samples

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_POS); // Use external trigger (rising edge)
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0); // and enable it within the trigger OR-mask

```

```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_FIFO_GATE); // Enables FIFO Gated Sampling

spcm_dwSetParam_i64 (hDrv, PRETRIGGER, 128); // Set the pretrigger to 128 samples
spcm_dwSetParam_i64 (hDrv, POSTTRIGGER, 512); // Set the posttrigger to 512 samples
spcm_dwSetParam_i64 (hDrv, SPC_LOOP, 1024); // 1024 gates will be recorded

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_NEG); // Use external trigger (falling edge)
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0); // and enable it within the trigger OR-mask
```

Timestamps

General information

The timestamp function is used to record trigger events relative to the beginning of the measurement, relative to a fixed time-zero point or synchronized to an external reset clock. The reset clock can come from a radio clock, a GPS signal or from any other external machine.

The timestamp is internally realized as a very wide counter that is running with the currently used sampling rate. The counter is reset either by explicit software command or depending on the mode by the start of the card. On receiving the trigger event the current counter value is stored in an extra FIFO memory.

This function is designed as an enhancement to the Multiple Recording mode and is also used together with the Gated Sampling and ABA mode, but can also be used with plain single acquisitions.

Each recorded timestamp consists of the number of samples that has been counted since the last counter reset has been done. The actual time in relation to the reset command can be easily calculated by the formula on the right. Please note that the timestamp recalculation depends on the currently used sampling rate. Please have a look at the clock chapter to see how to read out the sampling rate.

$$t = \frac{\text{Timestamp}}{\text{Sampling rate}}$$

If you want to know the time between two timestamps, you can simply calculate this by the formula on the right.

$$\Delta t = \frac{\text{Timestamp}_{n+1} - \text{Timestamp}_n}{\text{Sampling rate}}$$

The following registers can be used for the timestamp function:

Table 99: Spectrum API: timestamp related register and available timestamp commands

Register	Value	Direction	Description
SPC_TIMESTAMP_STARTTIME	47030	read/write	Return the reset time when using reference clock mode. Hours are placed in bit 16 to 23, minutes are placed in bit 8 to 15, seconds are placed in bit 0 to 7. Returned value is expressed as a UTC time.
SPC_TIMESTAMP_STARTDATE	47031	read/write	Return the reset date when using reference clock mode. The year is placed in bit 16 to 31, the month is placed in bit 8 to 15 and the day of month is placed in bit 0 to 7.
SPC_TIMESTAMP_TIMEOUT	47045	read/write	Set's a timeout in milli seconds for waiting of an reference clock edge. Writing a zero disables the timeout. Default value is zero.
SPC_TIMESTAMP_AVAILMODES	47001	read	Returns all available modes as a bitmap. Modes are listed below
SPC_TIMESTAMP_CMD	47000	read/write	Programs a timestamp mode and performs commands as listed below
SPC_TSMODE_DISABLE	0		Timestamp is disabled.
SPC_TS_RESET	1h		The counters are reset and the local PC time is stored for read out by SPC_TIMESTAMP_STARTTIME and SPC_TIMESTAMP_STARTDATE registers. Only usable with mode TSMODE_STANDARD
SPC_TSMODE_STANDARD	2h		Standard mode, counter is reset by explicit reset command SPC_TS_RESET or SPC_TS_RESET_WAITREFCLOCK.
SPC_TSMODE_STARTRESET	4h		Counter is reset on every card start, all timestamps are in relation to card start.
SPC_TS_RESET_WAITREFCLK	8h		Similar as SPC_TS_RESET, but aimed at SPC_TSCNT_REFCLKxxx modes: The counters are reset then the driver waits for the reference edge as long as defined by the timestamp timeout time. After detecting the edge, the local PC time is stored for read out by SPC_TIMESTAMP_STARTTIME and SPC_TIMESTAMP_STARTDATE registers. Only usable with mode TSMODE_STANDARD
SPC_TSCNT_INTERNAL	100h		Counter is running with complete width on sampling clock
SPC_TSCNT_REFCLKPOS	200h		Counter is split, upper part is running with external reference clock positive edge, lower part is running with sampling clock
SPC_TSCNT_REFCLKNEG	400h		Counter is split, upper part is running with external reference clock negative edge, lower part is running with sampling clock
SPC_TSIOACQ_ENABLE	1000h		Enables the trigger synchronous acquisition of the multi-purpose inputs with every stored timestamp in the upper 64 bit. See Multi-purpose I/O chapter for details on these inputs.
SPC_TSFEAT_NONE	0		No additional timestamp is created. The total number of stamps is only trigger related.
SPC_TSFEAT_STORE1STABA	10000h		Enables the creation of one additional timestamp for the first A area sample when using the optional ABA (dual-time-base) mode.
SPC_TSFEAT_TRGSRC	80000h		Reding this flag from the SPC_TIMESTAMP_AVAILMODES indicates that the card is capable of encoding the trigger source into the timestamp. Writing this flag to the SPC_TIMESTAMP_CMD register enables the storage of the trigger source in the upper 64 bit of the timestamp value.

Writing of SPC_TS_RESET and SPC_TS_RESET_WAITREFCLK to the command register can only have an effect on the counters, if the cards clock generation is already active and the timestamp mode has been written to the hardware. This is the case when the card either has already done an acquisition with enabled timestamps after the last reset or if the clock setup and timestamp mode has already been actively transferred to the card by issuing the M2CMD_CARD_WRITESETUP command.



Example for setting timestamp mode:

The timestamp mode must consist of one of the mode constants, one of the counter and one of the feature constants:

```
// setting timestamp mode to standard using internal clocking
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TSMODE_STANDARD | SPC_TSCNT_INTERNAL | SPC_TSFEAT_NONE);

// setting timestamp mode to start reset mode using internal clocking
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TSMODE_STARTRESET | SPC_TSCNT_INTERNAL | SPC_TSFEAT_NONE);

// setting timestamp mode to standard using external reference clock with positive edge
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TSMODE_STANDARD | SPC_TSCNT_REFCLOCKPOS | SPC_TSFEAT_NONE);
```

Timestamp modes

Standard mode

In standard mode the timestamp counter is set to zero once by writing the TS_RESET command to the command register. After that command the counter counts continuously independent of start and stop of acquisition. The timestamps of all recorded trigger events are referenced to this common zero time. With this mode you can calculate the exact time difference between different recordings and also within one acquisition (if using for example Multiple Recording).

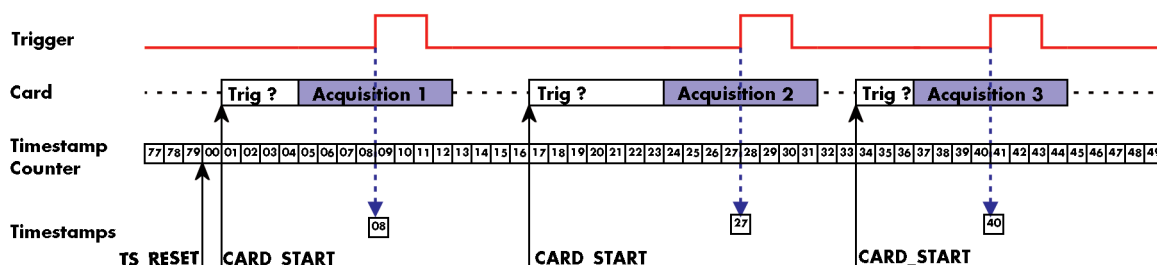


Image 59: drawing of timestamp acquisition in standard mode in relation to card start and trigger detection

The following table shows the valid values that can be written to the timestamp command register for this mode:

Table 100: Spectrum API: timestamp commands for standard mode

Register	Value	Direction	Description
SPC_TIMESTAMP_CMD	47000	read/write	Programs a timestamp mode and performs commands as listed below
SPC_TSMODE_DISABLE	0		Timestamp is disabled.
SPC_TS_RESET	1h		The timestamp counter is set to zero
SPC_TSMODE_STANDARD	2h		Standard mode, counter is reset by explicit reset command.
SPC_TSCNT_INTERNAL	100h		Counter is running with complete width on sampling clock

Please keep in mind that this mode only work sufficiently as long as you don't change the sampling rate between two acquisitions that you want to compare.



StartReset mode

In StartReset mode the timestamp counter is set to zero on every start of the card. After starting the card the counter counts continuously. The timestamps of one recording are referenced to the start of the recording. This mode is very useful for Multiple Recording and Gated Sampling (see according chapters for detailed information on these two optional modes).

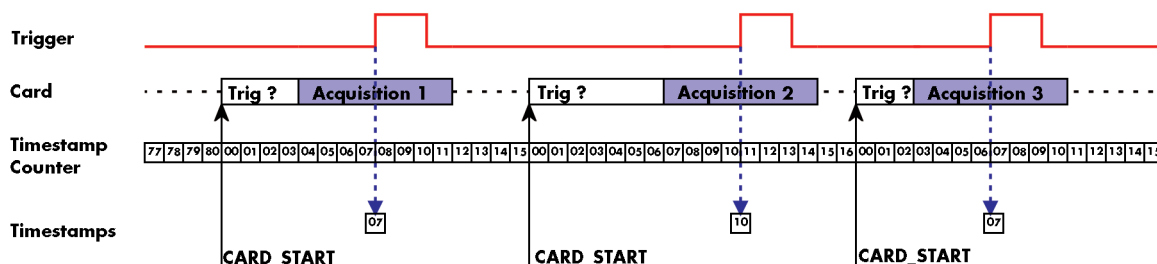


Image 60: drawing of timestamp acquisition in start-reset mode in relation to card start and trigger detection

The following table shows the valid values that can be written to the timestamp command register.

Table 101: Spectrum API: timestamp commands for star-reset mode

Register	Value	Direction	Description
SPC_TIMESTAMP_CMD	47000	read/write	Programs a timestamp mode and performs commands as listed below
SPC_TSMODE_DISABLE	0		Timestamp is disabled.
SPC_TSMODE_STARTRESET	4h		Counter is reset on every card start, all timestamps are in relation to card start.
SPC_TSCNT_INTERNAL	100h		Counter is running with complete width on sampling clock

Refclock mode

In addition to the counter counting the samples a second separate counter is utilized. An additional external signal is used, which affects both counters and needs to be fed in externally. This external reference clock signal will reset the sample counter and also increase the second counter. The second counter holds the number of the clock edges that have occurred on the external reference clock signal and the sample counter holds the position within the current reference clock period with the resolution of the sampling rate.

This mode can be used to obtain an absolute time reference when using an external radio clock or a GPS receiver. In that case the higher part is counting the seconds since the last reset and the lower part is counting the position inside the second using the current sampling rate.

Please keep in mind that as this mode uses an additional external signal and can therefore only be used when connecting an reference clock signal on the related connector on the card:



- X0 on M4i/M4x/M5i and related digitizerNETBOX products
- X1 on M2p and related digitizerNETBOX products

The counting is initialized with the timestamp reset command. Both counters will then be set to zero.

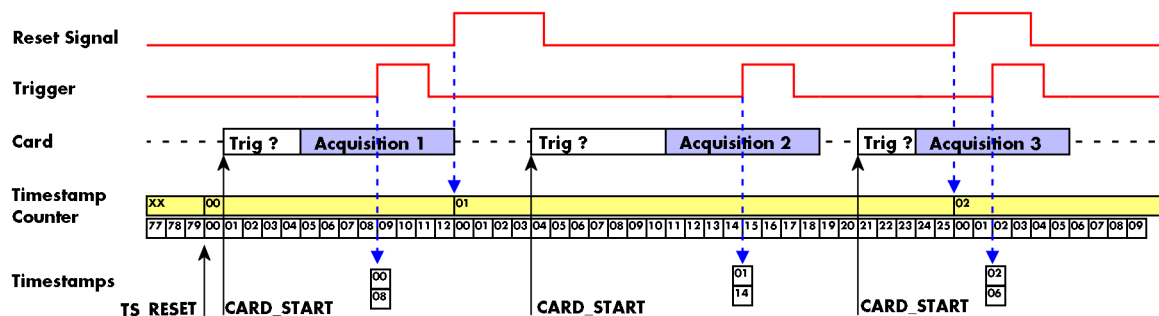


Image 61: drawing of timestamp acquisition in refclock mode in relation to card start and trigger detection

The following table shows the valid values that can be written to the timestamp command register for this mode:

Table 102: Spectrum API: timestamp commands for refclock mode

Register	Value	Direction	Description
SPC_TIMESTAMP_STARTTIME	47030	read/write	Return the reset time when using reference clock mode. Hours are placed in bit 16 to 23, minutes are placed in bit 8 to 15, seconds are placed in bit 0 to 7
SPC_TIMESTAMP_STARTDATE	47031	read/write	Return the reset date when using reference clock mode. The year is placed in bit 16 to 31, the month is placed in bit 8 to 15 and the day of month is placed in bit 0 to 7
SPC_TIMESTAMP_TIMEOUT	47045	read/write	Sets a timeout in milli seconds for waiting for a reference clock edge
SPC_TIMESTAMP_CMD	47000	read/write	Programs a timestamp mode and performs commands as listed below
SPC_TSMODE_DISABLE	0		Timestamp is disabled.
SPC_TS_RESET	1h		The counters are reset and the local PC time is stored for read out by SPC_TIMESTAMP_STARTTIME and SPC_TIMESTAMP_STARTDATE registers.
SPC_TS_RESET_WAITREFCLK	8h		Similar as SPC_TS_RESET, but aimed at SPC_TSCNT_REFCLOCKxxx modes: The counters are reset then the driver waits for the reference edge as long as defined by the timeout time. After detecting the edge, the local PC time is stored for read out by SPC_TIMESTAMP_STARTTIME and SPC_TIMESTAMP_STARTDATE registers.
SPC_TSMODE_STANDARD	2h		Standard mode, counter is reset by explicit reset command.
SPC_TSMODE_STARTRESET	4h		Counter is reset on every card start, all timestamps are in relation to card start.
SPC_TSCNT_REFCLOCKPOS	200h		Counter is split, upper part is running with external reference clock positive edge, lower part is running with sampling clock
SPC_TSCNT_REFCLOCKNEG	400h		Counter is split, upper part is running with external reference clock negative edge, lower part is running with sampling clock

To synchronize the external reference clock signal with the PC clock it is possible to perform a timestamp reset command which waits a specified time for the occurrence of the external clock edge. As soon as the clock edge is found the function stores the current PC time and date which can be used to get the absolute time. As the timestamp reference clock can also be used with other clocks that don't need to be synchronized with the PC clock the waiting time can be programmed using the SPC_TIMESTAMP_TIMEOUT register.

Example for initialization of timestamp reference clock and synchronization of a seconds signal with the PC clock:

