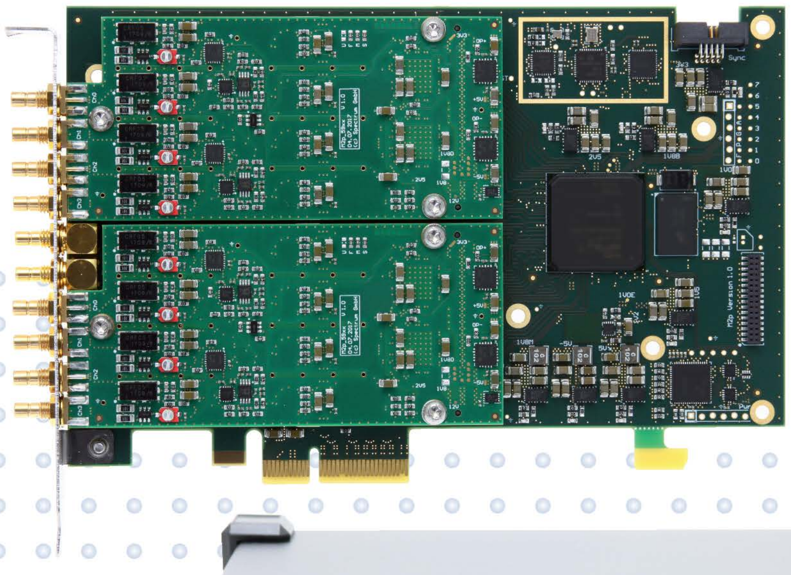
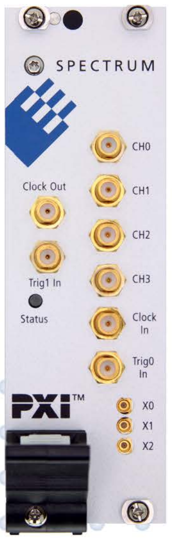
SPECTRUM
INSTRUMENTATION

Perfect fit – modular designed solutions





M5i.33xx-x16

**high-speed 12 bit Digitizer,
A/D converter board
for PCI Express bus and PXI 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

Table of Contents

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.
CE	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 user 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.

Table 2: Packing List

Item	Contained	Description
Card	Yes	Ordered card type inside ESD safety bag
Star-Hub M5i.xxxx-SH8-C2	Optional	Star-Hub mounted on card, containing 2 sync-cables
Star-Hub M5i.xxxx-SH8-C4		Star-Hub mounted on card, containing 4 sync-cables
Star-Hub M5i.xxxx-SH8-C8		Star-Hub mounted on card, containing 8 sync-cables
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

The M5i generation is the fastest streaming and high performance platform from Spectrum. The cards have been optimized for extremely fast data transfer and allow to read data for online analysis or offline storage with more than 12.8 GB/s using the PCI Express x16 Gen 3 bus interface.

The card comes with a standard memory of 2 GSamples which can be extended to 8 GSamples. A number of features like clock and trigger input and output and several multi-purpose I/O lines allow the integration into different application

General Information

The M5i.33xx is best suitable for applications that need ultra high sample rates, high resolution as well as a high input bandwidth. The fastest models are available with 3.0 GHz bandwidth, slower versions come with 2.0 GHz or 1.0 GHz bandwidth.

On the M5i.33xx every of the up to two channels has its own amplifier. Each input channel can be adapted to a wide variety of signal sources. This is done by software selecting a matching input range and input coupling. The user will easily find a matching solution from the offered different models. These versions are working with sample rates of 3.2 GS/s up to 10.0 GS/s and have one or two channels and can also be updated to a multi-channel system using the internal synchronization bus.

Data is written in the internal 2 GSample large memory which can be enlarged to a total of 8 GSamples. 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: Automatic test systems, Supersonics, CCD imaging systems, Vibration analysis, Radar, Sonar.

Different models of the M5i.33xx series

The following overview shows the different available models of the M5i.33xx series. They differ in the number of available channels. You can also see the model dependent location of the input connectors.

- **M5i.3330-x16**
- **M5i.3350-x16**
- **M5i.3360-x16**

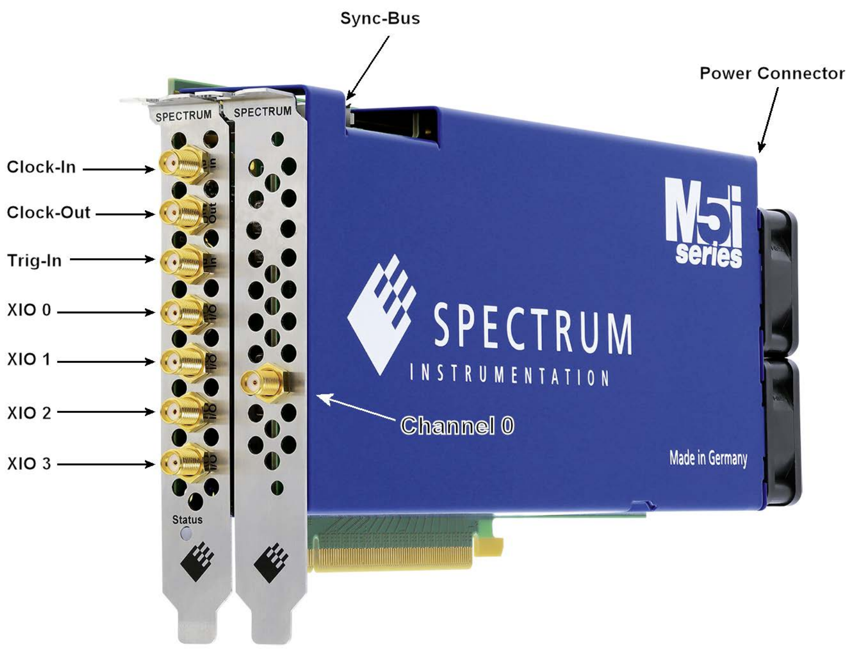


Image 1: Connector location for 1 channel cards

- **M5i.3321-x16**
- **M5i.3337-x16**
- **M5i.3357-x16**
- **M5i.3367-x16**

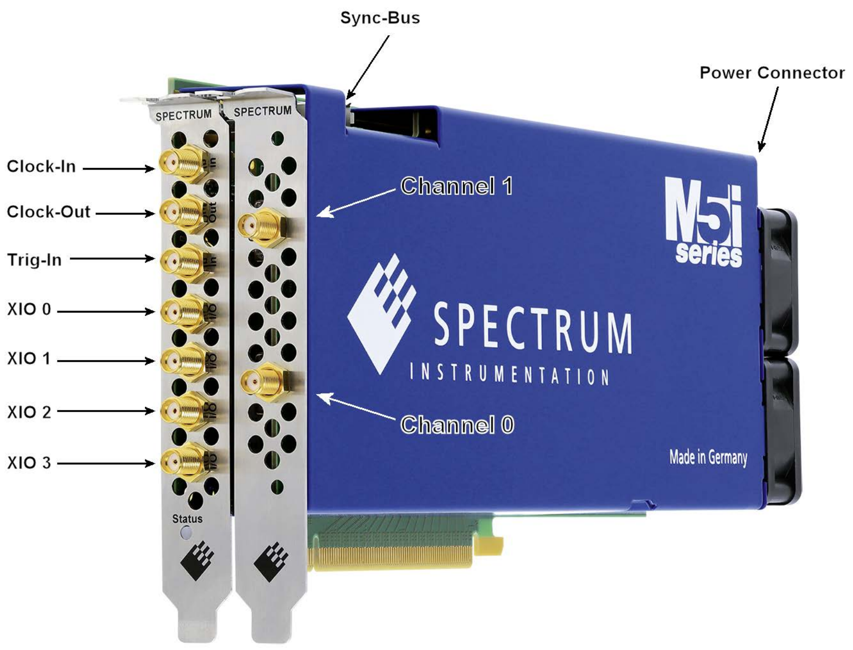


Image 2: Connector location for 2 channel cards

Additional options

Star-Hub

The Star-Hub module allows the synchronization of up to 8 M5i cards. It is possible to synchronize only cards of the same family with each other.

The Star-Hub is mounted on the backside of the M5i card and extends the M5i card to use a third PCIe bracket and hence requiring enough space for at least half-length PCI Express card. A free, full length, space is recommended for easier access to the cables and for enhanced air flow.

The module acts as a star distribution for clock and trigger signals. Every board is connected to the Star-Hub with a small cable of the same length; the carrier or master card, as well as all slave cards.
That minimizes the clock skew between the different cards. The picture shows the piggy-back module mounted on the backside of the carrier board schematically, without any plugged synchronization cables to achieve a better visibility.

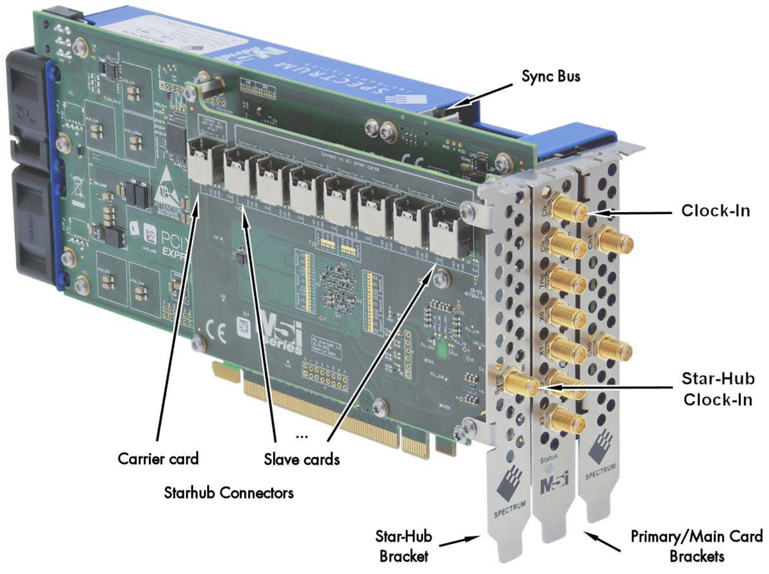


Image 3: M5i card showing mounted star-hub and the card's sync bus connector

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 slave boards is automatically recognized and checked by the driver when initializing the Star-Hub module. So no care must be taken on how to connect the slave cards. The master card must always be connected to its dedicated connector (the very left one as shown above). 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.

The Spectrum type plate

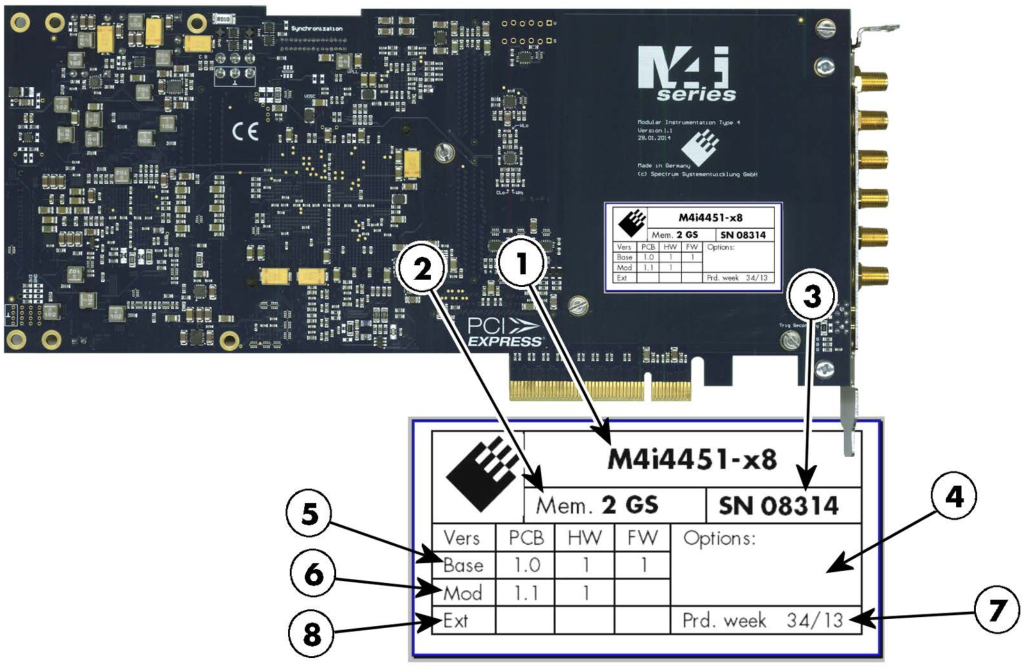


Image 4: M4i card backside showing type plate location and content

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 Diagrams

M5i.33xx Block Diagram

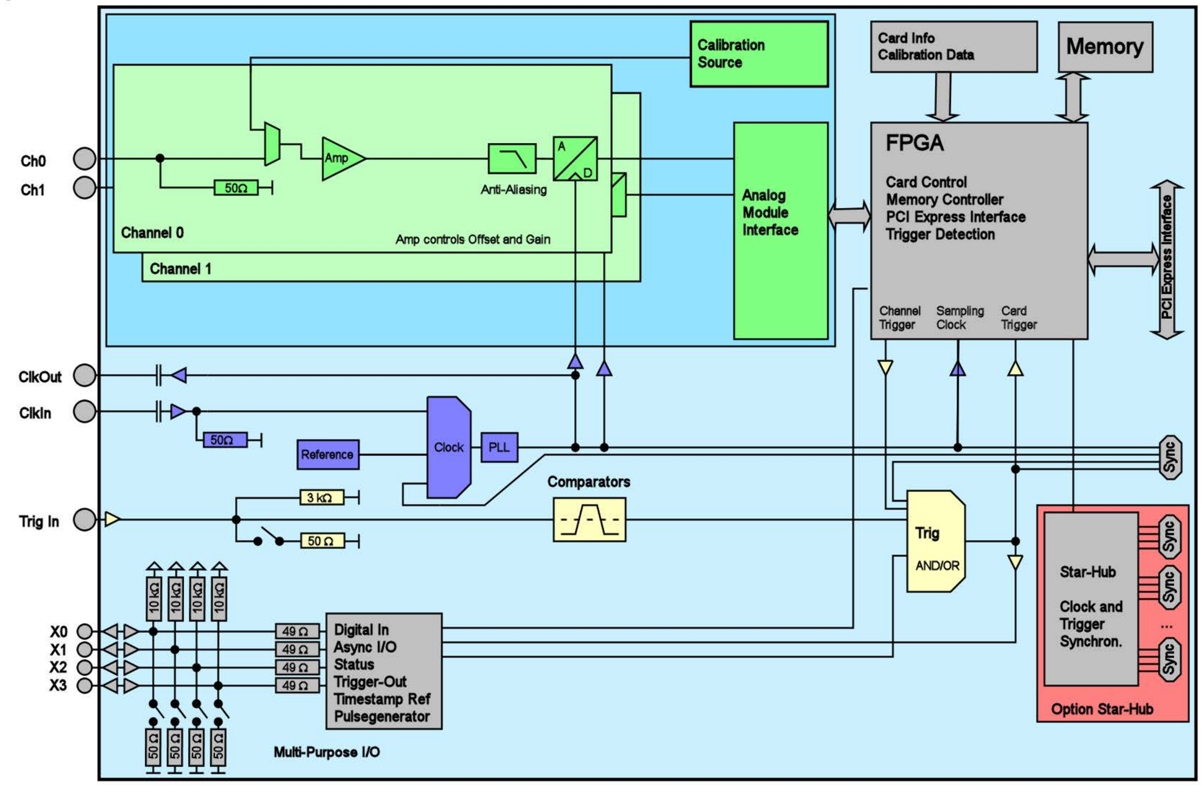


Image 5: M5i.33xx series block diagram

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.

Analog Inputs

Resolution		12 bit (can be switched by software to 8 bit to reduce data throughput)
Input Range	software programmable	±200 mV, ±500 mV, ±1 V, ±2.5 V
Input Type	fixed	Single-ended
Input Offset (single-ended)	software programmable	programmable to ±100% of input range in steps of 1%
ADC Differential non linearity (DNL)	ADC only	±0.3 LSB
ADC Integral non linearity (INL)	ADC only	±2.5 LSB
Offset error (full speed), DC signal	after warm-up and calibration	< 0.5% of range
Gain error (full speed), DC signal	after warm-up and calibration	< 0.5% of reading
Crosstalk: Signal 10 MHz, 50 Ω	any range, any channel	< -110 dB
Crosstalk: Signal 100 MHz, 50 Ω	any range, any channel	< -103 dB
Analog Input impedance	fixed	50 Ω
Analog input coupling	fixed	DC
Over voltage protection	input range ±200 mV	1.4 Vrms (16 dBm),max ±2.0 V peak input voltage
Over voltage protection	input range >= ±500 mV	5 Vrms (27 dBm),max ±7.5 V peak input voltage
Anti-Aliasing Filter (standard)		fixed at specified bandwidth (see table below)
Channel selection (single-ended inputs)	software programmable	1 or 2 channels (maximum is model dependent)
Calibration	Internal	Self-calibration is done on software command and corrects against the on-board references. Self-calibration should be issued after warm-up time.
Calibration	External	External calibration calibrates the on-board references used in self-calibration. All calibration constants are stored in non-volatile memory. A yearly external calibration is recommended.

	Input Range	M5i.3360-x16 M5i.3367-x16	M5i.3350-x16 M5i.3357-x16	M5i.3330-x16 M5i.3337-x16	M5i.3321-x16
lower bandwidth limit	all ranges	0 Hz (DC)	0 Hz (DC)	0 Hz (DC)	0 Hz (DC)
-3 dB bandwidth (minimum)	all ranges	4.7 GHz	3.0 GHz	2.0 GHz	1.0 GHz
-3 dB bandwidth (typical)	all ranges	4.8 GHz	3.1 GHz	2.2 GHz	1.1 GHz
Flatness within ±0.5 dB	all ranges	2.0 GHz	1.8 GHz	1.1 GHz	0.8 GHz

Trigger

Available trigger modes	software programmable	Channel Trigger, External, Software, Window, Re-Arm, Or/And, Delay
Channel trigger level resolution		12 bit
Trigger edge	software programmable	Rising edge, falling edge or both edges
Trigger delay	software programmable	0 up to (256 GS - 32) in steps of 32
Trigger holdoff (for Multi)	software programmable	0 up to (256 GS - 32) in steps of 32
Multi re-arming time	1 channel mode 2 channel mode	352 samples (+ programmed pretrigger) 176 samples (+ programmed pretrigger)
Pretrigger at Multi, FIFO	software programmable	32 up to (32 kSamples / channels) in steps of 32
Posttrigger at Standard Single	software programmable	32 up to (256 GS - 32) in steps of 32
Memory depth	software programmable	64 up to (Installed memory / channels) in steps of 32
Multiple Recording segment size	software programmable	64 up to (Installed memory / channels) in steps of 32
Internal/External trigger accuracy		1 sample

Timestamp modes	software programmable	Standard, Startreset, external reference clock (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/X2/X3 inputs at trigger time, trigger source (for OR trigger)	
Size per stamp		128 bit = 16 bytes	

External trigger		**Ext**	**X0, X1, X2, X3**
External trigger type		single level comparator	3.3V LVTTL logic inputs
External trigger impedance	software programmable	50 Ω or 3k Ω	For electrical specifications refer to „Multi Purpose I/O lines“ section.
External trigger input level		±5 V	
External trigger over voltage protection	50 Ω termination 3k Ω termination	±20 V 7 Vrms	
External trigger sensitivity (minimum required signal swing)		200 mVpp	
External trigger level	software programmable	±5 V with a stepsize of 10 mV	
External trigger bandwidth	50 Ω 3 kΩ 10 kΩ	DC to 2 GHz DC to 750 MHz n.a.	DC to 125 MHz n.a. DC to 125 MHz
Minimum external trigger pulse width		≥ 2 samples	≥ 2 samples
Resulting max detectable trigger frequency		[Current Samplerate]/2	[Current Samplerate]/2

Multi Purpose I/O lines (front-plate)

Number of multi purpose lines		four, named X0, X1, X2, X3
Input: available signal types	software programmable	Logic Trigger, Asynchronous Digital-In, Synchronous Digital-In, Timestamp Reference Clock
Input: impedance	software programmable	10 kΩ to 3.3 V or 50 Ω to GND
Input: maximum voltage level		-0.5 V to +4.0 V
Input: signal levels		3.3 V LVTTL (Low ≤ 0.8 V, High ≥ 2.0 V)
Input: bandwith		125 MHz
Output: available signal types	software programmable	Asynchronous Digital-Out, Trigger Output, Run, Arm, System Clock
Output: impedance		50 Ω
Output: signal levels		3.3 V LVTTL
Output: type		3.3V LVTTL, TTL compatible for high impedance loads
Output: drive strength		Capable of driving 50 Ω loads, maximum drive strength ±48 mA
Output: internal update rate	M5i.33xx	Current sampling clock ≤ 3.2 GS/s : 1/4 of sampling clock Current sampling clock > 3.2 GS/s and ≤ 6.4 GS/s : 1/8 of sampling clock
Output: min high/low time		4 ns
Output: max signal frequency		125 MHz

Option M5i.xxxx-PulseGen

Number of internal pulse generators		4
Number of pulse generator output lines		4 (Existing multi-purpose outputs X0 to X3)
Time resolution of pulse generator		Pulse generator's sampling rate is derived from instrument's sampling rate and value can be read out. Maximum possible pulse generator update rate is 33xx: 312.5 MS/s (3.2 ns)
Programmable output modes		Single-shot, multiple repetitions on trigger, gated
Programmable trigger sources		Software, Card Trigger, Other Pulse Generator, XIO lines.
Programmable trigger gate		None, ARM state, RUN state
Programmable length (frequency)		2 to 4G samples in steps of 1 (32 bit)
Programmable width (duty cycle)		1 to 4G samples in steps of 1 (32 bit)
Programmable delay		0 to 4G samples in steps of 1 (32 bit)
Programmable loops		0 to 4G samples in steps of 1 (32 bit) - 0 = infinite
Output level of digital pulse generators		Please see section of multi-purpose I/O lines

Clock

Clock Modes	software programmable	internal PLL, external reference clock, star-hub synchronization clock
Internal clock accuracy		≤ ±1 ppm
Clock setup range		base frequency or divided base frequency
Clock setup base frequencies	M5i.3321 M5i.333x M5i.335x/M5i.336x	3.2 GS/s, 2.5 GS/s, 2.0 GS/s 6.4 GS/s, 5.0 GS/s, 4.0 GS/s 10.0 GS/s, 8.0 GS/s, 5.0 GS/s
Clock setup divider		power of 2: 2, 4, 8, 16, 32, ... , 524288, 1048576
Clock setup examples	 M5i.332x M5i.333x M5i.335x/M5i.336x	Combination of any base frequency with any divider: 3.2, 2.5, 2.0, 1.6, 1.25, 1.0, 0.8 GS/s, ..., 1 kS/s 6.4, 5.0, 4.0, 3.2, 2.5, 2.0, 1.6. 1.25, 1.0, 0.8 GS/s, ..., 1 kS/s 10.0, 8.0, 6.4, 5.0, 4.0, 3.2, 2.5, 2.0, 1.6, 1.25, 1.0 GS/s, ..., 1 kS/s
External reference clock range	software programmable	≥ 2 MHz and ≤ 750 MHz in steps of 2 MHz
External reference clock input impedance		50 Ω fixed
External reference clock input coupling		AC coupling
External reference clock input edge		Rising edge
External reference clock input type		Single-ended, sine wave or square wave
External reference clock input swing	min max	200 mVpp 3 Vpp
External reference clock input max DC voltage		±10 V (with max 3.0 V difference between low and high level)
External reference clock input duty cycle requirement		45% to 55%
Clock setup granularity when using reference clock		divider: maximum sampling rate divided by: TBD
Internal reference clock output type		Single-ended, AC-coupled, LVPECL, 720 mVpp (typ)
Internal reference clock output frequency	M5i.3321 M5i.333x/335x/336x	clock setup base frequency / 64 (example: clock 3.2 GS/s -> output 50.000 MHz) clock setup base frequency / 128 (example: clock 4.0 GS/s -> output 31.25 MHz)
Star-Hub synchronization clock modes	software programmable	Internal clock, External reference clock
Channel to channel skew on one card		<12 ps
Skew between star-hub synchronized cards	software programmable	skew adjustable up to 200 ps (10 GS/s models) or 312 ps (6.4 GS/s and 3.2 GS/s models)

Block Average Signal Processing Option M5i.33xx

Averaging/Accumulation Modes	Software programmable	Standard or threshold defined averaging (TDA) for positive or negative pulses
Minimum Waveform Length		64 samples
Minimum Waveform Stepsize		32 samples
Maximum Waveform Length	1 channel active	1 MSamples
Maximum Waveform Length	2 channels active	512 kSamples
Minimum Number of Averages		2
Maximum Number of Averages		1024 (1k)
Data Output Format	fixed	32 bit signed integer
Re-Arming Time between waveforms	1 channel mode 2 channel mode	352 samples (+ programmed pretrigger) 176 samples (+ programmed pretrigger)
Re-Arming Time between end of average to start of next average		Two times the programmed segment length's (L) acquisition time: t = 2 * SegmentLen * ActiveChannels / Samplerate

Connectors

Analog Inputs (one for each single-ended input)	SMA female	Cable-Type: Cab-3mA-xx-xx
Trigger Input	SMA female	Cable-Type: Cab-3mA-xx-xx
Clock Input	SMA female	Cable-Type: Cab-3mA-xx-xx
Clock Output	SMA female	Cable-Type: Cab-3mA-xx-xx
Multi Purpose I/O	SMA female	Cable-Type: Cab-3mA-xx-xx
Power Connector	PCIe 6-pin power +12V+GND	Must be supplied by PC power supply

Connection Cycles

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

SMA connector	500 connection cycles
PCIe connector	50 connection cycles
PCIe power connector	30 connection cycles

Environmental and Physical Details

Dimension (Card, including rear fans)		L x H x W: 241 mm x 107 mm x 40 mm (double slot width)
Dimension (Card, rear fans, option star-hub)		L x H x W: 241 mm x 107 mm x 60 mm (three slots width)
Weight (M5i.33xx series)	maximum	780 g
Weight (Option Star-hub, including 8 cables)	maximum	150 g
Warm up time		30 minutes (running acquisition at full speed)
Operating temperature		0°C to 50°C
Storage temperature		-10°C to 70°C
Humidity		10% to 90%
Dimension of packing	1 card	470 mm x 250 mm x 130 cm
Volume weight of packing	1 card	4 kg

PCI Express specific details

PCIe connector type	x16 Generation 3 (Gen3)
PCIe slot compatibility (physical)	x16
PCIe slot compatibility (electrical)	x1, x2, x4, x8, x16 with PCIe Gen1, Gen2, Gen3, Gen4 or Gen5
Sustained streaming mode (Card-to-System):	> 12.8 GB/s (measured on PCIe x16 Gen3 with a chipset supporting a 512 bytes TLP) > 11.2 GB/s (measured on PCIe x16 Gen3 with a chipset supporting a 256 bytes TLP)
PCIe max card controller TLP	512 (lower values will limit maximum streaming speed)

Certification, Compliance, Warranty

Conformity Declaration	EN 17050-1:2010	General Requirements
EU Directives	2014/30/EU	EMC - Electromagnetic Compatibility
	2014/35/EU	LVD - Electrical equipment designed for use within certain voltage limits
	2011/65/EU	RoHS - Restriction of the use of certain hazardous substances in electrical and electronic equipment
	2006/1907/EC	REACH - Registration, Evaluation, Authorisation and Restriction of Chemicals
	2012/19/EU	WEEE - Waste from Electrical and Electronic Equipment
Compliance Standards	EN 61010-1: 2010	Safety regulations for electrical measuring, control, regulating and laboratory devices - Part 1: General requirement
	EN 61187:1994	Electrical and electronic measuring equipment - Documentation
	EN 61326-1:2021	Electrical equipment for measurement, control and laboratory use
	EN 61326-2-1:2021	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
	EN IEC 63000:2018	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

	Bus Connector		Power Connector*	
	3.3V	12 V	12 V	Total
M5i.3357-x16/M5i.3367-x16	0.3 A	n.a.	3.2 A	39 W
M5i.3350-x16/M5i.3360-x16	0.3 A	n.a.	3.0 A	37 W
M5i.3337-x16	0.3 A	n.a.	3.0 A	37 W
M5i.3330-x16	0.3 A	n.a.	2.8 A	35 W
M5i.3321-x16	0.3 A	n.a.	3.0 A	37 W

*A separate power connection to the card is mandatory. The card cannot be powered solely by the PCIe bus connector

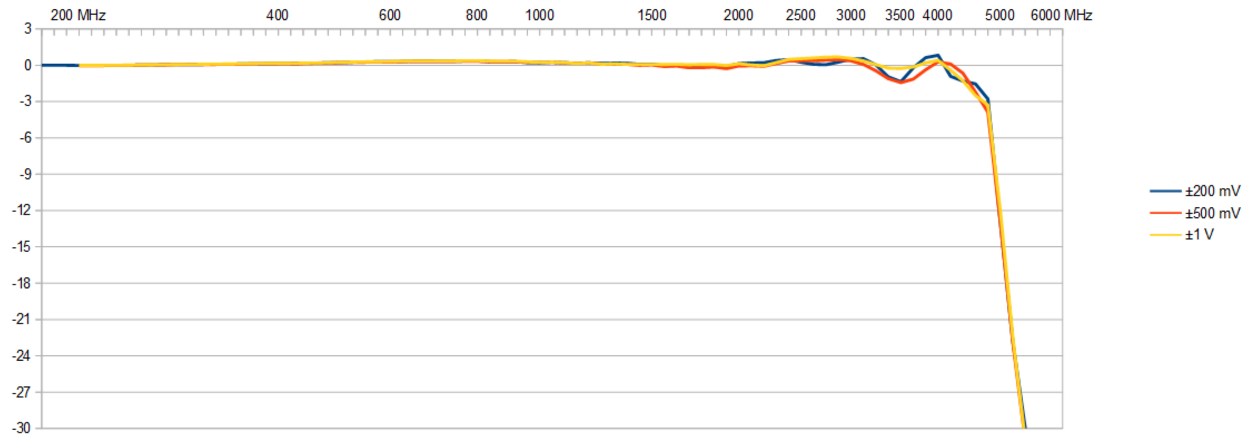
MTBF

MTBF	TBD hours

Frequency Response Plots

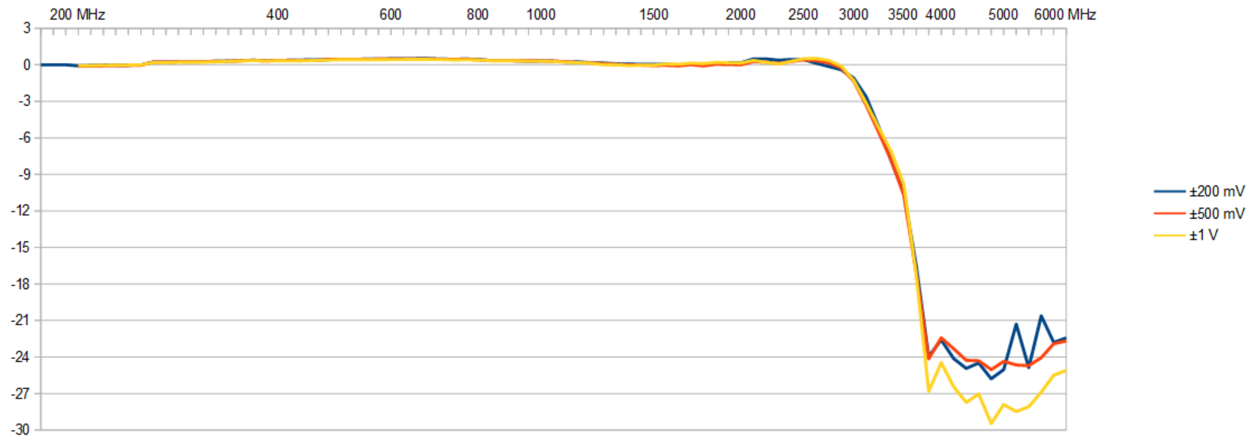
Frequency Response M5i.3360-x16, M5i.3367-x16

Sampling Rate: 10 GS/s, Bandwidth 4.7 GHz
50 Ω, DC coupling, no offset, no external filter



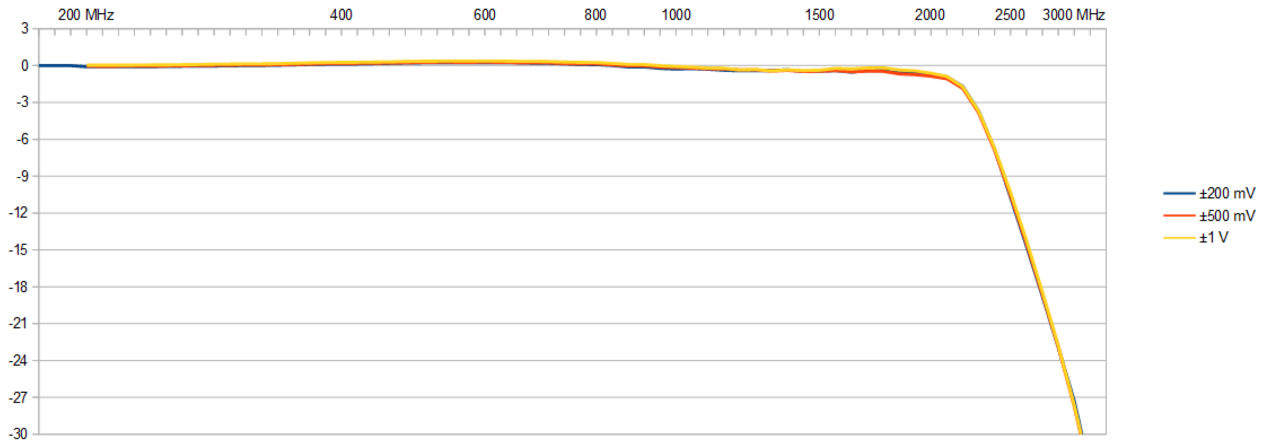
Frequency Response M5i.3350-x16, M5i.3357-x16

Sampling Rate: 10 GS/s, Bandwidth 3.0 GHz
50 Ω, DC coupling, no offset, no external filter



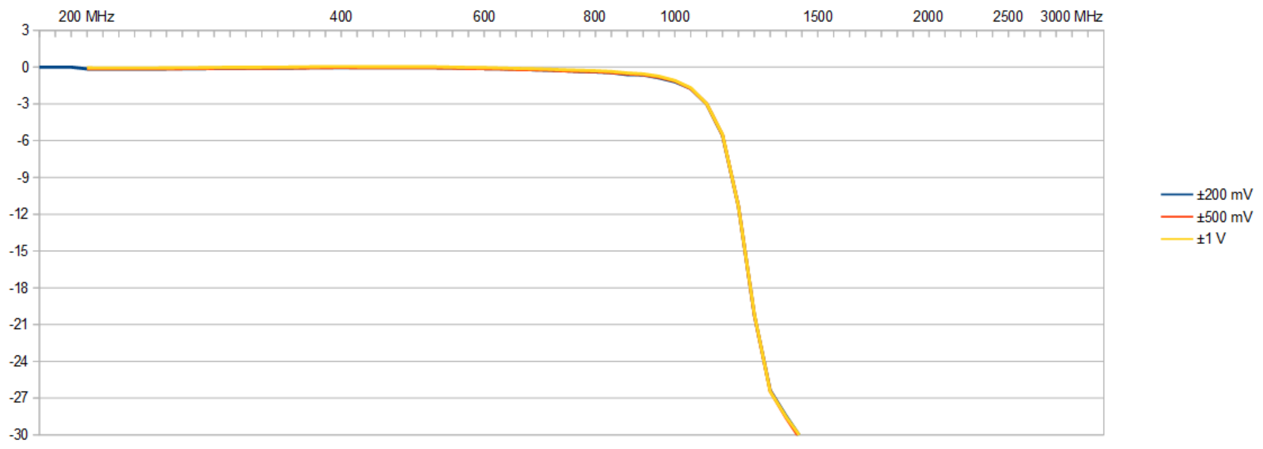
Frequency Response M5i.3330-x16, M5i.3337-x16

Sampling Rate: 6.4 GS/s, Bandwidth 2.0 GHz
50 Ω, DC coupling, no offset, no external filter



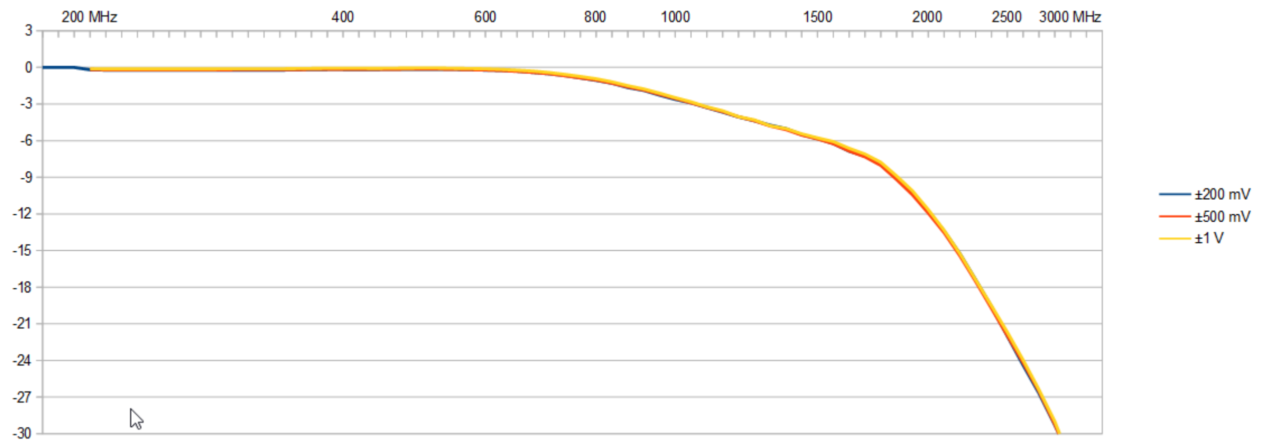
Frequency Response M5i.3321-x16 Standard

Sampling Rate: 3.2 GS/s, Bandwidth 1.0 GHz
50 Ω, DC coupling, no offset, no external filter
Standard Filter



Frequency Response M5i.3321-x16 with option -inptd

Sampling Rate: 3.2 GS/s, Bandwidth 1.0 GHz
50 Ω, DC coupling, no offset, no external filter
Option -inptd (input time domain optimization) Filter



Dynamic Parameters 10.0 GS/s 4.7 GHz models

	M5i.3360-x16 and M5i.3367-x16 - 12 Bit 10 GS/s (channel 0)											
Input Range	±200 mV						±500 mV					
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz
SNR (typ)	50.9 dB	50.3 dB	50.6 dB	50.0 dB	50.4 dB	50.9 dB	51.8 dB	51.5 dB	51.2 dB	50.9 dB	51.4 dB	50.8 dB
THD (typ)	-65.9 dB	-67.4 dB	-69.6 dB	-60.0 dB	-53.7 dB	-57.4 dB	-70.6 dB	-69.1 dB	-65.5 dB	-61.4 dB	-58.8 dB	-57.8 dB
SFDR (typ), incl. harm.	59.7 dB	57.6 dB	59.6 dB	58.1 dB	55.1 dB	57.9 dB	61.2 dB	59.3 dB	58.8 dB	58.2 dB	60.5 dB	58.4 dB
SFDR (typ), excl. harm.	59.7 dB	57.6 dB	59.6 dB	58.1 dB	60.7 dB	61.4 dB	61.2 dB	59.3 dB	58.8 dB	58.2 dB	63.9 dB	60.1 dB
SINAD/THD+N (typ)	50.8 dB	50.3 dB	50.6 dB	49.6 dB	48.7 dB	50.0 dB	51.7 dB	51.4 dB	51.1 dB	50.6 dB	50.7 dB	50.0 dB
ENOB (SINAD)	8.2 LSB	8.1 LSB	8.1 LSB	8.0 LSB	7.8 LSB	8.0 LSB	8.3 LSB	8.2 LSB	8.2 LSB	8.1 LSB	8.1 LSB	8.0 LSB
ENOB (SNR)	8.2 LSB	8.1 LSB	8.1 LSB	8.0 LSB	8.1 LSB	8.2 LSB	8.3 LSB	8.3 LSB	8.2 LSB	8.1 LSB	8.2 LSB	8.1 LSB

	M5i.3360-x16 and M5i.3367-x16 - 12 Bit 10 GS/s (channel 0)											
Input Range	±1 V						±2.5 V					
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz
SNR (typ)	51.3 dB	51.4 dB	51.3 dB	51.0 dB	51.3 dB	50.9 dB	51.3 dB	51.4 dB	51.3 dB	51.0 dB	51.3 dB	50.9 dB
THD (typ)	-70.0 dB	-67.3 dB	-68.9 dB	-61.1 dB	-58.3 dB	-57.4 dB	-70.0 dB	-67.3 dB	-68.9 dB	-61.1 dB	-58.3 dB	-57.4 dB
SFDR (typ), incl. harm.	59.4 dB	60.4 dB	58.9 dB	58.9 dB	59.1 dB	57.9 dB	59.4 dB	60.4 dB	58.9 dB	58.9 dB	59.1 dB	57.9 dB
SFDR (typ), excl. harm.	59.4 dB	60.4 dB	58.9 dB	58.9 dB	62.3 dB	61.4 dB	59.4 dB	60.4 dB	58.9 dB	58.9 dB	62.3 dB	61.4 dB
SINAD/THD+N (typ)	51.3 dB	51.3 dB	51.1 dB	50.6 dB	50.6 dB	50.1 dB	51.3 dB	51.3 dB	51.1 dB	50.6 dB	50.6 dB	50.1 dB
ENOB (SINAD)	8.2 LSB	8.2 LSB	8.2 dB	8.1 LSB	8.1 LSB	8.0 LSB	8.2 LSB	8.2 LSB	8.2 dB	8.1 LSB	8.1 LSB	8.0 LSB
ENOB (SNR)	8.2 LSB	8.2 LSB	8.2 dB	8.2 LSB	8.2 LSB	8.2 LSB	8.2 LSB	8.2 LSB	8.2 dB	8.2 LSB	8.2 LSB	8.2 LSB

Dynamic Parameters 10.0 GS/s 3.0 GHz models

	M5i.3350-x16 and M5i.3357-x16 - 12 Bit 10 GS/s (channel 0)											
Input Range	±200 mV						±500 mV					
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz
SNR (typ)	51.5 dB	52.0 dB	51.3 dB	51.0 dB	50.9 dB	50.8 dB	52.0 dB	52.0 dB	51.2 dB	52.3 dB	51.5 dB	51.3 dB
THD (typ)	-66.8 dB	-65.3 dB	-65.2 dB	-65.4 dB	-55.7 dB	-54.8 dB	-65.5 dB	-63.1 dB	-65.9 dB	-65.2 dB	-58.6 dB	-58.8 dB
SFDR (typ), incl. harm.	56.0 dB	55.7 dB	54.6 dB	54.8 dB	55.6 dB	55.0 dB	55.7 dB	64.7 dB	54.3 dB	58.9 dB	59.6 dB	52.5 dB
SFDR (typ), excl. harm.	56.0 dB	55.7 dB	54.6 dB	54.8 dB	55.6 dB	55.6 dB	55.7 dB	67.0 dB	54.3 dB	58.9 dB	59.6 dB	52.5 dB
SINAD/THD+N (typ)	51.3 dB	51.8 dB	51.2 dB	50.9 dB	49.6 dB	49.4 dB	51.8 dB	51.9 dB	51.1 dB	52.2 dB	50.7 dB	49.2 dB
ENOB (SINAD)	8.2 LSB	8.3 LSB	8.2 LSB	8.2 LSB	8.0 LSB	7.9 LSB	8.3 LSB	8.3 LSB	8.2 LSB	8.4 LSB	8.1 LSB	7.9 LSB
ENOB (SNR)	8.3 LSB	8.3 LSB	8.2 LSB	8.2 LSB	8.2 LSB	8.0 LSB	8.3 LSB	8.3 LSB	8.2 LSB	8.4 LSB	8.2 LSB	8.2 LSB

	M5i.3350-x16 and M5i.3357-x16 - 12 Bit 10 GS/s (channel 0)											
Input Range	±1 V						±2.5 V					
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz
SNR (typ)	51.7 dB	52.0 dB	51.7 dB	51.6 dB	51.2 dB	51.2 dB	52.0 dB	52.0 dB	51.2 dB	52.3 dB	51.5 dB	51.0 dB
THD (typ)	-66.4 dB	-66.5 dB	-66.5 dB	-64.7 dB	-58.5 dB	-60.5 dB	-65.5 dB	-63.1 dB	-65.9 dB	-65.2 dB	-58.6 dB	-64.4 dB
SFDR (typ), incl. harm.	55.8 dB	63.6 dB	55.9 dB	54.9 dB	59.5 dB	57.7 dB	55.7 dB	64.7 dB	54.3 dB	58.9 dB	59.6 dB	60.4 dB
SFDR (typ), excl. harm.	55.8 dB	63.7 dB	55.9 dB	54.9 dB	60.0 dB	57.7 dB	55.7 dB	67.0 dB	54.3 dB	58.9 dB	59.6 dB	60.4 dB
SINAD/THD+N (typ)	51.6 dB	51.9 dB	51.7 dB	51.5 dB	50.5 dB	51.2 dB	51.8 dB	51.9 dB	51.1 dB	52.2 dB	50.7 dB	50.9 dB
ENOB (SINAD)	8.3 LSB	8.3 LSB	8.3 dB	8.3 LSB	8.1 LSB	8.2 LSB	8.3 LSB	8.3 LSB	8.2 LSB	8.4 LSB	8.1 LSB	8.1 LSB
ENOB (SNR)	8.3 LSB	8.3 LSB	8.3 dB	8.3 LSB	8.2 LSB	8.2 LSB	8.3 LSB	8.3 LSB	8.2 LSB	8.4 LSB	8.2 LSB	8.2 LSB

Dynamic Parameters 6.4 GS/s models

	M5i.3330-x16 and M5i.3337-x16 - 12 Bit 6.4 GS/s (channel 0)											
Input Range	±200 mV						±500 mV					
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz
SNR (typ)	53.1 dB	53.1 dB	53.0 dB	52.6 dB	51.9 dB	50.1 dB	53.8 dB	53.2 dB	53.4 dB	53.0 dB	52.4 dB	50.3 dB
THD (typ)	-63.8 dB	-63.8 dB	-62.0 dB	-62.3 dB	-56.9 dB	-56.7 dB	-61.6 dB	-62.1 dB	-61.6 dB	-61.6 dB	-59.8 dB	-59.8 dB
SFDR (typ), incl. harm.	62.0 dB	61.6 dB	62.4 dB	62.5 dB	59.7 dB	57.2 dB	62.5 dB	64.2 dB	60.7 dB	62.2 dB	58.1 dB	60.0 dB
SFDR (typ), excl. harm.	62.0 dB	61.6 dB	62.6 dB	62.6 dB	64.5 dB	58.7 dB	65.0 dB	66.3 dB	60.6 dB	65.1 dB	58.1 dB	60.1 dB
SINAD/THD+N (typ)	52.8 dB	52.6 dB	52.3 dB	52.5 dB	51.6 dB	49.6 dB	53.4 dB	53.6 dB	52.8 dB	53.0 dB	51.9 dB	50.0 dB
ENOB (SINAD)	8.5 LSB	8.5 LSB	8.4 LSB	8.4 LSB	8.3 LSB	8.0 LSB	8.6 LSB	8.6 LSB	8.5 LSB	8.6 LSB	8.3 LSB	8.0 LSB
ENOB (SNR)	8.5 LSB	8.6 LSB	8.6 LSB	8.5 LSB	8.3 LSB	8.0 LSB	8.7 LSB	8.6 LSB	8.6 LSB	8.6 LSB	8.5 LSB	8.0 LSB

	M5i.3330-x16 and M5i.3337-x16 - 12 Bit 6.4 GS/s (channel 0)											
Input Range	±1 V						±2.5 V					
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz
SNR (typ)	53.4 dB	53.6 dB	53.3 dB	53.4 dB	52.5 dB	50.3 dB	53.5 dB	52.9 dB	53.5 dB	53.4 dB	51.9 dB	52.3 dB
THD (typ)	-63.8 dB	-63.5 dB	-63.5 dB	-62.6 dB	-59.9 dB	-59,7 dB	-64.0 dB	-61.0 dB	-61.2 dB	-60.9 dB	-58.9 dB	-59.5 dB
SFDR (typ), incl. harm.	62.0 dB	63.3 dB	65.1 dB	58.1 dB	60.4 dB	53.0 dB	62.2 dB	60.9 dB	63.6 dB	62.2 dB	58.7 dB	58.8 dB
SFDR (typ), excl. harm.	62.0 dB	63.4 dB	66.3 dB	58.1 dB	60.8 dB	53.0 dB	62.2 dB	53.9 dB	63.5 dB	63.0 dB	59.4 dB	58.9 dB
SINAD/THD+N (typ)	53.0 dB	53.2 dB	53.1 dB	52.6 dB	51.8 dB	49.6 dB	53.1 dB	52.9 dB	53.1 dB	52.9 dB	51.6 dB	51.5 dB
ENOB (SINAD)	8.5 LSB	8.5 LSB	8.6 LSB	8.4 LSB	8.3 LSB	8.0 LSB	8.5 LSB	8.5 LSB	8.6 LSB	8.5 LSB	8.3 LSB	8.3 LSB
ENOB (SNR)	8.6 LSB	8.6 LSB	8.6 LSB	8.6 LSB	8.5 LSB	8.1 LSB	8.6 LSB	8.6 LSB	8.6 LSB	8.6 LSB	8.3 LSB	8.4 LSB

Dynamic Parameters 3.2 GS/s models

	M5i.3321-x16 - 12 Bit 3.2 GS/s									
Input Range	±200 mV					±500 mV				
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz
SNR (typ)	54.1 dB	54.4 dB	54.7 dB	54.5 dB	54.5 dB	54.8 dB	55.0 dB	54.8 dB	54.6 dB	54.9 dB
THD (typ)	-64.3 dB	-63.4 dB	-62.3 dB	-61.1 dB	-59.5 dsB	-61.5 dB	-62.0 dB	-66.5 dB	-61.7 dB	-57.5 dB
SFDR (typ), incl. harm.	64.7 dB	65.4 dB	63.5 dB	61.9 dB	61.8 dB	72.9 dB	64.9 dB	65.6 dB	62.1 dB	60.3 dB
SFDR (typ), excl. harm.	65.1 dB	73.8 dB	71.6 dB	72.5 dB	69.7 dB	65.6 dB	72.8 dB	65.8 dB	69.1 dN	67.7 dB
SINAD/THD+N (typ)	53.7 dB	53.9 dB	54.0 dB	53.6 dB	53.3 dB	54.0 dB	54.2 dB	54.6 dB	53.9 dB	52.9 dB
ENOB (SINAD)	8.6 LSB	8.7 LSB	8.7 LSB	8.5 LSB	8.6 LSB	8.7 LSB	8.7 LSB	8.8 LSB	8.7 LSB	8.5 LSB
ENOB (SNR)	8.7 LSB	8.7 LSB	8.8 LSB	8.8 LSB	8.8 LSB	8.8 LSB	8.8 LSB	8.8 LSB	8.8 LSB	8.8 LSB

	M5i.3321-x16 - 12 Bit 3.2 GS/s									
Input Range	±1 V					±2.5 V				
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz
SNR (typ)	55.3 dB	55.3 dB	54.8 dB	54.8 dB	54.9 dB	54.8 dB	55.3 dB	54.8 dB	54.8 dB	54.9 dB
THD (typ)	-63.8 dB	-63.8 dB	-59.5 dB	-62.5 dB	-57.8 dB	-63.4 dB	-63.8 dB	-59.5 dB	-62.5 dB	-57.8 dB
SFDR (typ), incl. harm.	64.5 dB	66.3 dB	60.7 dB	63.5 dB	60.4 dB	62.5 dB	66.3 dB	60.7 dB	63.5 dB	60.4 dB
SFDR (typ), excl. harm.	65.3 dB	73.2 dB	67.4 dB	71.0 dB	68.9 dB	62.7 dB	73.2 dB	67.4 dB	71.0 dB	68.9 dB
SINAD/THD+N (typ)	54.2 dB	54.8 dB	53.6 dB	54.1 dB	53.1 dB	54.2 dB	54.8 dB	53.6 dB	54.1 dB	53.1 dB
ENOB (SINAD)	8.7 LSB	8.8 LSB	8.6 LSB	8.7 LSB	8.5 LSB	8.7 LSB	8.8 LSB	8.6 LSB	8.7 LSB	8.5 LSB
ENOB (SNR)	8.9 LSB	8.9 LSB	8.8 LSB	8.8 LSB	8.8 LSB	8.8 LSB	8.9 LSB	8.8 LSB	8.8 LSB	8.8 LSB

Dynamic Parameters 10.0 GS/s 4.7 GHz models (8-bit Mode)

The below dynamic parameters are measured using the 8-bit mode which reduces the resolution in hardware from 12 bit to 8 bit to save memory and data transfer bandwidth. Due to the hardware resolution being below the ENOB of all models, the dynamic parameters are similar for all models when switched to the 8-bit mode.

	M5i.3360-x16 and M5i.3367-x16 - 8 Bit 10 GS/s (channel 0)											
Input Range	±200 mV						±500 mV					
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz
SNR (typ)	46.8 dB	46.7 dB	46.6 dB	46.5 dB	46.5 dB	46.3 dB	46.8 dB	47.0 dB	46.8 dB	46.8 dB	46.9 dB	46.7 dB
THD (typ)	-65.2 dB	-66.1 dB	-67.0 dB	-60.6 dB	-54.3 dB	-54.0 dB	-65.4 dB	-66.2 dB	-65.3 dB	-60.6 dB	-58.9 dB	-56.7 dB
SFDR (typ), incl. harm.	55.3 dB	54.0 dB	55.2 dB	56.8 dB	55.6 dB	55.0 dB	56.1 dB	57.0 dB	53.5 dB	54.3 dB	56.2 dB	56.7 dB
SFDR (typ), excl. harm.	55.3 dB	54.0 dB	55.2 dB	56.8 dB	55.6 dB	56.4 dB	56.1 dB	57.0 dB	53.5 dB	55.3 dB	56.2 dB	56.7 dB
SINAD/THD+N (typ)	46.8 dB	46.7 dB	46.6 dB	46.3 dB	46.0 dB	45.7 dB	46.9 dB	47.0 dB	46.7 dB	46.7 dB	46.7 dB	46.5 dB
ENOB (SINAD)	7.5 LSB	7.5 LSB	7.5 LSB	7.4 LSB	7.4 LSB	7.3 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.4 LSB
ENOB (SNR)	7.5 LSB	7.5 LSB	7.5 LSB	7.4 LSB	7.4 LSB	7.4 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB

	M5i.3360-x16 and M5i.3367-x16 - 8 Bit 10 GS/s (channel 0)											
Input Range	±1 V						±2.5 V					
Test signal frequency	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz	10 MHz	40 MHz	70 MHz	240 MHz	600 MHz	1.2 GHz
SNR (typ)	47.0 dB	46.9 dB	46.9 dB	46.8 dB	46.9 dB	46.7 dB	46.8 dB	47.0 dB	46.8 dB	46.8 dB	46.9 dB	46.7 dB
THD (typ)	-65.6 dB	-65.8 dB	-66.5 dB	-61.9 dB	-58.9 dB	-57.9 dB	-65.4 dB	-66.2 dB	-65.3 dB	-60.6 dB	-58.9 dB	-56.7 dB
SFDR (typ), incl. harm.	56.1 dB	55.2 dB	55.2 dB	55.9 dB	54.7 dB	56.2 dB	56.1 dB	57.0 dB	53.5 dB	54.3 dB	56.2 dB	56.7 dB
SFDR (typ), excl. harm.	56.1 dB	55.2 dB	55.2 dB	55.9 dB	54.7 dB	56.2 dB	56.1 dB	57.0 dB	53.5 dB	55.3 dB	56.2 dB	56.7 dB
SINAD/THD+N (typ)	47.0 dB	46.9 dB	46.9 dB	46.7 dB	46.7 dB	46.5 dB	46.9 dB	47.0 dB	46.7 dB	46.7 dB	46.7 dB	46.5 dB
ENOB (SINAD)	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.4 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.4 LSB
ENOB (SNR)	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB	7.5 LSB

Dynamic parameters are measured at ±1 V input range (if no other range is stated) and 50Ω termination with the samplerate specified in the table. Measured parameters are averaged 20 times to get typical values. Test signal is a pure sine wave generated by a signal generator and a matching bandpass filter. Amplitude is >99% of FSR. SNR and RMS noise parameters may differ depending on the quality of the used PC. SNR = Signal to Noise Ratio, THD = Total Harmonic Distortion, SFDR = Spurious Free Dynamic Range, SINAD = Signal Noise and Distortion, ENOB = Effective Number of Bits.

RMS Noise Level (Zero Noise)

Standard Mode (12 Bit Resolution)

	M5i.3360-x16 and M5i.3367-x16 - 12 Bit 10 GS/s (Channel 0)							
Input Range	±200 mV		±500 mV		±1		±2.5 V	
Voltage resolution (1 LSB)	97 uV		244 uV		488 uV		1.22 mV	
DC, fixed 50 Ω, typical	3.9 LSB	381 uV	3.8 LSB	928 uV	4.3 LSB	2,1 mV	4.3 LSB	5.3 mV

	M5i.3360-x16 and M5i.3367-x16 - 12 Bit 5 GS/s							
Input Range	±200 mV		±500 mV		±1		±2.5 V	
Voltage resolution (1 LSB)	97 uV		244 uV		488 uV		1.22 mV	
DC, fixed 50 Ω, typical	4.1 LSB	398 uV	3.4 LSB	830 uV	3.6 LSB	1.8 mV	3.4 LSB	4.1 mV

	M5i.3350-x16 and M5i.3357-x16 - 12 Bit 10 GS/s (Channel 0)							
Input Range	±200 mV		±500 mV		±1		±2.5 V	
Voltage resolution (1 LSB)	97 uV		244 uV		488 uV		1.22 mV	
DC, fixed 50 Ω, typical	3.9 LSB	381 uV	3.8 LSB	928 uV	4.3 LSB	2,1 mV	4.3 LSB	5.3 mV

	M5i.3350-x16 and M5i.3357-x16 - 12 Bit 5 GS/s							
Input Range	±200 mV		±500 mV		±1		±2.5 V	
Voltage resolution (1 LSB)	97 uV		244 uV		488 uV		1.22 mV	
DC, fixed 50 Ω, typical	4.0 LSB	391 uV	3.3 LSB	806 uV	3.6 LSB	1.8 mV	2.9 LSB	3.5 mV

	M5i.3330-x16 and M5i.3337-x16 - 12 Bit 6.4 GS/s (Channel 0)							
Input Range	±200 mV		±500 mV		±1		±2.5 V	
Voltage resolution (1 LSB)	97 uV		244 uV		488 uV		1.22 mV	
DC, fixed 50 Ω, typical	3.7 LSB	361 uV	3.0 LSB	732 uV	3.8 LSB	1.9 mV	3.5 LSB	4.3 mV

	M5i.3330-x16 and M5i.3337-x16 - 12 Bit 3.2 GS/s							
Input Range	±200 mV		±500 mV		±1		±2.5 V	
Voltage resolution (1 LSB)	97 uV		244 uV		488 uV		1.22 mV	
DC, fixed 50 Ω, typical	3.0 LSB	293 uV	2.8 LSB	684 uV	3.0 LSB	1.5 mV	2.7 LSB	3.3 mV

	M5i.3321-x16 - 12 Bit 3.2 GS/s							
Input Range	±200 mV		±500 mV		±1		±2.5 V	
Voltage resolution (1 LSB)	97 uV		244 uV		488 uV		1.22 mV	
DC, fixed 50 Ω, typical	2.8 LSB	273 uV	2.3 LSB	562 uV	2.3 LSB	1.1 mV	2.9 LSB	3.5 mV

8-Bit acquisition mode (resolution reduced to 8 bit in hardware)

	M5i.3360-x16 and M5i.3367-x16 - 8 Bit 10 GS/s (Channel 0)							
Input Range	±200 mV		±500 mV		±1		±2.5 V	
Voltage resolution (1 LSB)	1.56 mV		3.9 mV		7.8 mV		19.5 mV	
DC, fixed 50 Ω, typical	0.5 LSB	780 uV	0.5 LSB	1.95 mV	0.5 LSB	3.9 mV	0.5 LSB	9.8 mV

M5i.33xx Order Information

The card is delivered with 2 GSample on-board memory and supports standard acquisition (Scope), FIFO acquisition (streaming), Multiple Recording and Timestamps. Operating system drivers for Windows/Linux 32 bit and 64 bit, examples for C/C++, LabVIEW (Windows), MATLAB (Windows and Linux), IVI, .NET, Delphi, Java, Python, Julia and a Base license of the oscilloscope software SBench 6 are included.

Adapter cables are not included. Please order separately!

PCI Express x16

Order no.	Bandwidth	Standard mem	1 channel	2 channels
M5i.3321-x16	1 GHz	2 GSamples	3.2 GS/s	3.2 GS/s
M5i.3330-x16	2 GHz	2 GSamples	6.4 GS/s	
M5i.3337-x16	2 GHz	2 GSamples	6.4 GS/s	3.2 GS/s
M5i.3350-x16	3 GHz	2 GSamples	10.0 GS/s	
M5i.3357-x16	3 GHz	2 GSamples	10.0 GS/s	5.0 GS/s
M5i.3360-x16	4.7 GHz	2 GSamples	10.0 GS/s	
M5i.3367-x16	4.7 GHz	2 GSamples	10.0 GS/s	5.0 GS/s

Options

Order no.	Option
M5i.xxxx-MEM8GS	Optional memory extension to 8 GSamples (16 GBytes)
M5i.3321-inptd	M5i.3321-x16 input stage optimized for time domain measurments with smooth step response.
M5i.xxxx-SH8-C2	Synchronization star-hub for up to 8 cards in one system, 2 synchronization cables included
M5i.xxxx-SH8-C4	Synchronization star-hub for up to 8 cards in one system, 4 synchronization cables included
M5i.xxxx-SH8-C8	Synchronization star-hub for up to 8 cards in one system, 8 synchronization cables included
Card-Upgrade	Upgrade for M5i.xxxx: Later installation of star-hub or inptd
M5i.xxxx-SyncCable	Additional synchronization cable for connecting star-hub to one card

Firmware Options

Order no.	Option
M5i.xxxx–spavg	Signal Processing Firmware Option: Block Average with TDA (later firmware-upgrade available)
M5i.xxxx-PulseGen	Firmware Option: adds 4 freely programmable digital pulse generators that use the XIO lines for output (later installation by firmware-upgrade available)

Services

Order no.	
Recal	Recalibration at Spectrum incl. calibration protocol

Standard Cables

		Order no.					
for Connections	Length	to BNC male	to BNC female	to SMA male	to SMA female	to SMB female	
Analog/Clk/Trig/XIO	80 cm	Cab-3mA-9m-80	Cab-3mA-9f-80	Cab-3mA-3mA-80		Cab-3f-3mA-80	
Analog/Clk/Trig/XIO	200 cm	Cab-3mA-9m-200	Cab-3mA-9f-200	Cab-3mA-3mA-200		Cab-3f-3mA-200	
Probes (short)	5 cm		Cab-3mA-9f-5				
Information	The standard adapter cables are based on RG174 cables and have a nominal attenuation of 0.3 dB/m at 100 MHz and 0.5 dB/m at 250 MHz. For high speed signals we recommend the low loss cables series CHF						

Low Loss Cables

Order No.	Option
CHF-3mA-3mA-200	Low loss cables SMA male to SMA male 200 cm
CHF-3mA-9m-200	Low loss cables SMA male to BNC male 200 cm
Information	The low loss adapter cables are based on MF141 cables and have an attenuation of 0.3 dB/m at 500 MHz and 0.5 dB/m at 1.5 GHz. They are recommended for signal frequencies of 200 MHz and above.

Amplifiers

Order no.	Bandwidth	Connection	Input Impedance	Coupling	Amplification
SPA.1841 [(2)]	2 GHz	SMA	50 Ohm	AC	x100 (40 dB)
SPA.1801 [(2)]	2 GHz	SMA	50 Ohm	AC	x10 (20 dB)
SPA.1601 [(2)]	500 MHz	BNC	50 Ohm	DC	x10 (20 dB)
Information	External Amplifiers with one channel, BNC/SMA female connections on input and output, manually adjustable offset, manually switchable settings. An external power supply for 100 to 240 VAC is included. Please be sure to order an adapter cable matching the amplifier connector type and matching the connector type for your A/D card input.				

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 M5i.xxxx-x16 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 M5i cards it is absolutely mandatory to have cooling fans installed in the chassis that exhaust the heat dissipation of the PC components.

- Make absolutely sure, that the on-board fans on the rear of the M5i card are not blocked by PC internal cabling or any other means.
- Make absolutely sure that the fan outlets on the front panel are all open and are not blocked by any outside obstacles.
- 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/PXIe cards to highest setting in the BIOS/UEFI.
- 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.

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 vulunable to mechanical damages when used not properly. Especially SMB and MMCX connctors 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.

M5i PCIe Cards

System Requirements

All Spectrum M5i.xxxx-x16 instrumentation cards are compliant to the PCI Express 3.0 standard and require in general two adjacent free 3/4 length PCI Express slot spaces. It is possible to install the card into a single PCIe slot if the adjacent slot is empty and has an accessible slot bracket. The adjacent slot can have a different slot type like PCI or can be unsoldered.

The main mounting slot must mechanically be a x16 slot, electrically all lane widths are supported, be it x1, x4, x8 or x16. For full data transfer performance using a Generation 3 slot is recommended but the card will also work in a Gen 1 or Gen 2 speed, but with lower transfer speed.

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 M5i 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 M5i cards mechanically require one PCI Express 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. Excessive bending of the card may damage the PCB permanently 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 M5i.xxxx-x16 PCI Express card in a PCIe 16 slot:

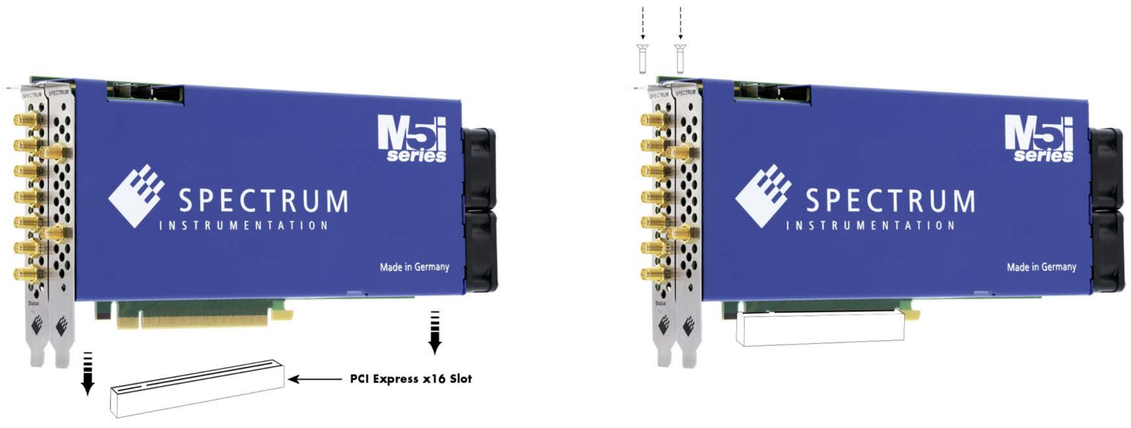


Image 6: Mounting M5i PCIe card into connector

Additional notes on PCIe x16 slot retention

M5i-xxxx-x16 cards 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.

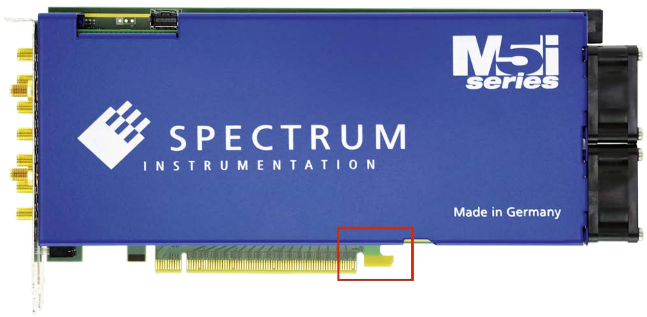


Image 7: M5i card location of the PCIe retention hook

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.

Providing additional power to M5i.xxxx-x16 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. The M5i PCIe cards are specified with a power consumption that exceeds the 25W of total power.

This is why all M5i cards **<u>must be supplied</u>** 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 prevent starting the card if there is no power connection In that case the front panel LED will shop a RED light.

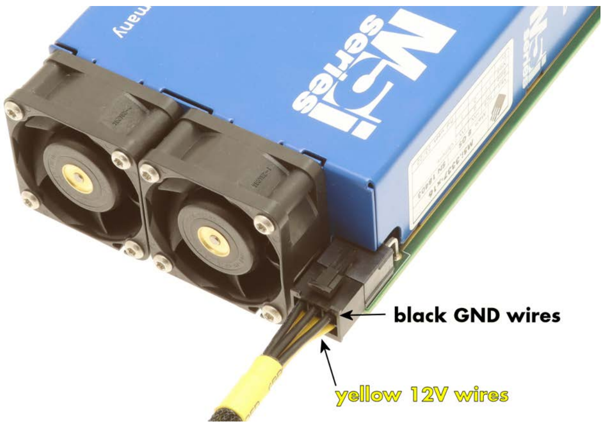


Image 8: M5i card mandatory power connection usage

Please <u>only connect 6-pin PCIe power cables</u> to the M5i 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 multiple boards synchronized by Star-Hub option

When fitting the card, please take care not the damage the motherboard with the lower edge of the three metal PCIe brackets.

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 ordered number of synchronization cables. All of them are matched to the same length, to achieve a minimal clock delay between the cards.

Only use the provided M5i miniature coaxial cables.

Remove the protective plastic caps on the cable before plugging. It is recommended to keep these protective caps, in case that you want to re-apply them to the cables, when they are not in use.

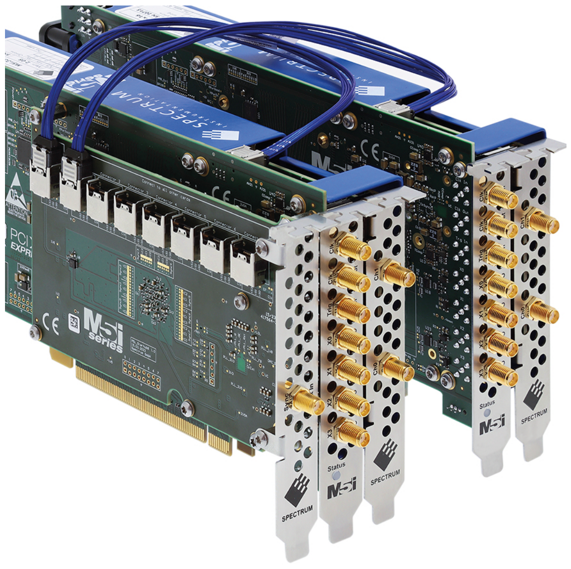


Image 9: Two M5i cards with Star-Hub

All of the cards, **including the master card, carrying the star-hub piggy-back module**, must be wired to the star-hub.

It is important, that the master card that is carrying the Star-Hub is connected to its dedicated "Carrier card" connector.
The software driver will detect, if the master card is not connected correctly and issue a Star-Hub connection error.

For all further slave cards that are to be synchronized, it does not matter which of the remaining available connectors on the Star-Hub module you use.

The software driver will automatically detect and enumerate all other synchronized boards automatically.

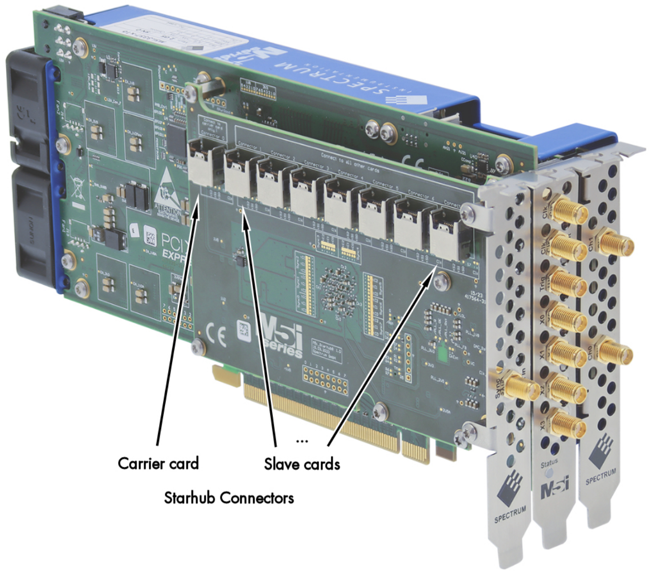


Image 10: M5i card showing mounted star-hub and the card's sync bus

All of the synchronization cables are secured against wrong insertion and will not latch into place, in case that the connector is not in the correct orientation.

BThe connectors latch in place once properly inserted and are hence locked and secured against unplugging unintentionally.

Consequently the metallic latch on the plugs is required to be pressed whilst removing any of the synchronization cables! The way that the connectors are oriented on either PCB allow for easy access of the plug's latch.#

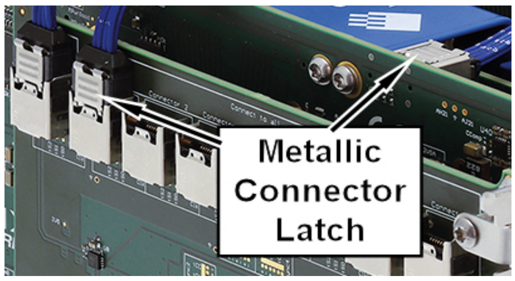


Image 11: Synchronisation connectors with marker on connector latch

Both of the synchronization cable plugs will easily slide into place and latch with a "click", when correctly inserted into the card and Star-Hub connectors. No excessive force is required.

The metallic shield of the connectors are very thin and might have slightly pointy corners. Caution is advised when inserting the cables.

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 M5i cards with the option „M5i.xxxx-SH8" installed, require three slots with ¾ PCIe length. On both sides of the connecting slot for the card, one additional slot in width is required to accommodate for the two slots required by any M5i.xxxx card plus the additional slot required, for the Star-Hub.

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 extra care to not bend the card in any direction while inserting it in the system. A bending of the card may damage the card's PCB and is not covered by the standard warranty.

Also take very special care, when inserting the board with its three brackets, not to damage any components on the mainboard, by very carefully observing the lower parts of all three PCIe brackets during the entire insertion process.

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 bent while being transported. When using freight forwarders, the transport and handling processes can be quite rough potentially subjecting the shopped 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!

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"

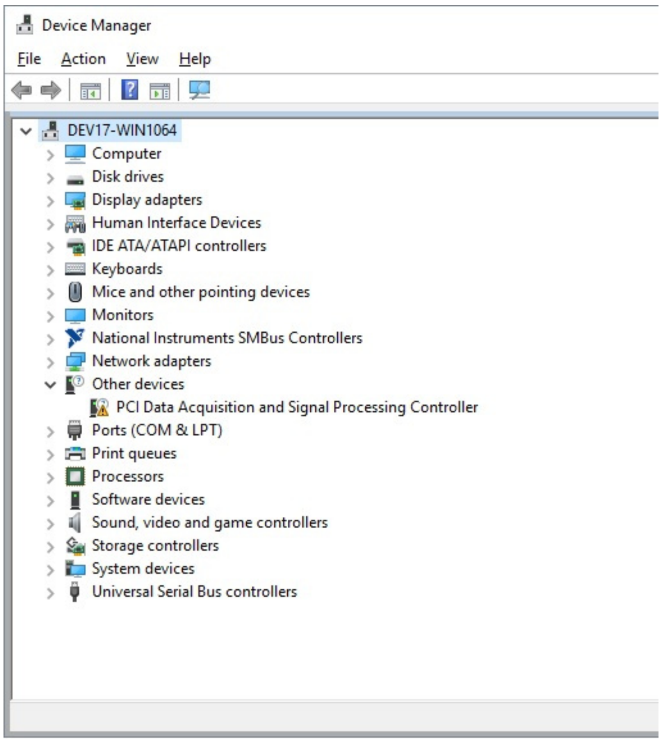
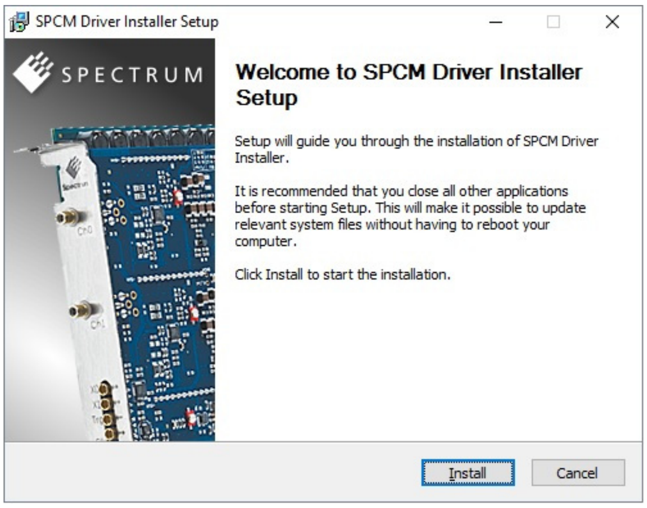


Image 12: Windows Device Manager showing a new Spectrum card

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.