```

spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TSMODE_STANDARD | SPC_TSCNT_REFCLOCKPOS);
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_TIMEOUT, 1500);
if (ERR_TIMESTAMP_SYNC == spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TS_RESET_WAITREFCLK))
    printf ("Synchronization with external clock signal failed\n");

// now we read out the stored synchronization clock and date
int32 lSyncDate, lSyncTime;
spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_STARTDATE, &lSyncDate);
spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_STARTTIME, &lSyncTime); // expressed as UTC time

// and print the start date and time information (European format: day.month.year hour:minutes:seconds)
printf ("Start date: %02d.%02d.%04d\n", lSyncDate & 0xff, (lSyncDate >> 8) & 0xff, (lSyncDate >> 16) & 0xffff);
printf ("Start time: %02d:%02d:%02d\n", (lSyncTime >> 16) & 0xff, (lSyncTime >> 8) & 0xff, lSyncTime & 0xff);

```

Reading out the timestamps

General

The timestamps are stored in an extra FIFO that is located in hardware on the card. This extra FIFO can read out timestamps using DMA transfer similar to the DMA transfer of the main sample data DMA transfer. The card has three completely independent busmaster DMA engines in hardware allowing the simultaneous transfer of both timestamp and sample data.

As seen in the picture there are separate FIFOs holding ABA (if available) and timestamp data.

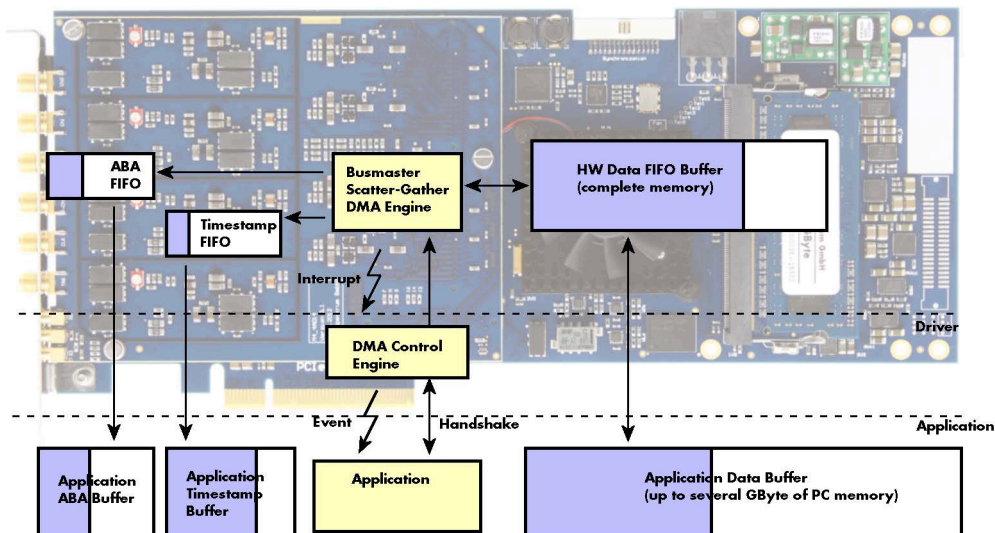


Image 62: Overview of acquisition data, ABA data and timestamp data DMA transfer

Although an M4i is shown here, this applies to M4x, M2p and M5i cards as well. Each FIFO has its own DMA channel, the way data is handled by the DMA engine is similar for both kinds of extra FIFOs and is also very similar to the main sample data transfer engine. Therefore additional information can be found in the chapter explaining the main data transfer.

Commands and Status information for extra transfer buffers.

As explained above the data transfer is performed with the same command and status registers like the card control and sample data transfer. It is possible to send commands for card control, data transfer and extra FIFO data transfer at the same time

Table 103: Spectrum API: extra DMA commands (ABA and Timestamp)

Register	Value	Direction	Description
SPC_M2CMD	100	write only	Executes a command for the card or data transfer
M2CMD_EXTRA_STARTDMA	100000h		Starts the DMA transfer for an already defined buffer.
M2CMD_EXTRA_WAITDMA	200000h		Waits until the data transfer has ended or until at least the amount of bytes defined by notify size are available. This wait function also takes the timeout parameter into account.
M2CMD_EXTRA_STOPDMA	400000h		Stops a running DMA transfer. Data is invalid afterwards.
M2CMD_EXTRA_POLL	800000h		Polls data without using DMA. As DMA has some overhead and has been implemented for fast data transfer of large amounts of data it is in some cases more simple to poll for available data. Please see the detailed examples for this mode. It is not possible to mix DMA and polling mode.

The extra FIFO data transfer can generate one of the following status information:

Table 104: Spectrum API: extra DMA status (ABA and Timestamp)

Register	Value	Direction	Description
SPC_M2STATUS	110	read only	Reads out the current status information
M2STAT_EXTRA_BLOCKREADY	1000h		The next data block as defined in the notify size is available. It is at least the amount of data available but it also can be more data.
M2STAT_EXTRA_END	2000h		The data transfer has completed. This status information will only occur if the notify size is set to zero.
M2STAT_EXTRA_OVERRUN	4000h		The data transfer had on overrun (acquisition) or underrun (replay) while doing FIFO transfer.
M2STAT_EXTRA_ERROR	8000h		An internal error occurred while doing data transfer.

Data Transfer using DMA

Data transfer consists of two parts: the buffer definition and the commands/status information that controls the transfer itself. Extra data transfer shares the command and status register with the card control, data transfer commands and status information.

The DMA based data transfer mode is activated as soon as the M2CMD_EXTRA_STARTDMA is given. Please see next chapter to see how the polling mode works.

Definition of the transfer buffer

Before any data transfer can start it is necessary to define the transfer buffer with all its details. The definition of the buffer is done with the `spcm_dwDefTransfer` function as explained in an earlier chapter. The following example will show the definition of a transfer buffer for timestamp data, definition for ABA data is similar:

```
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_TIMESTAMP, SPCM_DIR_CARDTOPC, 0, pvBuffer, 0, lLenOfBufferInBytes);
```

In this example the notify size is set to zero, meaning that we don't want to be notified until all extra data has been transferred. Please have a look at the sample data transfer in an earlier chapter to see more details on the notify size.

Please note that extra data transfer is only possible from card to PC and there's no programmable offset available for this transfer.

M5i cards only:

On M5i cards the `lLenOfBufferInBytes` parameter needs to be an integer multiple of 64 bytes.



Buffer handling

A data buffer handshake is implemented in the driver which allows to run the card in different data transfer modes. The software transfer buffer is handled as one large buffer for each kind of data (timestamp and ABA) which is on the one side controlled by the driver and filled automatically by busmaster DMA from the hardware extra FIFO buffer and on the other hand it is handled by the user who set's parts of this software buffer available for the driver for further transfer. The handshake is fulfilled with the following 3 software registers:

Table 105: Spectrum API: ABA and Timestamp DMA buffer handling registers

Register	Value	Direction	Description
SPC_ABA_AVAIL_USER_LEN	210	read	This register contains the currently available number of bytes that are filled with newly transferred slow ABA data. The user can now use this ABA data for own purposes, copy it, write it to disk or start calculations with this data.
SPC_ABA_AVAIL_USER_POS	211	read	The register holds the current byte index position where the available ABA bytes start. The register is just intended to help you and to avoid own position calculation
SPC_ABA_AVAIL_CARD_LEN	212	write	After finishing the job with the new available ABA data the user needs to tell the driver that this amount of bytes is again free for new data to be transferred.
SPC_TS_AVAIL_USER_LEN	220	read	This register contains the currently available number of bytes that are filled with newly transferred timestamp data. The user can now use these timestamps for own purposes, copy it, write it to disk or start calculations with the timestamps.
SPC_TS_AVAIL_USER_POS	221	read	The register holds the current byte index position where the available timestamp bytes start. The register is just intended to help you and to avoid own position calculation
SPC_TS_AVAIL_CARD_LEN	222	write	After finishing the job with the new available timestamp data the user needs to tell the driver that this amount of bytes is again free for new data to be transferred.

Directly after start of transfer the `SPC_XXX_AVAIL_USER_LEN` is every time zero as no data is available for the user and the `SPC_XXX_AVAIL_CARD_LEN` is every time identical to the length of the defined buffer as the complete buffer is available for the card for transfer.

The counter that is holding the user buffer available bytes (`SPC_XXX_AVAIL_USER_LEN`) is sticking to the defined notify size at the `DefTransfer` call. Even when less bytes already have been transferred you won't get notice of it if the notify size is programmed to a higher value.



Remarks

- The transfer between hardware FIFO buffer and application buffer is done with scatter-gather DMA using a busmaster DMA controller located on the card. Even if the PC is busy with other jobs data is still transferred until the application buffer is completely used.
- As shown in the drawing above the DMA control will announce new data to the application by sending an event. Waiting for an event is done internally inside the driver if the application calls one of the wait functions. Waiting for an event does not consume any CPU time and is therefore highly requested if other threads do lot of calculation work. However it is not necessary to use the wait functions and one can simply request the current status whenever the program has time to do so. When using this polling mode the announced available

bytes still stick to the defined notify size!

- If the on-board FIFO buffer has an overrun data transfer is stopped immediately.

Buffer handling example for DMA timestamp transfer (ABA transfer is similar, just using other registers)

```
int8* pcData = (int8*) pvAllocMemPageAligned (lBufSizeInBytes);

// we now define the transfer buffer with the minimum notify size of one page = 4 kByte
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_TIMESTAMP, SPCM_DIR_CARDTOPC, 4096, (void*) pcData, 0, lBufSizeInBytes);

do
{
    // we wait for the next data to be available. After this call we get at least 4k of data to proceed
    dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_EXTRA_STARTDMA | M2CMD_EXTRA_WAITDMA);

    if (!dwError)
    {
        // if there was no error we can proceed and read out the current amount of available data
        spcm_dwGetParam_i32 (hDrv, SPC_TS_AVAIL_USER_LEN, &lAvailBytes);
        spcm_dwGetParam_i32 (hDrv, SPC_TS_AVAIL_USER_POS, &lBytePos);

        printf ("We now have %d new bytes available\n", lAvailBytes);
        printf ("The available data starts at position %d\n", lBytePos);

        // we take care not to go across the end of the buffer
        if ((lBytePos + lAvailBytes) >= lBufSizeInBytes)
            lAvailBytes = lBufSizeInBytes - lBytePos;

        // our do function gets a pointer to the start of the available data section and the length
        vProcessTimestamps (&pcData[lBytePos], lAvailBytes);

        // the buffer section is now immediately set available for the card
        spcm_dwSetParam_i32 (hDrv, SPC_TS_AVAIL_CARD_LEN, lAvailBytes);
    }
}
while (!dwError); // we loop forever if no error occurs
```

The extra FIFO has a quite small size compared to the main data buffer. As the transfer is done initiated by the hardware using busmaster DMA this is not critical as long as the application data buffers are large enough and as long as the extra transfer is started BEFORE starting the card.



Data Transfer using Polling

If the extra data is quite slow and the delay caused by the notify size on DMA transfers is unacceptable for your application it is possible to use the polling mode. Please be aware that the polling mode uses CPU processing power to get the data and that there might be an overrun if your CPU is otherwise busy. You should only use polling mode in special cases and if the amount of data to transfer is not too high.

Most of the functionality is similar to the DMA based transfer mode as explained above.

The polling data transfer mode is activated as soon as the M2CMD_EXTRA_POLL is executed.

Definition of the transfer buffer

This is similar to the above explained DMA buffer transfer. The value „notify size“ is ignored and should be set to 4k (4096).

Buffer handling

The buffer handling is also similar to the DMA transfer. As soon as one of the registers SPC_TS_AVAIL_USER_LEN or SPC_ABA_AVAIL_USER_LEN is read the driver will read out all available data from the hardware and will return the number of bytes that has been read. In minimum this will be one DWORD = 4 bytes.

Buffer handling example for polling timestamp transfer (ABA transfer is similar, just using other registers)

```

int8* pcData = (int8*) pvAllocMemPageAligned (lBufSizeInBytes);

// we now define the transfer buffer with the minimum notify size of one page = 4 kByte
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_TIMESTAMP, SPCM_DIR_CARDTOPC, 4096, (void*) pcData, 0, lBufSizeInBytes);

// we start the polling mode
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_EXTRA_POLL);

// this is our polling loop
do
{
    spcm_dwGetParam_i32 (hDrv, SPC_TS_AVAIL_USER_LEN, &lAvailBytes);
    spcm_dwGetParam_i32 (hDrv, SPC_TS_AVAIL_USER_POS, &lBytesPos);

    if (lAvailBytes > 0)
    {
        printf ("We now have %d new bytes available\n", lAvailBytes);
        printf ("The available data starts at position %d\n", lBytesPos);

        // we take care not to go across the end of the buffer
        if ((lBytesPos + lAvailBytes) >= lBufSizeInBytes)
            lAvailBytes = lBufSizeInBytes - lBytesPos;

        // our do function get's a pointer to the start of the available data section and the length
        vProcessTimestamps (&pcData[lBytesPos], lAvailBytes);