SPCM Driver Installer Setup

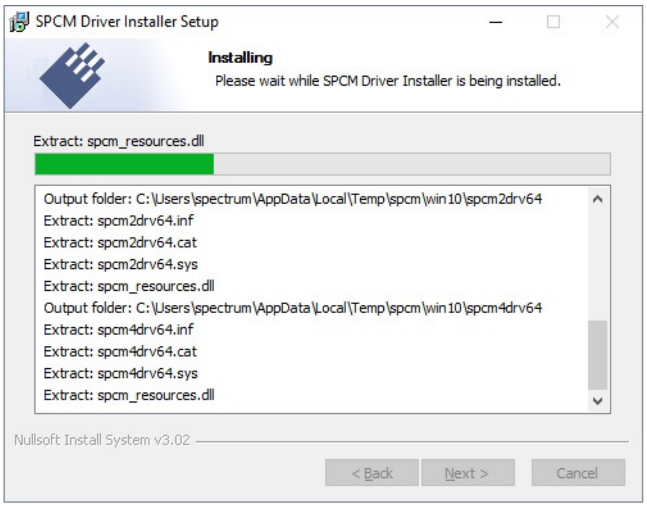
Welcome to SPCM Driver Installer Setup

Setup will guide you through the installation of SPCM Driver Installer.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Install to start the installation.

Image 13: Spectrum Driver Installer Welcome Screen


SPCM Driver Installer Setup

Installing

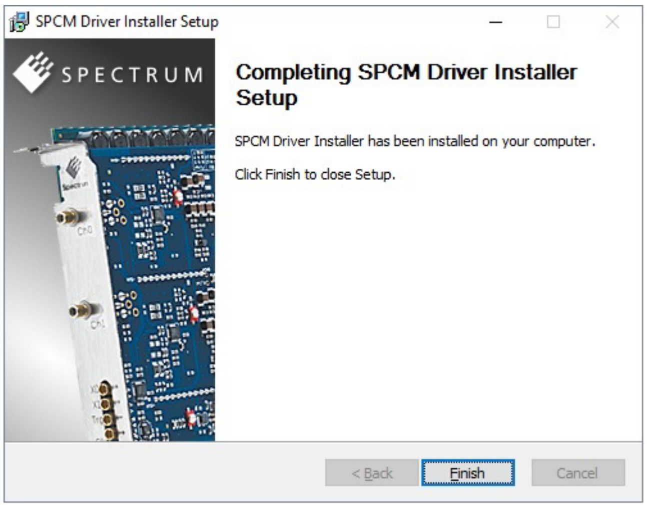
Please wait while SPCM Driver Installer is being installed.

Extract: spcm_resources.dll

```
Output folder: C:\Users\spectrum\AppData\Local\Temp\spcm\win10\spcm2drv64
Extract: spcm2drv64.inf
Extract: spcm2drv64.cat
Extract: spcm2drv64.sys
Extract: spcm_resources.dll
Output folder: C:\Users\spectrum\AppData\Local\Temp\spcm\win10\spcm4drv64
Extract: spcm4drv64.inf
Extract: spcm4drv64.cat
Extract: spcm4drv64.sys
Extract: spcm_resources.dll
```

Nullsoft Install System v3.02

Image 14: Spectrum Driver Installer - Progress


SPCM Driver Installer Setup

Completing SPCM Driver Installer Setup

SPCM Driver Installer has been installed on your computer.

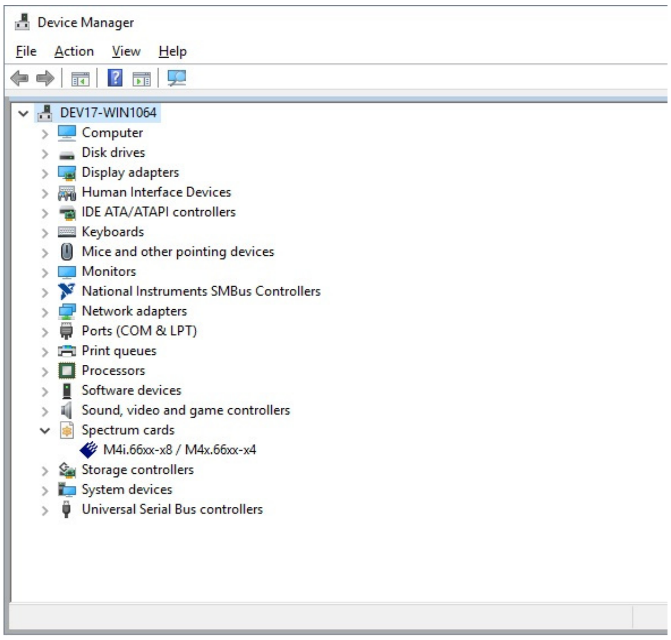
Click Finish to close Setup.

Image 15: 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.


Device Manager

File Action View Help

```
DEV17-WIN1064
  Computer
  Disk drives
  Display adapters
  Human Interface Devices
  IDE ATA/ATAPI controllers
  Keyboards
  Mice and other pointing devices
  Monitors
  National Instruments SMBus Controllers
  Network adapters
  Ports (COM & LPT)
  Print queues
  Processors
  Software devices
  Sound, video and game controllers
  Spectrum cards
    M4i.66xx-x8 / M4x.66xx-x4
  Storage controllers
  System devices
  Universal Serial Bus controllers
```

Image 16: 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_stdc++6.so - supporting libstdc++.so.6 on 32 bit systems
- spcm_linux_64bit_stdc++6.so - supporting libstdc++.so.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_prg -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

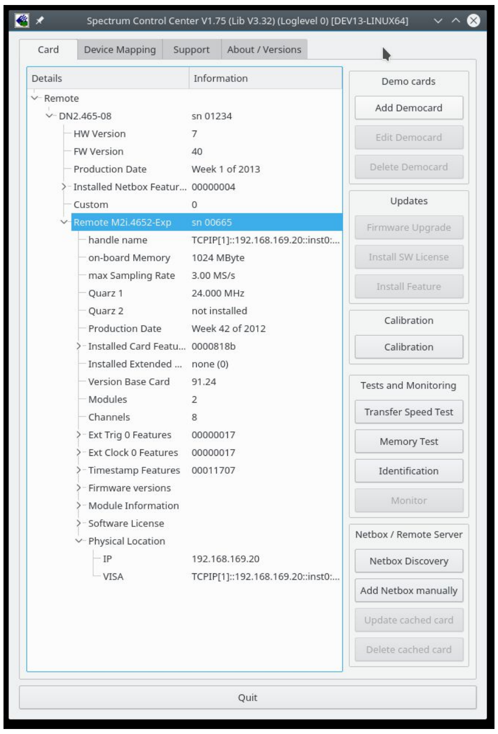
```
sudo dnf install SBench6
```

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)



Spectrum Control Center V1.75 (Lib V3.32) (Loglevel 0) [DEV13-LINUX64]

Details	Information
Remote	
DN2.465-08	sn 01234
HW Version	7
FW Version	40
Production Date	Week 1 of 2013
Installed Netbox Featur...	00000004
Custom	0
Remote M2i.4652-Exp	sn 00665
handle name	TCPIP[1]::192.168.169.20::inst0:...
on-board Memory	1024 MByte
max Sampling Rate	3.00 MS/s
Quarz 1	24.000 MHz
Quarz 2	not installed
Production Date	Week 42 of 2012
Installed Card Featu...	0000818b
Installed Extended ...	none (0)
Version Base Card	91.24
Modules	2
Channels	8
Ext Trig 0 Features	00000017
Ext Clock 0 Features	00000017
Timestamp Features	00011707
Firmware versions	
Module Information	
Software License	
Physical Location	
IP	192.168.169.20
VISA	TCPIP[1]::192.168.169.20::inst0:...

Image 17: Device Manager showing a new Spectrum card

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.

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

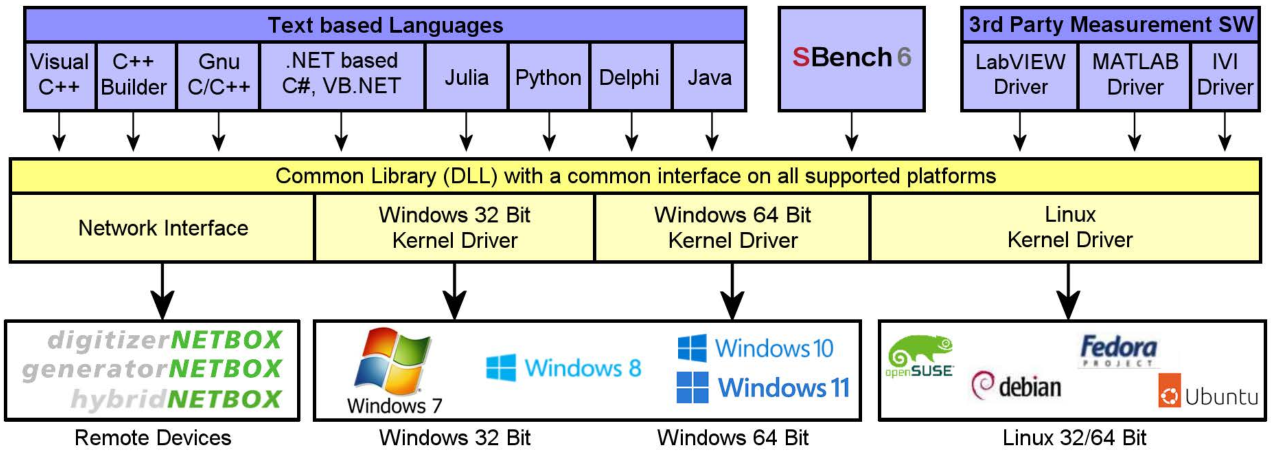


Image 18: Spectrum Kernel Driver, API Library and Software structure

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.

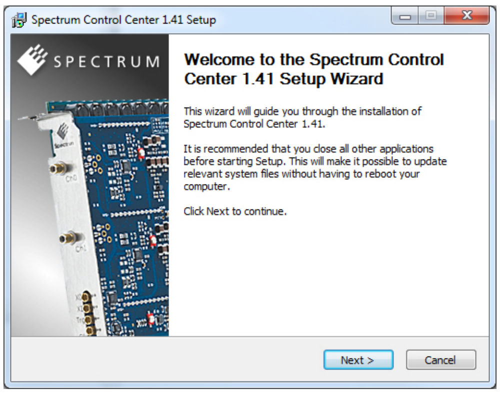
Card Control Center

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.


Spectrum Control Center 1.41 Setup

Welcome to the Spectrum Control Center 1.41 Setup Wizard

This wizard will guide you through the installation of Spectrum Control Center 1.41.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Image 19: 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

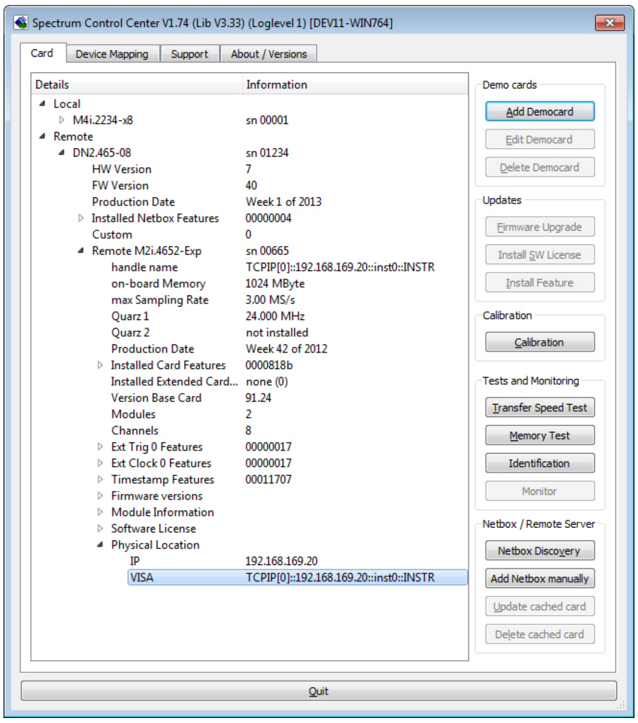
The Discovery function helps you to find and identify the Spectrum LXI instruments like digitizerNETBOX, generatorNETBOX or hybridNETBOX available to your computer on the network. The Discovery function will also locate Spectrum card products handled by an installed Spectrum Remote Server somewhere on 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.



Details	Information
Local	
M4i.2234-x8	sn 00001
Remote	
DN2.465-08	sn 01234
HW Version	7
FW Version	40
Production Date	Week 1 of 2013
Installed Netbox Features	00000004
Custom	0
Remote M2i.4652-Exp	sn 00665
handle name	TCPIP[0]::192.168.169.20::inst0::INSTR
on-board Memory	1024 MByte
max Sampling Rate	3.00 MS/s
Quarz 1	24.000 MHz
Quarz 2	not installed
Production Date	Week 42 of 2012
Installed Card Features	0000818b
Installed Extended Card...	none (0)
Version Base Card	91.24
Modules	2
Channels	8
Ext Trig 0 Features	00000017
Ext Clock 0 Features	00000017
Timestamp Features	00011707
Firmware versions	
Module Information	
Software License	
Physical Location	
IP	192.168.169.20
VISA	TCPIP[0]::192.168.169.20::inst0::INSTR

Image 20: 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.

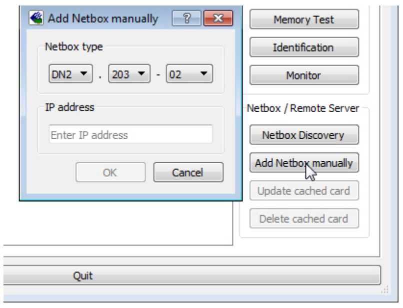


Image 21: Spectrum Control Center - entering an IP address for a NETBOX

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);
```

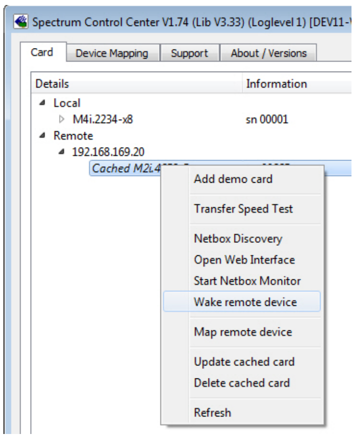


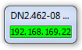
Image 22: Spectrum Control Center: wake on LAN for a cached card

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.

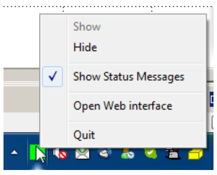


Image 23: Netbox Monitor activation

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.

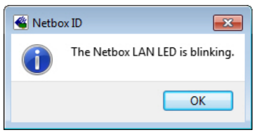
Left clicking on the tray icon will hide/show the small Netbox Monitor status window. Right clicking on the tray icon as shown in the picture on the right will open up a context menu. In here one can again select to hide/show the Netbox Monitor status window, one can directly open the web interface from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

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


Netbox ID

The Netbox LAN LED is blinking.

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:

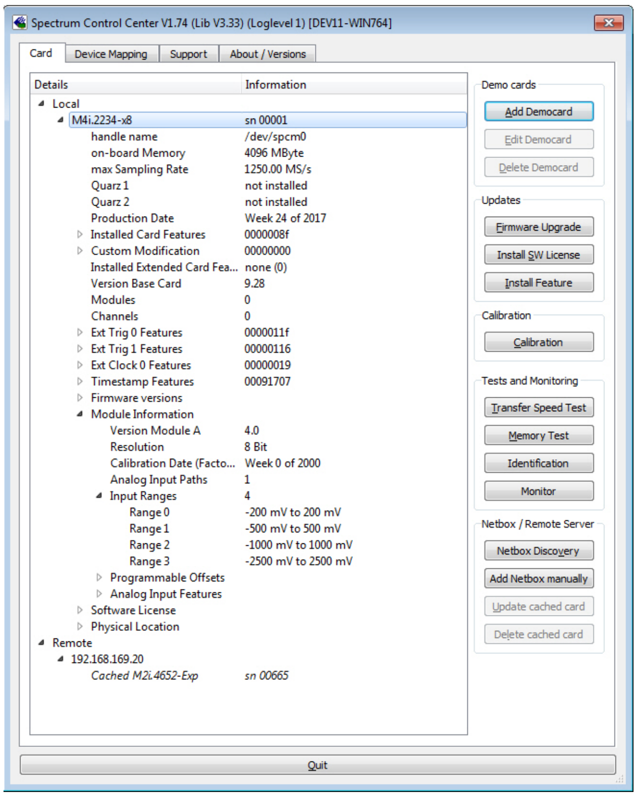
- On a digitizerNETBOX/generatorNETBOX/hybridNETBOX: the LAN LED light on the front plate of the device
- On local or remote M5i, M4i, M4x or M2p card: the indicator LED on the card's bracket

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.



```
Details                               Information
◢ Local
  ◢ M4i.2234-x8                       sn 00001
      handle name                     /dev/spcm0
      on-board Memory                 4096 MByte
      max Sampling Rate               1250.00 MS/s
      Quarz 1                         not installed
      Quarz 2                         not installed
      Production Date                 Week 24 of 2017
    ▷ Installed Card Features         0000008f
    ▷ Custom Modification             00000000
      Installed Extended Card Fea...  none (0)
      Version Base Card               9.28
      Modules                         0
      Channels                        0
    ▷ Ext Trig 0 Features             0000011f
    ▷ Ext Trig 1 Features             00000116
    ▷ Ext Clock 0 Features            00000019
    ▷ Timestamp Features              00091707
    ▷ Firmware versions
    ◢ Module Information
        Version Module A              4.0
        Resolution                    8 Bit
        Calibration Date (Facto...    Week 0 of 2000
        Analog Input Paths            1
      ◢ Input Ranges                  4
          Range 0                     -200 mV to 200 mV
          Range 1                     -500 mV to 500 mV
          Range 2                     -1000 mV to 1000 mV
          Range 3                     -2500 mV to 2500 mV
      ▷ Programmable Offsets
      ▷ Analog Input Features
    ▷ Software License
    ▷ Physical Location
◢ Remote
  ◢ 192.168.169.20
      Cached M2i.4652-Exp             sn 00665
```

Image 24: Spectrum Control Center: detailed hardware information on installed card

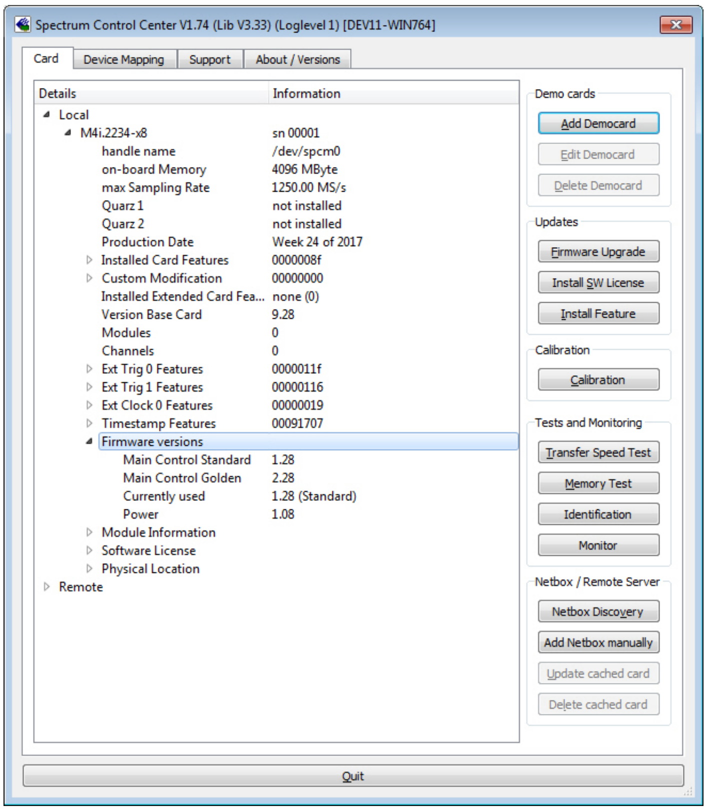
Firmware information

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.



```
Details                               Information
◢ Local
  ◢ M4i.2234-x8                       sn 00001
      handle name                     /dev/spcm0
      on-board Memory                 4096 MByte
      max Sampling Rate               1250.00 MS/s
      Quarz 1                         not installed
      Quarz 2                         not installed
      Production Date                 Week 24 of 2017
    ▷ Installed Card Features         0000008f
    ▷ Custom Modification             00000000
      Installed Extended Card Fea...  none (0)
      Version Base Card               9.28
      Modules                         0
      Channels                        0
    ▷ Ext Trig 0 Features             0000011f
    ▷ Ext Trig 1 Features             00000116
    ▷ Ext Clock 0 Features            00000019
    ▷ Timestamp Features              00091707
    ◢ Firmware versions
        Main Control Standard         1.28
        Main Control Golden           2.28
        Currently used                1.28 (Standard)
        Power                         1.08
    ▷ Module Information
    ▷ Software License
    ▷ Physical Location
▷ Remote
```

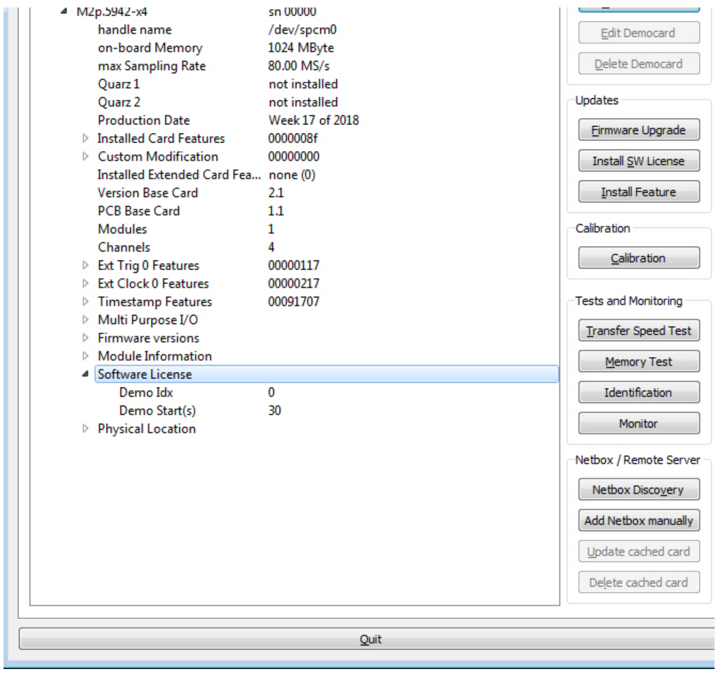
Image 25: Spectrum Control Center - showing firmware information of an installed card

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.



```
M2p.5942-x4                        sn 00000
  handle name                      /dev/spcm0
  on-board Memory                  1024 MByte
  max Sampling Rate                80.00 MS/s
  Quarz 1                          not installed
  Quarz 2                          not installed
  Production Date                  Week 17 of 2018
  Installed Card Features          0000008f
  Custom Modification              00000000
  Installed Extended Card Fea...   none (0)
  Version Base Card                2.1
  PCB Base Card                    1.1
  Modules                          1
  Channels                         4
  Ext Trig 0 Features              00000117
  Ext Clock 0 Features             00000217
  Timestamp Features               00091707
  Multi Purpose I/O
  Firmware versions
  Module Information
  Software License
    Demo Idx                       0
    Demo Start(s)                  30
  Physical Location
```

Image 26: Spectrum Control Center - showing firmware information of an installed card

Driver information

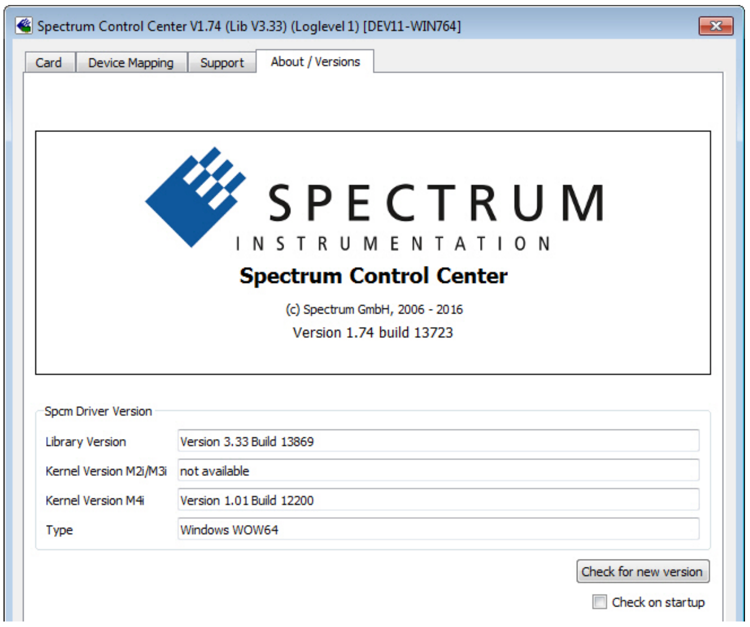
The Spectrum card control center also offers a way to gather information on the installed and used Spectrum driver.

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 form the control panel. For details in driver details within the control panel please stick to the section on driver installation in your hardware manual.



Spectrum Control Center V1.74 (Lib V3.33) (Loglevel 1) [DEV11-WIN764]

Spectrum Control Center

(c) Spectrum GmbH, 2006 - 2016

Version 1.74 build 13723

Spcm Driver Version

Library Version	Version 3.33 Build 13869
Kernel Version M2i/M3i	not available
Kernel Version M4i	Version 1.01 Build 12200
Type	Windows WOW64

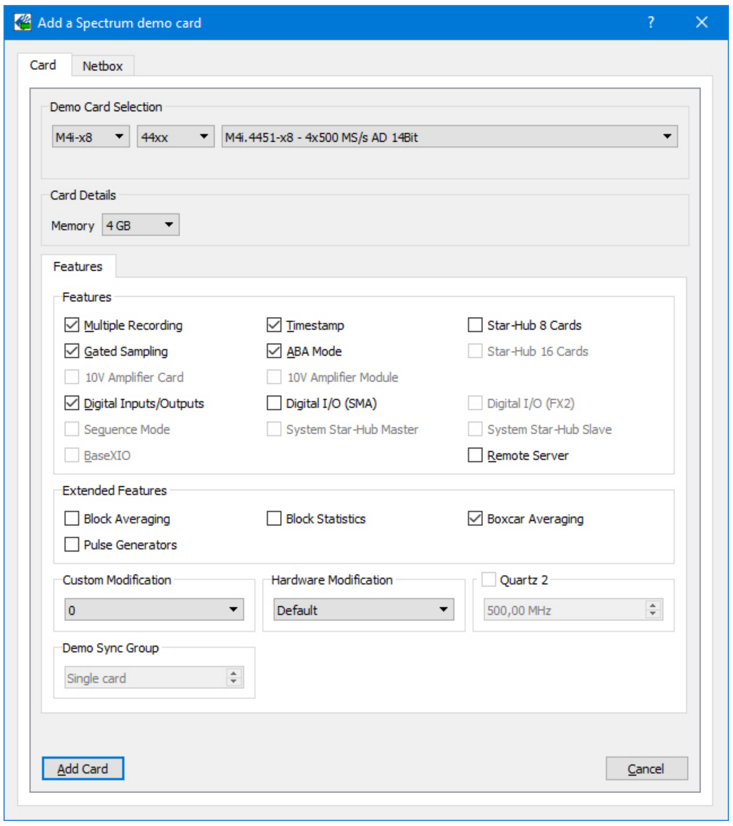
Image 27: Spectrum Control Center - showing driver information details

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.


Add a Spectrum demo card

Card | Netbox

Demo Card Selection: M4i-x8 | 44xx | M4i.4451-x8 - 4x500 MS/s AD 14Bit

Card Details: Memory 4 GB

Features

Features:
- [✓] Multiple Recording
- [✓] Timestamp
- [ ] Star-Hub 8 Cards
- [✓] Gated Sampling
- [✓] ABA Mode
- [ ] Star-Hub 16 Cards
- [ ] 10V Amplifier Card
- [ ] 10V Amplifier Module
- [✓] Digital Inputs/Outputs
- [ ] Digital I/O (SMA)
- [ ] Digital I/O (FX2)
- [ ] Sequence Mode
- [ ] System Star-Hub Master
- [ ] System Star-Hub Slave
- [ ] BaseXIO
- [ ] Remote Server

Extended Features:
- [ ] Block Averaging
- [ ] Block Statistics
- [✓] Boxcar Averaging
- [ ] Pulse Generators

Custom Modification: 0
Hardware Modification: Default
[ ] Quartz 2: 500,00 MHz
Demo Sync Group: Single card

Add Card | Cancel

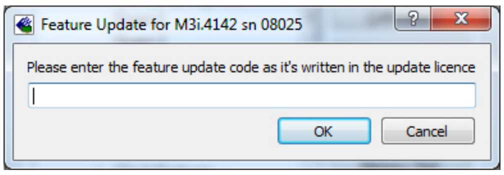
Image 28: Spectrum Control Center - adding a demo card to the sysstem

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.


Feature Update for M3i.4142 sn 08025

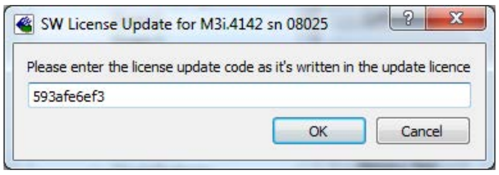
Please enter the feature update code as it's written in the update licence

OK | Cancel

Image 29: 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.


SW License Update for M3i.4142 sn 08025

Please enter the license update code as it's written in the update licence

593afe6ef3

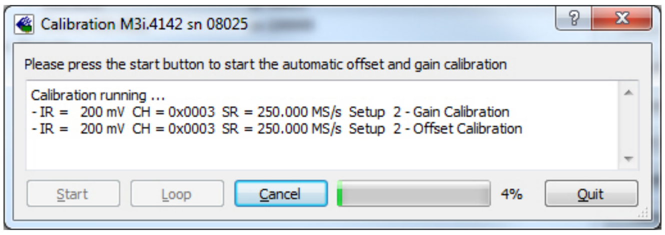
OK | Cancel

Image 30: Spectrum Control Center - software license installe

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


Calibration M3i.4142 sn 08025

Please press the start button to start the automatic offset and gain calibration

```
Calibration running ...
- IR =   200 mV  CH = 0x0003  SR = 250.000 MS/s  Setup  2 - Gain Calibration
- IR =   200 mV  CH = 0x0003  SR = 250.000 MS/s  Setup  2 - Offset Calibration
```

Start | Loop | Cancel | 4% | Quit

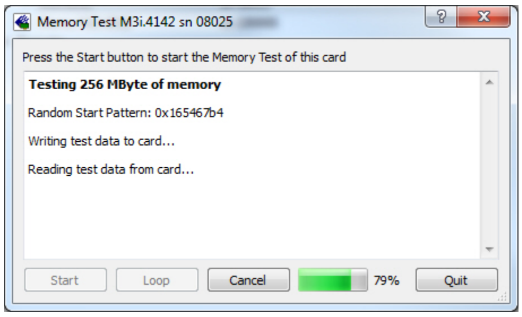
Image 31: Spectrum Control Center - running an on-board calibration

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.

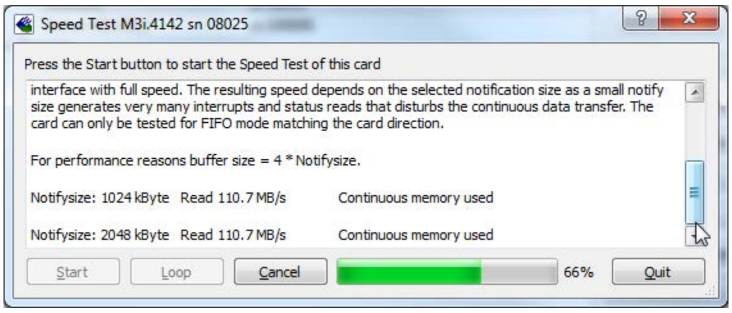

Memory Test M3i.4142 sn 08025
Press the Start button to start the Memory Test of this card
Start
Loop
Cancel
79%
Quit

Testing 256 MByte of memory
Random Start Pattern: 0x165467b4
Writing test data to card...
Reading test data from card...

Image 32: 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:


Speed Test M3i.4142 sn 08025
Press the Start button to start the Speed Test of this card
Start
Loop
Cancel
66%
Quit

interface with full speed. The resulting speed depends on the selected notification size as a small notify size generates very many interrupts and status reads that disturbs the continuous data transfer. The card can only be tested for FIFO mode matching the card direction.

For performance reasons buffer size = 4 * Notifysize.

```
Notifysize: 1024 kByte  Read 110.7 MB/s     Continuous memory used
Notifysize: 2048 kByte  Read 110.7 MB/s     Continuous memory used
```

Image 33: Spectrum Control Center - running a transfer speed test of one card

- 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.

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 channels * 2 Bytes/Sample * 28000000] = 112000000 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 MB/s = 108 * 1024 * 1024 = 113246208 Bytes/second.

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

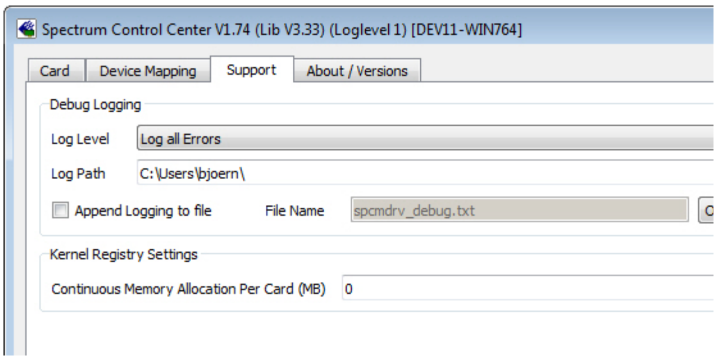
Debug logging for support cases

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.



Log Level: Log all Errors
Log Path: C:\Users\bjoern\
Append Logging to file: ☐
File Name: spcmdrv_debug.txt
Continuous Memory Allocation Per Card (MB): 0

Image 34: 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.

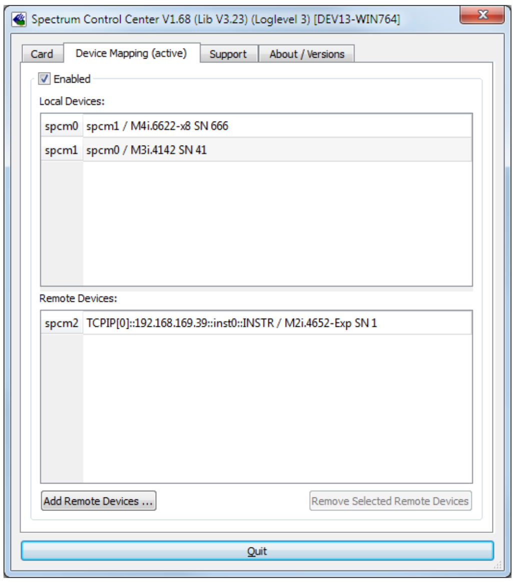
In the left column the re-mapped device name is visible that is given to the device in the right column with its original un-mapped device string.

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.


Spectrum Control Center V1.68 (Lib V3.23) (Loglevel 3) [DEV13-WIN764]
Card
Device Mapping (active)
Support
About / Versions
Add Remote Devices ...
Remove Selected Remote Devices
Quit

☑ Enabled

Local Devices:

spcm0	spcm1 / M4i.6622-x8 SN 666
spcm1	spcm0 / M3i.4142 SN 41

Remote Devices:

spcm2	TCPIP[0]::192.168.169.39::inst0::INSTR / M2i.4652-Exp SN 1

Image 35: Spectrum Control Center - using device mapping

Firmware upgrade

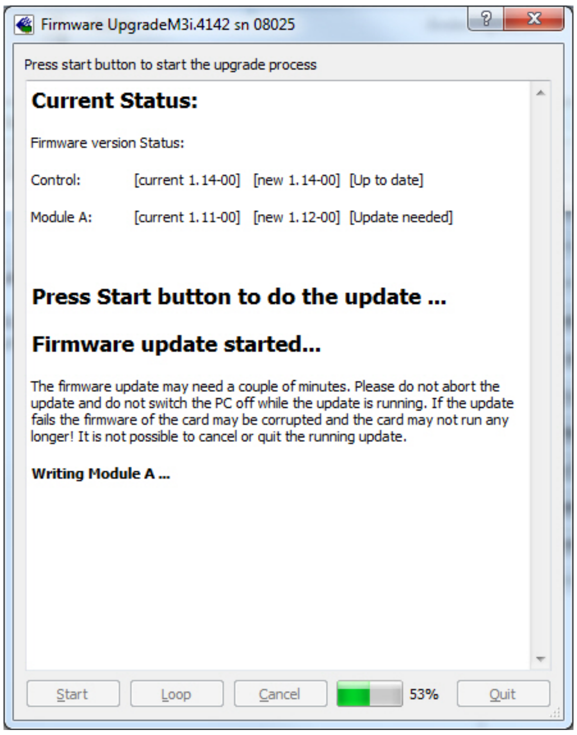
One of the major features of the card control center is the ability to update the card's firmware by an easy-to-use software. The latest firmware revisions can be found in the download section of our homepage under http://www.spectrum-instrumentation.com.

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.



Press start button to start the upgrade process

Current Status:

Firmware version Status:

```
Control:    [current 1.14-00]  [new 1.14-00]  [Up to date]
Module A:   [current 1.11-00]  [new 1.12-00]  [Update needed]
```

Press Start button to do the update ...

Firmware update started...

The firmware update may need a couple of minutes. Please do not abort the update and do not switch the PC off while the update is running. If the update fails the firmware of the card may be corrupted and the card may not run any longer! It is not possible to cancel or quit the running update.

Writing Module A ...

Image 36: 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

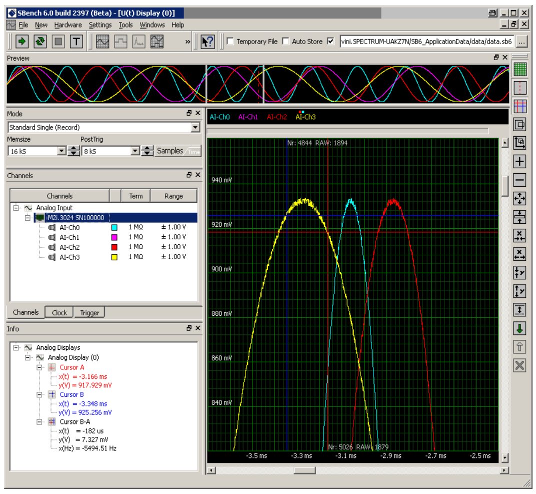


Image 37: SBench 6 overview of main functionality with demo data

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 3: list of C/C++ header files in driver

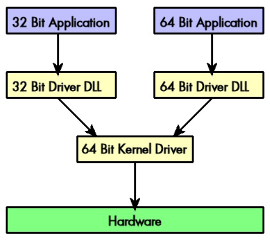
dlltyp.h	Includes the platform specific definitions for data types and function declarations. All data types are based on these definitions. The use of this type definition file allows the use of examples and programs on different platforms without changes to the program source. The header file supports 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 4: 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 5: 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 opended 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 opended 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 lRegister 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 6: 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	spcm_dwGetParam_i32	value is returned as 32 bit integer in plValue
32 bit register	read	spcm_dwGetParam_i64	value is returned as 64 bit integer in pllValue
32 bit register	read	spcm_dwGetParam_i64m	value is returned as 64 bit integer, the lower part in plValueLow, the upper part in plValueHigh. The upper part can be ignored as it's only a sign extension
32 bit register	read	spcm_dwGetParam_d64	value is returned as 64 bit double in pdValue
32 bit register	write	spcm_dwSetParam_i32	32 bit value can be directly written
32 bit register	write	spcm_dwSetParam_i64	64 bit value can be directly written, please be sure not to exceed the valid register value range
32 bit register	write	spcm_dwSetParam_i64m	32 bit value is written as llValueLow, the value llValueHigh needs to contain the sign extension of this value. In case of llValueLow being a value >= 0 llValueHigh can be 0, in case of llValueLow being a value < 0, llValueHigh has to be -1.
32 bit register	write	spcm_dwSetParam_d64	32 bit value needs to converted to double. Please make sure no to exceed the valid register range
64 bit register	read	spcm_dwGetParam_i32	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 ERR_EXCEEDSINT32 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	spcm_dwGetParam_i64	value is returned as 64 bit integer value in pllValue independent of the value of the internal register.
64 bit register	read	spcm_dwGetParam_i64m	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 plValueLow contains the 32 bit value and the upper part plValueHigh 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	spcm_dwGetParam_d64	value is returned as 64 bit double in pdValue. Please note that double values are limited to 2^48. Any larger value is not returned with full precision.
64 bit register	write	spcm_dwSetParam_i32	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 used.
64 bit register	write	spcm_dwSetParam_i64	the value has to be split into two parts. Be sure to fill the upper part lValueHigh 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	spcm_dwSetParam_i64m	the value can be written directly independent of the size.
64 bit register	write	spcm_dwSetParam_d64	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 _i64 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 ERRORTEXTLEN 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 SpeErr.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;
```

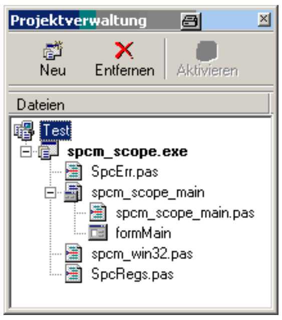


Image 38: Structure of the Delphi examples

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 ctype, 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 = 0x00000001l                      # hardware reset
M2CMD_CARD_WRITESETUP = 0x00000002l                 # write setup only
M2CMD_CARD_START = 0x00000004l                      # start of card (including writesetup)
M2CMD_CARD_ENABLETRIGGER = 0x00000008l              # 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 sEr-
rorTextBuffer);

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.