        // the buffer section is now immediately set available for the card
        spcm_dwSetParam_i32 (hDrv, SPC_TS_AVAIL_CARD_LEN, lAvailBytes);
    }
}
while (!dwError); // we loop forever if no error occurs

```

Comparison of DMA and polling commands

This chapter shows you how small the difference in programming is between the DMA and the polling mode:

	DMA mode	Polling mode
Define the buffer	spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_TIMESTAMP, SPCM_DIR...);	spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_TIMESTAMP, SPCM_DIR...);
Start the transfer	spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_EXTRA_STARTDMA)	spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_EXTRA_POLL)
Wait for data	spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_EXTRA_WAITDMA)	not in polling mode
Available bytes?	spcm_dwGetParam_i32 (hDrv, SPC_TS_AVAIL_USER_LEN, &lBytes);	spcm_dwGetParam_i32 (hDrv, SPC_TS_AVAIL_USER_LEN, &lBytes);
Min available bytes	programmed notify size	4 bytes
Current position?	spcm_dwGetParam_i32 (hDrv, SPC_TS_AVAIL_USER_LEN, &lBytes);	spcm_dwGetParam_i32 (hDrv, SPC_TS_AVAIL_USER_LEN, &lBytes);
Free buffer for card	spcm_dwSetParam_i32 (hDrv, SPC_TS_AVAIL_CARD_LEN, lBytes);	spcm_dwSetParam_i32 (hDrv, SPC_TS_AVAIL_CARD_LEN, lBytes);

Data format

Each timestamp is 128 bit long and internally mapped to two consecutive 64 bit (8 bytes) values. The lower 64 bit (counter value) contains the number of clocks that have been recorded with the currently used sampling rate since the last counter-reset has been done. The matching time can easily be calculated as described in the general information section at the beginning of this chapter.

The values the counter is counting and that are stored in the timestamp FIFO represent the moments the trigger event occurs internally. Compared to the real external trigger event, these values are delayed. This delay is fix and therefore can be ignored, as it will be identical for all recordings with the same setup.

Standard data format

When internally mapping the timestamp from 128 bit to two 64 bit values, the unused upper 64 bits are filled up with zeros.

Table 106: Spectrum API: timestamp sample format for standard mode

Timestamp Mode	16 th byte	...	11 th byte	10 th byte	9 th byte	8 th byte	7 th byte	6 th byte	5 th byte	4 th byte	3 rd byte	2 nd byte	1 st byte
Standard/StartReset	0h					64 bit wide Timestamp							
Refclock mode	0h					24 bit wide Refclock edge counter (seconds counter)				40 bit wide Timestamp			

Extended timestamp data format

Sometimes it is useful to store the level of additional external static signals together with a recording, such as e.g. control inputs of an external input multiplexer or settings of other external equipment. When programming a special flag the upper 64 bit of every 128 bit timestamp

value is not (as in standard data mode) filled up with leading zeros, but with the values of either the digital inputs (X3, X2, X1) or optionally also (X10 ..X3). The following table shows the resulting 128 bit timestamps.

Table 107: Spectrum API: timestamp sample format for extended mode

Timestamp Mode	16 th byte	...	14 th byte	13 th byte	...	9 th byte	8 th byte	7 th byte	6 th byte	5 th byte	4 th byte	3 rd byte	2 nd byte	1 st byte
Standard/StartReset	0h			Extra Data Word			64 bit wide Timestamp							
Refclock mode	0h			Extra Data Word			24 bit wide Refclock edge counter (seconds counter)				40 bit wide Timestamp			

The above mentioned „Extra Data Word“ contains the following 40bit wide data, depending on the selected timestamp data format:

Table 108: Spectrum API: timestamp extra data word format

Timestamp Data Format	Bit 39	...	Bit 32	Bit 31	...	Bit 26	Bit 25	...	Bit 16	Bit 15	...	Bit 13	Bit 12	Bit 11	Bit 10	...	Bit 0
no special data format is set	0h																
SPC_TSXIOACQ_ENABLE	X10 .. X3 (option)					0h					X2 .. X0					0h	
SPC_TSFEAT_TRGSR	0h					Trigger source bit-mask PXI sources (see table below)					0h					Trigger source bit-mask (Ch0 .. Force) (see table below)	
SPC_TSXIOACQ_ENABLE SPC_TSFEAT_TRGSR	X10 .. X3 (option)					0h					Trigger source bit-mask PXI sources (see table below)					X2 .. X0	

The multi-purpose lines X10...X3 are only available when the additional digital I/O option (DigSMA) is installed. For cards where this option is not installed, Bits 39 down to 32 are always zero. Depending on the chosen mode for acquiring digital data (see Multi-Purpose I/O chapter for details) either group (X2..X0) or (X10..X3) are active as digital input lines, whilst the inactive group is always zero.



The trigger sources are encoded as follows:

SPC_TRGSR_MASK_CH0	1h	Set when a trigger event occurring on channel 0 was leading to final trigger event.
SPC_TRGSR_MASK_CH1	2h	Set when a trigger event occurring on channel 1 was leading to final trigger event.
SPC_TRGSR_MASK_CH2	4h	Set when a trigger event occurring on channel 2 was leading to final trigger event.
SPC_TRGSR_MASK_CH3	8h	Set when a trigger event occurring on channel 3 was leading to final trigger event.
SPC_TRGSR_MASK_EXT0	100h	Set when a trigger event occurring on external trigger(Ext0) was leading to final trigger event.
SPC_TRGSR_MASK_EXT1	200h	Set when a trigger event occurring on external trigger(Ext1) was leading to final trigger event.
SPC_TRGSR_MASK_FORCE	400h	Set when a trigger event occurring by using the force trigger command is leading to final trigger event.
SPC_TRGSR_MASK_PXI0	10000h	M4x only: Set when a trigger event occurring on PXI trigger 0 was leading to final trigger event.
SPC_TRGSR_MASK_PXI1	20000h	M4x only: Set when a trigger event occurring on PXI trigger 1 was leading to final trigger event.
SPC_TRGSR_MASK_PXI2	40000h	M4x only: Set when a trigger event occurring on PXI trigger 2 was leading to final trigger event.
SPC_TRGSR_MASK_PXI3	80000h	M4x only: Set when a trigger event occurring on PXI trigger 3 was leading to final trigger event.
SPC_TRGSR_MASK_PXI4	100000h	M4x only: Set when a trigger event occurring on PXI trigger 4 was leading to final trigger event.
SPC_TRGSR_MASK_PXI5	200000h	M4x only: Set when a trigger event occurring on PXI trigger 5 was leading to final trigger event.
SPC_TRGSR_MASK_PXI6	400000h	M4x only: Set when a trigger event occurring on PXI trigger 6 was leading to final trigger event.
SPC_TRGSR_MASK_PXI7	800000h	M4x only: Set when a trigger event occurring on PXI trigger 7 was leading to final trigger event.
SPC_TRGSR_MASK_PXISTAR	1000000h	M4x only: Set when a trigger event occurring on PXI star-trigger was leading to final trigger event.
SPC_TRGSR_MASK_PXIDSTARB	2000000h	M4x only: Set when a trigger event occurring on PXI DStarB was leading to final trigger event.

Selecting the timestamp data format

Table 109: Spectrum API: timestamp data format register

Register	Value	Direction	Description
SPC_TIMESTAMP_CMD	47000	read/write	Programs a timestamp mode and performs commands as listed below
SPC_TSXIOACQ_ENABLE	1000h		Enables the trigger synchronous acquisition of the X0...X2 or X3...X10 inputs with every stored timestamp in the upper 64 bit.
SPC_TSFEAT_TRGSR	80000h		Enables the storage of the trigger source in the upper 64 bit of the timestamp value.

The selection between the different data format for the timestamps is done with a flag that is written to the timestamp command register. As this register is organized as a bitfield, the data format selection is available for all possible timestamp modes and different data modes can be combined.

Combination of Memory Segmentation Options with Timestamps

This topic should give you a brief overview how the timestamp option interacts with the options Multiple Recording and ABA mode for which the timestamps option has been made.

Multiple Recording and Timestamps

Multiple Recording is well matching with the timestamp option. If timestamp recording is activated each trigger event and therefore each Multiple Recording segment will get timestamped as shown in the drawing on the right.

Please keep in mind that the trigger events are timestamped, not the beginning of the acquisition. The first sample that is available is at the time position of [Timestamp - Pretrigger].

The programming details of the timestamp option is explained in an extra chapter.

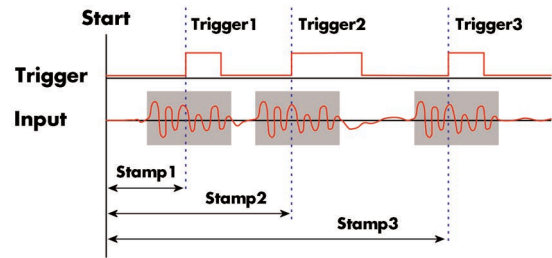


Image 63: drawing of Multiple Recording Acquisition with Timestamps

The following example shows the setup of the Multiple Recording mode together with activated timestamps recording and a short display of the acquired timestamps. The example doesn't care for the acquired data itself and doesn't check for error:

```
// setup of the Multiple Recording mode
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_MULTI); // Enable Standard Multiple Recording
spcm_dwSetParam_i64 (hDrv, SPC_SEGMENTSIZE, 1024); // Segment size is 1 kSamples, Posttrigger is 768
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, 768); // samples and pretrigger therefore 256 samples.
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, 4096); // 4 kSamples in total acquired -> 4 segments

// setup the Timestamp mode and make a reset of the timestamp counter
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TSMODE_STANDARD | SPC_TSCNT_INTERNAL);
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TSMODE_RESET);

// now we define a buffer for timestamp data and start the acquisition. Each timestamp is 128 bit = 16 bytes.
int64* pllStamps = (int64*) pvAllocMemPageAligned (16 * 4);
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_TIMESTAMP, SPCM_DIR_CARDTOPC, 0, (void*) pllStamps, 0, 4 * 16);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_EXTRA_STARTDMA);

// we wait for the end timestamps transfer which will be received if all segments have been recorded
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_EXTRA_WAITDMA);

// as we now have the timestamps we just print them and calculate the time in milli seconds
// for simplicity only the lower 64 bit part of the 128 bit stamp is used, hence only every
// second array element of pllStamps is used here.
int64 llSamplerate;
double dTime_ms;
spcm_dwGetParam_i64 (hDrv, SPC_SAMPLERATE, &llSamplerate);

for (int i = 0; i < 4; i++)
{
    dTime_ms = 1000.0 * pllStamps[2 * i] / llSamplerate;

    printf ("##d: %I64d samples = %.3f ms\n", i, pllStamps[2 * i], dTime_ms);
}
```

Option Star-Hub (M3i and M4i only)

Star-Hub introduction

The purpose of the Star-Hub is to extend the number of channels available for acquisition or generation by interconnecting multiple cards and running them simultaneously.

The Star-Hub option allows to synchronize several cards of the same M3i/M4i series that are mounted within one host system (PC):

- For the M3i series there are the two different versions available: a small version with 4 connectors (option SH4) for synchronizing up to four cards and a big version with 8 connectors (option SH8) for synchronizing up to eight cards.
- For the M4i series there are the two different mechanical versions available, with 8 connectors for synchronizing up to eight cards.

The Star-Hub allows synchronizing cards of the same family only. It is not possible to synchronize cards of different families!



Both versions are implemented as a piggy-back module that is mounted to one of the cards. For details on how to install several cards including the one carrying the Star-Hub module, please refer to the section on hardware installation.

Either which of the two available Star-Hub options is used, there will be no phase delay between the sampling clocks of the synchronized cards and either no delay between the trigger events. The card holding the Star-Hub is automatically also the clock master. Any one of the synchronized cards can be part of the trigger generation.

Star-Hub trigger engine

The trigger bus between an M3i/M4i card and the Star-Hub option consists of several lines. Some of them send the trigger information from the card's trigger engine to the Star-Hub and some receives the resulting trigger from the Star-Hub. All trigger events from the different cards connected are combined with OR on the Star-Hub.

While the returned trigger is identical for all synchronized cards, the sent out trigger of every single card depends on their trigger settings.

Star-Hub clock engine

The card holding the Star-Hub is the clock master for the complete system. If you need to feed in an external clock to a synchronized system the clock has to be connected to the master card. Slave cards cannot generate a Star-Hub system clock. As shown in the drawing on the right the clock master can use either the programmable quartz 1 or the external clock input to be broadcast to all other cards.

All cards including the clock master itself receive the distributed clock with equal phase information. This makes sure that there is no phase delay between the cards.

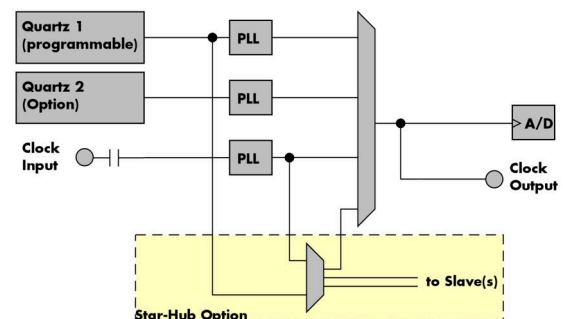


Table 110: star-hub clock overview diagram

Software Interface

The software interface is similar to the card software interface that is explained earlier in this manual. The same functions and some of the registers are used with the Star-Hub. The Star-Hub is accessed using its own handle which has some extra commands for synchronization setup. All card functions are programmed directly on card as before. There are only a few commands that need to be programmed directly to the Star-Hub for synchronization.

The software interface as well as the hardware supports multiple Star-Hubs in one system. Each set of cards connected by a Star-Hub then runs totally independent. It is also possible to mix cards that are connected with the Star-Hub with other cards that run independent in one system.

Star-Hub Initialization

The interconnection between the Star-Hubs is probed at driver load time and does not need to be programmed separately. Instead the cards can be accessed using a logical index. This card index is only based on the ordering of the cards in the system and is not influenced by the current cabling. It is even possible to change the cable connections between two system starts without changing the logical card order that is used for Star-Hub programming.

The Star-Hub initialization must be done AFTER initialization of all cards in the system. Otherwise the inter-connection won't be received properly.



The Star-Hubs are accessed using a special device name „sync“ followed by the index of the star-hub to access. The Star-Hub is handled completely like a physical card allowing all functions based on the handle like the card itself.

Example with 4 cards and one Star-Hub (no error checking to keep example simple)

```
drv_handle hSync;
drv_handle hCard[4];

for (i = 0; i < 4; i++)
{
    sprintf (s, "/dev/spcm%d", i);
    hCard[i] = spcm_hOpen (s);
}
hSync = spcm_hOpen ("sync0");

...

spcm_vClose (hSync);
for (i = 0; i < 4; i++)
    spcm_vClose (hCard[i]);
```

Example for a digitizerNETBOX or generatorNETBOX with two internal digitizer/generator modules, This example is also suitable for accessing a remote server with two cards installed:

```
drv_handle hSync;
drv_handle hCard[2];

for (i = 0; i < 2; i++)
{
    sprintf (s, "TCPIP::192.168.169.14::INST%d::INSTR", i);
    hCard[i] = spcm_hOpen (s);
}
hSync = spcm_hOpen ("sync0");

...

spcm_vClose (hSync);
for (i = 0; i < 2; i++)
    spcm_vClose (hCard[i]);
```

When opening the Star-Hub the cable interconnection is checked. The Star-Hub may return an error if it sees internal cabling problems or if the connection between Star-Hub and the card that holds the Star-Hub is broken. It can't identify broken connections between Star-Hub and other cards as it doesn't know that there has to be a connection.

The synchronization setup is done using bit masks where one bit stands for one recognized card. All cards that are connected with a Star-Hub are internally numbered beginning with 0. The number of connected cards as well as the connections of the star-hub can be read out after initialization. For each card that is connected to the star-hub one can read the index of that card:

Table 111: Spectrum API: star-hub related registers for reading detected connections

Register	Value	Direction	Description
SPC_SYNC_READ_NUMCONNECTORS	48991	read	Number of connectors that the Star-Hub offers at max. (available with driver V5.6 or newer)
SPC_SYNC_READ_SYNCCOUNT	48990	read	Number of cards that are connected to this Star-Hub
SPC_SYNC_READ_CARDIDX0	49000	read	Index of card that is connected to star-hub logical index 0 (mask 0x0001)
SPC_SYNC_READ_CARDIDX1	49001	read	Index of card that is connected to star-hub logical index 1 (mask 0x0002)
...		read	...
SPC_SYNC_READ_CARDIDX7	49007	read	Index of card that is connected to star-hub logical index 7 (mask 0x0080)
SPC_SYNC_READ_CARDIDX8	49008	read	M2i only: Index of card that is connected to star-hub logical index 8 (mask 0x0100)
...		read	...
SPC_SYNC_READ_CARDIDX15	49015	read	M2i only: Index of card that is connected to star-hub logical index 15 (mask 0x8000)
SPC_SYNC_READ_CABLECON0		read	Returns the index of the cable connection that is used for the logical connection 0. The cable connections can be seen printed on the PCB of the star-hub. Use these cable connection information in case that there are hardware failures with the star-hub cabling.
...	49100	read	...
SPC_SYNC_READ_CABLECON15	49115	read	Returns the index of the cable connection that is used for the logical connection 15.

In standard systems where all cards are connected to one star-hub reading the star-hub logical index will simply return the index of the card again. This results in bit 0 of star-hub mask being 1 when doing the setup for card 0, bit 1 in star-hub mask being 1 when setting up card 1

and so on. On such systems it is sufficient to read out the SPC_SYNC_READ_SYNCCOUNT register to check whether the star-hub has found the expected number of cards to be connected.