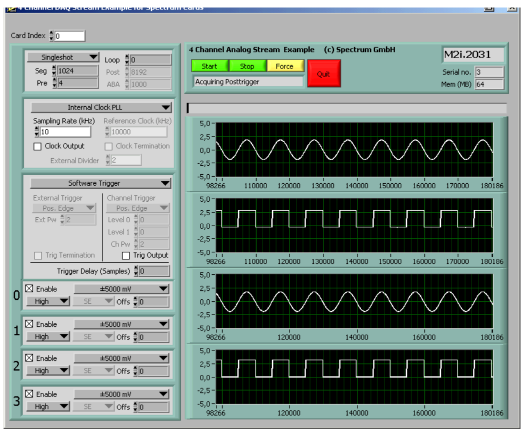
Examples

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.


4 Channel Analog Stream Example (c) Spectrum GmbH

Card Index 0

Singleshot; Seg 1024; Pre 4; Loop 0; Post 8192; ABA 1000

Internal Clock PLL; Sampling Rate (kHz) 10; Reference Clock (kHz) 10000; Clock Output; Clock Termination; External Divider 2

Software Trigger; External Trigger Pos. Edge; Ext Pw 2; Channel Trigger Pos. Edge; Level 0 0; Level 1 0; Ch Pw 2; Trig Termination; Trig Output; Trigger Delay (Samples) 0

0 Enable ±5000 mV High SE Offs 0

1 Enable ±5000 mV High SE Offs 0

2 Enable ±5000 mV High SE Offs 0

3 Enable ±5000 mV High SE Offs 0

Acquiring Posttrigger

M2i.2031; Serial no. 3; Mem (MB) 64

Image 39: 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.

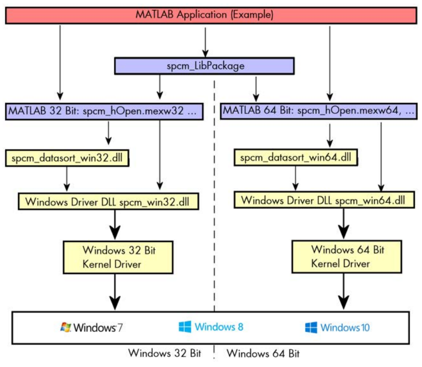

MATLAB Application (Example)
spcm_LibPackage
MATLAB 32 Bit: spcm_hOpen.mexw32 ...
MATLAB 64 Bit: spcm_hOpen.mexw64, ...
spcm_datasort_win32.dll
spcm_datasort_win64.dll
Windows Driver DLL spcm_win32.dll
Windows Driver DLL spcm_win64.dll
Windows 32 Bit Kernel Driver
Windows 64 Bit Kernel Driver
Windows 7
Windows 8
Windows 10
Windows 32 Bit
Windows 64 Bit

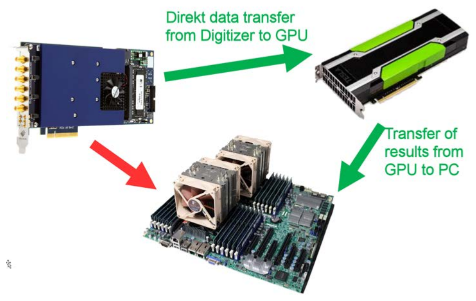
Image 40: Spectrum MATLAB driver structure

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.


Direkt data transfer from Digitizer to GPU
Transfer of results from GPU to PC
Image 41: 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 7: 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
                                                               // or alternatively
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST0::INSTR");     // Opens the remote board #0
                                                               // all other boards require an index:
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST1::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[ERRORTEXTLEN];

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 ocurred 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);
if (dwErrorCode)                                                    // check for an error
    {
    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 8: 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 9: available models and decimal and hexadecimal value of model

Card type	Card type as defined in regs.h	Value hexadecimal	Value decimal
M5i.3321-x16	TYP_M5I3321_X16	A3321h	668449
M5i.3330-x16	TYP_M5I3330_X16	A3330h	668464
M5i.3337-x16	TYP_M5I3337_X16	A3337h	668471
M5i.3350-x16	TYP_M5I3350_X16	A3350h	668496
M5i.3357-x16	TYP_M5I3357_X16	A3357h	668503
M5i.3360-x16	TYP_M5I3360_X16	A3360h	668512
M5i.3367-x16	TYP_M5I3367_X16	A3367h	668519

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 10: 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 11: 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 12: Spectrum API: register for reading back the PXIe card slot number

Register	Value	Direction	Description
SPC_PXIHWSLOTNO	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 13: 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 14: Spectrum API: Register overview of reading current firmware

Register	Value	Direction	Description						
				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 15: 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 16: 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 17: 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 18: 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 19: 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 20: 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_STARHUBSYSSLAVE	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, adding16 additional digital I/Os via SMB connectors.
SPCM_FEAT_DIG16_FX2	80000h	M2p: Set if option M2p.xxxx-DigFX2 is installed, adding16 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 21: Spectrum API: Extended feature register and available extended feature flags

Register	Value	Direction	Description
SPC_PCIEXTFEATURES	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 22: 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).

Table 22: Spectrum API: register overview of miscellaneous cards information

Register	Value	Direction	Description
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 register returns the basic type of the card:

Table 23: 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 24: 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_LINUX32	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_LINUX64	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 25: 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 26: 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 27: 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	000000FFh		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 28: 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.

Analog Inputs

Channel Selection

One key setting that influences all other possible settings is the channel enable register. A unique feature of the Spectrum cards is the possibility to program the number of channels you want to use. All on-board memory can then be used by these activated channels.

This description shows you the channel enable register for the complete card family. However, your specific board may have less channels depending on the card type that you have purchased and therefore does not allow you to set the maximum number of channels shown here.

Register		Value	Direction	Description
SPC_CHENABLE		11000	read/write	Sets the channel enable information for the next board run.
	CHANNEL0	1	Activates channel 0	
	CHANNEL1	2	Activates channel 1	

The channel enable register is set as a bitmap. That means one bit of the value corresponds to one channel to be activated. To activate more than one channel the values have to be combined by a bitwise OR.

Example showing how to activate 2 channels:

```
spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0 | CHANNEL1);
```

The following table shows all allowed settings for the channel enable register.

Channels to activate				
Ch0	Ch1	Values to program	Value as hex	Value as decimal
X		CHANNEL0	1h	1
	X	CHANNEL1	2h	2
X	X	CHANNEL0 | CHANNEL1	3h	3

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

To help user programs it is also possible to read out the number of activated channels that correspond to the currently programmed bitmap.

Register	Value	Direction	Description
SPC_CHCOUNT	11001	read	Reads back the number of currently activated channels.

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

```
spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0 | CHANNEL1);
spcm_dwGetParam_i32 (hDrv, SPC_CHENABLE, &lActivatedChannels);
spcm_dwGetParam_i32 (hDrv, SPC_CHCOUNT, &lChCount);

printf ("Activated channels bitmask is: 0x%08x\n", lActivatedChannels);
printf ("Number of activated channels with this bitmask: %d\n", lChCount);
```

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

```
Activated channels bitmask is: 0x00000003
Number of activated channels with this bitmask: 2
```

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.

Setting up the inputs

This analog acquisition board uses separate input stages and converters on each channel. This gives you the possibility to set up the desired and concerning your application best suiting input range also separately for each channel. All input stage related settings can easily be set by the corresponding input registers. The table below shows the available input stage registers and possible standard values for your type of board. As there are also modified versions available with different input ranges it is recommended to read out the currently available input ranges as shown later in this chapter.

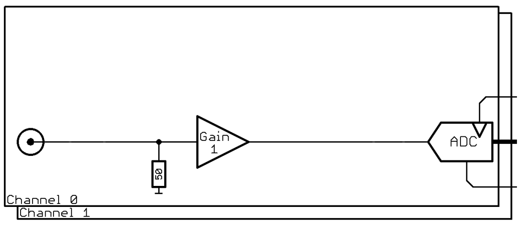


Image 42: complete input stage

Input ranges

This analog acquisition board has several different input ranges for each channel. This gives you the possibility to set up the desired and concerning your application best suiting input range also separately for each channel. The input ranges can easily be set by the corresponding input registers. The table below shows the available input registers and possible standard ranges for your type of board. As there are also modified versions available with different input ranges it is recommended to read out the currently available input ranges as shown later in this chapter.

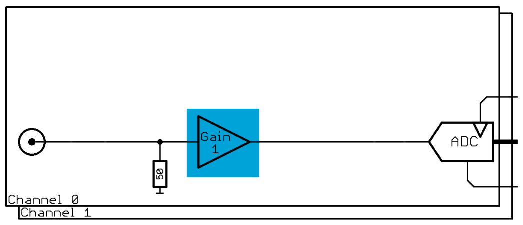


Image 43: input stage showing the gain amplifier

The available input rages are read out using the following registers.

Table 29: Spectrum API: registers for reading the installed input ranges from card EEPROM

Register	Value	Direction	Description
SPC_READIRCOUNT	3000	read	Returns the number of available input ranges.
SPC_READRANGEMIN0	4000	read	Reads the lower border of input range 0 in mV
SPC_READRANGEMIN1	4001	read	Reads the lower border of input range 1 in mV
...	...	...	
SPC_READRANGEMAX0	4100	read	Reads the upper border of input range 0 in mV
SPC_READRANGEMAX1	4101	read	Reads the upper border of input range 1 in mV
...	...	...	

The following example reads out the number of available input ranges and reads and prints the minimum and maximum value of all input ranges.

```
spcm_dwGetParam_i32 (hDrv, SPC_READIRCOUNT, &lNumberOfRanges);
for (i = 0; i < lNumberOfRanges; i++)
    {
    spcm_dwGetParam_i32 (hDrv, SPC_READRANGEMIN0 + i, &lMinimumInputRage);
    spcm_dwGetParam_i32 (hDrv, SPC_READRANGEMAX0 + i, &lMaximumInputRange);
    printf („Range %d: %d mV to %d mV\n“, i, lMinimumInputRange, lMaximumInputRange);
    }
```

The input range is selected individually for each channel. Please note that the correct input path needs to be set

Table 30: Spectrum API: input range settings register and available vales depending on installed low-voltage option

Register	Value	Direction	Description
SPC_AMP0	30010	read/write	Defines the input range of channel0.
SPC_AMP1	30110	read/write	Defines the input range of channel1.
	200	± 200 mV calibrated input range for the appropriate channel.	
	500	± 500 mV calibrated input range for the appropriate channel.	
	1000	± 1 V calibrated input range for the appropriate channel.	
	2500	± 2.5 V calibrated input range for the appropriate channel.	

Input offset

In most cases the external signals will not be symmetrically related to ground. If you want to acquire such asymmetrical signals, it is possible to use the smallest input range that matches the biggest absolute signal amplitude without exceeding the range.

The figure at the right shows this possibility. But in this example you would leave half of the possible resolution unused.

It is much more efficient if you shift the signal on-board to be as symmetrical as possible and to acquire it within the best possible range.

This results in a much better use of the converters resolution.

On this acquisition boards from Spectrum you have the possibility to adjust the input offset separately for each channel.

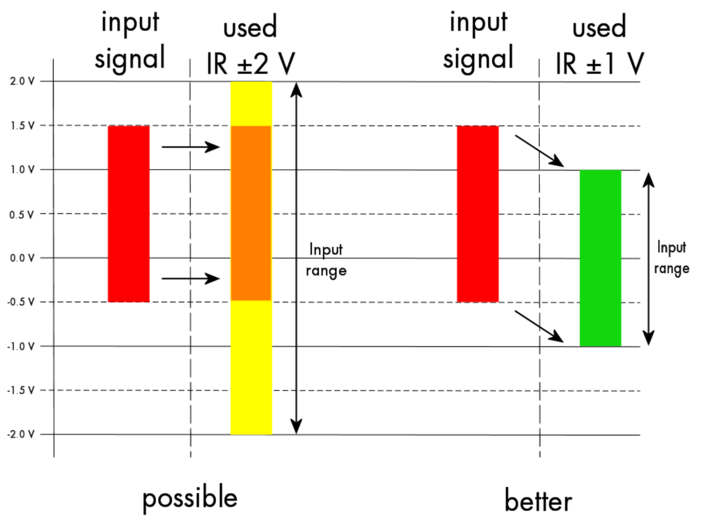


Image 44: Spectrum API: using the input offset shifting to optimize the usage of the input range

The example in the right figure shows signals with a range of ±1.0 V that have offsets up to ±1.0 V. So related to the desired input range these signals have offsets of ±100 %.

For compensating such offsets you can use the offset register for each channel separately. If you want to compensate the +100 % offset of the outer left signal, you would have to set the offset to -100 % to compensate it.

As the offset levels are relatively to the related input range, you have to calculate and set your offset again when changing the input's range.

The table below shows the offset registers and the possible offset ranges for your specific type of board.

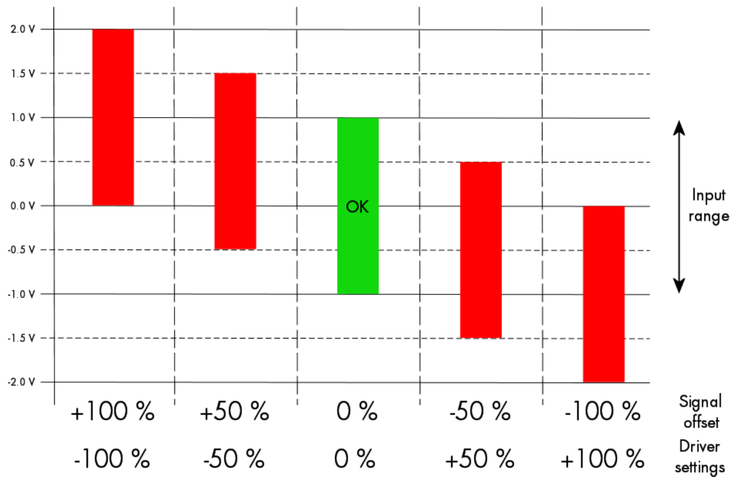


Image 45: Spectrum API: effects of different input offset setting

Image 46: Spectrum API: input offset registers and available register settingss

Register	Value	Direction	Description	Offset range
SPC_OFFS0	30000	read/write	Defines the input's offset and therefore shifts the input of channel0.	±100 % in steps of 1 %
SPC_OFFS1	30100	read/write	Defines the input's offset and therefore shifts the input of channel1.	±100 % in steps of 1 %

Read out of input features

Each input path (if multiple paths are available on the card) has different features 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 acquisition cards. Some of these features are not installed on your specific hardware:

Table 31: Spectrum API: register to read the analog input features and the meaning of the feature flags

Register	Value	Direction	Description
SPC_READAIFEATURES	3101	read	Returns a bit map with the available features of that input path. The possible return values are listed below.

	Value	Description
SPCM_AI_TERM	00000001h	Programmable input termination available
SPCM_AI_SE	00000002h	Input is single-ended. If available together with SPC_AI_DIFF: input type is software selectable
SPCM_AI_DIFF	00000004h	Input is differential. If available together with SPC_AI_SE: input type is software selectable
SPCM_AI_OFFSPERCENT	00000008h	Input offset programmable in per cent of input range
SPCM_AI_OFFSMV	00000010h	Input offset programmable in mV

	Value	Description
SPCM_AI_OVERRANGEDETECT	00000020h	Programmable overrange detection available
SPCM_AI_DCCOUPLING	00000040h	Input is DC coupled. If available together with AC coupling: coupling is software selectable
SPCM_AI_ACCOUPLING	00000080h	Input is AC coupled. If available together with DC coupling: coupling is software selectable
SPCM_AI_LOWPASS	00000100h	Input has a individually per channel selectable low pass filter (bandwidth limit)
SPCM_AI_ACDC_OFFS_COMP	00000200h	Input has a selectable offset compensation for HF-Path with AC/DC coupling/source mismatch.
SPCM_AI_GLOBALLOWPASS	00000800h	Card has a globally selectable low pass (affects all channels with the same setting)
SPCM_AI_AUTOCALOFFS	00001000h	Input offset can be auto calibrated on the card
SPCM_AI_AUTOCALGAIN	00002000h	Input gain can be auto calibrated on the card
SPCM_AI_AUTOCALOFFSNOIN	00004000h	Input offset can auto calibrated on the card if inputs are left open
SPCM_AI_HIGHIMP	00008000h	Input has a high impedance mode available
SPCM_AI_LOWIMP	00010000h	Input has a low impedance mode (50 Ohm) available
SPCM_AI_INDIVPULSEWIDTH	00100000h	Trigger pulsewidth is individually per channel programmable

The following example shows a setup of the input range of a two channel card.

Please note that this is a general example and the number of input channels may not match your card channels.

```
spcm_dwSetParam_i32 (hDrv, SPC_AMP0 ,    1000); // Set up channel0 to the range of ± 1.0 V
spcm_dwSetParam_i32 (hDrv, SPC_AMP1 ,     500); // Set up channel1 to the range of ± 0.5 V
```

Automatic on-board calibration of the offset and gain settings

All of the channels are calibrated in factory before the board is shipped. These values are stored in the on-board EEProm under the default settings. If you have asymmetrical signals, you can adjust the offset easily with the corresponding registers of the inputs as shown before.

To start the automatic offset adjustment, simply write the register, mentioned in the following table.

Before you start an automatic offset adjustment make sure, that no signal is connected to any input. Leave all the input connectors open and then start the adjustment. All the internal settings of the driver are changed, while the automatic offset compensation is in progress.

Table 32: Spectrum API: automatic offset compensation register and valid register settings

Register	Value	Direction	Description
SPC_ADJ_AUTOADJ	50020	write	Performs the automatic offset compensation in the driver either for all input ranges or only the actual.
ADJ_ALL	0	Automatic offset adjustment for all input ranges.	

As all settings are temporarily stored in the driver, the automatic adjustment will only affect these values. After exiting your program, all calibration information will be lost. To give you a possibility to save your own settings, most Spectrum card have at least one set of user settings that can be saved within the on-board EEPROM. The default settings of the offset and gain values are then read-only and cannot be written to the EEPROM by the user. If the card has no user settings the default settings may be overwritten.

You can easily either save adjustment settings to the EEPROM with SPC_ADJ_SAVE or recall them with SPC_ADJ_LOAD. These two registers are shown in the table below. The values for these EEPROM access registers are the sets that can be stored within the EEPROM. The amount of sets available for storing user offset settings depends on the type of board you use. The table below shows all the EEPROM sets, that are available for your board.

Table 33: Spectrum API: loading and storing calibration values to the EEPROM

Register	Value	Direction	Description
SPC_ADJ_LOAD	50000	write	Loads the specified set of settings from the EEPROM. The default settings are automatically loaded, when the driver is started.
		read	Reads out, what kind of settings have been loaded last.
SPC_ADJ_SAVE	50010	write	Stores the current settings to the specified set in the EEPROM.
		read	Reads out, what kind of settings have been saved last.
ADJ_DEFAULT	0	Default settings, no user settings available	

If you want to make an offset and gain adjustment on all the channels and store the data to the ADJ_DEFAULT set of the EEPROM you can do this the way, the following example shows.

```
spcm_dwSetParam_i32 (hDrv, SPC_ADJ_AUTOADJ,    ADJ_ALL  ); // Activate offset/gain adjustment on all channels
spcm_dwSetParam_i32 (hDrv, SPC_ADJ_SAVE   ,    ADJ_DEFAULT); // and store values to DEFAULT set in the EEPROM
```

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_WRITESETUP	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:

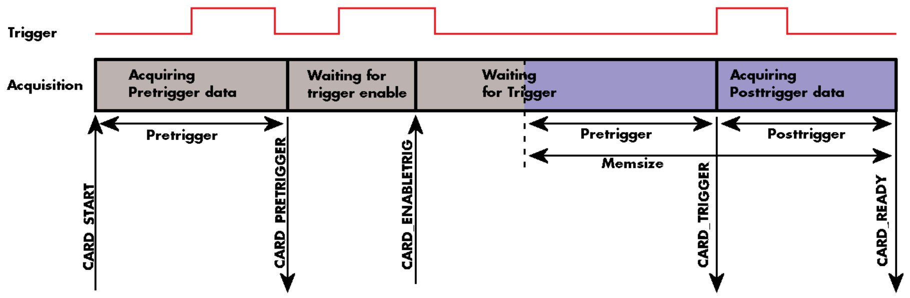


Image 47: Acquisition cards: graphical overview of acquisition status and card command interaction

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.

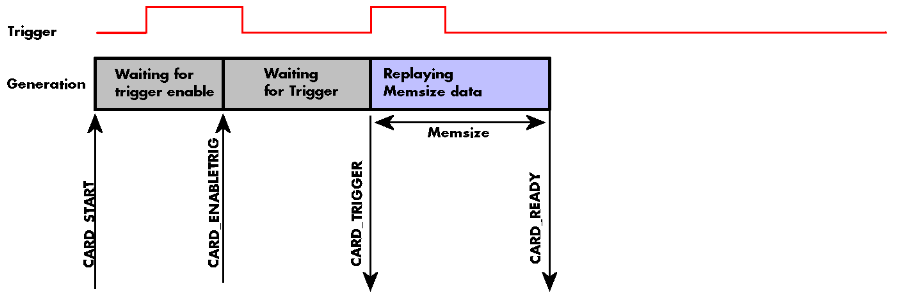


Image 48: Generation cards: graphical overview of generation status and card command interaction

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_GPUTOCARD	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);

// explicitely stop DMA tranfer 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).

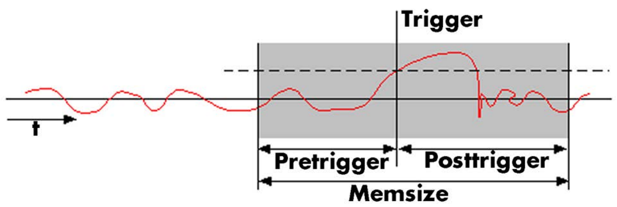


Image 49: standard acquisition mode and pretrigger/posttrigger/trigger relation

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 use 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_SEGMENTSIZE]. 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 chuncks
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. Please keep in mind that each sample needs 2 bytes of memory to be stored. 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: Spectrum API: Limits of pre trigger, post trigger and memory size

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
1 Ch	Standard Single	64	Mem	32	32	Mem - 32	32	32	256G - 32	32	not used			not used		
					(defined by mem and post)											
	Standard Multi	64	Mem	32	32	32k	32	32	Mem-32	32	64	Mem	32	not used		
					(defined by segment and post)			(Limited by max pretrigger)								
	FIFO Single	not used			32	32k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	32k	32	32	256G-32	32	64	pre+post	32	0 (∞)	4G - 1	1
					(defined by segment and post)			(Limited by max pretrigger)								
2 Ch	Standard Single	64	Mem/2	32	32	Mem/2 - 32	32	32	256G - 32	32	not used			not used		
					(defined by mem and post)											
	Standard Multi	64	Mem/2	32	32	16k	32	32	Mem/2-32	32	64	Mem/2	32	not used		
					(defined by segment and post)			(Limited by max pretrigger)								
	FIFO Single	not used			32	16k	32	not used			64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi	not used			32	16k	32	32	256G-32	32	64	pre+post	32	0 (∞)	4G - 1	1
					(defined by segment and post)			(Limited by max pretrigger)								

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:

Table 47: Spectrum-API: maximum memory sizes for different memory upgrade options

	Installed Memory	
	2 GSample (4 GByte)	**8 GSample (16 GByte)** **(Option: M5i.xxx-MEM8GS)**
Mem	2 GSample	8 GSample
Mem / 2	1 GSample	4 GSample

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:

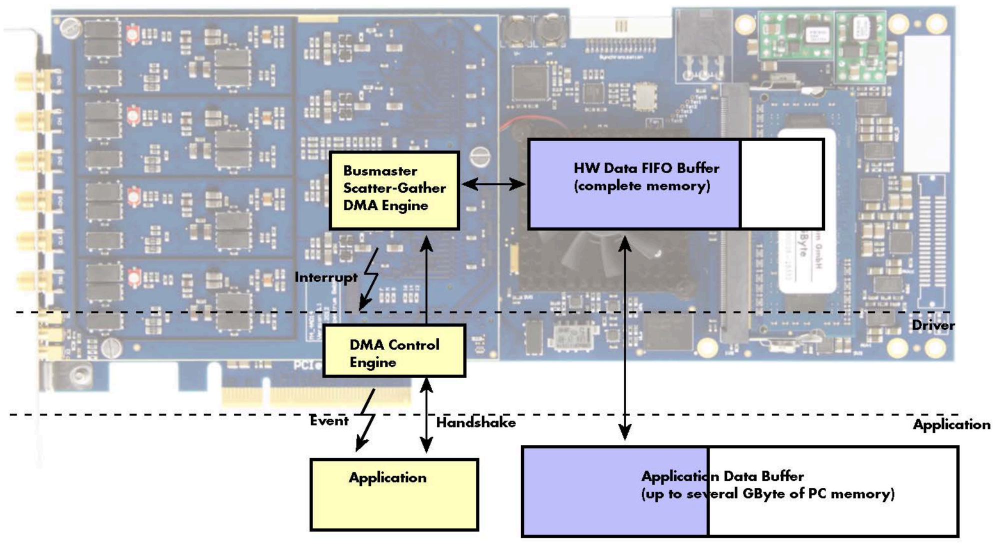

Busmaster Scatter-Gather DMA Engine
HW Data FIFO Buffer (complete memory)
Interrupt
Driver
DMA Control Engine
Event
Handshake
Application
Application
Application Data Buffer (up to several GByte of PC memory)

Image 50: 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 48: 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 49: 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.
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 unmatching length as DMA can than 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.

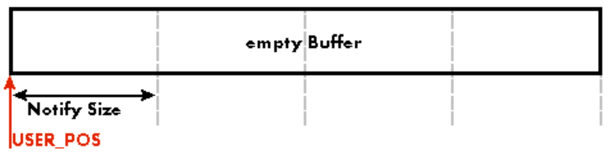
M5i:

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.

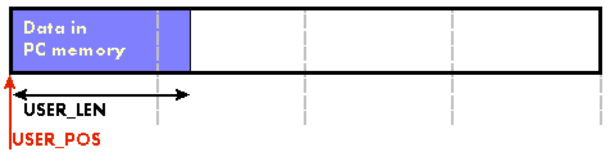
Step 1: Buffer definition

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.



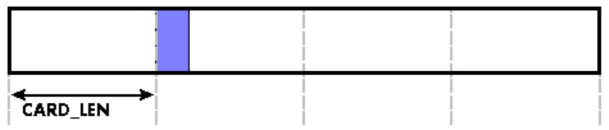
Step 2: Start and first data available

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.



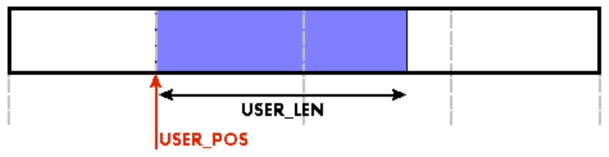
Step 3: set the first data available for card

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.



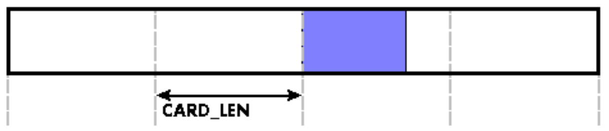
Step 4: next data available

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.

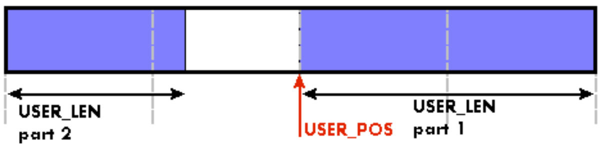


Step 5: set data available again

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.

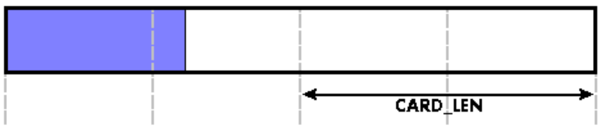


Step 6: roll over the end of buffer



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. Afte 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”, llBytesPos);

        // 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[llBytesPos], 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”, llBytesPos);

        // 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[llBytesPos], 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 organization

Data is organized in a multiplexed way in the transfer buffer. If using 2 channels data of first activated channel comes first, then data of second channel.

Table 50: M5i cards data organization

Activated Channels	Ch0	Ch1	Samples ordering in buffer memory starting with data offset zero																
1 channel	X		A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16
1 channel		X	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16
2 channels	X	X	A0	B0	A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	A8

The samples are re-named for better readability. A0 is sample 0 of channel 0, B4 is sample 4 of channel 1, and so on.

Sample format

The card is using 12 bit A/D samples, that are stored in twos complement in two 8 bit data byte. 12 bit resolution means that data is ranging from -2048...to...+2048. Data is stored in little-endian format, the upper 8 bit come first and the lower 8 bit second.:

Table 51: data sample format in standard mode and with digital inputs enable

	Standard Mode	Digital inputs enabled SPCM_XMODE_DIGIN
	M5i.33xx	M5i.33xx
Data bit	12 bit ADC resolution	12 bit ADC resolution
D15	ADX Bit 11 (sign extension)	Multi-Purpose XIO3
D14	ADX Bit 11 (sign extension)	Multi-Purpose XIO2
D13	ADX Bit 11 (sign extension)	Multi-Purpose XIO1
D12	ADX Bit 11 (sign extension)	Multi-Purpose XIO0
D11	ADx Bit 11 (MSB)	ADx Bit 11 (MSB)
D10	ADx Bit 10	ADx Bit 10
D9	ADx Bit 9	ADx Bit 9
D8	ADx Bit 8	ADx Bit 8
D7	ADx Bit 7	ADx Bit 7
D6	ADx Bit 6	ADx Bit 6
D5	ADx Bit 5	ADx Bit 5
D4	ADx Bit 4	ADx Bit 4
D3	ADx Bit 3	ADx Bit 3
D2	ADx Bit 2	ADx Bit 2
D1	ADx Bit 1	ADx Bit 1
D0	ADx Bit 0 (LSB)	ADx Bit 0 (LSB)

Converting ADC samples to voltage values

The Spectrum driver also contains a register that holds the value of the decimal value of the full scale representation of the installed ADC. This value should be used when converting ADC values (in LSB) into real-world voltage values, because this register also automatically takes any specialities into account, such as slightly reduced ADC resolution with reserved codes for gain/offset compensation.

Register	Value	Direction	Description
SPC_MIINST_MAXADCVALUE	1126	read	Contains the decimal code (in LSB) of the ADC full scale value.

In case of a board that uses an 8 bit ADC that provides the full ADC code (without reserving any bits) the returned value would be 128. The peak value for a ±1.0 V input range would be 1.0 V (or 1000 mV).

$$V_{In} = ADC_{Code} \times \frac{InputRange_{peak}}{ADC_{max}}$$

A returned sample value of for example +49 (decimal, two's complement, signed representation) would then convert to:

$$V_{in} = 49 \times \frac{1000\ \text{mV}}{128} = 382.81\ \text{mV}$$

A returned sample value of for example -55 (decimal) would then convert to:

$$V_{in} = -55 \times \frac{1000\ \text{mV}}{128} = -429.69\ \text{mV}$$

When converting samples that contain any additional data such as for example additional digital channels or over-range bits, this extra information must be first masked out and a proper sign-extension must be performed, before these values can be used as a signed two's complement value for above formulas.

Clock generation

Overview

The Spectrum M5i PCI Express (PCIe) cards offer a wide variety of different clock modes to match all the customers' needs. 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 on all M5i cards for clock generation.

The purpose of this chapter is to give you a guide to the best matching clock settings for your specific application and needs.

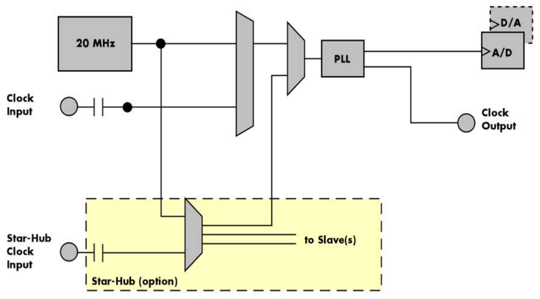


Image 51: M5i clock section overview

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 52: 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_EXTREFCLOCK	32	Enables internal PLL with external reference for sample clock generation	

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 (PLL with internal reference)

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 on-board oscillator acts as a reference to the internal PLL. The specification is found in the technical data section of this manual.

External Clock (reference 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 exact matching sample rate. The external clock is divided/multiplied using a PLL allowing a wide range of external clock modes.

Depending on whether the external reference clock is used with a single card or with multiple cards connected/synchronized via Star-Hub, the externally fed in clock must be either connected to the "Clock-In" connector of the card itself or connected to the Clock Input located on the additional PCIe bracket of the Star-Hub module.

Synchronization Clock (option Star-Hub)

The star-hub option allows the synchronization of up to 8 cards of the M5i series from Spectrum with a minimal phase delay between the different cards. The clock is distributed from the master card carrying the Star-Hub to all connected cards. For details on the synchronization option please take a look at the dedicated chapter 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 53: 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 54: 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 55: 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"
```

In all clock modes, the sampling rate can only be programmed as maximum samplingrate and divisions of this. Valid sampling rates are [max], [max/2], [max/4], [max/8], ... [max/131072]. Any programmed sampling rate in between will automatically be rounded to the next matching divided sampling clock.

Minimum internal sampling rate

The minimum internal sampling rates depend on the specific type of board. This value can be found in the technical data section of this manual.

Maximum internal sampling rate (standard clock mode)

Table 56: Spectrum API: maximum internal sampling rate depending on channel selection and model

activated Channels		M5i.3321	M5i.3330	M5i.3337	M5i.3350 M5i.3360	M5i.3357 M5i.3367
Ch0	Ch1					
X		3.2 GS/s	6.4 GS/s	6.4 GS/s	10.0 GS/s	10.0 GS/s
	X	3.2 GS/s	n.a.	3.2 GS/s[1]	n.a.	5.0 GS/s[1]
X	X	3.2 GS/s	n.a.	3.2 GS/s	n.a.	5.0 GS/s

[1] Card can also run with full speed (6.4 GS/s or 10.0 GS/s) on channel 1 but will have a reduced signal quality compared to using channel 0. It is not recommended to use channel 1 in this mode, as channel 0 is optimized for full speed acquisition.

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 57: 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 58: 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 59: 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
```

Depending on whether the external reference clock is used with a single M5i card or with multiple M5i cards connected/synchronized via Star-Hub, the externally fed in clock must be either connected to the "Clock-In" connector of the card itself or connected to the Clock Input located on the additional PCIe bracket of the Star-Hub module.

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 60: 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.

Trigger modes and related registers

General Description

The trigger modes of the Spectrum M5i series A/D and D/A cards are very extensive and give you the possibility to detect nearly any trigger event you can think of.

You can choose between more than 10 external trigger modes and up to 20 internal trigger modes (on analog acquisition cards) including software and channel trigger, depending on your type of board. Many of the channel trigger modes can be independently set for each input channel (on A/D boards only) resulting in a even bigger variety of modes. 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

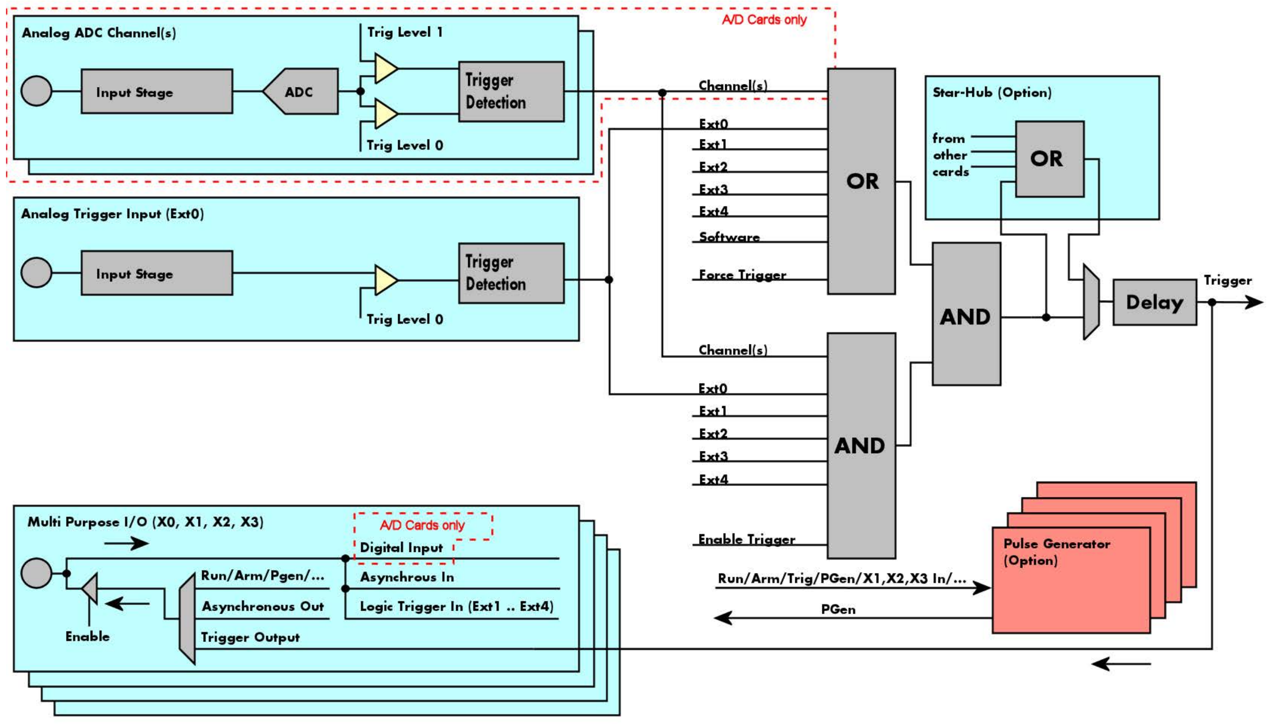


Image 52: M5i card trigger engine overview with the different trigger sources and trigger outputs

The trigger engine of the M5i 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.