```
spcm_dwGetParam_i32 (hSync, SPC_SYNC_READ_SYNCCOUNT, &lSyncCount);
for (i = 0; i < lSyncCount; i++)
{
    spcm_dwGetParam_i32 (hSync, SPC_SYNC_READ_CARDIDX0 + i, &lCardIdx);
    printf ("star-hub logical index %d is connected with card %d\n", i, lCardIdx);
}
```

In case of 4 cards in one system and all are connected with the star-hub this program excerpt will return:

```
star-hub logical index 0 is connected with card 0
star-hub logical index 1 is connected with card 1
star-hub logical index 2 is connected with card 2
star-hub logical index 3 is connected with card 3
```

Let's see a more complex example with two Star-Hubs and one independent card in one system. Star-Hub A connects card 2, card 4 and card 5. Star-Hub B connects card 0 and card 3. Card 1 is running completely independent and is not synchronized at all:

card	Star-Hub connection	card handle	star-hub handle	card index in star-hub	mask for this card in star-hub
card 0	-	/dev/spcm0		0 (of star-hub B)	0x0001
card 1	-	/dev/spcm1			-
card 2	star-hub A	/dev/spcm2	sync0	0 (of star-hub A)	0x0001
card 3	star-hub B	/dev/spcm3	sync1	1 (of star-hub B)	0x0002
card 4	-	/dev/spcm4		1 (of star-hub A)	0x0002
card 5	-	/dev/spcm5		2 (of star-hub A)	0x0004

Now the program has to check both star-hubs:

```
for (j = 0; j < lStarhubCount; j++)
{
    spcm_dwGetParam_i32 (hSync[j], SPC_SYNC_READ_SYNCCOUNT, &lSyncCount);
    for (i = 0; i < lSyncCount; i++)
    {
        spcm_dwGetParam_i32 (hSync[j], SPC_SYNC_READ_CARDIDX0 + i, &lCardIdx);
        printf ("star-hub %c logical index %d is connected with card %d\n", (!j ? 'A' : 'B'), i, lCardIdx);
    }
    printf ("\n");
}
```

In case of the above mentioned cabling this program excerpt will return:

```
star-hub A logical index 0 is connected with card 2
star-hub A logical index 1 is connected with card 4
star-hub A logical index 2 is connected with card 5

star-hub B logical index 0 is connected with card 0
star-hub B logical index 1 is connected with card 3
```

For the following examples we will assume that 4 cards in one system are all connected to one star-hub to keep things easier.

Setup of Synchronization

The synchronization setup only requires one additional register to enable the cards that are synchronized in the next run

Table 112: Spectrum API: synchronization enable mask register

Register	Value	Direction	Description
SPC_SYNC_ENABLEMASK	49200	read/write	Mask of all cards that are enabled for the synchronization

The enable mask is based on the logical index explained above. It is possible to just select a couple of cards for the synchronization. All other cards then will run independently. Please be sure to always enable the card on which the star-hub is located as this one is a must for the synchronization.

In our example we synchronize all four cards. The star-hub is located on card #2 and is therefore the clock master

```
spcm_dwSetParam_i32 (hSync, SPC_SYNC_ENABLEMASK, 0x000F); // all 4 cards are masked

// set the clock master to 100 MS/s internal clock
spcm_dwSetParam_i32 (hCard[2], SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hCard[2], SPC_SAMPLERATE, MEGA(100));

// set all the slaves to run synchronously with 100 MS/s
spcm_dwSetParam_i32 (hCard[0], SPC_SAMPLERATE, MEGA(100));
spcm_dwSetParam_i32 (hCard[1], SPC_SAMPLERATE, MEGA(100));
spcm_dwSetParam_i32 (hCard[3], SPC_SAMPLERATE, MEGA(100));
```

Setup of Trigger

Setting up the trigger does not need any further steps of synchronization setup. Simply all trigger settings of all cards that have been enabled for synchronization are connected together. All trigger sources and all trigger modes can be used on synchronization as well.

Having positive edge of external trigger on card 0 to be the trigger source for the complete system needs the following setup:

```
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_ORMASK, SPC_TMASK_EXT0);
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_EXT0_MODE, SPC_TM_POS);

spcm_dwSetParam_i32 (hCard[1], SPC_TRIG_ORMASK, SPC_TM_NONE);
spcm_dwSetParam_i32 (hCard[2], SPC_TRIG_ORMASK, SPC_TM_NONE);
spcm_dwSetParam_i32 (hCard[3], SPC_TRIG_ORMASK, SPC_TM_NONE);
```

Assuming that the 4 cards are analog data acquisition cards with 4 channels each we can simply setup a synchronous system with all channels of all cards being trigger source. The following setup will show how to set up all trigger events of all channels to be OR connected. If any of the channels will now have a signal above the programmed trigger level the complete system will do an acquisition:

```
for (i = 0; i < lSyncCount; i++)
{
    int32 lAllChannels = (SPC_TMASK0_CH0 | SPC_TMASK0_CH1 | SPC_TMASK0_CH2 | SPC_TMASK0_CH3);
    spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_CH_ORMASK0, lAllChannels);
    for (j = 0; j < 2; j++)
    {
        // set all channels to trigger on positive edge crossing trigger level 100
        spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_CH0_MODE + j, SPC_TM_POS);
        spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_CH0_LEVEL0 + j, 100);
    }
}
```

Run the synchronized cards

Running of the cards is very simple. The star-hub acts as one big card containing all synchronized cards. All card commands have to be omitted directly to the star-hub which will check the setup, do the synchronization and distribute the commands in the correct order to all synchronized cards. The same card commands can be used that are also possible for single cards:

Table 113: Spectrum API: star-hub synchronization commands

Register	Value	Direction	Description
SPC_M2CMD	100	write only	Executes a command for the card or data transfer
M2CMD_CARD_RESET	1h		Performs a hard and software reset of the card as explained further above
M2CMD_CARD_WRITESUP	2h		Writes the current setup to the card without starting the hardware. This command may be useful if changing some internal settings like clock frequency and enabling outputs.
M2CMD_CARD_START	4h		Starts the card with all selected settings. This command automatically writes all settings to the card if any of the settings has been changed since the last one was written. After card has been started none of the settings can be changed while the card is running.
M2CMD_CARD_ENABLETRIGGER	8h		The trigger detection is enabled. This command can be either send together with the start command to enable trigger immediately or in a second call after some external hardware has been started.
M2CMD_CARD_FORCETRIGGER	10h		This command forces a trigger even if none has been detected so far. Sending this command together with the start command is similar to using the software trigger.
M2CMD_CARD_DISABLETRIGGER	20h		The trigger detection is disabled. All further trigger events are ignored until the trigger detection is again enabled. When starting the card the trigger detection is started disabled.
M2CMD_CARD_STOP	40h		Stops the current run of the card. If the card is not running this command has no effect.

All other commands and settings need to be send directly to the card that it refers to.

This example shows the complete setup and synchronization start for our four cards:

```

spcm_dwSetParam_i32 (hSync, SPC_SYNC_ENABLEMASK, 0x000F); // all 4 cards are masked

// to keep it easy we set all card to the same clock and disable trigger
for (i = 0; i < 4; i++)
{
    spcm_dwSetParam_i32 (hCard[i], SPC_CLOCKMODE, SPC_CM_INTPLL);
    spcm_dwSetParam_i32 (hCard[i], SPC_SAMPLERATE, MEGA(100));
    spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_ORMASK, SPC_TM_NONE);
}

// card 0 is trigger master and waits for external positive edge
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_ORMASK, SPC_TMASK_EXT0);
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_EXT0_MODE, SPC_TM_POS);

// start the cards and wait for them a maximum of 1 second to be ready
spcm_dwSetParam_i32 (hSync, SPC_TIMEOUT, 1000);
spcm_dwSetParam_i32 (hSync, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);
if (spcm_dwSetParam_i32 (hSync, SPC_M2CMD, M2CMD_CARD_WAITREADY) == ERR_TIMEOUT)
    printf ("Timeout occurred - no trigger received within time\n");

```

Using one of the wait commands for the Star-Hub will return as soon as the card holding the Star-Hub has reached this state. However when synchronizing cards with different memory sizes there may be other cards that still haven't reached this level.



SH-Direct: using the Star-Hub clock directly without synchronization

Starting with driver version 1.26 build 1754 it is possible to use the clock from the Star-Hub just like an external clock and running one or more cards totally independent of the synchronized card. The mode is by example useful if one has one or more output cards that run continuously in a loop and are synchronized with Star-Hub and in addition to this one or more acquisition cards should make multiple acquisitions but using the same clock.

For all M2i cards is also possible to run the „slave“ cards with a divided clock. Therefore please program a desired divided sampling rate in the SPC_SAMPLERATE register (example: running the Star-Hub card with 10 MS/s and the independent cards with 1 MS/s). The sampling rate is automatically adjusted by the driver to the next matching value.

What is necessary?

- All cards need to be connected to the Star-Hub
- The card(s) that should run independently can not hold the Star-Hub
- The card(s) with the Star-Hub must be setup to synchronization even if it's only one card
- The synchronized card(s) have to be started prior to the card(s) that run with the direct Star-Hub clock

Setup

At first all cards that should run synchronized with the Star-Hub are set-up exactly as explained before. The card(s) that should run independently and use the Star-Hub clock need to use the following clock mode:

Table 114: Spectrum API: clock mode register and settings for SH Direct mode

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_SHDIRECT	128		Uses the clock from the Star-Hub as if this was an external clock

When using SH_Direct mode, the register call to SPC_CLOCKMODE enabling this mode must be written before initiating a card start command to any of the connected cards. Also it is not allowed to be modified later in the programming sequence to prevent the driver from calculating wrong sample rates.



Example

In this example we have one generator card with the Star-Hub mounted running in a continuous loop and one acquisition card running independently using the SH-Direct clock.

```
// setup of the generator card
spcm_dwSetParam_i32 (hCard[0], SPC_CARDMODE, SPC_REC_STD_SINGLE);
spcm_dwSetParam_i32 (hCard[0], SPC_LOOPS, 0); // infinite data replay
spcm_dwSetParam_i32 (hCard[0], SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hCard[0], SPC_SAMPLERATE, MEGA(1));
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_ORMASK, SPC_TM_SOFTWARE);

spcm_dwSetParam_i32 (hSync, SPC_SYNC_ENABLEMASK, 0x0001); // card 0 is the generator card
spcm_dwSetParam_i32 (hSync, SPC_SYNC_CLKMASK, 0x0001); // only for M2i/M3i cards: set ClkMask

// Setup of the acquisition card (waiting for external trigger)
spcm_dwSetParam_i32 (hCard[1], SPC_CARDMODE, SPC_REC_STD_SINGLE);
spcm_dwSetParam_i32 (hCard[1], SPC_CLOCKMODE, SPC_CM_SHDIRECT);
spcm_dwSetParam_i32 (hCard[1], SPC_SAMPLERATE, MEGA(1));
spcm_dwSetParam_i32 (hCard[1], SPC_TRIG_ORMASK, SPC_TM_MASK_EXT0);
spcm_dwSetParam_i32 (hCard[1], SPC_TRIG_EXT0_MODE, SPC_TM_POS);

// now start the generator card (sync!) first and then the acquisition card
spcm_dwSetParam_i32 (hSync, SPC_TIMEOUT, 1000);
spcm_dwSetParam_i32 (hSync, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);

// start first acquisition
spcm_dwSetParam_i32 (hCard[1], SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_CARD_WAITREADY);

// process data

// start next acquisition
spcm_dwSetParam_i32 (hCard[1], SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_CARD_WAITREADY);