On A/D cards each analog input channel has two trigger level comparators to detect edges as well as windowed triggers. All card types have also different external trigger sources. One main trigger source (Ext0/Trig0) with one analog level comparator. Additionally four multi purpose inputs/outputs can be used as additional logic (TTL) trigger sources as well. These lines can also be software programmed to either inputs or outputs some extended status signals.

The Enable trigger allows the user to enable or disable all trigger sources (including channel trigger on A/D cards 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 related 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 main external trigger (external analog window trigger), the secondary external trigger (external analog comparator trigger and software trigger and a channel OR mask.

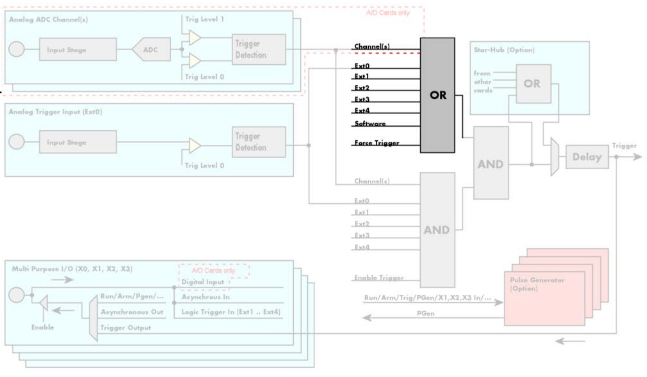


Image 53: Trigger overview - trigger OR mask

Every trigger source of the M5i 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 (A/D cards only) 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.

The table below shows the relating register for the general OR mask and the possible constants that can be written to it.

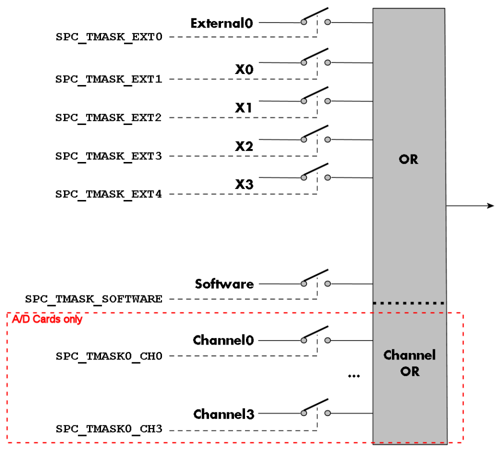


Image 54: trigger OR mask details

Table 61: Spectrum API: external trigger OR mask related software 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	0h		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) trigger 0 (Trig In) for the OR mask. The card will trigger if the programmed condition for this input is valid.
SPC_TMASK_EXT1	4h		Enables the X0 (logic) trigger for the OR mask. The card will trigger if the programmed condition for this input is valid.
SPC_TMASK_EXT2	8h		Enables the X1 (logic) trigger for the OR mask. The card will trigger if the programmed condition for this input is valid.
SPC_TMASK_EXT3	10h		Enables the X2 (logic) trigger for the OR mask. The card will trigger if the programmed condition for this input is valid.
SPC_TMASK_EXT4	20h		Enables the X3 (logic) trigger for the OR mask. The card will trigger if 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 explicitely cleared, as otherwise the software trigger will override other modes.

The following example shows, how to setup the OR mask, for the two external trigger inputs, ORing them together. When using just a single trigger, only this particular trigger must be used in the OR mask register, respectively. As an example a simple edge detection has been chosen for Ext1 input and a window edge detection has been chosen for Ext0 input. 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_EXT0_LEVEL0, 1800);      // External trigger level set to 1.8 V
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_POS);  // Setting up to detect positive edges

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT1_MODE, SPC_TM_NEG);  // Setting up X1 logic trigger for falling edges

// Enable both external triggers within the OR mask, by ORing the mask flags together
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT1 | SPC_TMASK_EXT0);
```

The table below is showing the registers for the channel OR mask (A/D cards only) and the possible constants that can be written to it.

Table 62: Spectrum API: channel trigger OR mask related software register 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…3) for the channel OR mask are set, if available.
SPC_TRIG_CH_ORMASK0	40460	read/write	Includes the analog channels (0…3) 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.	

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_LEVEL0, 0);             // Trigger level is zero crossing
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE, SPC_TM_POS);      // Setting up channel 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.

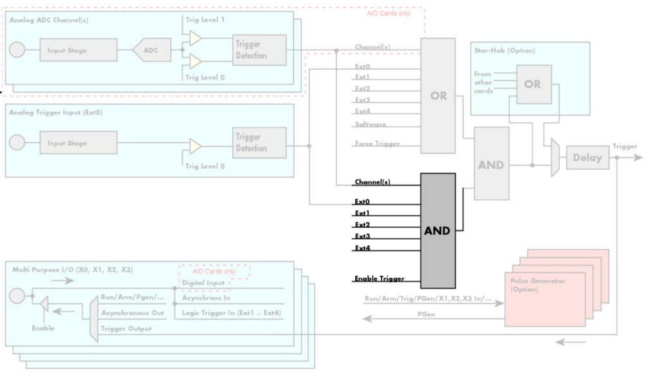


Image 55: Trigger overview - trigger AND mask

Every trigger source of the M5i 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 (A/D cards only) 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.

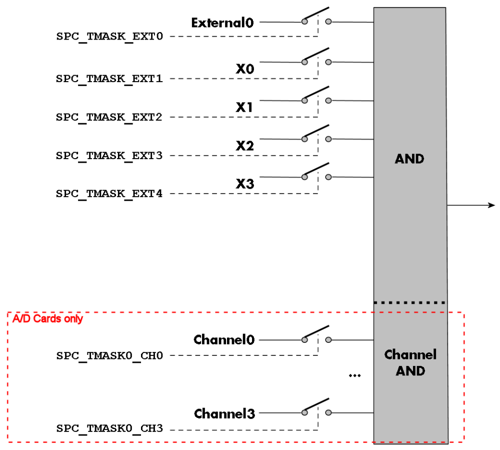


Image 56: trigger AND mask details

The table below shows the relating register for the general AND mask and the possible constants that can be written to it.

Table 63: Spectrum API: external trigger AND mask related software register 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) trigger 0 for the AND mask. The card will trigger if the programmed condition for this input is valid.	
	SPC_TMASK_EXT1	4h	Enables the X0 (logic) trigger for the AND mask. The card will trigger if the programmed condition for this input is valid.	
	SPC_TMASK_EXT2	8h	Enables the X1 (logic) trigger for the AND mask. The card will trigger if the programmed condition for this input is valid.	
	SPC_TMASK_EXT3	10h	Enables the X2 (logic) trigger for the AND mask. The card will trigger if the programmed condition for this input is valid.	
	SPC_TMASK_EXT4	20h	Enables the X3 (logic) trigger for the AND mask. The card will trigger if 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_LEVEL0, 2000);       // Trigger level is 2.0 V (2000 mV)
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 (A/D cards only) and all the constants for the different channels.

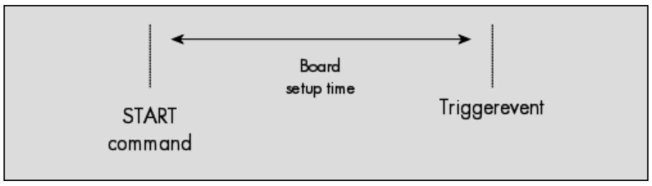
Table 64: Spectrum API: channel trigger AND mask related software register and available settings

Register		Value	Direction	Description
SPC_TRIG_CH_AVAILANDMASK0		40470	read	Bitmask, in which all bits of the below mentioned sources/channels (0…3) for the channel AND mask are set, if available.
SPC_TRIG_CH_ANDMASK0		40480	read/write	Includes the analog or digital channels (0…3) 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.	

The following example shows, how to setup the AND mask for a channel trigger. As an example a simple level 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_ANDMASK0, SPC_TMASK_CH0);// Enable channel0 trigger within AND mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_LEVEL0, 0);             // channel level to detect is zero level
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE, SPC_TM_HIGH);     // Setting up ch0 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 65: 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 66: 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 67: 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 M5i 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 M5i 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].

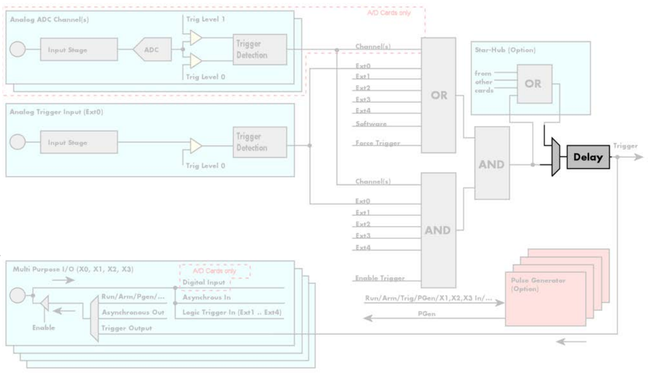


Image 57: trigger engine overview with marked trigger delay stage

The following table shows the related register and the possible values. A value of 0 disables the trigger delay.

Table 68: 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…[256G -32] in steps of 32 (12 bit cards)		Defines the additional trigger delay in number of sample clocks. The trigger delay can be programmed up to (256GSamples - 32) = 274877906912. Stepsize is 32 samples for 12 bit cards.

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“.

Trigger holdoff

All the cards of the Spectrum M5i series allow the user to program a trigger holdoff time when using the segmented acquisition or generation called Multiple Recording/Multiple Replay. This can be useful when observing and analyzing certain signals that are packeted or bursty in nature.

Using a trigger holdoff will result in an artificially inserted dead-time after each posttrigger area, in which the trigger engine will reject all detected trigger events. The holdoff value is programmed in samples and the resulting holdoff time will therefore be [Programmed Delay] / [Sampling Rate].

The following table shows the related register and the possible values. A value of 0 disables the trigger holdoff.

Table 69: Spectrum API: trigger holdoff related registers and settings for these

Register	Value	Direction	Description
SPC_TRIG_AVAILHOLDOFF	40802	read	Contains the maximum available holdoff as a decimal integer value.
SPC_TRIG_HOLDOFF	40811	read/write	Defines the trigger holdoff for the card's trigger engine for segmented modes (Multi, ABA, Gate).
	0		No additional holdoff will be added.
	32…[256G -32] in steps of 32 (12 bit cards)		Defines the trigger holdoff in number of sample clocks. The trigger holdoff can be programmed up to (256GSamples - 32) = 274877906912. Stepsize is 32 samples for 12 bit cards.

The example shows, how to use the trigger holdoff command:

```
spcm_dwSetParam_i64 (hDrv, SPC_TRIG_HOLDOFF, 1984); // A trigger holdoff is set to 1984 sample clocks
```

Trigger Counter

The number of acquired trigger events is counted in hardware and can be read out while the acquisition is running or after the acquisition has finished. The trigger events are counted both in standard mode as well as in FIFO mode.

Table 70: Spectrum API: trigger counter register and register return values

Register	Value	Direction	Description
SPC_TRIGGERCOUNTER	200905	read	Returns the number of trigger events that has been acquired since the acquisition start. The internal trigger counter has 48 bits. It is therefore necessary to read out the trigger counter value with 64 bit access or 2 x 32 bit access if the number of trigger events exceed the 32 bit range.

The trigger counter feature needs at least driver version V2.17 and firmware version V20 (M2i series), V10 (M3i series), V6 (M4i/M4x series) or V1 (M2p and M5i series). Please update the driver and the card firmware to these versions to use this feature. Trying to use this feature without the proper firmware version will issue a driver error.

On M2i and M3i cards, using the trigger counter information allows to determine how many Multiple Recording segments have been acquired and can perform a memory flush by issuing Force trigger commands to read out all data. This is helpful if the number of trigger events is not known at the start of the acquisition. In that case one will do the following steps:

- Program the maximum number of segments that one expects or use the FIFO mode with unlimited segments
- Set a timeout to be sure that there are no more trigger events acquired. Alternatively one can manually proceed as soon as it is clear from the application that all trigger events have been acquired
- Read out the number of acquired trigger segments
- Issue a number of Force Trigger commands to fill the complete memory (standard mode) or to transfer the last FIFO block that contains valid data segments
- Use the trigger counter value to split the acquired data into valid data with a real trigger event and invalid data with a force trigger event.

Main analog external trigger (Ext0)

The M5i series has one primary external trigger input consisting of an input stage with programmable either 3 kOhm or 50 Ohm input termination and one comparator that can be programmed in the range of ±5000 mV. Using a comparator offers a wide range of different logic levels for the available trigger modes that are supported like edge, level.

The external analog trigger can be easily combined with channel trigger or with the additional logic triggers via the multi-purpose I/O lines, when being programmed as an additional external trigger input. The programming of the masks is shown in the chapters above.

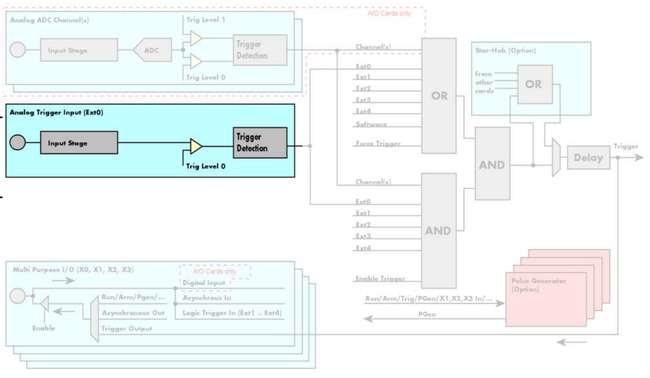


Image 58: trigger engine overview and external trigger

Trigger Mode

Please find the main external (analog) trigger input modes below. A detailed description of the modes follows in the next chapters..

Table 71: Spectrum API: external trigger mode registers and available settings therefore

Register		Value	Direction	Description
SPC_TRIG_EXT0_AVAILMODES		40500	read	Bitmask showing all available trigger modes for external 0 (Ext0) = main analog trigger input
SPC_TRIG_EXT0_MODE		40510	read/write	Defines the external trigger mode for the external SMA connector trigger input. The trigger need to be added to either OR or AND mask input to be activated.
	SPC_TM_NONE	00000000h	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 (crossing level 0 from below to above)	
	SPC_TM_NEG	00000002h	Trigger detection for negative edges (crossing level 0 from above to below)	
	SPC_TM_BOTH	00000004h	Trigger detection for positive and negative edges (any crossing of level 0)	
	SPC_TM_HIGH	00000008h	Trigger detection for HIGH levels (signal above level 0)	
	SPC_TM_LOW	00000010h	Trigger detection for LOW levels (signal below level 0)	

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: trigger or mask and setup for external trigger

Register		Value	Direction	Description
SPC_TRIG_ORMASK		40410	read/write	Defines the OR mask for the different trigger sources.
	SPC_TMASK_EXT0	2h	Enable primary external trigger input for the OR mask	

Trigger Input Termination

The external trigger input is a high impedance input with 3 kOhm termination against GND. It is possible to program a 50 Ohm termination by software to terminate fast trigger signals correctly. If you enable the termination, please make sure, that your trigger source is capable to deliver the needed current. Please check carefully whether the source is able to fulfill the trigger input specification given in the technical data section.

Register	Value	Direction	Description
SPC_TRIG_TERM	40110	read/write	A „1“ sets the 50 Ohm termination for external trigger signals. A „0“ sets the high impedance termination

Please note that the signal levels will drop by 50% if using the 50 Ohm termination and your source also has 50 Ohm output impedance (both terminators will then work as a 1:2 divider). In that case it will be necessary to reprogram the trigger levels to match the new signal levels. In case of problems receiving a trigger please check the signal level of your source while connected to the terminated input.

Trigger level

All of the external (analog) trigger modes listed above require a trigger level to be set (except SPC_TM_NONE of course). The meaning of the trigger levels is depending on the selected mode and can be found in the detailed trigger mode description that follows.

Trigger level for the external (analog) trigger is to be programmed in mV:

Table 73: Spectrum API: software registers for external trigger levels

Register	Value	Direction	Description	Range
SPC_TRIG_EXT_AVAIL0_MIN	42340	read	returns the minimum trigger level to be programmed in mV	
SPC_TRIG_EXT_AVAIL0_MAX	42341	read	returns the maximum trigger level to be programmed in mV	
SPC_TRIG_EXT_AVAIL0_STEP	42342	read	returns the step size of trigger level to be programmed in mV	
SPC_TRIG_EXT0_LEVEL0	42320	read/write	Trigger level 0 for external trigger Ext0	-5000 mV to +5000 mV

Detailed description of the external analog trigger modes

For all external analog trigger modes shown below, either the OR mask or the AND must contain the external trigger to activate the external input as trigger source:.

Table 74: Spectrum API: software registers to program external trigger

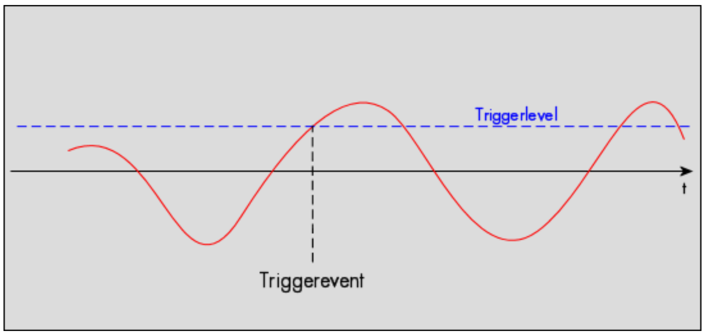
Register	Value	Direction	Description
SPC_TRIG_ORMASK	40410	read/write	Defines the events included within the trigger OR mask of the card.
SPC_TRIG_ANDMASK	40430	read/write	Defines the events included within the trigger AND mask of the card.
SPC_TMASK_EXT0	2h	Enables the main external (analog) trigger 0 for the mask.	

The following pages explain the available modes in detail.

Trigger on positive edge

The trigger input is continuously sampled with the selected sample rate. If the programmed trigger level is crossed by the trigger signal from lower values to higher values (rising edge) then the trigger event will be detected.

This edge triggered external trigger mode correspond to the trigger possibilities of usual oscilloscopes.

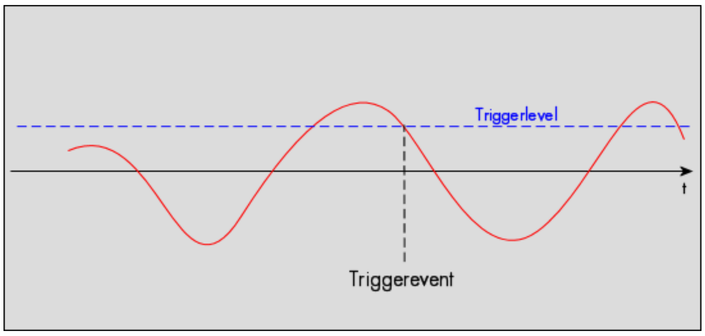


Register	Value	Direction	set to	Value
SPC_TRIG_EXT0_MODE	40510	read/write	SPC_TM_POS	1h
SPC_TRIG_EXT0_LEVEL0	42320	read/write	Set it to the desired trigger level in mV	mV

Trigger on negative edge

The trigger input is continuously sampled with the selected sample rate. If the programmed trigger level is crossed by the trigger signal from higher values to lower values (falling edge) then the trigger event will be detected.

This edge triggered external trigger mode correspond to the trigger possibilities of usual oscilloscopes.

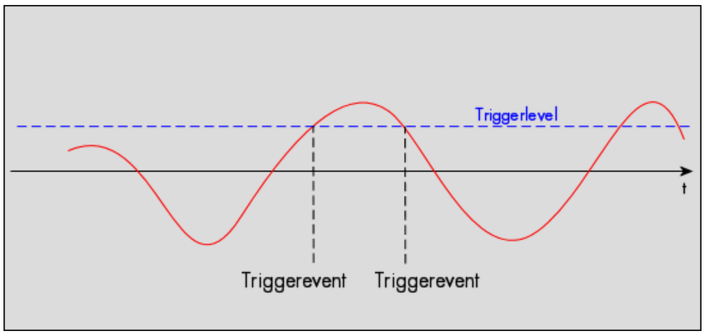


Register	Value	Direction	set to	Value
SPC_TRIG_EXT0_MODE	40510	read/write	SPC_TM_NEG	2h
SPC_TRIG_EXT0_LEVEL0	42320	read/write	Set it to the desired trigger level in mV	mV

Trigger on positive and negative edge

The trigger input is continuously sampled with the selected sample rate. If the programmed trigger level is crossed by the trigger signal (either rising or falling edge) the trigger event will be detected.

This edge triggered external trigger mode correspond to the trigger possibilities of usual oscilloscopes.

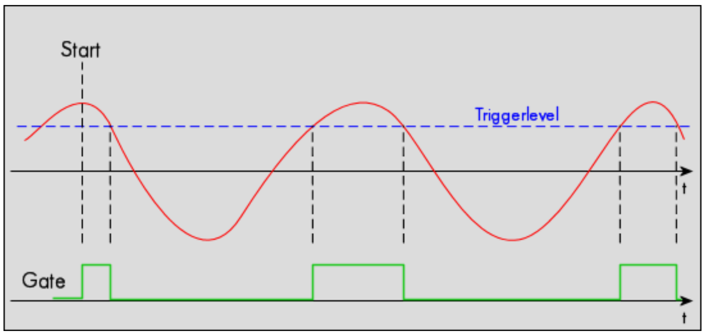


Register	Value	Direction	set to	Value
SPC_TRIG_EXT0_MODE	40510	read/write	SPC_TM_BOTH	4h
SPC_TRIG_EXT0_LEVEL0	42320	read/write	Set it to the desired trigger level in mV	mV

High level trigger

This trigger mode will generate an internal gate signal that can be useful in conjunction with a second trigger mode to gate that second 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 above the programmed 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 above the programmed trigger level.

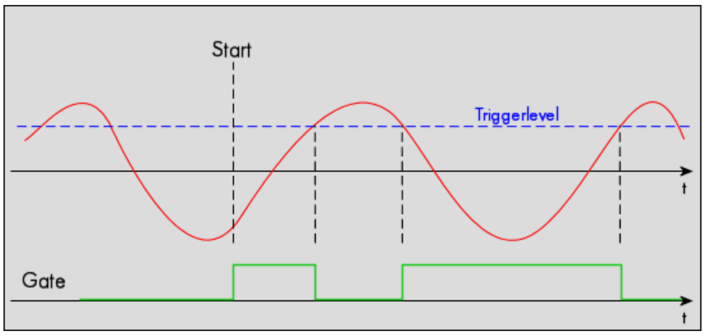


Register	Value	Direction	set to	Value
SPC_TRIG_EXT0_MODE	40510	read/write	SPC_TM_HIGH	00000008h
SPC_TRIG_EXT0_LEVEL0	42320	read/write	Set it to the upper trigger level in mV	mV

Low level trigger

This trigger mode will generate an internal gate signal that can be useful in conjunction with a second trigger mode to gate that second 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 above the programmed 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 below the programmed trigger level.



Register	Value	Direction	set to	Value
SPC_TRIG_EXT0_MODE	40510	read/write	SPC_TM_LOW	00000010h
SPC_TRIG_EXT0_LEVEL0	42320	read/write	Set it to the upper trigger level in mV	mV

External logic trigger (X0, X1, X2, X3)

The four multi purpose I/O lines of the M5i series can be set up as additional logic (TTL) triggers.

The external logic triggers can be easily combined with the external analog trigger as well as the channel trigger.The programming of the masks is shown in the chapters above.

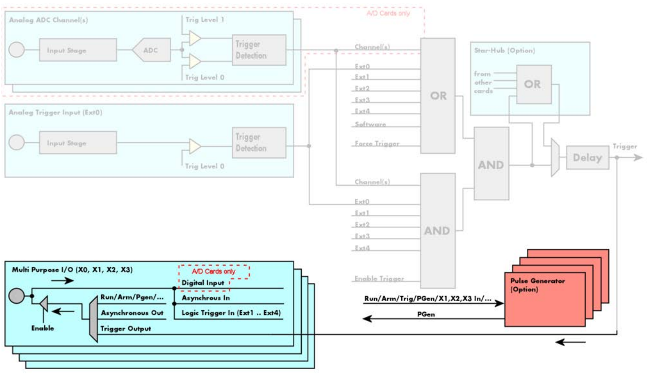


Image 59: trigger engine overview and multi purpose trigger

Trigger Mode

Please find the main external (analog) trigger input modes below. A detailed description of the modes follows in the next chapters..

Table 75: Spectrum API: external logic trigger registers and settings for them

Register	Value	Direction	Description
SPC_TRIG_EXT1_AVAILMODES	40501	read	Bitmask showing all available trigger modes for external 1 (X0) = logic trigger input
SPC_TRIG_EXT2_AVAILMODES	40502	read	Bitmask showing all available trigger modes for external 2 (X1) = logic trigger input
SPC_TRIG_EXT3_AVAILMODES	40514	read	Bitmask showing all available trigger modes for external 3 (X2) = logic trigger input
SPC_TRIG_EXT4_AVAILMODES	40516	read	Bitmask showing all available trigger modes for external 4 (X3) = logic trigger input
SPC_TRIG_EXT1_MODE	40511	read/write	Defines the external trigger mode for the external X0 connector trigger input. The trigger need to be added to either OR or AND mask input to be activated.
SPC_TRIG_EXT2_MODE	40512	read/write	Defines the external trigger mode for the external X1 connector trigger input. The trigger need to be added to either OR or AND mask input to be activated.
SPC_TRIG_EXT3_MODE	40513	read/write	Defines the external trigger mode for the external X2 connector trigger input. The trigger need to be added to either OR or AND mask input to be activated.
SPC_TRIG_EXT4_MODE	40519	read/write	Defines the external trigger mode for the external X3 connector trigger input. The trigger need to be added to either OR or AND mask input to be activated.
SPC_TM_NONE	00000000h	Channel is not used for trigger detection. This is as with the trigger masks another possibility for disabling channels.	
SPC_TM_POS	00000001h	Sets the trigger mode for external logic (TTL) trigger to detect positive edges.	
SPC_TM_NEG	00000002h	Sets the trigger mode for external logic (TTL) trigger to detect negative edges.	
SPC_TM_BOTH	00000004h	Sets the trigger mode for external logic (TTL) trigger to detect positive and negative edges	
SPC_TM_HIGH	00000008h	Sets the trigger mode for external logic (TTL) trigger to detect HIGH levels.	
SPC_TM_LOW	00000010h	Sets the trigger mode for external logic (TTL) trigger to detect LOW levels.	

For all external edge and level trigger modes, the OR mask must contain the corresponding input, as the following table shows:

Table 76: Spectrum API: trigger OR mask register an settings for external logic trigger

Register	Value	Direction	Description
SPC_TRIG_ORMASK	40410	read/write	Defines the OR mask for the different trigger sources.
SPC_TMASK_EXT1	4h	Enable logic trigger X0 input for the OR mask	
SPC_TMASK_EXT2	8h	Enable logic trigger X1 input for the OR mask	
SPC_TMASK_EXT3	10h	Enable logic trigger X2 input for the OR mask	
SPC_TMASK_EXT4	20h	Enable logic trigger X3 input for the OR mask	

Input Termination

The logic trigger inputs are high impedance input with a 10 kOhm termination against GND. It is possible to program a 50 Ohm termination by software to terminate fast trigger signals correctly. If you enable the termination, please make sure, that your trigger source is capable to deliver the needed current. Please check carefully whether the source is able to fulfill the trigger input specification given in the technical data section.

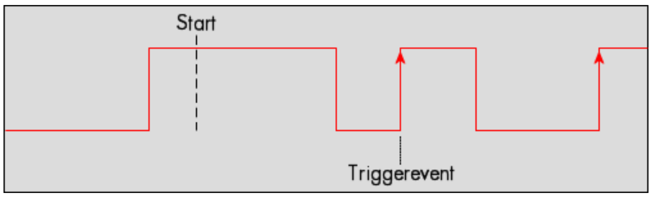
Register	Value	Direction	Description
SPC_X0_TERM	600100	read/write	A „1“ enables the 50 Ohm termination for X0. A „0“ sets the high impedance termination.
SPC_X1_TERM	600101	read/write	A „1“ enables the 50 Ohm termination for X0. A „0“ sets the high impedance termination.
SPC_X2_TERM	600102	read/write	A „1“ enables the 50 Ohm termination for X0. A „0“ sets the high impedance termination.
SPC_X3_TERM	600103	read/write	A „1“ enables the 50 Ohm termination for X0. A „0“ sets the high impedance termination.

When enabling the 50 Ohm termination on the multi-purpose inputs, make sure that the source can drive such loads while still maintaining the required Low and High levels, as specified in the technical data section.

Detailed description of the logic trigger modes

Positive (rising) edge TTL trigger

This mode is for detecting the rising edges of an external TTL signal. 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 actual recording/replay has finished and the board is armed and waiting for a trigger again.



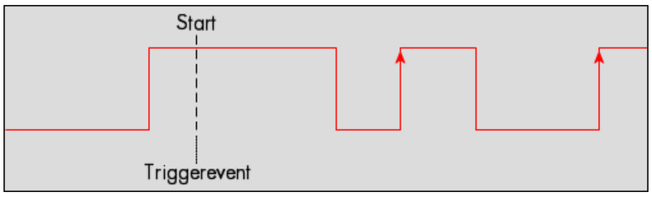
Register	Value	Direction	set to	Value
SPC_TRIG_EXT1_MODE SPC_TRIG_EXT2_MODE SPC_TRIG_EXT3_MODE SPC_TRIG_EXT4_MODE	40511 40512 40513 40519	read/write	SPC_TM_POS	1h

Example on how to set up the board for positive TTL trigger:

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_POS);// Set up ext. TTL trigger to detect positive edges
```

HIGH level TTL trigger

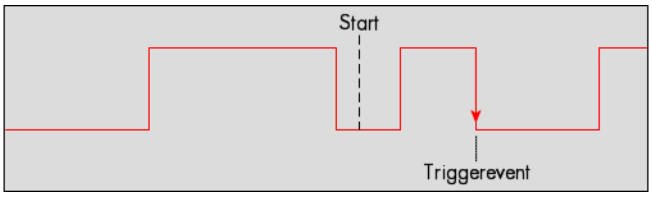
This mode is for detecting the HIGH levels of an external TTL signal. The board will trigger on the first HIGH level that is detected after starting the board. If this condition is fulfilled when the board is started, a trigger event will be detected.
The next trigger event will then be detected, if the actual recording/replay has finished and the board is armed and waiting for a trigger again.



Register	Value	Direction	set to	Value
SPC_TRIG_EXT1_MODE SPC_TRIG_EXT2_MODE SPC_TRIG_EXT3_MODE SPC_TRIG_EXT4_MODE	40511 40512 40513 40519	read/write	SPC_TM_HIGH	8h

Negative (falling) edge TTL trigger

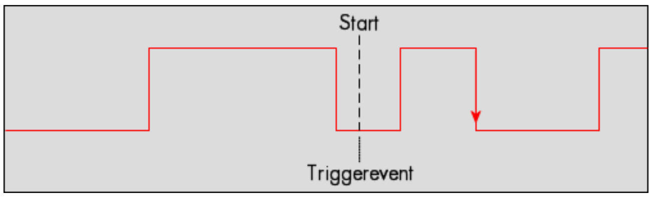
This mode is for detecting the falling edges of an external TTL signal. The board will trigger on the first falling edge that is detected after starting the board.
The next trigger event will then be detected, if the actual recording/replay has finished and the board is armed and waiting for a trigger again.



Register	Value	Direction	set to	Value
SPC_TRIG_EXT1_MODE SPC_TRIG_EXT2_MODE SPC_TRIG_EXT3_MODE SPC_TRIG_EXT4_MODE	40511 40512 40513 40519	read/write	SPC_TM_NEG	2h

LOW level TTL trigger

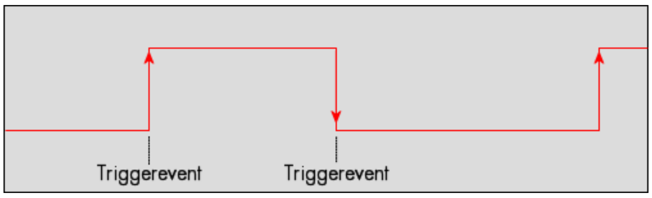
This mode is for detecting the LOW levels of an external TTL signal. The board will trigger on the first LOW level that is detected after starting the board. If this condition is fulfilled when the board is started, a trigger event will be detected.
The next trigger event will then be detected, if the actual recording/replay has finished and the board is armed and waiting for a trigger again.



Register	Value	Direction	set to	Value
SPC_TRIG_EXT1_MODE SPC_TRIG_EXT2_MODE SPC_TRIG_EXT3_MODE SPC_TRIG_EXT4_MODE	40511 40512 40513 40519	read/write	SPC_TM_LOW	10h

Positive (rising) and negative (falling) edges TTL trigger

This mode is for detecting the rising and falling edges of an external TTL signal. 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/replay has finished and the board is armed and waiting for a trigger again.



Register	Value	Direction	set to	Value
SPC_TRIG_EXT1_MODE SPC_TRIG_EXT2_MODE SPC_TRIG_EXT3_MODE SPC_TRIG_EXT4_MODE	40511 40512 40513 40519	read/write	SPC_TM_BOTH	4h

Channel Trigger

Overview of the channel trigger registers

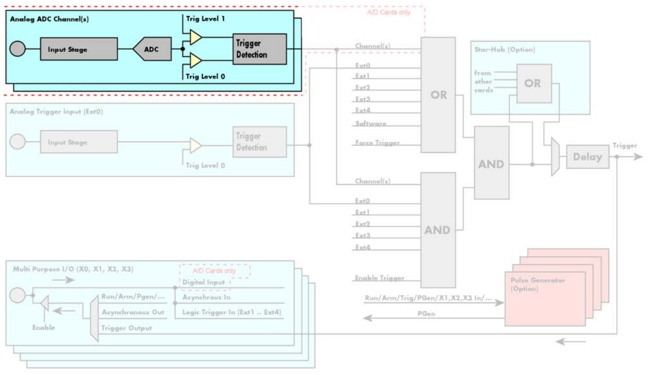


Image 60: trigger engine overview and channel trigger

The channel trigger modes are the most common modes, compared to external equipment like oscilloscopes. The huge variety of different channel trigger modes enable you to observe nearly any part of the analog signal. This chapter is about to explain the different modes in detail. To enable the channel trigger, you have to set the channel triggermode register accordingly. Therefore you have to choose, if you either want only one channel to be the trigger source, or if you want to combine two or more channels to a logical OR or a logical AND trigger.

For all channel trigger modes, the OR mask must contain the corresponding input channels (channel 0 taken as example here):.

Table 77: Spectrum API: channel trigger OR mask register

Register	Value	Direction	Description
SPC_TRIG_CH_ORMASK0	40460	read/write	Defines the OR mask for the channel trigger sources.
SPC_TMASK0_CH0	1h	Enables channel0 input for the channel OR mask	

The following table shows the according registers for the two general channel trigger modes. It lists the maximum of the available channel mode registers for your card's series. So it can be that you have less channels installed on your specific card and therefore have less valid channel mode registers. If you try to set a channel, that is not installed on your specific card, an error message will be returned.

Table 78: Spectrum API: channel trigger register and available settings for these

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, if available.
SPC_TRIG_CH0_MODE	40610	read/write	Sets the trigger mode for channel 0. Channel 0 must be enabled in the channel OR/AND mask.
SPC_TRIG_CH1_MODE	40611	read/write	Sets the trigger mode for channel 1. Channel 1 must be enabled in the channel OR/AND mask.
SPC_TM_NONE	00000000h	Channel is not used for trigger detection. This is as with the trigger masks another possibility for disabling channels.	
SPC_TM_POS	00000001h	Enables the trigger detection for positive edges	
SPC_TM_NEG	00000002h	Enables the trigger detection for negative edges	
SPC_TM_BOTH	00000004h	Enables the trigger detection for positive and negative edges	
SPC_TM_LOW	00000010h	Enables the trigger detection for LOW levels	
SPC_TM_HIGH	00000008h	Enables the trigger detection for HIGH levels	
SPC_TM_POS | SPC_TM_REARM	01000001h	Trigger detection for positive edges on level 0. Trigger is armed when crossing level 1 to avoid false trigger on noise	
SPC_TM_NEG | SPC_TM_REARM	01000002h	Trigger detection for negative edges on level 1. Trigger is armed when crossing level 0 to avoid false trigger on noise	
SPC_TM_WINENTER	00000020h	Enables the window trigger for entering signals	
SPC_TM_WINLEAVE	00000040h	Enables the window trigger for leaving signals	
SPC_TM_INWIN	00000080h	Enables the window trigger for inner signals	
SPC_TM_OUTSIDEWIN	00000100h	Enables the window trigger for outer signals	
SPC_TM_POS | SPC_TM_HYSTERESIS	20000001h	Enables the trigger detection for positive edges with hysteresis	
SPC_TM_NEG | SPC_TM_HYSTERESIS	20000002h	Enables the trigger detection for negative edges with hysteresis	
SPC_TM_POS | SPC_TM_REARM | SPC_TM_HYSTERESIS	21000001h	Trigger detection for positive edges with hysteresis on level 0. Trigger is armed when crossing level 1 to avoid false trigger on noise	
SPC_TM_NEG | SPC_TM_REARM | SPC_TM_HYSTERESIS	21000002h	Trigger detection for negative edges with hysteresis on level 1. Trigger is armed when crossing level 0 to avoid false trigger on noise	
SPC_TM_LOW | SPC_TM_HYSTERESIS	20000010h	Enables the trigger detection for LOW levels with hysteresis	
SPC_TM_HIGH | SPC_TM_HYSTERESIS	20000008h	Enables the trigger detection for HIGH levels with hysteresis	

If you want to set up a two channel board to detect only a positive edge on channel 0, you would have to setup the board like the following example. Both of the examples either for the single trigger source and the OR trigger mode do not include the necessary settings for the trigger levels. These settings are detailed described in the following paragraphs.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_NONE);     // disable software trigger
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ORMASK0, SPC_TMASK0_CH0); // Enable channel 0 in the OR mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE, SPC_TM_POS );      // Set triggermode of Ch 0 to positive edge
```

If you want to set up a two channel board to detect a trigger event on either a positive edge on channel 0 or a negative edge on channel 1 you would have to set up your board as the following example shows.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_NONE);   // disable software trigger
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ORMASK0, SPC_TMASK0_CH0 | SPC_TMASK0_CH1); // Enable Ch 0 & Ch 1
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE, SPC_TM_POS );    // Set triggermode of Ch 0 to positive edge
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH1_MODE, SPC_TM_NEG );    // Set triggermode of Ch 1 to negative edge
```

Channel trigger level

All of the channel trigger modes listed above require at least one trigger level to be set (except SPC_TM_NONE of course). Some modes like the window triggers require even two levels (upper and lower level) to be set.

After the data has been sampled, the upper N data bits are compared with the N bits of the trigger levels. The following table shows the level registers and the possible values they can be set to for your specific card.

As the trigger levels are compared to the digitized data, the trigger levels depend on the channels input range. For every input range available to your board there is a corresponding range of trigger levels. On the different input ranges the possible stepsize for the trigger levels differs as well as the maximum and minimum values. The table further below gives you the absolute trigger levels for your specific card series.

12 bit resolution for the trigger levels:

Table 79: Spectrum API: channel trigger level registers

Register	Value	Direction	Description	Range
SPC_TRIG_CH0_LEVEL0	42200	read/write	Trigger level 0 channel 0: main trigger level / upper level if 2 levels used	-2047 to +2047
SPC_TRIG_CH1_LEVEL0	42201	read/write	Trigger level 0 channel 1: main trigger level / upper level if 2 levels used	-2047 to +2047
SPC_TRIG_CH0_LEVEL1	42300	read/write	Trigger level 1 channel 0: auxiliary trigger level / lower level if 2 levels used	-2047 to +2047
SPC_TRIG_CH1_LEVEL1	42301	read/write	Trigger level 1 channel 1: auxiliary trigger level / lower level if 2 levels used	-2047 to +2047

12bit trigger level representation depending on selected input range

Table 80: Spectrum API: standard input ranges and representation of trigger level settings in voltage

	Input ranges					
Triggerlevel	**±200 mV**	**±500 mV**	**±1 V**	**±2 V**	**±5 V**	**±10 V**
2047	+199.902 mV	+499.756 mV	+999.512 mV	+1999.023 mV	+4999.559 mV	+9999.117 mV
2046	+199.805 mV	+499.512 mV	+999.023 mV	+1999.047 mV	+4998.117 mV	+9999.234 mV
...						
1024	+100.000 mV	+250.000 mV	+500.000 mV	+1000.000 mV	+2500.000 mV	+5000.000 mV
...						
2	+0.195 mV	+0.488 mV	+0.977 mV	+1.953 mV	+4.883 mV	+9.766 mV
1	+0.098 mV	+0.244 mV	+0.488 mV	+0.977 mV	+2.441 mV	+4.8835 mV
0	0 V	0 V	0 V	0 V	0 V	0 V
-1	-0.098 mV	-0.244 mV	-0.488 mV	-0.977 mV	-2.441 mV	-4.8835 mV
-2	-0.195 mV	-0.488 mV	-0.977 mV	-1.953 mV	-4.883 mV	-9.766 mV
...						
-1024	-100.000 mV	-250.000 mV	-500.000 mV	-1000.000 mV	-2500.000 mV	-5000.000 V
...						
-2046	-199.805 mV	-499.512 mV	-999.023 mV	-1999.047 mV	-4998.117 mV	-9999.234 mV
-2047	-199.902 mV	-499.756 mV	-999.512 mV	-1999.023 mV	-4999.559 mV	-9999.117 mV
Step size	**97.66 µV**	**244.14 µV**	**488.28 µV**	**976.56 µV**	**2.441 mV**	**4.883 mV**

The following example shows, how to set up a one channel board to trigger on channel 0 with rising edge. It is assumed, that the input range of channel 0 is set to the the ±200 mV range. The decimal value for SPC_TRIG_CH0_LEVEL0 corresponds then with 5.0 mV, which is the resulting trigger level.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_NONE);     // disable default software trigger
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE,   SPC_TM_POS);     // Setting up channel trig (rising edge)
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_LEVEL0, 51);             // Sets 12bit triggerlevel to 5.0 mV
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ORMASK0, SPC_TMASK0_CH0); // and enable it within the OR mask
```

Reading out the number of possible trigger levels

The Spectrum driver also contains a register that holds the value of the maximum possible different trigger levels considering the above mentioned exclusion of the most negative possible value. This is useful, as new drivers can also be used with older hardware versions, because you can check the trigger resolution during run time. The register is shown in the following table:

Table 81: Spectrum API: trigger level count register

Register	Value	Direction	Description
SPC_READTRGLVLCOUNT	2500	r	Contains the number of different possible trigger levels meaning ± of the value.

In case of a board that uses 8 bits for trigger detection the returned value would be 127, as either the zero and 127 positive and negative values are possible.The resulting trigger step width in mV can easily be calculated from the returned value. It is assumed that you know the actually selected input range.

$$\text{Trigger step width} = \frac{\text{Input Range}_{max}}{\text{Number of trigger levels + 1}}$$

To give you an example on how to use this formula we assume, that the ±1.0 V input range is selected and the board uses 8 bits for trigger detection. The result would be 7.81 mV, which is the step width for your type of board within the actually chosen input range.

$$\text{Trigger step width} = \frac{+1000\text{ mV}}{127 + 1}$$

Detailed description of the channel trigger modes

For all channel trigger modes, the OR mask must contain the corresponding input channels (channel 0 taken as example here):.

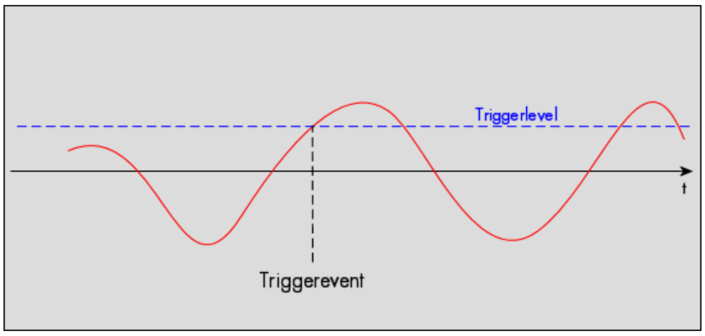
Table 82: Spectrum API: channel trigger OR mask register

Register	Value	Direction	Description
SPC_TRIG_CH_ORMASK0	40460	read/write	Defines the OR mask for the channel trigger sources.
SPC_TMASK0_CH0	1h	Enables channel0 input for the channel OR mask	

Channel trigger on positive edge

The analog input is continuously sampled with the selected sample rate. If the programmed trigger level is crossed by the channel's signal from lower values to higher values (rising edge) then the triggerevent will be detected.

These edge triggered channel trigger modes correspond to the trigger possibilities of usual oscilloscopes.

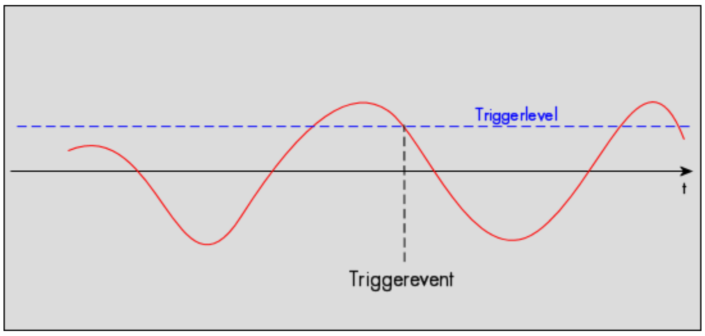


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS	1h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

Channel trigger on negative edge

The analog input is continuously sampled with the selected sample rate. If the programmed trigger level is crossed by the channel's signal from higher values to lower values (falling edge) then the triggerevent will be detected.

These edge triggered channel trigger modes correspond to the trigger possibilities of usual oscilloscopes.

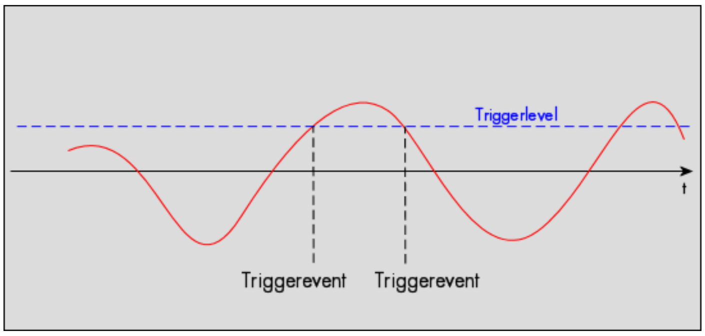


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG	2h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

Channel trigger on positive and negative edge

The analog input is continuously sampled with the selected sample rate. If the programmed trigger level is crossed by the channel's signal (either rising or falling edge) the triggerevent will be detected.

These edge triggered channel trigger modes correspond to the trigger possibilities of usual oscilloscopes.

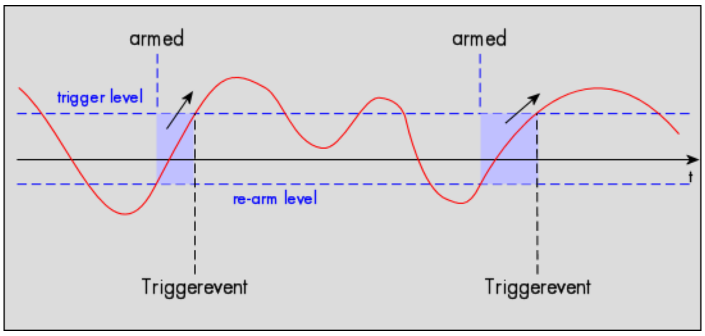


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_BOTH	4h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

Channel re-arm trigger on positive edge

The analog input is continuously sampled with the selected sample rate. If the programmed re-arm level is crossed from lower to higher values, the trigger engine is armed and waiting for trigger. If the programmed trigger level is crossed by the channel's signal from lower values to higher values (rising edge) then the triggerevent will be detected and the trigger engine will be disarmed. A new trigger event is only detected if the trigger engine is armed again.

The re-arm trigger modes can be used to prevent the board from triggering on wrong edges in noisy signals.

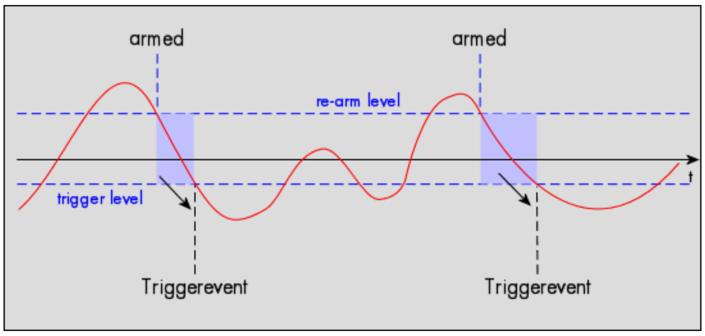


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS | SPC_TM_REARM	01000001h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the re-arm level relatively to the channel's input range	board dependant

Channel re-arm trigger on negative edge

The analog input is continuously sampled with the selected sample rate. If the programmed re-arm level is crossed from higher to lower values, the trigger engine is armed and waiting for trigger. If the programmed trigger level is crossed by the channel's signal from higher values to lower values (falling edge) then the triggerevent will be detected and the trigger engine will be disarmed. A new trigger event is only detected, if the trigger engine is armed again.

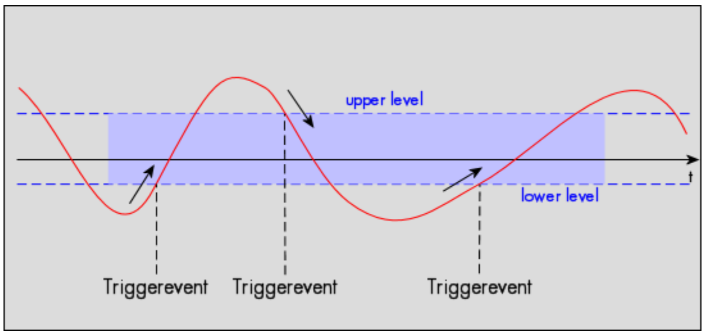
The re-arm trigger modes can be used to prevent the board from triggering on wrong edges in noisy signals.



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG | SPC_TM_REARM	01000002h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Defines the re-arm level relatively to the channels's input range	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

Channel window trigger for entering signals

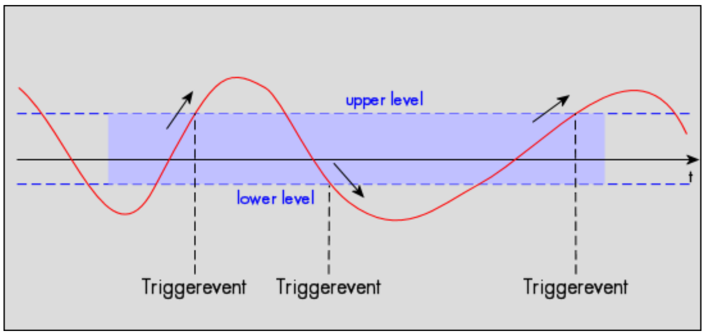
The analog input is continuously sampled with the selected sample rate. The upper and the lower level define a window. Every time the signal enters the window from the outside, a triggerevent will be detected.



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_WINENTER	00000020h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant

Channel window trigger for leaving signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower level define a window. Every time the signal leaves the window from the inside, a triggerevent will be detected.



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_WINLEAVE	00000040h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant

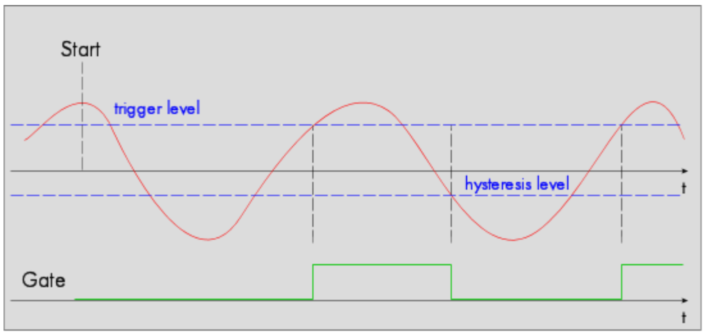
Channel hysteresis trigger on positive edge

This trigger mode will generate an internal gate signal that can be useful for masking a second trigger event generated by a different mode. The analog input is continuously sampled with the selected sample rate.

If the programmed trigger level is crossed by the channel's signal from lower values to higher values (rising edge) the gate starts.

When the signal crosses the programmed hysteresis level from higher values to lower values (falling edge) then the gate will stop.

As this mode is purely edge-triggered, the high level at the cards start time does not trigger the board.



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS | SPC_TM_HYSTERESIS	20000001h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the hysteresis level relatively to the channel's input range	board dependant

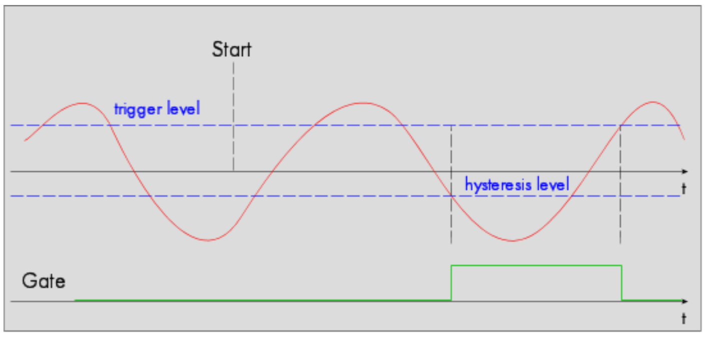
Channel hysteresis trigger on negative edge

This trigger mode will generate an internal gate signal that can be useful for masking a second trigger event generated by a different mode. The analog input is continuously sampled with the selected sample rate.

If the programmed trigger level is crossed by the channel's signal higher values to lower values (falling edge) the gate starts.

When the signal crosses the programmed hysteresis level from lower values to higher values (rising edge) then the gate will stop.

As this mode is purely edge-triggered, the low level at the cards start time does not trigger the board.

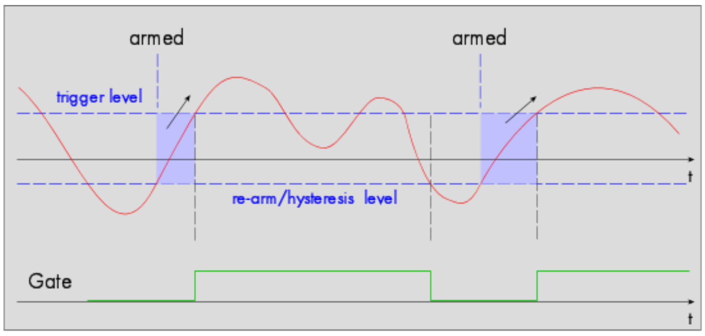


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG | SPC_TM_HYSTERESIS	20000002h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the hysteresis level relatively to the channel's input range	board dependant

Channel re-arm hysteresis trigger on positive edge

This trigger mode will generate an internal gate signal that can be useful for masking a second trigger event generated by a different mode. The analog input is continuously sampled with the selected sample rate.

If the programmed re-arm/hysteresis level is crossed from lower to higher values, the trigger engine is armed and waiting for trigger. If the programmed trigger level is crossed by the channel's signal from lower values to higher values (rising edge) then the gate starts and the trigger engine will be disarmed. If the programmed re-arm/hysteresis level is crossed by the channel's signal from higher values to lower values (falling edge) the gate stops.



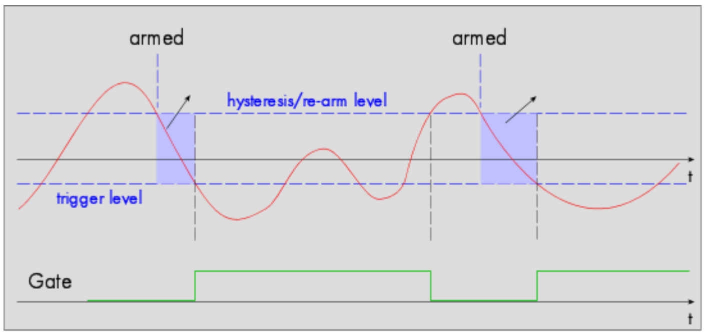
A new trigger event is only detected, if the trigger engine is armed again. The re-arm trigger modes can be used to prevent the board from triggering on wrong edges in noisy signals.

Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS | SPC_TM_REARM | SPC_TM_HYSTERESIS	21000001h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the re-arm and hysteresis level relatively to the channel's input range	board dependant

Channel re-arm hysteresis trigger on negative edge

This trigger mode will generate an internal gate signal that can be useful for masking a second trigger event generated by a different mode. The analog input is continuously sampled with the selected sample rate.

If the programmed re-arm/hysteresis level is crossed from higher to lower values, the trigger engine is armed and waiting for trigger. If the programmed trigger level is crossed by the channel's signal from higher values to lower values (falling edge) then the gate starts and the trigger engine will be disarmed. If the programmed re-arm/hysteresis level is crossed by the channel's signal from lower values to higher values (rising edge) the gate stops.



A new trigger event is only detected, if the trigger engine is armed again. The re-arm trigger modes can be used to prevent the board from triggering on wrong edges in noisy signals.

Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG | SPC_TM_REARM | SPC_TM_HYSTERESIS	21000002h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Defines the trigger level relatively to the channel's input range	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the re-arm and hysteresis level relatively to the channel's input range	board dependant

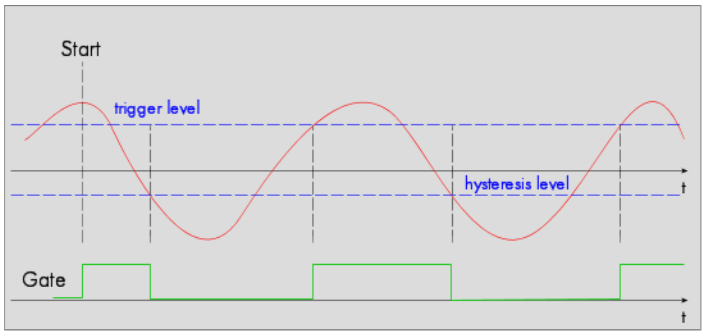
High level hysteresis trigger

This trigger mode will generate an internal gate signal that can be useful for masking a second trigger event generated by a different mode. The analog input is continuously sampled with the selected sample rate.

If the signal is equal or higher than the programmed trigger level the gate starts.

When the signal is lower than the programmed hysteresis level the gate will stop.

As this mode is level-triggered, the high level at the cards start time does trigger the board.



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_HIGH | SPC_TM_HYSTERESIS	20000008h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the hysteresis level relatively to the channel's input range	board dependant

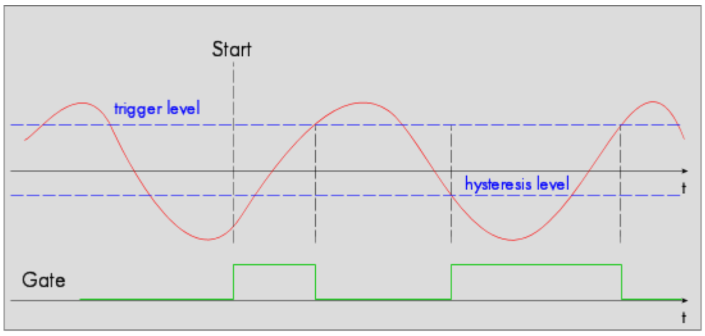
Low level hysteresis trigger

This trigger mode will generate an internal gate signal that can be useful for masking a second trigger event generated by a different mode. The analog input is continuously sampled with the selected sample rate.

If the signal is equal or lower than the programmed trigger level the gate starts.

When the signal is higher than the programmed hysteresis level the gate will stop.

As this mode is level-triggered, the high level at the cards start time does trigger the board.



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_LOW | SPC_TM_HYSTERESIS	20000010h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the hysteresis level relatively to the channel's input range	board dependant

Multi Purpose I/O Lines

On-board I/O lines (X0, X1, X2, X3)

The M5i series cards and the related digitizerNETBOX and generatorNETBOX products have four multi purpose I/O lines that can be used for a wide variety of functions to help the interconnection with external equipment. The functionality of these multi purpose I/O lines can be software programmed and each of these lines can either be used for input or output.

The multi purpose I/O lines may be used as status outputs such as trigger output or internal arm/run as well as for asynchronous I/O to control external equipment as well as additional digital input lines that are sampled synchronously with the analog data.

The multi purpose I/O lines are available on the front plate and labeled with X0 (line 0) through X3 (line 3). As default these lines are switched off.

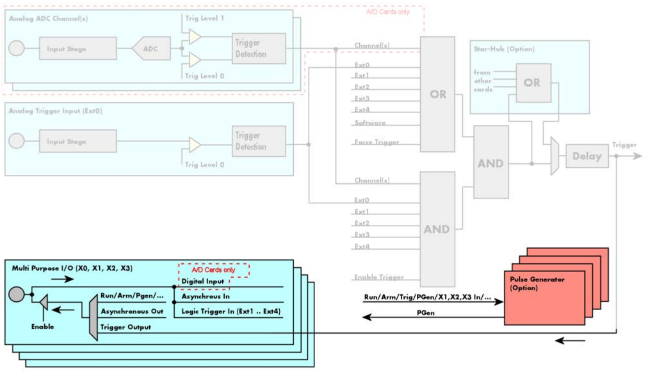


Image 61: trigger overview with multi-purpose lines marked

As default (power-on and after reset command) the I/O capable lines are switched off and hence are not actively driven. Hence the on-board 10k Ohm pull-up resistors are pulling these lines to logic HIGH. If a logic LOW is required, external lower-value (1k Ohm) pull-down resistors might be used.

Please be careful when programming these lines as an output whilst maybe still being connected with an external signal source, as that may damage components either on the external equipment or on the card itself.

Programming the behavior

Each multi purpose I/O line can be individually programmed. Please check the available modes by reading the SPCM_X0_AVAILMODES, SPCM_X1_AVAILMODES, SPCM_X2_AVAILMODES and SPCM_X3_AVAILMODES register first. The available modes may differ from card to card and may be enhanced with new driver/firmware versions to come.

Table 83: Spectrum API: multi-purpose I/O lines registers and available register settings

Register	Value	Direction	Description
SPCM_X0_AVAILMODES	47210	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X0)
SPCM_X1_AVAILMODES	47211	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X1)
SPCM_X2_AVAILMODES	47212	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X2)
SPCM_X3_AVAILMODES	47213	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X3)
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_X3_MODE	47203	read/write	Defines the mode for (X3). Only one mode selection is possible to be set at a time

SPCM_XMODE_DISABLE	00000000h	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_DIGIN	00000004h	A/D cards only: Connector is programmed for synchronous digital input. For each analog channel, one digital channel X0/X1/X2 is integrated into the ADC data stream. Depending on the ADC resolution of your card the resulting merged samples can have different formats. Please check the „Sample format" chapter for more details. Please note that automatic sign extension of analog data is ineffective as soon as one digital input line is activated and the software must properly mask out the digital bits.
SPCM_XMODE_DIGOUT	00000008h	D/A cards only: Connector is programmed for synchronous digital output. Digital channels can be „included" within the analog samples and synchronously replayed along. Requires additional MODE bits to be set along with this flag, as explained later on.
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 FIFO single 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_CONTOUTMARK	00002000h	Generator Cards only: outputs a HIGH pulse as continuous marker signal for continuous replay mode. The marker signal length is ½ of the programmed memory size.
SPCM_XMODE_PULSEGEN	00080000h	(optional): Connector reflects the output of the same index pulse generator (X0 output from pulse generator 0, X1 from pulse generator 1 etc.). For details on the pulse generator option please consult the "Pulse Generator (Option)" chapter.

Please note that a change to the SPCM_X0_MODE, SPCM_X1_MODE, SPCM_X2_MODE or SPCM_X3_MODE register 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.

Input Termination

The logic trigger inputs are high impedance input with a 10 kOhm termination against GND. It is possible to program a 50 Ohm termination by software to terminate fast trigger signals correctly. If you enable the termination, please make sure, that your trigger source is capable to deliver the needed current. Please check carefully whether the source is able to fulfill the trigger input specification given in the technical data section.

Register	Value	Direction	Description
SPC_X0_TERM	600100	read/write	A „1“ enables the 50 Ohm termination for X0. A „0“ sets the high impedance termination.
SPC_X1_TERM	600101	read/write	A „1“ enables the 50 Ohm termination for X0. A „0“ sets the high impedance termination.
SPC_X2_TERM	600102	read/write	A „1“ enables the 50 Ohm termination for X0. A „0“ sets the high impedance termination.
SPC_X3_TERM	600103	read/write	A „1“ enables the 50 Ohm termination for X0. A „0“ sets the high impedance termination.

When enabling the 50 Ohm termination on the multi-purpose inputs, make sure that the source can drive such loads while still maintaining the required Low and High levels, as specified in the technical data section.

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 84: Spectrum API: asynchronous I/O register settings of the multi-purpose I/O registers

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, connector X2 is linked to bit 2 while connector X3 is linked to bit 3 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 behavior of trigger output

As the driver of the M5i 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 85: Spectrum API: additional trigger output register for compatibility with older hardware

Register	Value	Direction	Description
SPC_TRIG_OUTPUT	40100	read/write	M2i style trigger output programming. Write a „1“ to enable: - X3 disabled (SPCM_X3_MODE = SPCM_XMODE_DISABLE) . - 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). Write a „0“ to disable all three outputs: - SPCM_X0_MODE .. SPCM_X3_MODE = SPCM_XMODE_DISABLE

The SPC_TRIG_OUTPUT register overrides the multi purpose I/O settings done by SPCM_X0_MODE, SPCM_X1_MODE, SPCM_X2_MODE and SPCM_X3_MODE and vice versa.
Do not use both methods together from within one program.

Synchronous digital inputs

The cards of the M5i series allow a very detailed setup on how to optionally record synchronous digital channels along with analog acquisition. The SPC_DIGMODEx register allows the setup separately for every analog channel. The table below shows the related registers and the values that correspond with the different possibilities. The mask and mode and mode values have to properly be combined. This is shown in the example below the tables:

Table 86: Spectrum API: digital input options registers

Register	Value	Direction	Description
SPC_DIGMODE0	47250	read/write	Set the digital input sources for channel 0.
SPC_DIGMODE1	47251	read/write	Set the digital input sources for channel 1.
SPC_DIGMODE2	47252	read/write	Set the digital input sources for channel 2.
SPC_DIGMODE3	47253	read/write	Set the digital input sources for channel 3.
SPCM_DIGMODE_OFF	00000000h		Disable acquisition of digital data for the masked analog bit (see masks below).
SPCM_DIGMODE_X0	21084000h		Enable acquisition of multi-purpose input X0 for the masked analog bit (see masks below).
SPCM_DIGMODE_X1	294A5000h		Enable acquisition of multi-purpose input X1 for the masked analog bit (see masks below).
SPCM_DIGMODE_X2	318C6000h		Enable acquisition of multi-purpose input X2 for the masked analog bit (see masks below).
SPCM_DIGMODE_X3	39CE7000h		Enable acquisition of multi-purpose input X3 for the masked analog bit (see masks below).
DIGMODEMASK_BIT15	F8000000h		Enable acquisition of a digital source (sources see above) into bit15 of the analog sample.
DIGMODEMASK_BIT14	07C00000h		Enable acquisition of a digital source (sources see above) into bit14 of the analog sample.
DIGMODEMASK_BIT13	003E0000h		Enable acquisition of a digital source (sources see above) into bit13 of the analog sample.
DIGMODEMASK_BIT12	0001F000h		Enable acquisition of a digital source (sources see above) into bit12 of the analog sample.

Each mask constant has to be bitwise AND combined with a source/mode constant, to define which digital source will be inserted at which position of the analog sample. The SPC_DIGMODEx register defines then, what analog channel this is applied to.

The driver will automatically scale the analog samples prior to inserting the digital channels to keep the channel at the maximum possible resolution.

Sample Format

The card is using 12 bit A/D samples, that are stored in twos complement in two 8 bit data byte. 12 bit resolution means that data is ranging from -2048…to…+2048. Data is stored in little-endian format, the upper 8 bit come first and the lower 8 bit second.:

Table 87: data sample format in standard mode and with digital inputs enable

Data bit	Standard Mode M5i.33xx 12 bit ADC resolution	Digital inputs enabled SPCM_XMODE_DIGIN M5i.33xx 12 bit ADC resolution
D15	ADX Bit 11 (sign extension)	Multi-Purpose XIO3
D14	ADX Bit 11 (sign extension)	Multi-Purpose XIO2
D13	ADX Bit 11 (sign extension)	Multi-Purpose XIO1
D12	ADX Bit 11 (sign extension)	Multi-Purpose XIO0
D11	ADx Bit 11 (MSB)	ADx Bit 11 (MSB)
D10	ADx Bit 10	ADx Bit 10
D9	ADx Bit 9	ADx Bit 9
D8	ADx Bit 8	ADx Bit 8
D7	ADx Bit 7	ADx Bit 7
D6	ADx Bit 6	ADx Bit 6
D5	ADx Bit 5	ADx Bit 5
D4	ADx Bit 4	ADx Bit 4
D3	ADx Bit 3	ADx Bit 3
D2	ADx Bit 2	ADx Bit 2
D1	ADx Bit 1	ADx Bit 1
D0	ADx Bit 0 (LSB)	ADx Bit 0 (LSB)

Please note that automatic sign extension of analog data is ineffective as soon as one digital input line is activated and the software must properly mask out all the digital bits from the samples.

The digital source has to be properly set to input direction to be a valid digital source. Also the analog channel into that the digital signals shall be routed to must be activated properly for acquisition as described in the „Channel Selection“ passage.

The following example shows how to enable a different number of digital channels (one and two respectively) on two different analog channels:

```
spcm_dwSetParam_i32 (hDrv, SPCM_X1_MODE, SPCM_XMODE_DIGIN);  // X1 set to synchronous input
spcm_dwSetParam_i32 (hDrv, SPCM_X2_MODE, SPCM_XMODE_DIGIN);  // X2 set to synchronous input
spcm_dwSetParam_i32 (hDrv, SPCM_X3_MODE, SPCM_XMODE_DIGIN);  // X3 set to synchronous input

// Enable acquisition of X1 input into bit15 of analog channel 0.
// Resulting Ch0 A/D samples will be 12bit samples with 3 bit sign extension and digital bit in MSB.
uint32 dwValue = (DIGMODEMASK_BIT15 & SPCM_DIGMODE_X1);
spcm_dwSetParam_i32 (hDrv, SPC_DIGMODE0, dwValue);

// Enable acquisition of X2 input into bit15 and X3 input into bit14 of analog channel 1.
// Resulting Ch1 A/D samples will be 14bit.
dwValue = (DIGMODEMASK_BIT15 & SPCM_DIGMODE_X2) | (DIGMODEMASK_BIT14 & SPCM_DIGMODE_X3);
spcm_dwSetParam_i32 (hDrv, SPC_DIGMODE1, dwValue);
```

The following example shows how to enable all four digital channels provided via the multi-purpose lines X0, X1, X2 and X3 to one channel:

```
spcm_dwSetParam_i32 (hDrv, SPCM_X0_MODE, SPCM_XMODE_DIGIN);  // X0 set to synchronous input
spcm_dwSetParam_i32 (hDrv, SPCM_X1_MODE, SPCM_XMODE_DIGIN);  // X1 set to synchronous input
spcm_dwSetParam_i32 (hDrv, SPCM_X2_MODE, SPCM_XMODE_DIGIN);  // X2 set to synchronous input
spcm_dwSetParam_i32 (hDrv, SPCM_X3_MODE, SPCM_XMODE_DIGIN);  // X3 set to synchronous input

// define and clear a temporary variable
uint32 dwValue = 0;

// add four sources (X0, X1, X2 and X3) at three different positions (bit15, bit14 and bit13)
dwValue |= (DIGMODEMASK_BIT15 & SPCM_DIGMODE_X3);
dwValue |= (DIGMODEMASK_BIT14 & SPCM_DIGMODE_X2);
dwValue |= (DIGMODEMASK_BIT13 & SPCM_DIGMODE_X1);
dwValue |= (DIGMODEMASK_BIT12 & SPCM_DIGMODE_X0);