// process data
```

Error Handling

The Star-Hub error handling is similar to the card error handling and uses the function `spcm_dwGetErrorInfo_i32`. Please see the example in the card error handling chapter to see how the error handling is done.

Option Remote Server

Introduction

Using the Spectrum Remote Server (order code -SPc-RServer) it is possible to access the M2i/M3i/M4i/M4x/M2p/M5i card(s) installed in one PC (server) from another PC (client) via local area network (LAN), similar to using a digitizerNETBOX, generatorNETBOX or hybridNETBOX.

It is possible to use different operating systems on both server and client. For example the Remote Server is running on a Linux system and the client is accessing them from a Windows system.

The Remote Server software requires, that the option „SPc-RServer“ is installed on at least one card installed within the server side PC. You can either check this with the Control Center in the "Installed Card features" node or by reading out the feature register, as described in the „Installed features and options“ passage, earlier in this manual.

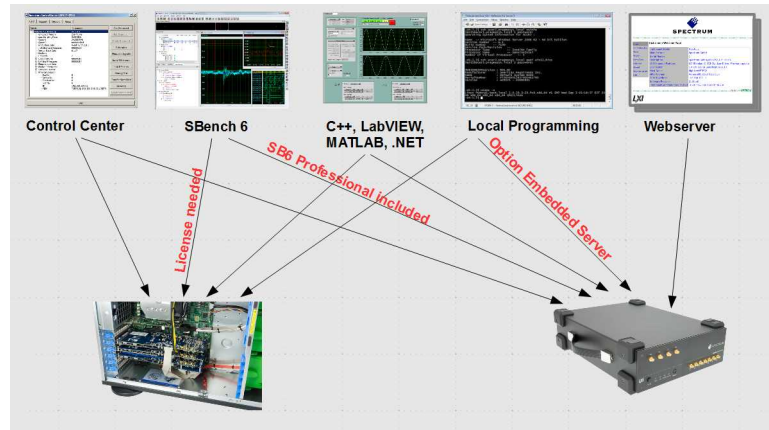


Image 64: Overview of remote server option interaction in comparison to NETBOX devices

To run the Remote Server software, it is required to have least version 3.18 of the Spectrum SPCM driver installed. Additionally at least on one card in the server PC the feature flag SPCM_FEAT_REMOTESERVER must be set.



Installing and starting the Remote Server

Windows

Windows users find the Control Center installer on the USB-Stick under „Install\win\spcm_remote_install.exe“. After the installation has finished there will be a new start menu entry in the Folder "Spectrum GmbH" to start the Remote Server. To start the Remote Server automatically after login, just copy this shortcut to the Autostart directory.

Linux

Linux users find the versions of the installer for the different StdC libraries under under /Install/linux/spcm_control_center/ as RPM packages.

To start the Remote Server type "spcm_remote_server" (without quotation marks). To start the Remote Server automatically after login, add the following line to the .bashrc or .profile file (depending on the used Linux distribution) in the user's home directory:

```
spcm_remote_server&
```

```
Spectrum Remote Server (32-bit)
SPCM Remote Server 1.3.17767
Providing access to the following cards:
M2p.5968-x4 sn13573
Listening on the following IPs:
192.168.56.1
192.168.169.42
Remote Server started successfully.
Press q + Enter to stop the Remote Server and close the program.
```

Detecting the digitizerNETBOX/generatorNETBOX/hybridNETBOX

Before accessing the digitizerNETBOX/generatorNETBOX/hybridNETBOX one has to determine the IP address of the device. Normally that can be done using one of the two methods described below:

Discovery Function

The digitizerNETBOX/generatorNETBOX/hybridNETBOX responds to the VISA described Discovery function. The next chapter will show how to install and use the Spectrum control center to execute the discovery function and to find the Spectrum hardware. As the discovery function is a standard feature of all LXI devices there are other software packages that can find the device using the discovery function:

- Spectrum control center (limited to Spectrum remote products)
- free LXI System Discovery Tool from the LXI consortium (www.lxi-standard.org)
- Measurement and Automation Explorer from National Instruments (NI MAX)
- Keysight Connection Expert from Keysight Technologies

Additionally the discovery procedure can also be started from ones own specific application:

```
#define TIMEOUT_DISCOVERY 5000 // timeout value in ms

const uint32 dwMaxNumRemoteCards = 50;

char* pszVisa[dwMaxNumRemoteCards] = { NULL };
char* pszIdn[dwMaxNumRemoteCards] = { NULL };

const uint32 dwMaxIdnStringLen = 256;
const uint32 dwMaxVisaStringLen = 50;

// allocate memory for string list
for (uint32 i = 0; i < dwMaxNumRemoteCards; i++)
{
    pszVisa[i] = new char [dwMaxVisaStringLen];
    pszIdn[i] = new char [dwMaxIdnStringLen];
    memset (pszVisa[i], 0, dwMaxVisaStringLen);
    memset (pszIdn[i], 0, dwMaxIdnStringLen);
}

// first make discovery - check if there are any LXI compatible remote devices
dwError = spcm_dwDiscovery ((char**)pszVisa, dwMaxNumRemoteCards, dwMaxVisaStringLen, TIMEOUT_DISCOVERY);

// second: check from which manufacturer the devices are
spcm_dwSendIDNRequest ((char**)pszIdn, dwMaxNumRemoteCards, dwMaxIdnStringLen);

// Use the VISA strings of these devices with Spectrum as manufacturer
// for accessing remote devices without previous knowledge of their IP address
```

Finding the digitizerNETBOX/generatorNETBOX/hybridNETBOX in the network

As the digitizerNETBOX/generatorNETBOX/hybridNETBOX is a standard network device it has its own IP address and host name and can be found in the computer network. The standard host name consist of the model type and the serial number of the device. The serial number is also found on the type plate on the back of the digitizerNETBOX/generatorNETBOX/hybridNETBOX chassis.

As default DHCP (IPv4) will be used and an IP address will be automatically set. In case no DHCP server is found, an IP will be obtained using the AutoIP feature. This will lead to an IPv4 address of 169.254.x.y (with x and y being assigned to a free IP in the network) using a subnet mask of 255.255.0.0.

The default IP setup can also be restored, by using the „LAN Reset“ button on the device.

If a fixed IP address should be used instead, the parameters need to be set according to the current LAN requirements.

Windows 7, Windows 8, Windows 10 and Windows 11

Under Windows 7, Windows 8, Windows 10 and Windows 11 the digitizerNETBOX, generatorNETBOX and hybridNETBOX devices are listed under the „other devices“ tree with their given host name.

A right click on the digitizerNETBOX or generatorNETBOX device opens the properties window where you find further information on the device including the IP address.

From here it is possible to go the website of the device where all necessary information are found to access the device from software.

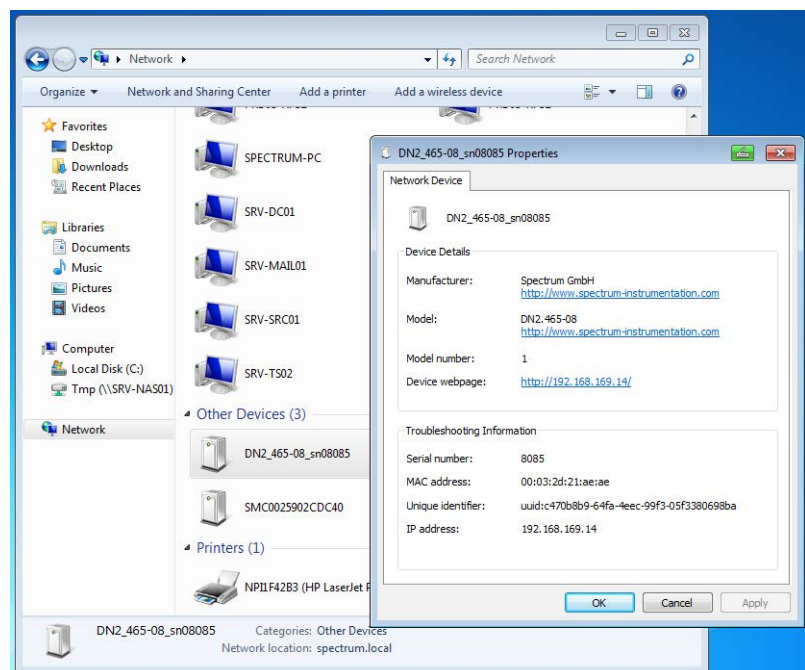


Image 65: Windows screenshot: finding a remote Spectrum device like digitizerNETBOX

Troubleshooting

If the above methods do not work please try one of the following steps:

- Ask your network administrator for the IP address of the digitizerNETBOX/generatorNETBOX and access it directly over the IP address.
- Check your local firewall whether it allows access to the device and whether it allows to access the ports listed in the technical data section.
- Check with your network administrator whether the subnet, the device and the ports that are listed in the technical data section are accessible from your system due to company security settings.

Accessing remote cards

To detect remote card(s) from the client PC, start the Spectrum Control Center on the client and click "Netbox Discovery". All discovered cards will be listed under the "Remote" node.

Using remote cards instead of using local ones is as easy as using a digitizerNETBOX and only requires a few lines of code to be changed compared to using local cards.

Instead of opening two locally installed cards like this:

```
hDrv0 = spcm_hOpen ("/dev/spcm0"); // open local card spcm0
hDrv1 = spcm_hOpen ("/dev/spcm1"); // open local card spcm1
```

one would call `spcm_hOpen()` with a VISA string as a parameter instead:

```
hDrv0 = spcm_hOpen ("TCPIP::192.168.1.2::inst0::INSTR"); // open card spcm0 on a Remote Server PC
hDrv1 = spcm_hOpen ("TCPIP::192.168.1.2::inst1::INSTR"); // open card spcm1 on a Remote Server PC
```

to open cards on the Remote Server PC with the IP address 192.168.1.2. The driver will take care of all the network communication.

Appendix

Error Codes

The following error codes could occur when a driver function has been called. Please check carefully the allowed setup for the register and change the settings to run the program.

Table 115: Spectrum API: driver error codes and error description

error name	value (hex)	value (dec.)	error description
ERR_OK	0h	0	Execution OK, no error.
ERR_INIT	1h	1	An error occurred when initializing the given card. Either the card has already been opened by another process or an hardware error occurred.
ERR_TYP	3h	3	Initialization only: The type of board is unknown. This is a critical error. Please check whether the board is correctly plugged in the slot and whether you have the latest driver version.
ERR_FNCNOTSUPPORTED	4h	4	This function is not supported by the hardware version.
ERR_BRDREMAP	5h	5	The board index re map table in the registry is wrong. Either delete this table or check it carefully for double values.
ERR_KERNELVERSION	6h	6	The version of the kernel driver is not matching the version of the DLL. Please do a complete re-installation of the hardware driver. This error normally only occurs if someone copies the driver library and the kernel driver manually.
ERR_HWRVVERSION	7h	7	The hardware needs a newer driver version to run properly. Please install the driver that was delivered together with the card.
ERR_ADRRANGE	8h	8	One of the address ranges is disabled (fatal error), can only occur under Linux.
ERR_INVALIDHANDLE	9h	9	The used handle is not valid.
ERR_BOARDNOTFOUND	Ah	10	A card with the given name has not been found.
ERR_BOARDINUSE	Bh	11	A card with given name is already in use by another application.
ERR_EXPHW64BITADR	Ch	12	Express hardware version not able to handle 64 bit addressing -> update needed.
ERR_FWVERSION	Dh	13	Firmware versions of synchronized cards or for this driver do not match -> update needed.
ERR_SYNCPROTOCOL	Eh	14	Synchronization protocol versions of synchronized cards do not match -> update needed
ERR_LASTERR	10h	16	Old error waiting to be read. Please read the full error information before proceeding. The driver is locked until the error information has been read.
ERR_BOARDINUSE	11h	17	Board is already used by another application. It is not possible to use one hardware from two different programs at the same time.
ERR_ABORT	20h	32	Abort of wait function. This return value just tells that the function has been aborted from another thread. The driver library is not locked if this error occurs.
ERR_BOARDLOCKED	30h	48	The card is already in access and therefore locked by another process. It is not possible to access one card through multiple processes. Only one process can access a specific card at the time.
ERR_DEVICE_MAPPING	32h	50	The device is mapped to an invalid device. The device mapping can be accessed via the Control Center.
ERR_NETWORKSETUP	40h	64	The network setup of a digitizerNETBOX has failed.
ERR_NETWORKTRANSFER	41h	65	The network data transfer from/to a digitizerNETBOX has failed.
ERR_FWPOWERCYCLE	42h	66	Power cycle (PC off/on) is needed to update the card's firmware (a simple OS reboot is not sufficient !)
ERR_NETWORKTIMEOUT	43h	67	A network timeout has occurred.
ERR_BUFFERSIZE	44h	68	The buffer size is not sufficient (too small).
ERR_RESTRICTEDACCESS	45h	69	The access to the card has been intentionally restricted.
ERR_INVALIDPARAM	46h	70	An invalid parameter has been used for a certain function.
ERR_TEMPERATURE	47h	71	The temperature of at least one of the card's sensors measures a temperature, that is too high for the hardware.
ERR_REG	100h	256	The register is not valid for this type of board.
ERR_VALUE	101h	257	The value for this register is not in a valid range. The allowed values and ranges are listed in the board specific documentation.
ERR_FEATURE	102h	258	Feature (option) is not installed on this board. It's not possible to access this feature if it's not installed.
ERR_SEQUENCE	103h	259	Command sequence is not allowed. Please check the manual carefully to see which command sequences are possible.
ERR_READABORT	104h	260	Data read is not allowed after aborting the data acquisition.
ERR_NOACCESS	105h	261	Access to this register is denied. This register is not accessible for users.
ERR_TIMEOUT	107h	263	A timeout occurred while waiting for an interrupt. This error does not lock the driver.
ERR_CALLTYPE	108h	264	The access to the register is only allowed with one 64 bit access but not with the multiplexed 32 bit (high and low double word) version.
ERR_EXCEEDSINT32	109h	265	The return value is int32 but the software register exceeds the 32 bit integer range. Use double int32 or int64 accesses instead, to get correct return values.
ERR_NOWRITEALLOWED	10Ah	266	The register that should be written is a read-only register. No write accesses are allowed.
ERR_SETUP	10Bh	267	The programmed setup for the card is not valid. The error register will show you which setting generates the error message. This error is returned if the card is started or the setup is written.
ERR_CLOCKNOTLOCKED	10Ch	268	Synchronization to external clock failed: no signal connected or signal not stable. Please check external clock or try to use a different sampling clock to make the PLL locking easier.
ERR_MEMINIT	10Dh	269	On-board memory initialization error. Power cycle the PC and try another PCIe slot (if possible). In case that the error persists, please contact Spectrum support for further assistance.
ERR_POWERSUPPLY	10Eh	270	On-board power supply error. Power cycle the PC and try another PCIe slot (if possible). In case that the error persists, please contact Spectrum support for further assistance.
ERR_ADCCOMMUNICATION	10Fh	271	Communication with ADC failed. Power cycle the PC and try another PCIe slot (if possible). In case that the error persists, please contact Spectrum support for further assistance.
ERR_CHANNEL	110h	272	The channel number may not be accessed on the board: Either it is not a valid channel number or the channel is not accessible due to the current setup (e.g. Only channel 0 is accessible in interlace mode)

Table 115: Spectrum API: driver error codes and error description

error name	value (hex)	value (dec.)	error description
ERR_NOTIFYSIZE	111h	273	The notify size of the last spcm_dwDefTransfer call is not valid. The notify size must be a multiple of the page size of 4096. For data transfer it may also be a fraction of 4k in the range of 16, 32, 64, 128, 256, 512, 1k or 2k. For ABA and timestamp the notify size can be 2k as a minimum.
ERR_RUNNING	120h	288	The board is still running, this function is not available now or this register is not accessible now.
ERR_ADJUST	130h	304	Automatic card calibration has reported an error. Please check the card inputs.
ERR_PRETRIGGERLEN	140h	320	The calculated pretrigger size (resulting from the user defined posttrigger values) exceeds the allowed limit.
ERR_DIRMISMATCH	141h	321	The direction of card and memory transfer mismatch. In normal operation mode it is not possible to transfer data from PC memory to card if the card is an acquisition card nor it is possible to transfer data from card to PC memory if the card is a generation card.
ERR_POSTEXCDSEGMENT	142h	322	The posttrigger value exceeds the programmed segment size in multiple recording/ABA mode. A delay of the multiple recording segments is only possible by using the delay trigger!
ERR_SEGMENTINMEM	143h	323	Memsizes is not a multiple of segment size when using Multiple Recording/Replay or ABA mode. The programmed segment size must match the programmed memory size.
ERR_MULTIPLEPW	144h	324	Multiple pulsewidth counters used but card only supports one at the time.
ERR_NOCHANNELPWOR	145h	325	The channel pulsewidth on this card can't be used together with the OR conjunction. Please use the AND conjunction of the channel trigger sources.
ERR_ANDORMASKOVLAP	146h	326	Trigger AND mask and OR mask overlap in at least one channel. Each trigger source can only be used either in the AND mask or in the OR mask, no source can be used for both.
ERR_ANDMASKEDGE	147h	327	One channel is activated for trigger detection in the AND mask but has been programmed to a trigger mode using an edge trigger. The AND mask can only work with level trigger modes.
ERR_ORMASKLEVEL	148h	328	One channel is activated for trigger detection in the OR mask but has been programmed to a trigger mode using a level trigger. The OR mask can only work together with edge trigger modes.
ERR_EDGEPERMOD	149h	329	This card is only capable to have one programmed trigger edge for each module that is installed. It is not possible to mix different trigger edges on one module.
ERR_DOLEVELMINDIFF	14Ah	330	The minimum difference between low output level and high output level is not reached.
ERR_STARHUBENABLE	14Bh	331	The card holding the star-hub must be enabled when doing synchronization.
ERR_PATPWSMALLEEDGE	14Ch	332	Combination of pattern with pulsewidth smaller and edge is not allowed.
ERR_XMODESETUP	14Dh	333	The chosen setup for (SPCM_X0_MODE .. SPCM_X19_MODE) is not valid. See hardware manual for details.
ERR_AVRG_LSA	14Eh	334	Setup for Average LSA Mode not valid. Check Threshold and Replacement values for chosen AVRGMODE.
ERR_PCICHECKSUM	203h	515	The check sum of the card information has failed. This could be a critical hardware failure. Restart the system and check the connection of the card in the slot.
ERR_MEMALLOC	205h	517	Internal memory allocation failed. Please restart the system and be sure that there is enough free memory.
ERR_EEPROMLOAD	206h	518	Timeout occurred while loading information from the on-board EEPROM. This could be a critical hardware failure. Please restart the system and check the PCI connector.
ERR_CARDNOSUPPORT	207h	519	The card that has been found in the system seems to be a valid Spectrum card of a type that is supported by the driver but the driver did not find this special type internally. Please get the latest driver from www.spectrum-instrumentation.com and install this one.
ERR_CONFIGACCESS	208h	520	Internal error occurred during config writes or reads. Please contact Spectrum support for further assistance.
ERR_FIFOHWOVERRUN	301h	769	FIFO acquisition: Hardware buffer overrun in FIFO mode. The complete on-board memory has been filled with data and data wasn't transferred fast enough to PC memory. FIFO replay: Hardware buffer underrun in FIFO mode. The complete on-board memory has been replayed and data wasn't transferred fast enough from PC memory. If acquisition or replay throughput is lower than the theoretical bus throughput, check the application buffer setup.
ERR_FIFOFINISHED	302h	770	FIFO transfer has been finished, programmed data length has been transferred completely.
ERR_TIMESTAMP_SYNC	310h	784	Synchronization to timestamp reference clock failed. Please check the connection and the signal levels of the reference clock input.
ERR_STARHUB	320h	800	The auto routing function of the Star-Hub initialization has failed. Please check whether all cables are mounted correctly.
ERR_INTERNAL_ERROR	FFFFh	65535	Internal hardware error detected. Please check for driver and firmware update of the card.

Spectrum Knowledge Base

You will also find additional help and information in our knowledge base available on our website:

<https://spectrum-instrumentation.com/support/knowledgebase/index.php>

Temperature sensors

The M4i/M4x card series has integrated temperature sensors that allow to read out different internal temperatures. These functions are also available for the internal M4i cards inside the digitizerNETBOX, generatorNETBOX or hybridNETBOX series. In here the temperature can be read out for every internal card separately.

Temperature read-out registers

Up to three different temperature sensors can be read-out for each M4i and M4x card. Depending on the specific card type not all of these temperature sensors are used. The temperature can be read in different temperature scales at any time:

Table 116: Spectrum API: temperature read-out registers of internal temperature sensors

Register	Value	Direction	Description
SPC_MON_TK_BASE_CTRL	500022	read	Base card temperature in Kelvin
SPC_MON_TK_MODULE_0	500023	read	Module temperature 0 in Kelvin
SPC_MON_TK_MODULE_1	500024	read	Module temperature 1 in Kelvin
SPC_MON_TC_BASE_CTRL	500025	read	Base card temperature in degrees Celsius
SPC_MON_TC_MODULE_0	500026	read	Module temperature 0 in degrees Celsius
SPC_MON_TC_MODULE_1	500027	read	Module temperature 1 in degrees Celsius
SPC_MON_TF_BASE_CTRL	500028	read	Base card temperature in degrees Fahrenheit
SPC_MON_TF_MODULE_0	500029	read	Module temperature 0 in degrees Fahrenheit
SPC_MON_TF_MODULE_1	500030	read	Module temperature 1 in degrees Fahrenheit

Temperature hints

- Monitoring of the temperature figures is recommended for environments where the operating temperature can reach or even exceed the specified operating temperature. Please see technical data section for specified operating temperatures.
- The temperature sensors can be used to optimize the system cooling.

77xx temperatures and limits

The following description shows the meaning of each temperature figure on the 77xx series and also gives maximum ratings that should not be exceeded. All figures given in degrees Celsius:

Table 117: Spectrum API: temperature limits

Sensor Name	Sensor Location	Typical figure at 25°C environment temperature	Maximum temperature
BASE_CTRL	Inside FPGA	50°C ±5°C	80°C
MODULE_0	not used	n.a.	n.a.
MODULE_1	Front-End	60°C ±5°C	90°C

Details on M4i/M4x cards status LED

Every M4i card has a two-color status LED mounted within the multi-purpose I/O connector field on the card bracket.



The same two-color LED is located on the bracket of the M4x cards as well.



This chapter explains the different color codes and offers some possible solutions in case of an error condition.

Table 118: card status LED color and blink coding

Condition	LED color	Status	Solution
Off	Off	Card not powered	Power on the PC.
Error	Static: red	Power supply error	Restart the PC. In case that the error persists, please contact Spectrum support for further assistance.
	Fast blinking (aprox. 4 Hz): red - green - red - green ...	Power supply error	Restart the PC. In case that the error persists, please contact Spectrum support for further assistance.
	Blinking: red - off - red - off ...	Over temperature error	Power down the PC, let the card cool down and restart the system. Please make sure that you have a proper cooling fan installed to supply the M4i card in the PCIe slot with a constant air flow.
	Strobed blinking: long red - short off - short red - short off ...	FPGA boot error	This error code is available with Power Firmware V1.8 or newer. Power down the PC and restart the system. In case that this error is occurring after a firmware update please contact Spectrum support for assistance on how to boot the card's golden recovery image.
	Slow blinking (aprox. 1 Hz): red - green - red - green ...	PCI Express link training has not finished	1) Power down the PC, un-plug and re-plug the card to verify that there is a proper contact between the card and the slot. 2) Try another PCI Express slot, maybe the currently used one is not properly working. In case that the above steps did not help, please contact Spectrum support for assistance.
O.K.	Static: green	Card is ready for operation (at full PCIe speed)	A full speed PCIe link has been established (PCIe x8, Gen 2) and the card is ready for operation.
	Slow blinking (approx. 1 Hz): green - off - green - off ...	Indicator mode on (at full PCIe speed)	To ease the identification of a specific card in a multi-card system without un-installing the card it is possible to activate the card identification status by software. This mode changes the static „Ready for Operation“ green into a blinking state.
	Static: green & red (yellow)	Card is ready for operation (at reduced PCIe speed)	A reduced speed PCIe link has been established either with less than all of the possible 8 lanes and/or the card is installed in a PCIe Gen 1 slot. The card is ready for operation, but the data transfer throughput over the PCI Express bus is reduced. For getting the highest PCIe performance please consult your PC's or motherboard's manual for details on the PCI Express slots of your system.
	Slow blinking (aprox. 1 Hz): yellow - off - yellow - off ...	Indicator mode on (at reduced PCIe speed)	To ease the identification of a specific card in a multi-card system without un-installing the card it is possible to activate the card identification status by software. This mode changes the static „Ready for Operation“ yellow into a blinking state.

Turning on card identification LED

To enable/disable the cards LED indicator mode or to read out the current setting, please use the following register:

Table 119: Spectrum API: card identification LED register

Register	Value	Direction	Description
SPC_CARDIDENTIFICATION	201500	read/write	Writing a '1' turns on the LED card indicator mode, writing a '0' turns off the LED indicator mode.

The default for the card identification register is the OFF state.

Pin assignment of the digital VHDCI connector

The 68 pin VHDCI multipin connector is the main connector for all of Spectrum's fast digital boards. The pinning is designed for standard SCSI cables where pin 1 is twisted with pin 35, pin 2 with pin 36 and so on. Please note that there are also cables on the market that have the twisted pair between pin 1 with pin 2, pin 35 with pin 36 and so on. These cables are not recommended for usage as this will result in increased crosstalk and decreased signal quality.

Pin Assignment for 32 channel single-ended version

Table 120: VHDCI connector pin assignment

Connector A				Connector B			
A1	D0	A35	GND	B1	D16	B35	GND
A2	D1	A36	GND	B2	D17	B36	GND
A3	GND	A37	GND	B3	GND	B37	GND
A4	D2	A38	GND	B4	D18	B38	GND
A5	D3	A39	GND	B5	D19	B39	GND
A6	GND	A40	GND	B6	GND	B40	GND
A7	D4	A41	GND	B7	D20	B41	GND
A8	D5	A42	GND	B8	D21	B42	GND
A9	GND	A43	GND	B9	GND	B43	GND
A10	D6	A44	GND	B10	D22	B44	GND
A11	D7	A45	GND	B11	D23	B45	GND
A12	GND	A46	GND	B12	GND	B46	GND
A13	D8	A47	GND	B13	D24	B47	GND
A14	D9	A48	GND	B14	D25	B48	GND
A15	GND	A49	GND	B15	GND	B49	GND
A16	D10	A50	GND	B16	D26	B50	GND
A17	D11	A51	GND	B17	D27	B51	GND
A18	GND	A52	GND	B18	GND	B52	GND
A19	D12	A53	GND	B19	D28	B53	GND
A20	D13	A54	GND	B20	D29	B54	GND
A21	GND	A55	GND	B21	GND	B55	GND
A22	D14	A56	GND	B22	D30	B56	GND
A23	D15	A57	GND	B23	D31	B57	GND
A24	GND	A58	GND	B24	GND	B58	GND
A25	TrigIn	A59	GND	B25	X0In	B59	GND
A26	X2Out	A60	GND	B26	X0Out	B60	GND
A27	Strobe	A61	GND	B27	X1In	B61	GND
A28	Power ²	A62	Power ²	B28	RFU ¹	B62	RFU ¹
A29	Power ²	A63	Power ²	B29	RFU ¹	B63	RFU ¹
A30	Power ²	A64	Power ²	B30	RFU ¹	B64	RFU ¹
A31	GND	A65	GND	B31	GND	B65	GND
A32	ClkOut	A66	GND	B32	X1Out	B66	GND
A33	GND	A67	GND	B33	GND	B67	GND
A34	ClkIn-DC	A68	GND	B34	ClkIn-AC+	B68	ClkIn-AC-

² 3.3V, max 500 mA combined

¹: Reserved for future use: do not connect

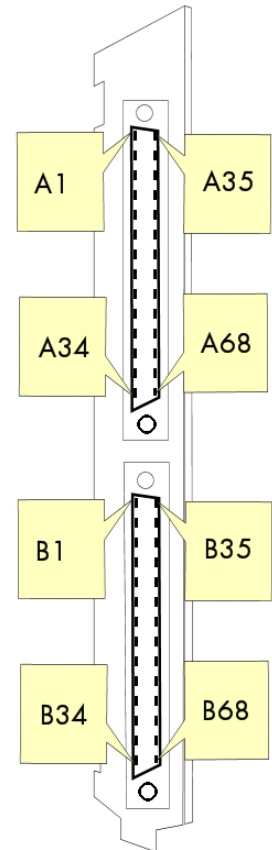


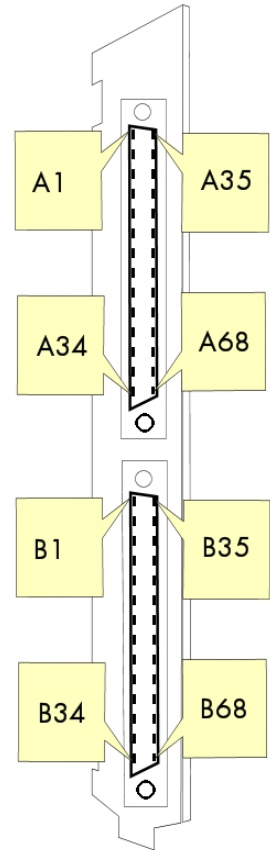
Image 66: VHDCI connector drawing and pin position

Pin Assignment for 32 channel differential (LVDS) version**Connector A**

A1	D0+	A35	D0-
A2	D1+	A36	D1-
A3	GND	A37	GND
A4	D2+	A38	D2-
A5	D3+	A39	D3-
A6	GND	A40	GND
A7	D4+	A41	D4-
A8	D5+	A42	D5-
A9	GND	A43	GND
A10	D6+	A44	D6-
A11	D7+	A45	D7-
A12	GND	A46	GND
A13	D8+	A47	D8-
A14	D9+	A48	D9-
A15	GND	A49	GND
A16	D10+	A50	D10-
A17	D11+	A51	D11-
A18	GND	A52	GND
A19	D12+	A53	D12-
A20	D13+	A54	D13-
A21	GND	A55	GND
A22	D14+	A56	D14-
A23	D15+	A57	D15-
A24	GND	A58	GND
A25	TrigIn+	A59	TrigIn-
A26	X2Out+	A60	X2Out-
A27	Strobe+	A61	Strobe-
A28	Power ²	A62	Power ²
A29	Power ²	A63	Power ²
A30	Power ²	A64	Power ²
A31	GND	A65	GND
A32	ClkOut+	A66	ClkOut-
A33	GND	A67	GND
A34	ClkIn-DC+	A68	ClkIn-DC-

² 3.3V, max 500 mA combined**Connector B**

B1	D16+	B35	D16-
B2	D17+	B36	D17-
B3	GND	B37	GND
B4	D18+	B38	D18-
B5	D19+	B39	D19-
B6	GND	B40	GND
B7	D20+	B41	D20-
B8	D21+	B42	D21-
B9	GND	B43	GND
B10	D22+	B44	D22-
B11	D23+	B45	D23-
B12	GND	B46	GND
B13	D24+	B47	D24-
B14	D25+	B48	D25-
B15	GND	B49	GND
B16	D26+	B50	D26-
B17	D27+	B51	D27-
B18	GND	B52	GND
B19	D28+	B53	D28-
B20	D29+	B54	D29-
B21	GND	B55	GND
B22	D30+	B56	D30-
B23	D31+	B57	D31-
B24	GND	B58	GND
B25	X0In+	B59	X0In-
B26	X0Out+	B60	X0Out-
B27	X1In+	B61	X1In-
B28	RFU ¹	B62	RFU ¹
B29	RFU ¹	B63	RFU ¹
B30	RFU ¹	B64	RFU ¹
B31	GND	B65	GND
B32	X1Out+	B66	X1Out-
B33	GND	B67	GND
B34	ClkIn-AC+	B68	ClkIn-AC-

¹: Reserved for future use: do not connect

Continuous memory for increased data transfer rate



The continuous memory buffer has been added to the driver version 1.36. The continuous buffer is not available in older driver versions. Please update to the latest driver if you wish to use this function.

Background

All modern operating systems use a very complex memory management strategy that strictly separates between physical memory, kernel memory and user memory. The memory management is based on memory pages (normally 4 kByte = 4096 Bytes). All software only sees virtual memory that is translated into physical memory addresses by a memory management unit based on the mentioned pages.

This will lead to the circumstance that although a user program allocated a larger memory block (as an example 1 MByte) and it sees the whole 1 MByte as a virtually continuous memory area this memory is physically located as spread 4 kByte pages all over the physical memory. No problem for the user program as the memory management unit will simply translate the virtual continuous addresses to the physically spread pages totally transparent for the user program.

When using this virtual memory for a DMA transfer things become more complicated. The DMA engine of any hardware can only access physical addresses. As a result the DMA engine has to access each 4 kByte page separately. This is done through the Scatter-Gather list. This list is simply a linked list of the physical page addresses which represent the user buffer. All translation and set-up of the Scatter-Gather list is done inside the driver without being seen by the user. Although the Scatter-Gather DMA transfer is an advanced and powerful technology it has one disadvantage: For each transferred memory page of data it is necessary to also load one Scatter-Gather entry (which is 16 bytes on 32 bit systems and 32 bytes on 64 bit systems). The little overhead to transfer (16/32 bytes in relation to 4096 bytes, being less than one percent) isn't critical but the fact that the continuous data transfer on the bus is broken up every 4096 bytes and some different addresses have to be accessed slows things down.

The solution is very simple: everything works faster if the user buffer is not only virtually continuous but also physically continuous. Unfortunately it is not possible to get a physically continuous buffer for a user program. Therefore the kernel driver has to do the job and the user program simply has to read out the address and the length of this continuous buffer. This is done with the function `spcm_dwGetContBuf` as already mentioned in the general driver description. The desired length of the continuous buffer has to be programmed to the kernel driver for load time and is done differently on the different operating systems. Please see the following chapters for more details.

Next we'll see some measuring results of the data transfer rate with/without continuous buffer. You will find more results on different motherboards and systems in the application note number 6 „Bus Transfer Speed Details“. Also with newer M5i/M4i/M4x/M2p cards the gain in speed is not as impressive, as it is for older cards, but can be useful in certain applications and settings. As this is also system dependent, your improvements may vary. This can not only depend on the system hardware but also on the used operating system, as in some cases Linux does seem to benefit more than Windows for newer cards.

Bus Transfer Speed Details (M2i/M3i cards in an example system)

Mode	PCI 33 MHz slot		PCI-X 66 MHz slot		PCI Express x1 slot	
	read	write	read	write	read	write
User buffer	109 MB/s	107 MB/s	195 MB/s	190 MB/s	130 MB/s	138 MB/s
Continuous kernel buffer	125 MB/s	122 MB/s	248 MB/s	238 MB/s	160 MB/s	170 MB/s
Speed advantage	15%	14%	27%	25%	24%	23%

Bus Transfer Standard Read/Write Transfer Speed Details (M4i.44xx card in an example system)

Mode	Notifysize 16 kByte		Notifysize 64 kByte		Notifysize 512 kByte		Notifysize 2048 kByte		Notifysize 4096 kByte	
	read	write	read	write	read	write	read	write	read	write
User buffer	243 MB/s	132 MB/s	793 MB/s	464 MB/s	2271 MB/s	1352 MB/s	2007 MB/s	1900 MB/s	2687 MB/s	2284 MB/s
Continuous kernel buffer	239 MB/s	133 MB/s	788 MB/s	457 MB/s	2270 MB/s	1470 MB/s	2555 MB/s	2121 MB/s	2989 MB/s	2549 MB/s
Speed advantage	-1.6%	+0.7%	-0.6%	-1.5%	0%	+8.7%	+27.3%	+11.6%	+11.2%	+11.6%

Bus Transfer FIFO Read Transfer Speed Details (M4i.44xx card in an example system)

Mode	Notifysize 4 kByte	Notifysize 8 kByte	Notifysize 16 kByte	Notifysize 32 kByte	Notifysize 64 kByte	Notifysize 256 kByte	Notifysize 1024 kByte	Notifysize 2048 kByte	Notifysize 4096 kByte
	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read
User buffer	455 MB/s	858 MB/s	1794 MB/s	2005 MB/s	3335 MB/s	3386 MB/s	3369 MB/s	3331 MB/s	3335 MB/s
Continuous kernel buffer	540 MB/s	833 MB/s	1767 MB/s	1965 MB/s	3216 MB/s	3386 MB/s	3389 MB/s	3388 MB/s	3389 MB/s
Speed advantage	+18.6%	-2.9%	-1.5%	-2.0%	-3.5%	0%	+0.6%	+1.7%	+1.6%

Bus Transfer FIFO Read Transfer Speed Details (M2p.5942 card in an example system)

Mode	Notifysize 4 kByte	Notifysize 8 kByte	Notifysize 16 kByte	Notifysize 32 kByte	Notifysize 64 kByte	Notifysize 256 kByte	Notifysize 1024 kByte	Notifysize 2048 kByte	Notifysize 4096 kByte
	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read
User buffer	282 MB/s	462 MB/s	597 MB/s	800 MB/s	800 MB/s	799 MB/s	799 MB/s	799 MB/s	797 MB/s
Continuous kernel buffer	279 MB/s	590 MB/s	577 MB/s	800 MB/s	800 MB/s	800 MB/s	800 MB/s	800 MB/s	799 MB/s
Speed advantage	-1.1%	+27.7%	-3.4%	+0.0%	+0.0%	0%	+0.1%	+0.1%	+0.3%

Setup on Linux systems

On Linux systems the continuous buffer setting is done via the command line argument `contmem_mb` when loading the kernel driver module:

```
insmod spcm.ko contmem_mb=4
```

As memory allocation is organized completely different compared to Windows the amount of data that is available for a continuous DMA buffer is unfortunately limited to a maximum of 8 MByte. On most systems it will even be only 4 MBytes.

To use a larger continuous buffer you can use the Continuous Memory Allocator (CMA). To allocate continuous memory this way you pass „cma=xyz“ as kernel boot parameter, with xyz being the size of the continuous memory, e.g. „cma=128M“ for 128 MByte.



Your kernel needs to have CMA support enabled to use this.
You can check this with „grep CONFIG_CMA /boot/config-\$(uname -r)“.

To enable CMA in our `spcm4` kernel driver module edit the Makefile for the kernel driver module and uncomment the line `#EXTRA_CFLAGS += -DSPCM4_USE_CMA` by removing the `#` in front. Then recompile the kernel module and load it as described above, like so as example:

```
insmod spcm4.ko contmem_mb=128
```



Using a continuous buffer of this size will need root privileges for the using program on most systems!

Setup on Windows systems

The continuous buffer settings is done with the Spectrum Control Center using a setup located on the „Support“ page. Please fill in the desired continuous buffer settings as MByte. After setting up the value the system needs to be restarted as the allocation of the buffer is done during system boot time.

If the system cannot allocate the amount of memory it will divide the desired memory by two and try again. This will continue until the system can allocate a continuous buffer. Please note that this try and error routine will need several seconds for each failed allocation try during boot up procedure. During these tries the system will look like being crashed. It is then recommended to change the buffer settings to a smaller value to avoid the long waiting time during boot up.

Continuous buffer settings should not exceed 1/4 of system memory. During tests the maximum amount that could be allocated was 384 MByte of continuous buffer on a system with 4 GByte memory installed.

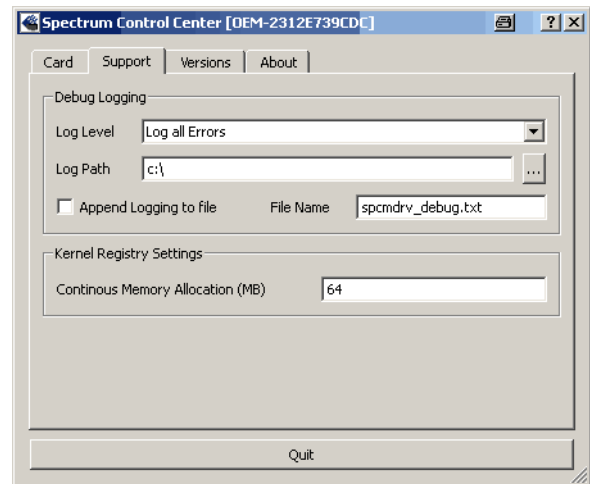


Image 67: setting up continuous memory buffer in Spectrum Control Center

Usage of the buffer

The usage of the continuous memory is very simple. It is just necessary to read the start address of the continuous memory from the driver and use this address instead of a self allocated user buffer for data transfer.

Function `spcm_dwGetContBuf`

This function reads out the internal continuous memory buffer (in bytes) if one has been allocated. If no buffer has been allocated the function returns a size of zero and a NULL pointer.

```
uint32 _stdcall spcm_dwGetContBuf_i64 ( // Return value is an error code
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType,                  // type of the buffer to read as listed above under SPCM_BUF_XXXX
    void** ppvDataBuffer,              // address of available data buffer
    uint64* pqwContBufLen);            // length of available continuous buffer

uint32 _stdcall spcm_dwGetContBuf_i64m ( // Return value is an error code
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType,                  // type of the buffer to read as listed above under SPCM_BUF_XXXX
    void** ppvDataBuffer,              // address of available data buffer
    uint32* pdwContBufLenH,            // high part of length of available continuous buffer
    uint32* pdwContBufLenL);           // low part of length of available continuous buffer
```

Please note that it is not possible to free the continuous memory for the user application.

Example

The following example shows a simple standard single mode data acquisition setup (for a card with 12/14/16 bit per resolution one sample equals 2 bytes) with the read out of data afterwards. To keep this example simple there is no error checking implemented.

```
int32 lMemsize = 16384; // recording length is set to 16 kSamples

spcm_dwSetParam_i64 (hDrv, SPC_CHENABLE, CHANNEL0); // only one channel activated
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_SINGLE); // set the standard single recording mode
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, lMemsize); // recording length in samples
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, 8192); // samples to acquire after trigger = 8k

// now we start the acquisition and wait for the interrupt that signalizes the end
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_CARD_WAITREADY);

// we now try to use a continuous buffer for data transfer or allocate our own buffer in case there's none
spcm_dwGetContBuf_i64 (hDrv, SPCM_BUF_DATA, &pvData, &qwContBufLen);
if (qwContBufLen < (2 * lMemsize))
    pvData = pvAllocMemPageAligned (lMemsize * 2); // assuming 2 bytes per sample

// read out the data
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 0, pvData, 0, 2 * lMemsize);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