// and write value to channel 0 digmode register. Resulting Ch0 A/D samples will be 13bit.
spcm_dwSetParam_i32 (hDrv, SPC_DIGMODE0, dwValue);
```

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.

The following table shows the register for defining the structure of the segments to be recorded with each trigger event.

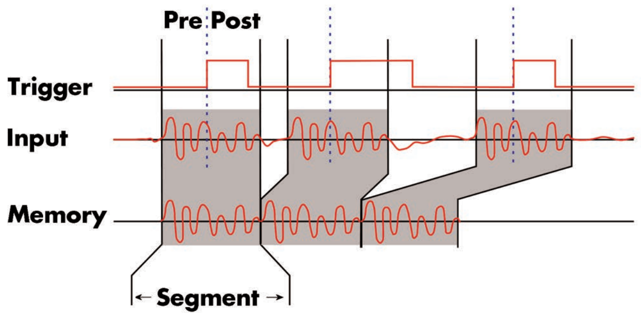


Image 62: Drawing of Multiple Recording acquisition

Table 88: 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: [pretrigger] = [segment size] - [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 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 89: 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 90: 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 91: 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 92: 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. Please keep in mind that each sample needs 2 bytes of memory to be stored. 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 93: Spectrum API: Limits of pre trigger, post trigger and memory size

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
1 Ch	Standard Single	64	Mem	32	32	Mem - 32	32	32	256G - 32	32		not used			not used	
						(defined by mem and post)										
	Standard Multi	64	Mem	32	32	32k	32	32	Mem-32	32	64	Mem	32		not used	
						(defined by segment and post)			(Limited by max pretrigger)							
	FIFO Single		not used		32	32k	32		not used		64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi		not used		32	32k	32	32	256G-32	32	64	pre+post	32	0 (∞)	4G - 1	1
						(defined by segment and post)			(Limited by max pretrigger)							
2 Ch	Standard Single	64	Mem/2	32	32	Mem/2 - 32	32	32	256G - 32	32		not used			not used	
						(defined by mem and post)										
	Standard Multi	64	Mem/2	32	32	16k	32	32	Mem/2-32	32	64	Mem/2	32		not used	
						(defined by segment and post)			(Limited by max pretrigger)							
	FIFO Single		not used		32	16k	32		not used		64	8G - 32	32	0 (∞)	4G - 1	1
	FIFO Multi		not used		32	16k	32	32	256G-32	32	64	pre+post	32	0 (∞)	4G - 1	1
						(defined by segment and post)			(Limited by max pretrigger)							

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:

Table 94: Spectrum-API: maximum memory sizes for different memory upgrade options

	Installed Memory	
	2 GSample (4 GByte)	**8 GSample (16 GByte)** **(Option: M5i.xxx-MEM8GS)**
Mem	2 GSample	8 GSample
Mem / 2	1 GSample	4 GSample

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.

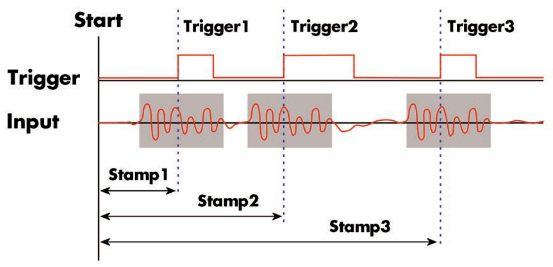


Image 63: drawing of Multiple Recording Acquisition with Timestamps

Trigger Modes

When using Multiple Recording all of the card's trigger modes can be used including the 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 8 bit Storage (Low-Resolution)

Overview

Starting with firmware version V5, the cards of the M5i.33xx series allow to optionally reduce the resolution of the A/D samples from their native 12 bit down to 8 bit resolution, such that each sample will only occupy one byte in memory instead of the standard two bytes required. This does not only doubles the size of the installed on-board memory, but also reduces the required bandwidth over the PCIe bus as well as to the storage devices, such as SSD or HDD.

Available acquisition modes

The following modes are compatible with the data conversion modes of the M5i.33xx series cards and hence can be used to acquire data in a reduced sample resolution:

Table 95: Spectrum API: 8 bit storage mode acquisition mode registers

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_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.

Please note the different limits of the memory settings (pre trigger, post trigger etc.) below compared to using the native card resolution.

Enabling hardware data conversion

The data conversion modes allow the conversion of acquired sample data in on the fly within the firmware from the card's native resolution of 12bit down to 8bit and the proper one should be chosen, depending on the cards original or native resolution:

Table 96: Spectrum API: data conversion registers and register settings

Register		Value	Direction	Description
SPC_DATACONVERSION		201400	read/write	Defines the data conversion mode.
SPC_AVAILDATACONVERSION		201401	read	Read out the available data conversion modes.
	SPCM_DC_NONE	0h	The original data format will be used and no hardware data conversion will be done.	
	SPCM_DC_12BIT_TO_8BIT	200h	12 bit input data is assumed and the resulting samples will be 8bit.	

Sample format

The hardware data conversion shifts the original data words down by either six bits or eight bits, no matter what their content is or what channel they belong to. In case that any digital channels are included included in the original samples, these might also be shifted down, depending on their original location in the samples:

Table 97: Spectrum API: sample format for different cards with data conversion mode activated

Data bit	**Data Conversion disabled** **M5i.33xx** **12 bit ADC resolution**	**Data Conversion enabled** **SPCM_DC_12BIT_TO_8BIT** **M5i.33xx** **reduced to 8 bit sample resolution**
D15	ADx Bit 11 (sign extension)	not used
D14	ADx Bit 11 (sign extension)	
D13	ADx Bit 11 (sign extension)	
D12	ADx Bit 11 (sign extension)	
D11	ADx Bit 11 (MSB)	
D10	ADx Bit 10	
D9	ADx Bit 9	
D8	ADx Bit 8	
D7	ADx Bit 7	D11 (MSB)
D6	ADx Bit 6	D10
D5	ADx Bit 5	D9
D4	ADx Bit 4	D8
D3	ADx Bit 3	D7
D2	ADx Bit 2	D6
D1	ADx Bit 1	D5
D0	ADx Bit 0 (LSB)	D4 (LSB)

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. Please keep in mind that each sample with enabled data conversion only needs 1 bytes of memory to be stored. 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: Spectrum API: Limits of pre trigger, post trigger and memory size when using 8 bit mode

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
1 Ch	Standard Single	128	Mem	64	64	Mem - 64	64	64	256G - 64	64		not used			not used	
						(defined by mem and post)										
	Standard Multi	128	Mem	64	64	32k	64	64	Mem-64	64	128	Mem	64		not used	
						(defined by segment and post)			(Limited by max pretrigger)							
	FIFO Single		not used		64	32k	64		not used		128	8G - 64	64	0 (∞)	4G - 1	1
	FIFO Multi		not used		64	32k	64	64	256G-64	64	128	pre+post	64	0 (∞)	4G - 1	1
						(defined by segment and post)			(Limited by max pretrigger)							
2 Ch	Standard Single	128	Mem/2	64	64	Mem/2 - 64	64	64	256G - 32	64		not used			not used	
						(defined by mem and post)										
	Standard Multi	128	Mem/2	64	64	16k	64	64	Mem/2-64	64	128	Mem/2	64		not used	
						(defined by segment and post)			(Limited by max pretrigger)							
	FIFO Single		not used		64	16k	64		not used		128	8G - 64	64	0 (∞)	4G - 1	1
	FIFO Multi		not used		64	16k	64	64	256G-64	64	128	pre+post	64	0 (∞)	4G - 1	1
						(defined by segment and post)			(Limited by max pretrigger)							

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:

Table 99: Spectrum-API: maximum memory sizes for different memory upgrade options when using 8 bit mode

	Installed Memory	
	4 GSample (4 GByte)	**16 GSample (16 GByte)** **(Option: M5i.xxx-MEM8GS)**
Mem	4 GSample	16 GSample
Mem / 2	2 GSample	8 GSample

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.

Converting reduced ADC samples to voltage values

When converting the reduced samples into voltage values the same principles and formulas apply as for the native 12 bit samples, as described earlier in this manual. However the instead of reading out the native ADC resolution from the driver, the reduced 8 bit resolution must be used instead.

Now that the board uses 8 bit samples that provides the full ADC code (without reserving any bits) the new value for ADC_{max} would be 128. The the peak value for a ±1.0 V input range would be 1.0 V (or 1000 mv).

$$V_{In} = ADC_{Code} \times \frac{InputRange_{peak}}{ADC_{max}}$$

A returned reduced sample value of for example +49 (decimal, two's complement, signed representation) would then convert to:

$$V_{in} = 49 \times \frac{1000 \text{ mV}}{128} = 382.81 \text{ mV}$$

A returned sample value of for example -55 (decimal) would then convert to:

$$V_{in} = -55 \times \frac{1000 \text{ mV}}{128} = -429.69 \text{ mV}$$

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 100: 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_REFCLOCKxxx 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_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
SPC_TSXIOACQ_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-timebase) 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).

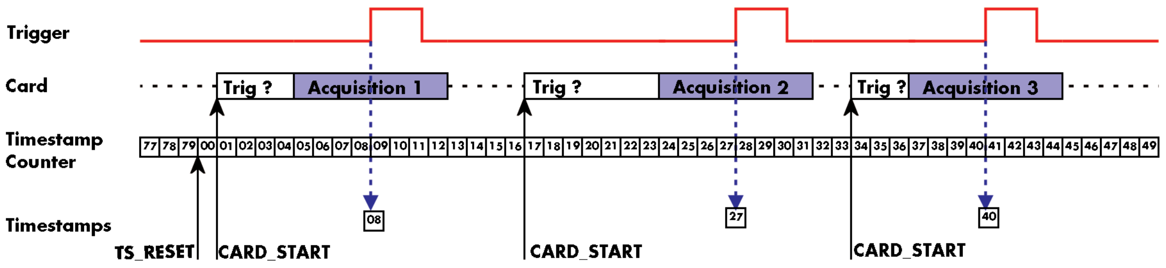


Image 64: drawing of timestamp acquisition in standard mode in relation to card start and trigger detection

The following table shows the valid values that can be written to the timestamp command register for this mode:

Table 101: 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).

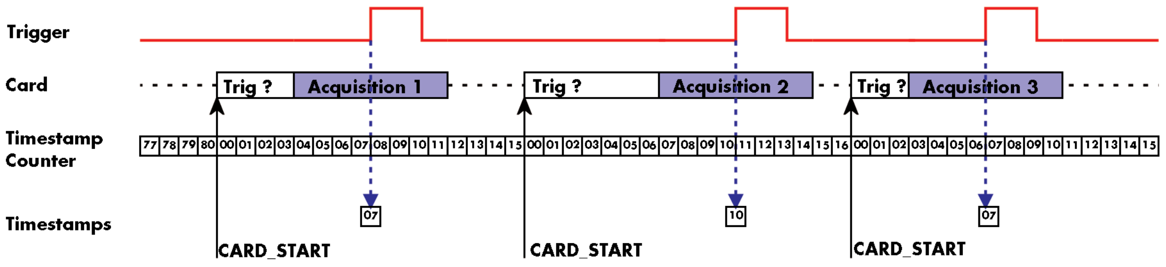


Image 65: drawing of timestamp acquisition in start-reset mode in relation to card start and trigger detection

The following table shows the valid values that can be written to the timestamp command register.

Table 102: 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.

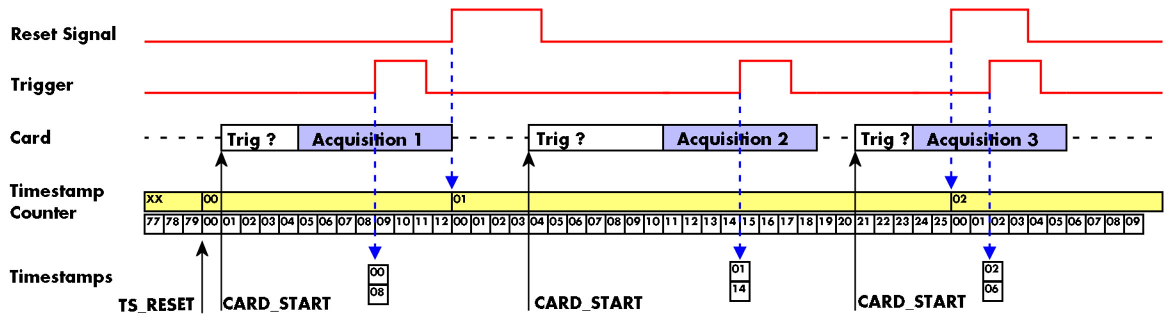


Image 66: drawing of timestamp acquisition in refclock mode in relation to card start and trigger detection

The following table shows the valid values that can be written to the timestamp command register for this mode:

Table 103: 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_TIME-STAMP_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 sam-pling 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.

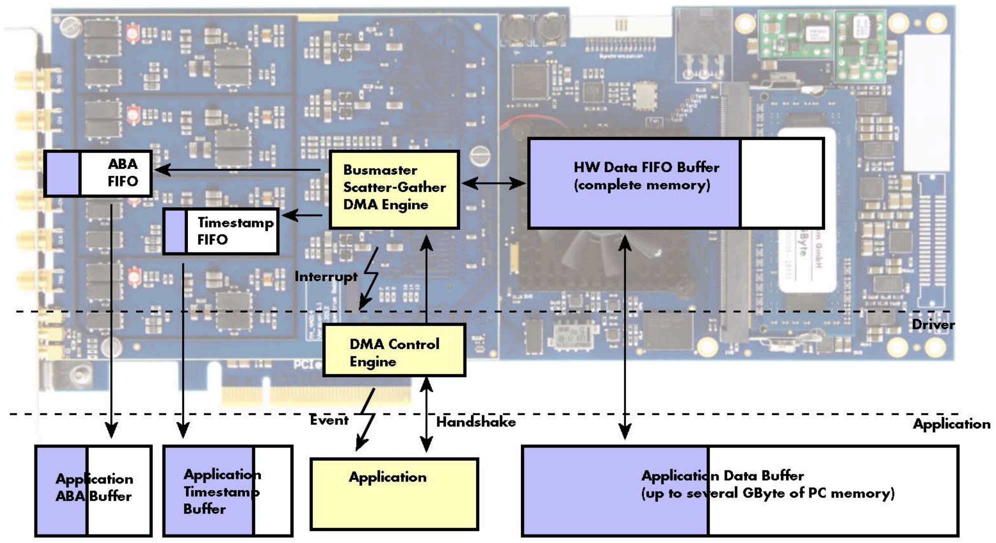

ABA FIFO
Busmaster Scatter-Gather DMA Engine
HW Data FIFO Buffer (complete memory)
Timestamp FIFO
Interrupt
Driver
DMA Control Engine
Event
Handshake
Application
Application ABA Buffer
Application Timestamp Buffer
Application
Application Data Buffer (up to several GByte of PC memory)

Image 67: 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 104: 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 105: Spectrum APUI: 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 106: 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", lBytesPos);

        // 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[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
```

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, &lBytePos);

    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 ((lBytePos + lAvailBytes) >= lBufSizeInBytes)
            lAvailBytes = lBufSizeInBytes - lBytePos;

        // 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 107: timestamp data format depending on the selected timestamp acquisition mode

Timestamp Mode	**16th byte**	**...**	**11th byte**	**10th byte**	**9th byte**	**8th byte**	**7th byte**	**6th byte**	**5th byte**	**4th byte**	**3rd byte**	**2nd byte**	**1st 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 an external. 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 the digital inputs (X3, X2, X1, X0). The following table shows the resulting 128 bit timestamps.

Table 108: extended timestamp data format depending on the selected timestamp acquisition mode

Timestamp Mode	16th byte	...	15h byte	14th byte	...	9th byte	8th byte	7th byte	6th byte	5th byte	4th byte	3rd byte	2nd byte	1st 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 48bit wide data, depending on the selected timestamp data format:

Table 109: timestamp extended data word format depending on the selected acquisition features

Timestamp Data Format	Bit 47	...	Bit 32	Bit 31	...	Bit 28	Bit 27	...	Bit 16	Bit 15	...	Bit 13	Bit 12	Bit 11	Bit 10	...	Bit 0
no special data format is set	0h																
SPC_TSXIOACQ_ENABLE	0h			0h						X3 .. X0				0h			
SPC_TSFEAT_TRGSRC	0h			Trigger source bitmask (X3, X2, X1, X0) (see table below)			0h								Trigger source bit-mask (Ch0 .. Force) (see table below)		
SPC_TSXIOACQ_ENABLE | SPC_TS-FEAT_TRGSRC	0h			Trigger source bitmask (X3, X2, X1, X0) (see table below)			0h			X3 .. X0				0h	Trigger source bit-mask (Ch0 .. Force) (see table below)		

The trigger sources are encoded as follows:

SPC_TRGSRC_MASK_CH0	1h	Set when a trigger event occurring on channel 0 was leading to final trigger event.
SPC_TRGSRC_MASK_CH1	2h	Set when a trigger event occurring on channel 1 was leading to final trigger event.
SPC_TRGSRC_MASK_CH2	4h	Set when a trigger event occurring on channel 2 was leading to final trigger event.
SPC_TRGSRC_MASK_CH3	8h	Set when a trigger event occurring on channel 3 was leading to final trigger event.
SPC_TRGSRC_MASK_EXT0	100h	Set when a trigger event occurring on external trigger(Ext0) was leading to final trigger event.
SPC_TRGSRC_MASK_FORCE	400h	Set when a trigger event occurring by using the force trigger command is leading to final trigger event.
SPC_TRGSRC_MASK_X0	10000000h	Set when a trigger event occurring on TTL trigger(X1) is leading to final trigger event.
SPC_TRGSRC_MASK_X1	20000000h	Set when a trigger event occurring on TTL trigger(X1) is leading to final trigger event.
SPC_TRGSRC_MASK_X2	40000000h	Set when a trigger event occurring on TTL trigger(X2) is leading to final trigger event.
SPC_TRGSRC_MASK_X3	80000000h	Set when a trigger event occurring on TTL trigger(X3) is leading to final trigger event.

Selecting the timestamp data format

Table 110: Spectrum API: timestamp command register and settings for different timestamp data formats

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 X1...X19 inputs with every stored timestamp in the upper 64 bit.	
	SPC_TSFEAT_TRGSRC	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 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.

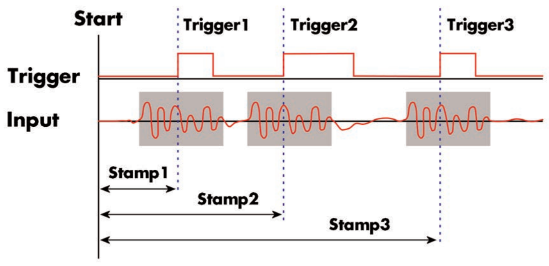


Image 68: drawing of Multiple Recording Acquisition with Timestamps

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 acquistion. 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);
    }
```

Pulse Generator (Firmware Option)

General Information

The pulse generator module provides a versatile timing synchronization interface between the acquisition/replay functionality of the card and external equipment.

The module consists of four pulse generators, where each generator allows for (in)dependent generation of individual pulses, pulse trains or a continuous stream of pulses that can be output on a Multi-Purpose I/O Line, greatly enhancing the versatility of the XIO lines.

The versatile trigger capabilities allow for external or internal triggering. Moreover, the pulse generators can trigger each other, hence allowing for cascading of up to four pulse repetition time scales.

The outputs of the pulse generators are intrinsically synchronized to the card acquisition/replay functionality and its sampling clock, hence allowing for reproducible enabling or switching of external signals (e.g., for signal actuating). Other use cases might be pulse broadening, pulse delaying, or just pulse generation.

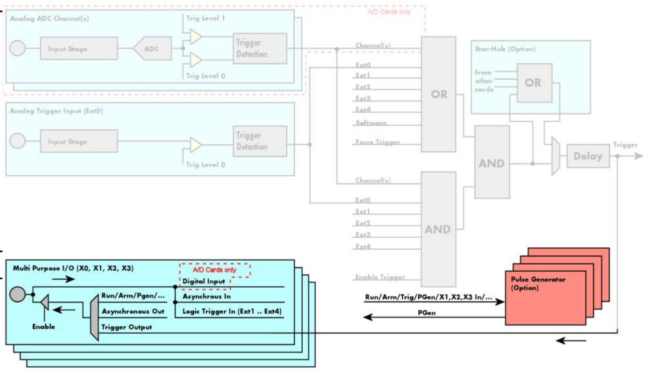


Image 69: overview block diagram of multi-purpose I/O lines and pulse generators

The generation of the pulse trains and timing signals is performed inside the FPGA of the card and is working in parallel to any other functionality of the card (such as data acquisition or replay), and hence not reducing the performance.

Feature Overview

- Four pulse generators are available
- Single-shot, multiple repetitions or continuous/infinite repetition of pulses
- Individual control of pulse length/duty cycle
- External or internal triggering/starting individually for each pulse generator
- Individual trigger delay per pulse generator allowing for phase shifting
- Internal cascading of pulse generators possible allowing up to four repetition time scales.

The “standard” modes of the multi purpose I/O lines are still available, as described in the “Multi Purpose I/O Lines” section. This chapter focuses on the additional functionality, available with the pulse generator firmware option installed.

The multi purpose I/O lines are available on the front plate and labelled with X0 (line 0), X1 (line 1), X2 (line 2) and X3 (line 3). As default these lines are switched off.

As default (power-on and after reset command) the I/O capable lines are switched off and hence are not actively driven. Hence the on-board 10k Ohm pull-up resistors are pulling these lines to logic HIGH. If a logic LOW is required, external lower-value (1k Ohm) pull-down resistors might be used.

Please be careful when programming these lines as an output whilst maybe still being connected with an external signal source, as that may damage components either on the external equipment or on the card itself.

Principle of Operation

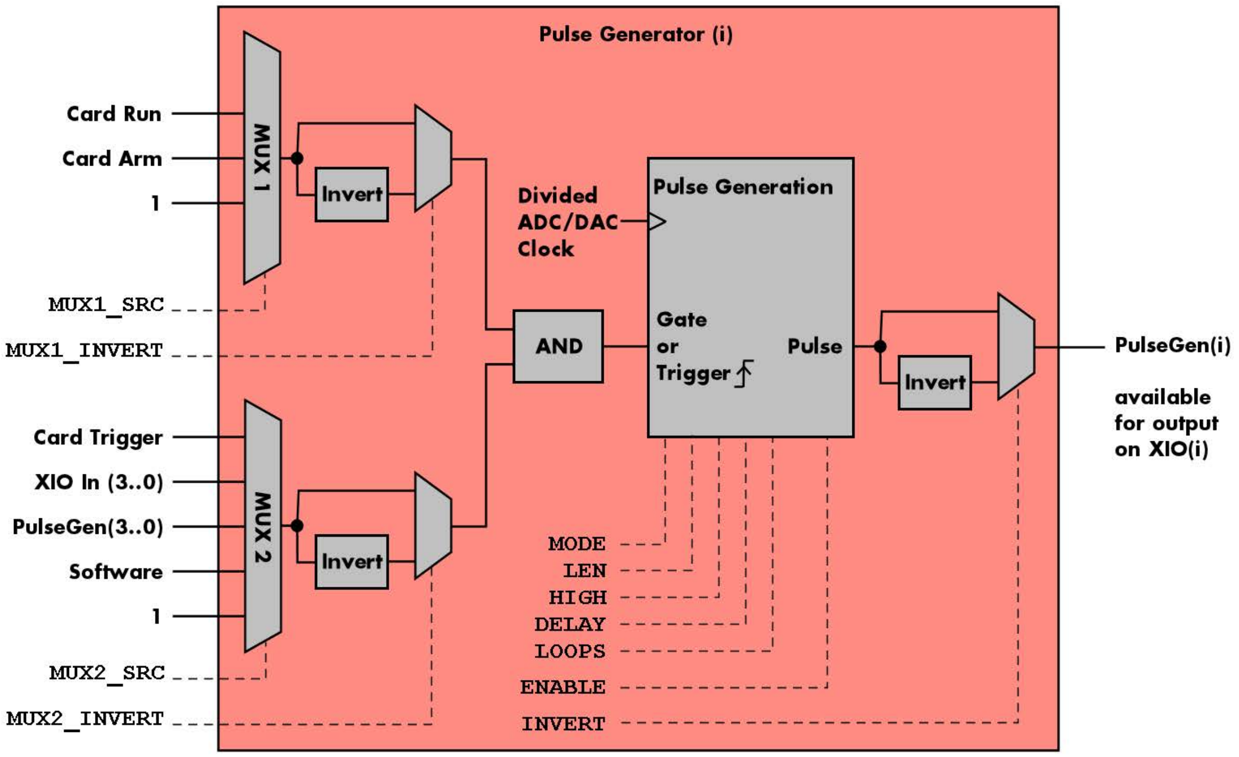


Image 70: overview block diagram of the pulse generator

All of the four available pulse generator units are identical in their feature set and individually programmable.

As shown above, each unit consists of:

- A dedicated trigger setup consisting of two multiplexers MUX1 and MUX2 combining various signals
- A programmable inverter on the output of each multiplexer
- A static logic AND gate combining the outputs of both multiplexers to form a trigger/gate for the pulse generating unit
- The pulse generating unit itself with its trigger signal driven by the AND gate
- A final programmable output inverter

The pulse generator unit is clocked with an FPGA internal clock, which is a divided version derived from the acquisition or generation sampling rate. Since the division ratio is depending on the used card type, the number of active channels and the sampling rate, an dedicated read only register allows to read out the frequency value by the following register:

Table 111: Spectrum API: pulse generator clock frequency read register

Register	Value	Direction	Description
SPC_XIO_PULSEGEN_CLOCK	602000	read	Returns the clock driving the pulse generator in Hz.

The following short excerpt shows which parameters need to be defined first and how to read out the clock rate at which the pulse generator units then are clocked:

```
...
// first set up the parameters, that influence the pulse generator’s clock rate
spcm_dwSetParam_i32 (hCard, SPC_CHENABLE, CHANNEL0);  // channel enable
spcm_dwSetParam_i64 (hCard, SPC_SAMPLERATE, MEGA(1)); // desired acquisition/generation sampling rate
...
// afterwards read out the divided clock rate, clocking the pulse generator units
int64 llPulseGenClock_Hz = 0;
spcm_dwGetParam_i64 (hCard, SPC_XIO_PULSEGEN_CLOCK, &llPulseGenClock_Hz);
```

See the end of this chapter for a more complete example setup of a pulse generator unit.

Changing the card settings while pulse generators are active will cause a stop and restart of the pulse generators automatically issued by the driver to the pulse generators.

Setting up the Pulse Generator

Enabling, disabling and resetting a pulse generator

Each pulse generator unit can be enabled and disabled separately:

Table 112: Spectrum API: pulse generator enable registers

Register	Value	Direction	Description
SPC_XIO_PULSEGEN_ENABLE	601500	read/write	Bitmask to enable any combination of the four different pulse generators.
SPCM_PULSEGEN_ENABLE0	1h		Enable pulse generator 0. When disabled, the output (prior to the output inverter) is set to logic LOW.
SPCM_PULSEGEN_ENABLE1	2h		Enable pulse generator 1. When disabled, the output (prior to the output inverter) is set to logic LOW.
SPCM_PULSEGEN_ENABLE2	4h		Enable pulse generator 2. When disabled, the output (prior to the output inverter) is set to logic LOW.
SPCM_PULSEGEN_ENABLE3	8h		Enable pulse generator 3. When disabled, the output (prior to the output inverter) is set to logic LOW.

Disabling a unit will act as a reset dedicated to this single unit. A disabled pulse generator will output a logic LOW prior to the programmable output inverter, hence with an active output inverter the final output of a disabled pulse generator will be logically HIGH.

Defining the basic pulse parameters

The two basic properties for generating a (repetitive) pulsed output is to define the length (or period) and define how much of the waveform should the output be HIGH:

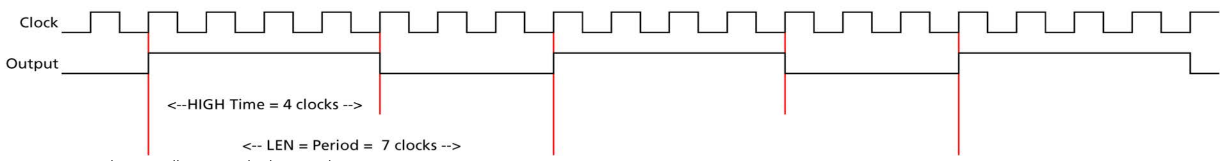


Image 71: timing diagram illustrating the basic pulse parameters

The pulse generator will upon start (trigger) first set the output HIGH for the programmed amount of time. Afterwards it will set the waveform LOW for the remaining time until the programmed length (period) has been reached. As a result, the number of clock cycles during which the output is LOW calculates to: LOW = LEN - HIGH. In the example above with LEN = 7 and HIGH = 4, the signal will be LOW for the remaining 3 clock cycles.

The following table shows the registers required to set the total length of the pulse to be generated. The length is defined in clock cycles:

Table 113: Spectrum API: pulse generator length/period register

Register	Value	Direction	Description
SPC_XIO_PULSEGEN_AVAILLEN_MIN	602001	read	Returns the minimum length (period) of the pulse generator's output pulses in clock cycles.
SPC_XIO_PULSEGEN_AVAILLEN_MAX	602002	read	Returns the maximum length (period) of the pulse generator's output pulses in clock cycles.
SPC_XIO_PULSEGEN_AVAILLEN_STEP	602003	read	Returns the step size the pulse generator's output pulses in clock cycles.
SPC_XIO_PULSEGEN0_LEN	601001	read/write	Define the length of the pulse period generated by pulse generator 0 in clock cycles.
SPC_XIO_PULSEGEN1_LEN	601101	read/write	Define the length of the pulse period generated by pulse generator 1 in clock cycles.
SPC_XIO_PULSEGEN2_LEN	601201	read/write	Define the length of the pulse period generated by pulse generator 2 in clock cycles.
SPC_XIO_PULSEGEN3_LEN	601301	read/write	Define the length of the pulse period generated by pulse generator 3 in clock cycles.

The second parameter that needs to be defined is the amount of clock pulses that force the output to a logic HIGH. The following table shows the registers required to set the total length of the pulse to be generated:

Table 114: Spectrum API: pulse generator HIGH time registers

Register	Value	Direction	Description
SPC_XIO_PULSEGEN_AVAILHIGH_MIN	602004	read	Returns the minimum HIGH time of the pulse generator's output pulses in clock cycles.
SPC_XIO_PULSEGEN_AVAILHIGH_MAX	602005	read	Returns the maximum HIGH time of the pulse generator's output pulses in clock cycles.
SPC_XIO_PULSEGEN_AVAILHIGH_STEP	602006	read	Returns the step size the pulse generator's HIGH time in clock cycles.
SPC_XIO_PULSEGEN0_HIGH	601002	read/write	Define the HIGH time for the pulse generated by pulse generator 0 in clock cycles.
SPC_XIO_PULSEGEN1_HIGH	601102	read/write	Define the HIGH time for the pulse generated by pulse generator 1 in clock cycles.
SPC_XIO_PULSEGEN2_HIGH	601202	read/write	Define the HIGH time for the pulse generated by pulse generator 2 in clock cycles.
SPC_XIO_PULSEGEN3_HIGH	601302	read/write	Define the HIGH time for the pulse generated by pulse generator 3 in clock cycles.

These two settings alone allow for the creation of periodic signals with the freely programmable duty cycle. Setting the HIGH time to half the LEN will result is a clock-like signal with half the time being HIGH and half the time being LOW, hence having a 50% duty-cycle signal.

Since the output of the pulse generator can only change with every edge of its clock input, the speed of this clock ultimately defines the granularity at which the pulses can be configured. The lower the period of the generated pulse signal the finer this granularity becomes with regards to the output signal frequency.

For example, when creating an output with the maximum output frequency of Clk/2 (with LEN = 2 and HIGH = 1), the only possible remaining configuration is a duty-cycle of 50%. And with a output at frequency with Clk/3 (with LEN=3 and HIGH either 1 or 2) the duty-cycle is either 33% or 66%, but cannot be 50%.

In addition to defining the length/period of a single pulse, one can also define how often a pulse should be replayed repeatedly. The choice can be made between repeating the pulses infinitely (until being explicitly stopped) or to pre-define a number of repetitions:

Table 115: Spectrum API: pulse generator loops/pulse repetition registers

Register	Value	Direction	Description
SPC_XIO_PULSEGEN_AVAILLOOPS_MIN	602010	read	Returns the minimum number of times, the output of a pulse generator can be repeated.
SPC_XIO_PULSEGEN_AVAILLOOPS_MAX	602011	read	Returns the maximum number of times, the output of a pulse generator can be repeated.
SPC_XIO_PULSEGEN_AVAILLOOPS_STEP	602012	read	Returns the step size when defining the repetition of pulse generator's output.
SPC_XIO_PULSEGEN0_LOOPS	601004	read/write	Define the number of repetitions of the output period when triggered for pulse generator 0.
SPC_XIO_PULSEGEN1_LOOPS	601104	read/write	Define the number of repetitions of the output period when triggered for pulse generator 1.
SPC_XIO_PULSEGEN2_LOOPS	601204	read/write	Define the number of repetitions of the output period when triggered for pulse generator 2.
SPC_XIO_PULSEGEN3_LOOPS	601304	read/write	Define the number of repetitions of the output period when triggered for pulse generator 3.
0			Upon a trigger event the output of the pulse generator will run infinitely until being disabled or reset.
1 ... [4G - 2]			Upon a trigger event the output period will replayed the defined number of times.

Delaying (phase shifting) the Outputs

As mentioned above the pulse generator will always start with the first portion of the period to be HIGH and then will set the output LOW for the remaining number of cycles within the chosen length.

When using the delay, it is possible to delay the initial HIGH portion of the pulse generator(s) by a defined amount of clock cycles. This in combination with a common starting point (start/trigger) allows for the generation of phase shifted signals as shown below for two of the pulse generators. Both are set up with identical LEN and HIGH parameters, but the additional delay for pulse generator 0 (PGen0) is kept at the default of zero clock cycles, whilst PGen1is delayed by 5 clock cycles:

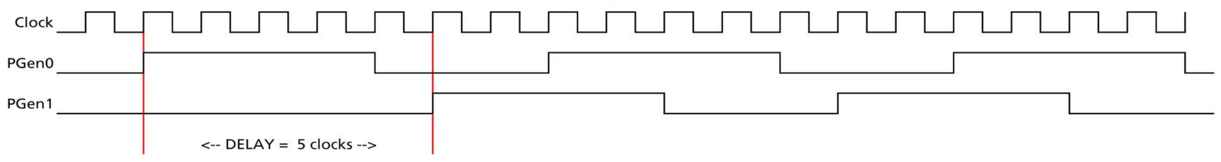


Image 72: timing diagram illustrating delaying a pulse generator output

The amount of additional delay can be set individually for each pulse generator, by using the following registers:

Table 116: Spectrum API: pulse generator delay/phase shift registers

Register	Value	Direction	Description
SPC_XIO_PULSEGEN_AVAILDELAY_MIN	602007	read	Returns the minimum delay of the pulse generator's output in clock cycles.
SPC_XIO_PULSEGEN_AVAILDELAY_MAX	602008	read	Returns the maximum delay of the pulse generator's output in clock cycles.
SPC_XIO_PULSEGEN_AVAILDELAY_STEP	602009	read	Returns the step size of the pulse generator's output delay in clock cycles.
SPC_XIO_PULSEGEN0_DELAY	601003	read/write	Define how much the output of pulse generator 0 is delayed after trigger in clock cycles.
SPC_XIO_PULSEGEN1_DELAY	601103	read/write	Define how much the output of pulse generator 1 is delayed after trigger in clock cycles.
SPC_XIO_PULSEGEN2_DELAY	601203	read/write	Define how much the output of pulse generator 2 is delayed after trigger in clock cycles.
SPC_XIO_PULSEGEN3_DELAY	601303	read/write	Define how much the output of pulse generator 3 is delayed after trigger in clock cycles.

Defining the trigger behavior

Each pulse generator can be set up to react on its trigger input in three different ways, depending on the application's need:

Table 117: Spectrum API: pulse generator mode registers with their available settings

Register	Value	Direction	Description
SPC_XIO_PULSEGEN0_MODE	601000	read/write	Defines the behavior of pulse generator 0 on how to react on its trigger event.
SPC_XIO_PULSEGEN1_MODE	601100	read/write	Defines the behavior of pulse generator 1 on how to react on its trigger event.
SPC_XIO_PULSEGEN2_MODE	601200	read/write	Defines the behavior of pulse generator 2 on how to react on its trigger event.
SPC_XIO_PULSEGEN3_MODE	601300	read/write	Defines the behavior of pulse generator 3 on how to react on its trigger event.
SPCM_PULSEGEN_MODE_GATED	1		Pulse generator will start if the trigger condition or "gate" is met and will stop, if either the gate becomes inactive or the defined number of LOOPS have been generated. Will reset its loop counter, when the gate becomes LOW.
SPCM_PULSEGEN_MODE_TRIGGERED	2		The pulse generator will start if the trigger condition is met and will replay the defined number of loops before re-arming itself and waiting for another trigger event. Changes in the trigger signal while replaying will be ignored.
SPCM_PULSEGEN_MODE_SINGLESHOT	3		The pulse generator will start if the trigger condition is met and will replay the defined number of loops once.

For simplicity, the waveforms below will show the modes principle, without any additionally programmed delay, and also omitting the intrinsic pipeline delay from the trigger event to the output's reaction.

Continuously triggered output

After enabling the pulse generator, it will detect trigger events. Upon each trigger, the programmed number of pulses are generated, as defined by the LEN, HIGH, DELAY and LOOPS parameters explained above. After finishing the programmed number of triggers, it will automatically arm itself again and wait for the next trigger.

In contrast to the Gated mode (see below), once a trigger has been detected the trigger input is ignored and the pulse train will finish independent from any activity on the trigger input. Only when is has finished the current generation, a new trigger will be detected:

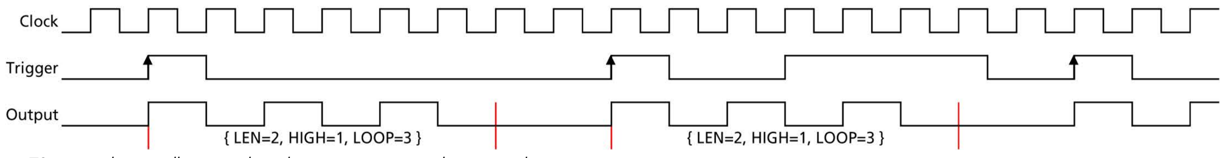


Image 73: timing diagram illustrating the pulse generator triggered output mode

Single Shot triggering

This mode is similar to the triggered mode, but after enabling the pulse generator it will only detect one single trigger. Upon that trigger, the programmed number of pulses are generated, as defined by the LEN, HIGH, DELAY and LOOPS parameters explained above:

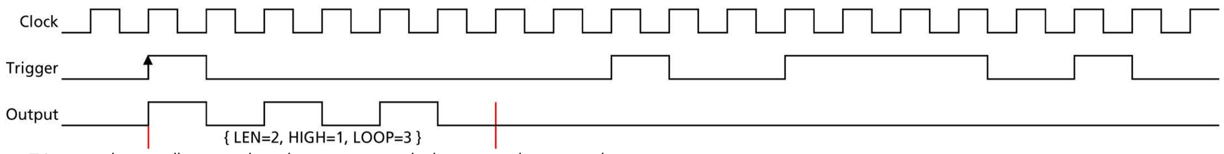


Image 74: timing diagram illustrating the pulse generator single-shot triggered output mode

Afterwards the pulse generator will not detect any further triggers, until being reset by re-enabling:

Continuously gated Output

After enabling the pulse generator, it will detect trigger events. Upon each trigger, the programmed number of pulses are generated, as defined by the LEN, HIGH, DELAY and LOOPS parameters explained above and as long as the trigger condition or gate is still valid (HIGH). If the gate ends, this will stop the output and reset all internal counters back to start. So, each time the gate turns HIGH, the sequence (number of pulses as defined by the LEN, HIGH, DELAY and LOOPS) starts again from its beginning:

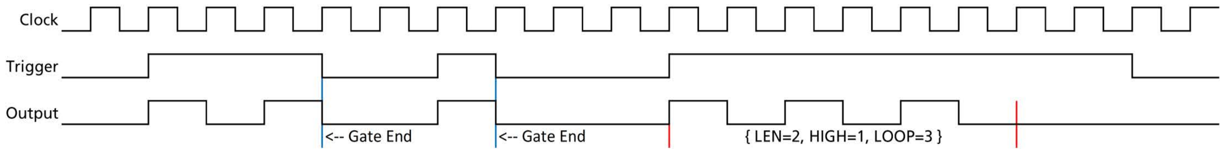


Image 75: timing diagram illustrating the pulse generator gated output mode

Configuring the pulse generator’s trigger source

The various possible signals that can logically be combined to form a trigger event for a pulse generator are split up into two portions each consisting of a multiplexer (MUX).

Multiplexer 1

The first multiplexer, MUX1, selects between two different sources and also allows to be completely unused by utilizing a logical ‘1’ or HIGH level, being transparent to the following AND condition combining the two multiplexers:

Table 118: Spectrum API: pulse generator trigger MUX1 registers with their available settings

Register	Value	Direction	Description
SPC_XIO_PULSEGEN0_MUX1_SRC	601005	read/write	Selects the input source for MUX1 for pulse generator 0.
SPC_XIO_PULSEGEN1_MUX1_SRC	601105	read/write	Selects the input source for MUX1 for pulse generator 1.
SPC_XIO_PULSEGEN2_MUX1_SRC	601205	read/write	Selects the input source for MUX1 for pulse generator 2.
SPC_XIO_PULSEGEN3_MUX1_SRC	601305	read/write	Selects the input source for MUX1 for pulse generator 3.
SPCM_PULSEGEN_MUX1_SRC_UNUSED	0		Inputs of MUX1 are not used in creating the trigger condition and instead a static logic HIGH is used for MUX1.
SPCM_PULSEGEN_MUX1_SRC_RUN	1		This input of MUX1 reflects the current run state of the card. If acquisition/output is running the signal is HIGH. If card has stopped the signal is LOW. The signal is identical to XIO output using SPCM_XMODE_RUNSTATE.
SPCM_PULSEGEN_MUX1_SRC_ARM	2		This input of MUX1 reflects 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 already been detected. the signal is LOW. The signal is identical to XIO output using SPCM_XMODE_ARMSTATE.

By having the two status lines ARM and RUN available as input, it is either possible to generate pulses depending only on the card’s RUN or ARM state (e.g., currently running or currently not running enabling the inverter of MUX1 output) or to mask other trigger conditions from MUX2 to only be passed upon the card’s acquisition/replay RUN or ARM state.

Multiplexer 2

The second multiplexer can be transparent and hence unused or allows to select various sources for starting the pulse creation:

- Allowing a start command issued by the application software by issuing a force trigger command
- Any one of the other pulse generator unit outputs to create pulses or pulse trains with up to four repetition time scales
- The card's acquisition or replay trigger output
- An external logic signal coming in from any of the multi-purpose XIO input capable lines

Table 119: Spectrum API: pulse generator trigger MUX2 registers with their available settings

Register	Value	Direction	Description
SPC_XIO_PULSEGEN0_MUX2_SRC	601006	read/write	Selects the input source for MUX2 for pulse generator 0.
SPC_XIO_PULSEGEN1_MUX2_SRC	601106	read/write	Selects the input source for MUX2 for pulse generator 1.
SPC_XIO_PULSEGEN2_MUX2_SRC	601206	read/write	Selects the input source for MUX2 for pulse generator 2.
SPC_XIO_PULSEGEN3_MUX2_SRC	601306	read/write	Selects the input source for MUX2 for pulse generator 3.
SPCM_PULSEGEN_MUX2_SRC_UNUSED	0		No input of MUX2 is used in creating the trigger condition for the pulse generator. A static logic HIGH is used, so that the MUX output is transparent for the following AND gate.
SPCM_PULSEGEN_MUX2_SRC_SOFTWARE	1		This input reflects the positive edge generated by issuing the SPCM_PULSEGEN_CMD_FORCE command.
SPCM_PULSEGEN_MUX2_SRC_CARDTRIGGER	2		This input of MUX2 reflects the trigger detection of the acquisition/replay. The trigger output goes HIGH as soon as the card's main trigger is recognized. After end of acquisition/replay it is LOW again. In Multiple Recording/Gated Sampling/ABA mode it goes LOW after the acquisition of the current segment stops. In FIFO single mode the trigger output is HIGH until FIFO mode is stopped. The signal is identical to what a XIO output is providing when using SPCM_XMODE_TRIGOUT.
SPCM_PULSEGEN_MUX2_SRC_PULSEGEN0	3		Input to MUX2 is set to output of pulse generator 0/1/2 or 3. This can be used to cascade pulse generators for creating up to four pulse repetition time scales. Each pulse generator can select to be triggered by any of the other pulse generator's output. Selecting its own pulse generator's output as a trigger (loopback) is not allowed and will lead to a driver error.
SPCM_PULSEGEN_MUX2_SRC_PULSEGEN1	4		
SPCM_PULSEGEN_MUX2_SRC_PULSEGEN2	5		
SPCM_PULSEGEN_MUX2_SRC_PULSEGEN3	6		
SPCM_PULSEGEN_MUX2_SRC_XIO0	7		Input to MUX2 is set to the input signal coming in from multi-purpose line of X0. M2p: Since X0 is an output only, it therefore is not allowed to be used as an input.
SPCM_PULSEGEN_MUX2_SRC_XIO1	8		Input to MUX2 is set to the input signal coming in from multi-purpose line of X1.
SPCM_PULSEGEN_MUX2_SRC_XIO2	9		Input to MUX2 is set to the input signal coming in from multi-purpose line of X2.
SPCM_PULSEGEN_MUX2_SRC_XIO3	10		Input to MUX2 is set to the input signal coming in from multi-purpose line of X3. M4i/M4x: Since X3 is not available, it therefore is not allowed to be used as an input.

The output of the following command register is connected to all pulse generator units in parallel in a synchronous fashion:

Table 120: Spectrum API: pulse generator command register for trigger forcing by software

Register	Value	Direction	Description
SPC_XIO_PULSEGEN_COMMAND	601501	write only	Executes a command for the pulse generator option.
SPCM_PULSEGEN_CMD_FORCE	1h		Generate a single rising edge, that is common for all pulse generator engines. This allows to start/trigger the output of all enabled pulse generators synchronously by issuing a software command.

This allows to start any number of pulse generators set to MUX2_SRC_SOFTWARE to be started at the same instant even from software, useful when requiring pulses with a known and static phase relation.

Additional trigger configuration (changing the active edge or level)

Please note that the Trigger/Gate input to the "Pulse Generation" portion is always HIGH-active. Depending on the selected pulse generator configuration it is triggering on the rising edge or the logic HIGH state. The two programmable inverters at the multiplexer outputs can be used to trigger on the falling edge or a logical LOW instead.

To access the three programmable inverters and to optionally change whether triggering on a rising edge (the trigger signal changing its state from LOW to HIGH) or on the valid level (the trigger being logically HIGH), following registers can be used:

Table 121: Spectrum API: pulse generator additional configuration registers with the available settings

Register	Value	Direction	Description
SPC_XIO_PULSEGEN0_CONFIG	601007	read/write	Bitmask with additional configuration for pulse generator 0.
SPC_XIO_PULSEGEN1_CONFIG	601107	read/write	Bitmask with additional configuration for pulse generator 1.
SPC_XIO_PULSEGEN2_CONFIG	601207	read/write	Bitmask with additional configuration for pulse generator 2.
SPC_XIO_PULSEGEN3_CONFIG	601307	read/write	Bitmask with additional configuration for pulse generator 3.
SPCM_PULSEGEN_CONFIG_MUX1_INVERT	1h		When bit is set, the output of MUX1 is logically inverted.
SPCM_PULSEGEN_CONFIG_MUX2_INVERT	2h		When bit is set, the output of MUX2 is logically inverted.
SPCM_PULSEGEN_CONFIG_INVERT	4h		When bit is set, the output of the pulse generator is logically inverted.
SPCM_PULSEGEN_CONFIG_HIGH	8h		As default the pulse generator's trigger input is sensitive only to a rising edge. When using this configuration, the input will not look for an active edge, but rather detect a HIGH level. This is similar to the distinction of the card's main trigger modes, when choosing between SPC_TM_POS and SPC_TM_HIGH.

Since the register is implemented as a bitmask, any combination of the above configuration flags is possible.