// ... Use the data here for analysis/calculation/storage

// delete our own buffer in case we have created one
if (qwContBufLen < (2 * lMemsize))
    vFreeMemPageAligned (pvData, lMemsize * 2);
```


Abbreviations

Table 121: Abbreviations used throughout the Spectrum documents

Abbreviation	Long Name	Description
s	Second	
ms	Milli Second	1/1000 second; 1 ms is the time between two samples when running at 1 kS/s
us (µs)	Micro Second	1/1000000 second or 1/1000 milli second; 1 ms is the time between two samples when running at 1 MS/s
ns	Nano Second	1/1000000000 second or 1/1000 micro second; 1 ns is the time between two samples when running at 1 GS/s
ps	Pico Second	1/1000000000000 second or 1/1000 nano second
Sample		One sample represents one data word that has been acquired on the same time position. Each sample consist of either one byte (8 bit resolution) or two bytes (12, 14 and 16 bit resolution)
Byte		The smallest storage unit
kB	Kilo Bytes	1024 (2 ¹⁰) Bytes
MB	Mega Bytes	1024 x 1024 (2 ²⁰) Bytes
GB	Giga Bytes	1024 x 1024 x 1024 (2 ³⁰) Bytes
Hz	Hertz	1 Hertz is one event/sample per second
kHz	Kilo Hertz	1000 Hertz
MHz	Mega Hertz	1000000 Hertz or 1000 kHz
GHz	Giga Hertz	1000000000 Hertz or 1000 MHz
kS/s	kilo Samples per Second	1000 samples per second
MS/s	Mega Samples per Second	1000 kilo samples (1000000 samples) per second
GS/s	Giga Samples per Second	1000 Mega samples (1000000000 samples) per second
PCIe	PCI Express	The PCI Express bus is a point to point connection allowing full speed for every single slot. The Express bus is freely scaling and is available with 1 lane (x1), 4 lanes (x4), 8 lanes (x8) and 16 lanes (x16)
PXI	PCI eXtensions for Instrumentation	Based on the CompactPCI 3U standard the PXI (PCI eXtensions for Instrumentation) enhancement was defined especially for the measurement user. In this specification additional lines for measurement purposes are defined.
PXIe	PXI Express	PXI Express or PXIe is a subset of the PXI standard that replaces PXI's parallel data bus with a high speed serial interface.
PLL	Phase Lock Loop	A clock device which generates a new clock phase-locked to a given reference clock.
LED	Light-Emitting Diode	A semiconductor device that emits light and is often used as a status light or indicator.
API	Application Programming Interface	A type of software interface, offering a service to access/control specific hardware or other pieces of software.
CPU	Central Processing Unit	The central processor of a computer/PC system.
GPU	Graphics Processing Unit	An co-processor specifically tailored for fast and efficient and massively parallel calculations of certain data structures. Often, but not exclusively, located on a separate PCIe graphics card or co-processing card. s
CUDA	Compute Unified Device Architecture	A proprietary API for Nvidia GPUs to perform "general purpose" as in non-graphic related processing on GPUs rather than the CPU.
DMA	Direct Memory Access	A method to transfer data directly between a device (card) and PC memory.
RDMA	Remote Direct Memory Access	A method to transfer data directly between two devices (cards).
RMA	Return Manufacturer Authorization	
WEEE	Waste Electrical and Electronic Equipment)	

List of Figures

Image 1: M4i card showing mounted star-hub and sync bus connector.....	15
Image 2: M4i card showing mounted digital option	15
Image 3: M4i card backside showing type plate location and content	16
Image 4: Mounting M4i PCIe card into connector	23
Image 5: M4i card slot retention with perforation.....	24
Image 6: M4i card additional power connection usage	24
Image 7: M4i card with digital option mechanical installation and position of screws	25
Image 8: M4i cards with star-hub ex installed and connecting cables.....	25
Image 9: Installation of PXle cards with connector handling and mounting screws.....	26
Image 10: Windows Device Manager showing a new Spectrum card	27
Image 11: Spectrum Driver Installer Welcome Screen.....	27
Image 12: Spectrum Driver Installer - Progress	28
Image 13: Spectrum Driver Installer - finished	28
Image 14: Windows Device Manager showing properly installed Spectrum card	28
Image 15: Device Manager showing a new Spectrum card	32
Image 16: Spectrum Kernel Driver, API Library and Software structure.....	34
Image 17: Spectrum Control Center Installer.....	34
Image 18: Spectrum Control Center showing detail card information	35
Image 19: Spectrum Control Center - entering an IP address for a NETBOX	35
Image 20: Spectrum Control Center: wake on LAN for a cached card.....	35
Image 21: Netbox Monitor activation.....	36
Image 22: Spectrum Control Center: detailed hardware information on installed card	37
Image 23: Spectrum Control Center - showing firmware information of an installed card	37
Image 24: Spectrum Control Center - showing firmware information of an installed card	38
Image 25: Spectrum Control Center - showing driver information details.....	38
Image 26: Spectrum Control Center - adding a demo card to the system	39
Image 27: Spectrum Control Center - feature update, code entry	39
Image 28: Spectrum Control Center - software license install	39
Image 29: Spectrum Control Center - running an on-board calibration.....	39
Image 30: Spectrum Control Center - performing memory test	40
Image 31: Spectrum Control Center - running a transfer speed test of one card	40
Image 32: Spectrum Control Center - activate debug logging for support cases	40
Image 33: Spectrum Control Center - using device mapping	41
Image 34: Spectrum Control Center - doing a firmware update for one device	41
Image 35: SBench 6 overview of main functionality with demo data	42
Image 36: Structure of the Delphi exmaples	51
Image 37: LabVIEW driver oscilloscope example.....	58
Image 38: Spectrum MATLAB driver structure	58
Image 39: GPU usage with SCAPP SDK: data transfer options	59
Image 40: Acquisition cards: graphical overview of acquisition status and card command interaction.....	74
Image 41: Generation cards: graphical overview of generation status and card command interaction	74
Image 42: standard acquisition mode and pretrigger/posttrigger/trigger relation	77
Image 43: Overview of buffer handling for DMA transfers showing and the interaction with the DMA engine	80
Image 44: M4i/M4x clock section overview	84
Image 45: Trigger Engine Overview. Red marked parts not available on all card types	89
Image 46: trigger engine overview with trigger OR mask shown.....	90
Image 47: trigger engine OR mask details	90
Image 48: trigger engine overview with trigger AND mask shown	91
Image 49: trigger engine AND mask details.....	92
Image 50: trigger engine overview with marked trigger delay stage	94
Image 51: trigger engine overview with marked main external trigger Ext0/Trg0	95
Image 52: trigger engine overview with pattern trigger marked	96
Image 53: single pattern matching timing diagram	98
Image 54: pattern and edge matching timing diagram showing the edge detection	99
Image 55: trigger engine overview with multi-purpose trigger lines highlighted	100
Image 56: Drawing of Multiple Recording acquisition	102
Image 57: drawing of Multiple Recording Acquisition with Timestamps	104
Image 58: Drawing of Gated Sampling mode.....	106
Image 59: drawing of timestamp acquisition in standard mode in relation to card start and trigger detection	110
Image 60: drawing of timestamp acquisition in start-reset mode in relation to card start and trigger detection	110
Image 61: drawing of timestamp acquisition in refclock mode in relation to card start and trigger detection	111
Image 62: Overview of acquisition data, ABA data and timestamp data DMA transfer	112
Image 63: drawing of Multiple Recording Acquisition with Timestamps	117
Image 64: Overview of remote server option interaction in comparison to NETBOX devices.....	124
Image 65: Windows screenshot: finding a remote Spectrum device like digitizerNETBOX	125
Image 66: VHDCI connector drawing and pin position.....	131
Image 67: setting up continuous memory buffer in Spectrum Control Center	135

List of Tables

Table 1: Symbols and Safety Labels	8
Table 2: Packing List M4i card (PCI Express)	11
Table 3: Packing List M4x card (PXI Express)	11
Table 4: list of C/C++ header files in driver	42
Table 5: C/C++ type declarations for drivers and examples	44
Table 6: C/C++ type naming convention throughout drivers and examples	44
Table 7: Spectrum driver API functions overview and differentiation between 32 bit and 64 bit registers	47
Table 8: Spectrum API: Command register and basic commands	60
Table 9: Spectrum API: Card Type Register	63
Table 10: Spectrum API: list of card type codes for M4i.77xx series	63
Table 11: Spectrum API: hardware and PCB version register overview	63
Table 12: Spectrum API: extension module hardware and PCB version register	63
Table 13: Spectrum API: register for reading back the PXIe card slot number	63
Table 14: Spectrum API: Register overview of firmware versions	64
Table 15: Spectrum API: Register overview of reading current firmware	64
Table 16: Spectrum API: production date register	64
Table 17: Spectrum API: calibration date register	65
Table 18: Spectrum API: hardware serial number register	65
Table 19: Spectrum API: maximum sampling rate register	65
Table 20: Spectrum API: installed memory registers. 32 bit read is limited to a maximum of 1 GByte	65
Table 21: Spectrum API: Feature Register and available feature flags	65
Table 22: Spectrum API: Extended feature register and available extended feature flags	66
Table 23: Spectrum API: register overview of miscellaneous cards information	66
Table 24: Spectrum API: register card function type and possible types	67
Table 25: Spectrum API: register driver type information and possible driver types	67
Table 26: Spectrum API: driver version read register	67
Table 27: Spectrum API: kernel driver version read register	67
Table 28: Spectrum API: custom modification register and different bitmasks to split the register in various hardware parts	68
Table 29: Spectrum API: command register and reset command	68
Table 30: Spectrum API: channel enable register	69
Table 31: Spectrum API: digital input features read register and register details	69
Table 32: Spectrum API: threshold registers and register settings	70
Table 33: Spectrum API: input termination registers	70
Table 34: Spectrum API: card mode and read out of available card mode software registers	71
Table 35: Spectrum API: possible values for the card mode register. Description of the different card modes	72
Table 36: Spectrum API: card command register and different commands with descriptions	72
Table 37: Spectrum API: timeout definition register	73
Table 38: Spectrum API: card status register and possible status values with descriptions of the status	73
Table 39: Spectrum API: memory test register	76
Table 40: Spectrum API: Command register and commands for DMA transfers	76
Table 41: Spectrum API: status register and status codes for DMA data transfer	76
Table 42: Spectrum API: card mode register and standard single mode setup	77
Table 43: Spectrum API: memory size and posttrigger registers for standard single mode	77
Table 44: Spectrum API: card mode register and standard FIFO mode setup	78
Table 45: Spectrum API: setup registers for standard FIFO mode	78
Table 46: Limits of pre trigger, post trigger, memory size depending on selected mode and activated channels	79
Table 47: Spectrum API: registers for DMA buffer handling	80
Table 48: Spectrum API: content of DMA buffer handling registers for different use cases	80
Table 49: Spectrum API: clock mode register and available clock modes	84
Table 50: Spectrum API: clock mode register and internal clock mode	85
Table 51: Spectrum API: samplerate register	85
Table 52: Spectrum API: clock output and clock output frequency register	85
Table 53: Spectrum API: clock oversampling readout register	85
Table 54: Spectrum API: clock mode register and external reference clock setup	86
Table 55: Spectrum API: reference clock register and available settings	86
Table 56: Spectrum API: clock output and clock output frequency register	86
Table 57: Spectrum API: clock mode register and PXI reference clock usage	87
Table 58: Spectrum API: direct external clock mode register settings	87
Table 59: Spectrum API: programmable clock delay registers	87
Table 60: Spectrum API: clock edge selection registers	88
Table 61: Spectrum API: clock threshold register	88
Table 62: Spectrum API: clock programmable termination register	88
Table 63: Spectrum API: general trigger OR mask register and available settings	90
Table 64: Spectrum API: channel trigger OR mask registers and available settings	91
Table 65: Spectrum API: general trigger AND mask registers and available settings	92
Table 66: Spectrum API: channel trigger AND mask registers and available settings	93
Table 67: Spectrum API: software register and register setting for software trigger	93
Table 68: Spectrum API: command register and force trigger command	93
Table 69: Spectrum API: command register and trigger enable/disable command	94
Table 70: Spectrum API: trigger delay registers and available settings	94

Table 71: Spectrum API: external trigger Ext0 registers and register settings	95
Table 72: Spectrum API: external trigger Ext0 OR mask settings.....	95
Table 73: Spectrum API: channel trigger mode registers and available register settings	96
Table 74: Spectrum API: channel OR and AND mask registers and available register settings	96
Table 75: Spectrum API: trigger mode registers and positive edge settings.....	97
Table 76: Spectrum API: trigger mode registers and high level settings	97
Table 77: Spectrum API: trigger mode registers and negative edge settings.....	97
Table 78: Spectrum API: trigger mode registers and low level settings	98
Table 79: Spectrum API: trigger mode registers and both edges settings	98
Table 80: Spectrum API: example of trigger settings registers for pattern matching	99
Table 81: Spectrum API: example of trigger settings registers for pattern and edge matching	99
Table 82: Spectrum API: multi-purpose registers and available register settings	100
Table 83: Spectrum API: registers for setting up asynchronous I/O of multi-purpose lines	100
Table 84: Spectrum API: trigger output compatibility register	101
Table 85: Spectrum API: trigger threshold register and register settings.....	101
Table 86: Spectrum API: multi-purpose I/O programmable termination registers	101
Table 87: Spectrum API: software registers for Multiple Recording mode setup	102
Table 88: Spectrum API: card mode register and multiple recording settings	102
Table 89: Spectrum API: memory and loop registers with related multiple recording settings	102
Table 90: Spectrum API: card mode register and multiple replay FIFO mode settings.....	102
Table 91: Spectrum API: loops register settings when using Multiple Replay FIFO mode.....	103
Table 92: Limits of pre trigger, post trigger, memory size depending on selected mode and activated channels.....	103
Table 93: Spectrum API: registers and settings for Gated Sampling mode	106
Table 94: Spectrum API: card mode register and settings for Gated Sampling standard mode.....	106
Table 95: Spectrum API: memsize and loops register and register settings for Gated Replay mode	106
Table 96: Spectrum API: card mode register and Gated Sampling FIFO mode settings	106
Table 97: Spectrum API: Gated Sampling FIFO mode loops register settings	106
Table 98: Limits of pre trigger, post trigger, memory size depending on selected mode and activated channels.....	107
Table 99: Spectrum API: timestamp related register and available timestamp commands	109
Table 100: Spectrum API: timestamp commands for standard mode	110
Table 101: Spectrum API: timestamp commands for star-reset mode	111
Table 102: Spectrum API: timestamp commands for refclock mode	111
Table 103: Spectrum API: extra DMA commands (ABA and Timestamp)	112
Table 104: Spectrum API: extra DMA status (ABA and Timestamp).....	113
Table 105: Spectrum API: ABA and Timestamp DMA buffer handling registers	113
Table 106: Spectrum API: timestamp sample format for standard mode	115
Table 107: Spectrum API: timestamp sample format for extended mode.....	116
Table 108: Spectrum API: timestamp extra data word format	116
Table 109: Spectrum API: timestamp data format register	116
Table 110: star-hub clock overview diagram	118
Table 111: Spectrum API: star-hub related registers for reading detected connections.....	119
Table 112: Spectrum API: synchronization enable mask register	120
Table 113: Spectrum API: star-hub synchronization commands	121
Table 114: Spectrum API: clock mode register and settings for SH Direct mode	122
Table 115: Spectrum API: driver error codes and error description	127
Table 116: Spectrum API: temperature read-out registers of internal temperature sensors	129
Table 117: Spectrum API: Spectrum API: temperature limits	129
Table 118: card status LED color and blink coding	130
Table 119: Spectrum API: card identification LED register	130
Table 120: VHDCI connector pin assignment	131
Table 121: Abbreviations used throughout the Spectrum documents	137



Spectrum Instrumentation GmbH

Ahrensfelder Weg 13-17 | 22927 Grosshansdorf | Germany

Phone +49 (0)4102-69 56-0 | Fax +49 (0)4102-69 56-66

info@spec.de



Spectrum Instrumentation Corp

401 Hackensack Ave, 4th Floor | Hackensack, NJ 07601 | USA

Phone +1 (201) 562-1999 | Fax +1 (201) 342-7598

sales@spectrum-instrumentation.com



spectrum-instrumentation.com