```
// enable the inverters on MUX1 and MUX2 outputs for pulse generator 2
int32 lPulseGenConfig = (SPCM_PULSEGEN_CONFIG_MUX1_INVERT | SPCM_PULSEGEN_CONFIG_MUX2_INVERT);

spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN2_CONFIG, lPulseGenConfig);
```

Configuring Multi Purpose lines to output generated pulses

Each of the up to four on-board multi purpose I/O lines can be programmed to output the pulses generated by its corresponding pulse generator unit, making it available for any external devices.

Please check the available modes by reading the SPCM_X0_AVAILMODES, SPCM_X1_AVAILMODES, SPCM_X2_AVAILMODES and SPCM_X3_AVAILMODES register first. The available modes may differ from card to card and may be enhanced with new driver/firmware versions to come.

Table 122: Spectrum API: XIO lines and mode software registers with their reduced to the settings required for outputting pulses

Register	Value	Direction	Description
SPCM_X0_AVAILMODES	600300	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X0)
SPCM_X1_AVAILMODES	600301	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X1)
SPCM_X2_AVAILMODES	600302	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X2)
SPCM_X3_AVAILMODES	600303	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X3)
SPCM_X0_MODE	600200	read/write	Defines the mode for (X0). Only one mode selection is possible to be set at a time
SPCM_X1_MODE	600201	read/write	Defines the mode for (X1). Only one mode selection is possible to be set at a time
SPCM_X2_MODE	600202	read/write	Defines the mode for (X2). Only one mode selection is possible to be set at a time
SPCM_X3_MODE	600203	read/write	Defines the mode for (X3). Only one mode selection is possible to be set at a time
SPCM_XMODE_DISABLE	00000000h	No mode selected. Output is tristate (default setup)	
...	...	For all other modes please see chapter “Multi Purpose I/O Lines”.	
SPCM_XMODE_PULSEGEN	00080000h	A/D and D/A cards only (optional): Connector reflects the output of the same index pulse generator (X1 can output pulses from pulse generator 1, X2 can output pulses from pulse generator 2, ... etc.). On M4i/M4x cards with three XIO lines (X0, X1, X2) and four pulse generators, pulses from pulse generator 3 cannot be output, but can still be used in cascading configurations to trigger another pulse generator.	

Please note that a change to the SPCM_X0_MODE, SPCM_X1_MODE, SPCM_X2_MODE or SPCM_X3_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.

Programming Example

The following example shows in principle, the steps required for generating a single, repetitive pulse with one of the pulse generators and how to output that pulse on the matching multi-purpose I/O line:

```
// First we set up the channel selection and the clock.
// For this example we enable only one channel to be able to use max sampling rate on all card types.
spcm_dwSetParam_i32 (hCard, SPC_CHENABLE, CHANNEL0);

// Read out the max. supported sampling rate ...
int64 llMaxSR = 0;
spcm_dwGetParam_i64 (hCard, SPC_PCISAMPLERATE, &llMaxSR);

// ... and use this as the card's sampling rate
spcm_dwSetParam_i64 (hCard, SPC_SAMPLERATE, llMaxSR);

// Read out the clock, at which the pulse generator will run with the above set sampling rate.
int64 llPulseGenClock_Hz = 0;
spcm_dwGetParam_i64 (hCard, SPC_XIO_PULSEGEN_CLOCK, &llPulseGenClock_Hz);

// Configure X0 to output signal from corresponding pulse generator 0
spcm_dwSetParam_i32 (hCard, SPCM_X0_MODE, SPCM_XMODE_PULSEGEN);

// Setup pulse generator 0 (output on X0)
// to generate a continuous signal with 1 MHz and ~50% duty-cycle
int32 lLenFor1MHz = static_cast < int32 > (llPulseGenClock_Hz / MEGA(1));
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_MODE, SPCM_PULSEGEN_MODE_TRIGGERED);
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_LEN,  lLenFor1MHz);

// An integer division by 2 will be truncated if lLenFor1MHz is an odd number,
// resulting in a slightly shorter HIGH than LOW time.
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_HIGH,  lLenFor1MHz / 2);

// Set LOOPS to 0: repeat infinitely
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_LOOPS, 0);

// Configure pulse generator to be triggered/started by software force command
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_MUX1_SRC, SPCM_PULSEGEN_MUX1_SRC_UNUSED);
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_MUX2_SRC, SPCM_PULSEGEN_MUX2_SRC_SOFTWARE);

// Enable the selected pulse generator and hence arm its trigger detection
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN_ENABLE, SPCM_PULSEGEN_ENABLE0);

// Write the settings to the card:
// This will update the clock section to generate the programmed frequencies
// (SPC_SAMPLERATE) and also write the pulse generator settings to the card.
spcm_dwSetParam_i32 (hCard, SPC_M2CMD, M2CMD_CARD_WRITESETUP);

// Start all armed pulse generators (in this case just one) by a software command
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN_COMMAND, SPCM_PULSEGEN_CMD_FORCE);

// Wait until a key is pressed
printf ("\nPress a key to stop the pulse generator(s) ");
cGetch ();

// Stop all running pulse generators
spcm_dwSetParam_i32(hCard, SPC_XIO_PULSEGEN_ENABLE, 0);
spcm_dwSetParam_i32(hCard, SPC_M2CMD, M2CMD_CARD_WRITESETUP);
```

Spectrum provides a dedicated programming example for the pulse generator feature as part of the standard example package. This example is showing different and more complex configurations than shown above, e.g., cascading of multiple pulse generators for more complex pulse generation time scales.

Option Star-Hub

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 M5i series that are mounted within one host system (PC). Up to eight cards can be synchronized, including the card carrying the Star-Hub option.

You can order the Star-Hub option with 2, 4 or 8 synchronization cables included. Additional synchronization cables for extending an existing setup or as a replacement can be ordered separately.

The Star-Hub allows synchronizing cards of the same family only. It is not possible to synchronize cards of different families!

The Star-Hub is 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.

When the cards are synchronized using Star-Hub options, any phase delay between the sampling clocks of the synchronized cards is kept to a minimum, and for triggering no additional delay between the trigger events is injected. 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 M5i 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.

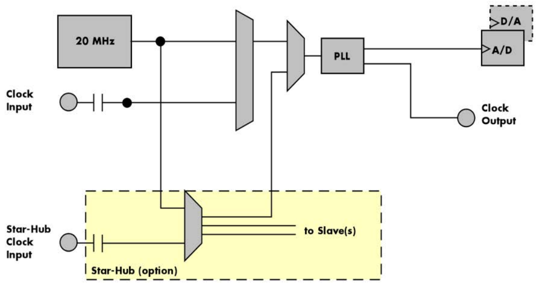


Table 123: star-hub clock overview diagram

Depending on whether the external reference clock is used with a single card or with multiple cards connected/synchronized via Star-Hub, the externally fed in clock must be either connected to the "Clock-In" connector of the card itself or connected to the Clock Input located on the additional PCIe bracket of the Star-Hub module.

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 interconnection 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 124: 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 cabeling.
...	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 125: 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 therefor 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_TMASK_CH2 | SPC_TMASK_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 126: 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_WRITESETUP	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 occured - 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.

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.

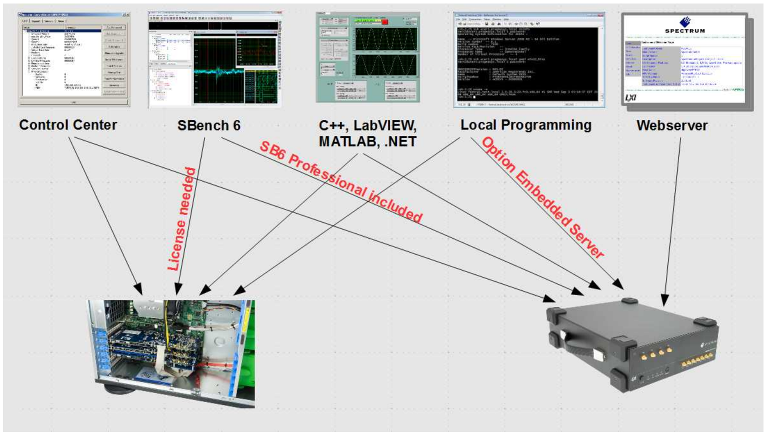


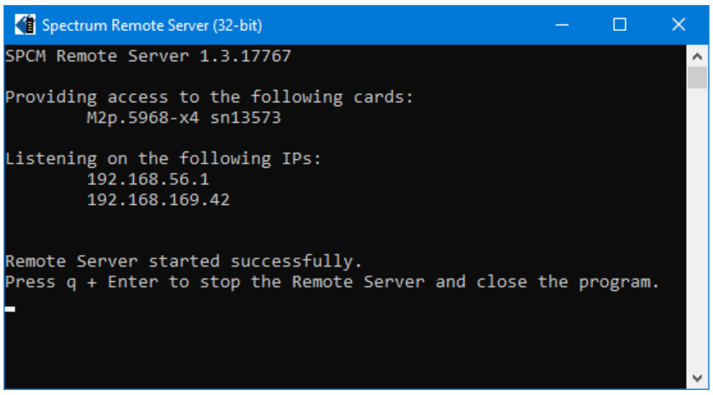
Image 76: Overview of remote server option interaction in comparison to NETBOX devices

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.


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.
```

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&
```

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.lxistandard.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.

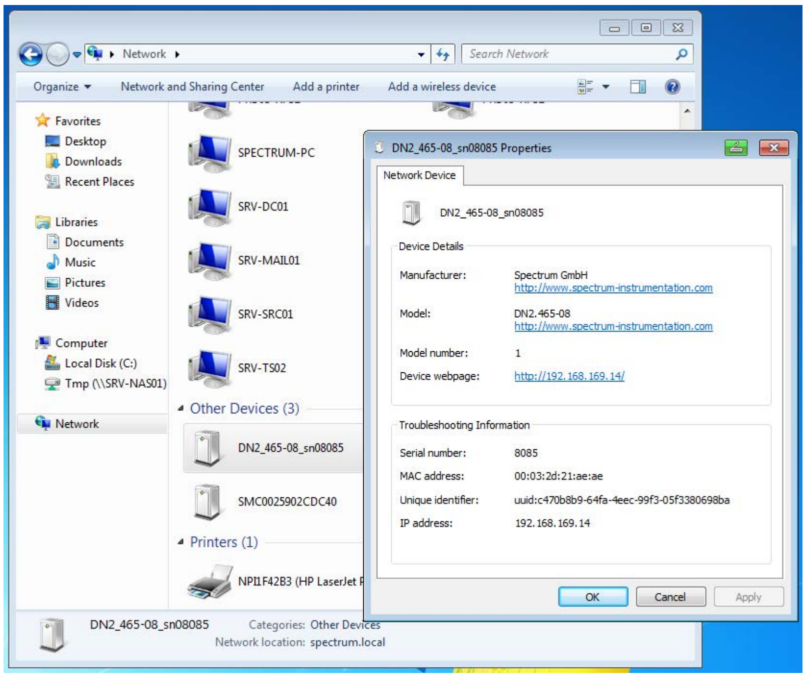
If a fixed IP address should be used instead, the parameters need to be set according to the current LAN requirements.

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.



DN2_465-08_sn08085 Properties

Network Device

DN2_465-08_sn08085

Device Details

Manufacturer:	Spectrum GmbH http://www.spectrum-instrumentation.com
Model:	DN2.465-08 http://www.spectrum-instrumentation.com
Model number:	1
Device webpage:	http://192.168.169.14/

Troubleshooting Information

Serial number:	8085
MAC address:	00:03:2d:21:ae:ae
Unique identifier:	uuid:c470b8b9-64fa-4eec-99f3-05f3380698ba
IP address:	192.168.169.14

DN2_465-08_sn08085 Categories: Other Devices
Network location: spectrum.local

Image 77: 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.

Mode Block Average (Firmware Option)

Overview

General Information

The Block Average Module improves the fidelity of any repetitive signal by reducing its random noise components. The Module allows multiple triggered acquisitions to be made and the triggered segments to accumulated. The process reduces random noise improving the visibility of the repetitive signal. The averaged signal has an enhanced measurement resolution and increased signal-to-noise (SNR) ratio.

The complete averaging process is performed inside the FPGA of the digitizer and involves no CPU load at all. Averaging also reduces the amount of data that needs to be transferred to the host PC further reducing CPU demand and speeding up measurement times.

The averager has multiple modes and can either accumulate unconditionally or selectively, only including samples that exceed a programmable threshold level (Threshold Defined Averaging, TDA).

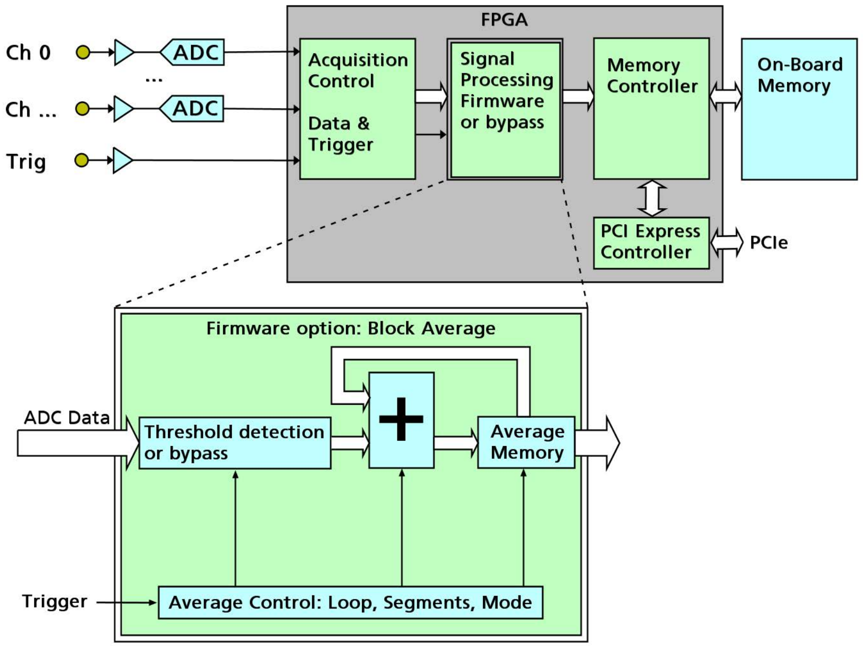


Image 78: block average FPGA option - block diagram

The Block Average mode is fully compatible with streaming (FIFO) mode so that the digitizer can accumulate and average signals for hours or days without losing a single event. The Module takes advantage of an advanced trigger circuit, with very fast re-arm time, so that signals can be averaged at ultra-fast rates, as specified in the technical data section.

The signal processing firmware also includes the standard digitizer firmware so that normal digitizer operation can be performed with no limitations.

Principle of operation

In Block Average mode the acquisition works very similar to the Multiple Recording mode.

The memory is segmented and with each trigger condition a predefined number of samples, a segment, is acquired.

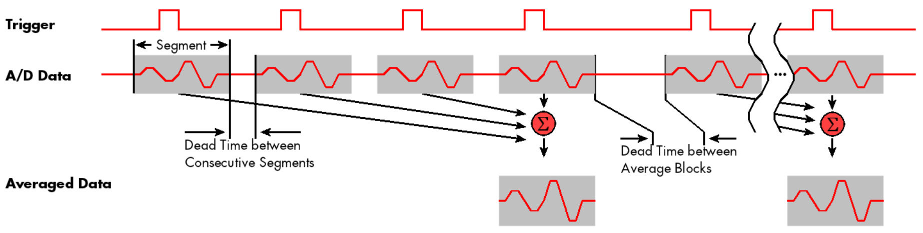

Trigger
A/D Data
Averaged Data
Segment
Dead Time between Consecutive Segments
Dead Time between Average Blocks
Σ
...

Image 79: block average FPGA option - principle of operation

The Block Average option now takes a programmable number of these acquired consecutive data segments and averages them sample by sample over one another.

The result of one averaging operation is a segment with summed values, that has the same length as each original „RAW“ segment, but each sample now consists of the sum of all samples of the averaged segment at the same location in relation to the trigger signal.

In order to get any meaningful results out of the Block Average operation, a repetitive signal is required along with a stable trigger condition.

Setting up the Acquisition

The Block Average mode allows the acquisition of data blocks with multiple trigger events without restarting the hardware.

With each trigger event, one RAW segment will be acquired (as shown) and the segment is then accumulated by the average firmware. If the number of accumulations is reached, the data is transferred to the on-board memory. So, the on-board memory will again be divided into several segments of the same size to hold the processed data, with the data now being 32bit per sample.
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.

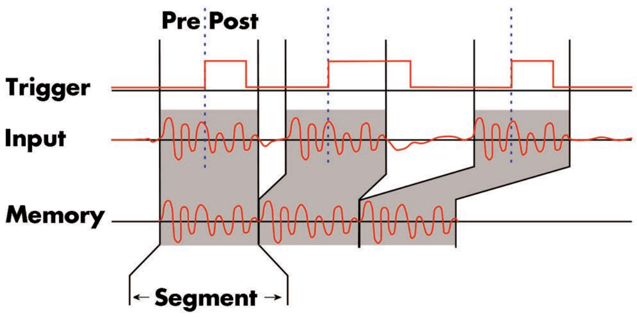


Image 80: timing diagram of block average acquisition

The following table shows the register for defining the structure of the segments to be recorded with each trigger event.

Table 127: Spectrum API: software registers and register settings for programming the block average mode

Register	Value	Direction	Description
SPC_POSTTRIGGER	10100	read/write	Defines the number of samples per channel to be recorded after the trigger event.
SPC_SEGMENTSIZE	10010	read/write	Size of one triggered segment (in RAW samples) as well as the averaged segment (in 32bit samples). The total number of samples to be recorded per channel after detection of one trigger event includes the time recorded before the trigger (pre trigger = segmentsize - posttrigger).
SPC_AVERAGES	10050	read/write	Defines the number of triggered segments that are averaged sample per sample over one another.

Each segment consist of pretrigger and 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: [pretrigger] = [segment size] - [posttrigger].

When using Block Averaging 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 triggered 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 a multiple of the segment size. The table below shows the register for enabling Block Average. For detailed information on how to setup and start the standard acquisition mode please refer to the according chapter earlier in this manual.

Table 128: Spectrum API: card mode registers and register settings for standard block average mode

Register		Value	Direction	Description
SPC_CARDMODE		9500	read/write	Defines the used operating mode
	SPC_REC_STD_AVERAGE	20000h	Enables Block Averaging for standard acquisition with 32 bit wide result data.	

The total number of samples to be recorded to the on-board memory in Standard Mode is defined by the SPC_MEMSIZE register.

Register	Value	Direction	Description
SPC_MEMSIZE	10000	read/write	Defines the total number of samples to be recorded per channel.

FIFO Mode

The Block Averaging in FIFO Mode is similar to the Block Averaging 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 Block Averaging.
The advantage of Block Averaging 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 Block Averaging. For detailed information how to setup and start the board in FIFO mode please refer to the according chapter earlier in this manual.

Table 129: Spectrum API: card mode registers and register settings for FIFO block average mode

Register		Value	Direction	Description
SPC_CARDMODE		9500	read/write	Defines the used operating mode
	SPC_REC_FIFO_AVERAGE	200000h	Enables Block Averaging for FIFO acquisition with 32 bit wide result data.	

The number of segments to be recorded must be set separately with the register shown in the following table:

Table 130: Spectrum API: block average mode loop register and register settings

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 averaged 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. Please keep in mind, that each averaged sample needs 4 bytes (32 bit) of memory to be stored. The required size in memory depends on the selected average mode. 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:

Table 131: Spectrum API: Limits of pre trigger, post trigger, memory size, averages and loops for block averaging

Activated Channels	Used Mode	Memory size SPC_MEMSIZE			Pre trigger (defined by segment and post)			Post trigger SPC_POSTTRIGGER (limited by max. pretrigger)			Segment size SPC_SEGMENTSIZE (limited by max. AverageMem)			Loops SPC_LOOPS			Number ofAverages SPC_AVERAGES		
		Min	Max	Step	Min	Max	Step	Min	Max	Step	Min	Max	Step	Min	Max	Step	Min	Max	Step
1 Ch	Standard Average	64	Mem	32	32	64k	32	32	1M-32	32	64	1M	32		not used		2	1024	1
	FIFO Average		not used											0 (∞)	4G - 1	1			
2 Ch	Standard Average	64	Mem/2	32	32	32k	32	32	512k-32	32	64	512k	32		not used		2	1024	1
	FIFO Average		not used											0 (∞)	4G - 1	1			

All figures listed here are given in samples. An entry of [512k - 32] means [512 kSamples - 32] = 524256 samples.

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

Table 132: Spectrum-API: maximum memory sizes for different memory upgrade options

	Installed Memory	
	2 GSample (RAW samples) equals 1 GSample (32bit Average Samples)	**8 GSample (RAW samples) equals 4 GSample (32bit Average Samples)** **(Option: M5i.xxx-MEM8GS)**
Mem	1 GSample	4 GSample
Mem / 2	512 MSample	2 GSample

Averaging Modes

The M5i block averaging unit allows to select which of the RAW samples coming from the ADC are used for accumulation/averaging and which are not. This can be defined on a per channel base:

Table 133: Spectrum API: block average mode registers for controlling accumulation (normal or threshold defined averaging TDA)

Register		Value	Direction	Description
SPC_AVAILAVRGMODES		424291	read	Bitmask in which all bits of the below mentioned AVRGMODEs are set, if supported by the firmware.
SPC_AVRGMODE_CH0		42430	read/write	Defines the averaging mode for channel 0.
SPC_AVRGMODE_CH1		42431	read/write	Defines the averaging mode for channel 1.
	AVRGMODE_NORMAL	0h	Unconditional averaging: every RAW sample will be accumulated.	
	AVRGMODE_TDA_HIGH	1h	Threshold Defined Averaging: Only RAW samples above a programmed threshold will be accumulated. Samples below that threshold will be replaced by a programmed replacement value. The corresponding threshold value must be more positive than the corresponding replacement value.	
	AVRGMODE_TDA_LOW	2h	Threshold Defined Averaging: Only RAW samples below a programmed threshold will be accumulated. Samples above that threshold will be replaced by a programmed replacement value. The corresponding threshold value must be more negative than the corresponding replacement value.	

Standard Average Mode

The default "normal" mode does simply use all incoming samples for accumulation, and hence requires no further configuration to be done.

Threshold Defined Averaging (TDA)

Certain applications have a very low-level signal of interest with its rare events sitting barely above a noisy baseline. One such application is time of flight mass spectroscopy (TOFMS).

The process of averaging in itself already reduces random noise with each accumulation step. To further enhance the visibility of the signal of interest, even when synchronous noise is present, the M5i averager includes the Threshold Defined Averaging (TDA) mode. This mode allows samples below a set threshold to either be excluded from the accumulation process, or replaced with a user defined (hence known) value.

For the decision on which samples to include and which to replace, defining a threshold level is required:

Table 134: Spectrum API: TDA - threshold level registers

Register	Value	Direction	Description	Range
SPC_TDA_THRESHOLD_CH0	42440	read/write	Defines the threshold value for threshold defined averaging on channel 0 in LSB.	-2047 to +2047
SPC_TDA_THRESHOLD_CH1	42441	read/write	Defines the threshold value for threshold defined averaging on channel 1 in LSB.	-2047 to +2047

The ADC samples that do not exceed this threshold will be dropped and replaced with a constant value as defined by:

Table 135: Spectrum API: TDA - replacement value registers

Register	Value	Direction	Description	Range
SPC_TDA_REPLACEMENT_CH0	42450		Sets replacement sample for threshold defined averaging on channel 0 in LSB.	-2047 to +2047
SPC_TDA_REPLACEMENT_CH1	42451		Sets replacement sample for threshold defined averaging on channel 1 in LSB.	-2047 to +2047

The output of the "level selection" block, shown in the averager's block diagram using AVRGMODE_TDA_HIGH would look the image on the right.

All samples that are above a threshold level are forwarded to the accumulator as is, while all samples below are replaced with the defined value, which is then forwarded.

So, the accumulator would see data segments on its input that are similar to the waveform shown at the bottom of the image. For best results, the threshold level should be set outside the level of noise.

When using TDA_HIGH, the replacement value must be lower than the threshold value and both levels must be defined in LSB of the ADC. As a result, the extracted events will be "positive" pulses.

TDA_LOW works the other way round. Only samples that are below the threshold level will be forwarded to the accumulator, while those above it are replaced before being forwarded. In this case the replacement value must be higher than the threshold level and the extracted events will be "negative" pulses.

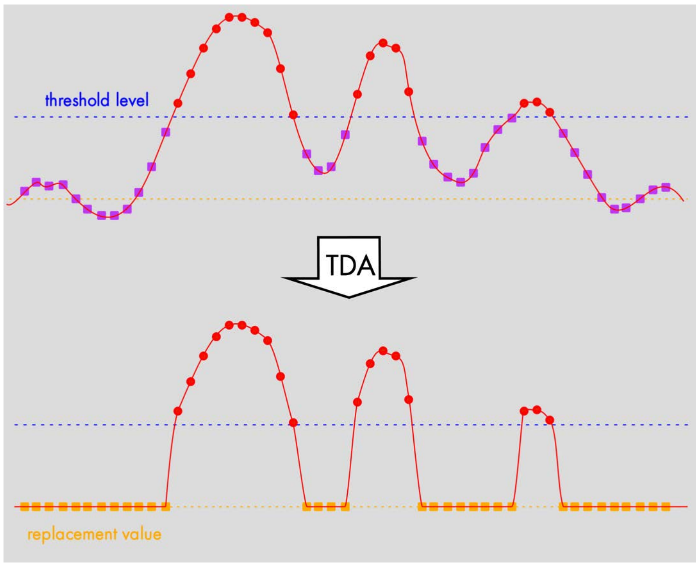


Image 81: visualization of the effects of threshold defined averaging (example for positive pulses)

If a sample does not exceed the threshold over all RAW segments that create one summed segment, it's final value will be a known value: (NoOfAverages * Replacement).

To remove the absolute replacement value from all samples, for normalization, the application software can subtract (NoOfAverages * Replacement) from every sample in post-processing.

For increasing the number of averages per segment, beyond what's possible in the firmware, the application software can simply sum up multiple already averaged segments.

Clock Modes

When using Block Averaging all of the card's clock modes can be used. Also all of the allowed sample rates can be used. For detailed information on the available clock modes, please take a look at the relating chapter earlier in this manual.

Trigger Modes

When using Block Averaging all of the card's trigger modes can be used. For detailed information on the available trigger modes, please take a look at the relating chapter earlier in this manual.

Output Data Format

When using Block Averaging mode, the resulting samples will be 32 bit signed integer values per channel, that each consist of the sum of a particular sample over all averaged segments. The following table illustrates this with the first four of 'S+1' samples of one channel (A0, A1, A2, A3, ..., A_S) that are N times averaged (summed):

Table 136: Spectrum API: block average mode output sample format

	Samples of one segment with segment size S+1 over time					
Triggered Segment No. 1	A0(1)	A1(1)	A2(1)	A3(1)	...	$A_S(1)$
Triggered Segment No. 2	A0(2)	A1(2)	A2(2)	A3(2)	...	$A_S(2)$
...	...	...	...	...	...	...
Triggered Segment No. N	A0(N)	A1(N)	A2(N)	A3(N)	...	$A_S(N)$
Resulting averaged Samples	$\sum_{i=1}^{N} \mathrm{A0(i)}$	$\sum_{i=1}^{N} \mathrm{A1(i)}$	$\sum_{i=1}^{N} \mathrm{A2(i)}$	$\sum_{i=1}^{N} \mathrm{A3(i)}$	...	$\sum_{i=1}^{N} \mathrm{AS(i)}$

The resulting „resolution" of the samples increases with the number of averages. For example averaging 12 bit RAW samples two times results in a final resolution of 13 bit, averaging them four times results in a sample with 14 bit „resolution".

By not dividing down the samples by the number of averages in the firmware and providing the user application with the 32 bit wide sums, one can take full advantage of the enhanced resolution by using proper data formats in the application software.

Data organization

Data is organized in a multiplexed way in the transfer buffer the same way as the RAW samples would be. If using 2 channels data of first activated channel comes first, then data of second channel:

Table 137: Spectrum API: block average mode data organization

Activated Channels	Ch0	Ch1	32bit wide averaged samples ordering in buffer memory starting with data offset zero																
1 channel	X		A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16
1 channel		X	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16
2 channels	X	X	A0	B0	A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	A8

The samples are re-named for better readability. A0 is sample 0 of channel 0, B4 is sample 4 of channel 1, and so on. The averaged samples now just have a wider format of 32 bit independent of the original RAW sample resolution.

Programming examples

The following example shows how to set up the card for Block Average in standard mode with 32 bit wide output data.

```
// define some parameters via variables
uint32 dwNoOfChannels =     2;            // Two active channels (e.g. Ch0 and Ch1)
uint64 qwNumberOfSegments = 4;            // four averaged segments will be acquired
uint64 qwSegmentSize =      1024;         // Set the segment size to 1024 samples
uint64 qwPosttrigger =      768;          // Set the posttrigger to 768 samples and therefore
                                          // the pretrigger will be 256 samples

uint64 qwSetMemsize = qwSegmentSize * qwNumberOfSegments; // calculate memsize

// for averaging the number of bytes per sample is fixed to 4 (32 bit samples)
// and memory for all channels is needed
uint64 qwMemInBytes = qwSetMemsize * sizeof(int32) * dwNoOfChannels;
void* pvBuffer = pvAllocMemPageAligned (qwMemInBytes);

// set up DMA transfer with the card
spcm_dwDefTransfer_i64 (stCard.hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 0, pvBuffer, 0, qwMemInBytes);

// configure acquisition
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_AVERAGE);  // Enables Standard Averaging
spcm_dwSetParam_i32 (hDrv, SPC_AVERAGES, 100);                  // 100 triggered acquisitions will be
                                                                // averaged for one output segment
spcm_dwSetParam_i64 (hDrv, SPC_SEGMENTSIZE, qwSegmentSize);
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, qwPosttrigger);
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE,     qwSetMemsize);

spcm_dwSetParam_i32 (hDrv, SPC_AVRGMODE_CH0, AVRGMODE_NORMAL); // normal avareging for Ch0
spcm_dwSetParam_i32 (hDrv, SPC_AVRGMODE_CH1, AVRGMODE_HIGH);   // TDA for Ch1
spcm_dwSetParam_i32 (hDrv, SPC_TDA_THRESHOLD_CH1,     512);    // sample below threshold of +512 LSB ...
spcm_dwSetParam_i32 (hDrv, SPC_TDA_REPLACEMENT_CH1, -2040);    // ... will be replaced with value -2040

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE,  SPC_TM_POS); // Set triggermode 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 Block Average in FIFO mode.

```
uint64 qwNumberOfSegments = 256;           // 256 averaged segments will be acquired
uint64 qwSegmentSize =      2048;          // Set the segment size to 2048 samples
uint64 qwPosttrigger =      1920;          // Set the posttrigger to 1920 samples and therefore
                                           // the pretrigger will be 128 samples

// FIFO buffer setup not shown here for simplicity. See FIFO buffer setup in according chapter for details.

// configure acquisition
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_FIFO_AVERAGE); // Enables FIFO Averaging
spcm_dwSetParam_i32 (hDrv, SPC_AVERAGES, 100);                  // 100 triggered acquisitions will be
                                                                // averaged for one output segment
spcm_dwSetParam_i64 (hDrv, SPC_SEGMENTSIZE, qwSegmentSize);
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, qwPosttrigger);
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE,     qwSetMemsize);
spcm_dwSetParam_i64 (hDrv, SPC_LOOPS        qwNumberOfSegments);
spcm_dwSetParam_i32 (hDrv, SPC_AVRGMODE_CH0, AVRGMODE_NORMAL);  // normal averaging (one channel assumed)

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE,  SPC_TM_POS); // Set triggermode to ext. TTL mode (rising edge)
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0); // and enable it within the trigger OR-mask
```

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 138: 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_HWDRVVERSION	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.P ower 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 138: 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	Memsize 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_ANDORMASKOVRLAP	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_PATPWSMALLEDGE	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 occured 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 and fan speed sensors

The M5i card series has multiple integrated sensors that allow to read out different internal temperatures and the fan speed. Theses functions are also available for the internal M5i cards inside the digitizerNETBOX, generatorNETBOX or hybridNETBOX series. In here the temperature can be read out for every internal card separately.

Base Card Temperature read-out registers

Up to three different temperature sensors can be read-out for each M5i base 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 139: Spectrum API: base card read-out registers of internal temperature sensors

Register	Value	Direction	Description
SPC_MON_TK_BASE_CTRL	500022	read	Base card FPGA temperature in Kelvin
SPC_MON_TK_BASE_0	500099	read	Base card fan intake temperature in Kelvin
SPC_MON_TK_BASE_1	500102	read	Base card power supply temperature in Kelvin
SPC_MON_TC_BASE_CTRL	500025	read	Base card FPGA temperature in degrees Celsius
SPC_MON_TC_BASE_0	500100	read	Base card fan intake temperature in degrees Celsius
SPC_MON_TC_BASE_1	500103	read	Base card power supply temperature in degrees Celsius
SPC_MON_TF_BASE_CTRL	500028	read	Base card FPGA temperature in degrees Fahrenheit
SPC_MON_TF_BASE_0	500101	read	Base card fan intake temperature in degrees Fahrenheit
SPC_MON_TF_BASE_1	500104	read	Base card power supply temperature in degrees Fahrenheit

Front-end Module read-out registers

Up to three different temperature sensors can be read-out for each M5i front-end module. 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 140: Spectrum API: front-end module read-out registers of internal temperature sensors

Register	Value	Direction	Description
SPC_MON_TK_MODULE_0	500023	read	ADC temperature in Kelvin
SPC_MON_TK_MODULE_1	500024	read	Analog front-end temperature in Kelvin
SPC_MON_TK_MODULE_2	500089	read	Front-end module power supply temperature in Kelvin
SPC_MON_TC_MODULE_0	500026	read	ADC temperature in degrees Celsius
SPC_MON_TC_MODULE_1	500027	read	Analog front-end temperature in degrees Celsius
SPC_MON_TC_MODULE_2	500090	read	Front-end module power supply temperature in degrees Celsius
SPC_MON_TF_MODULE_0	500029	read	ADC temperature in degrees Fahrenheit
SPC_MON_TF_MODULE_1	500030	read	Analog front-end temperature in degrees Fahrenheit
SPC_MON_TF_MODULE_2	500091	read	Front-end module power supply temperature in degrees Fahrenheit

Base Card fan speed read-out registers

For the two fans of the M5i base card, the temperature controlled rotation speed can be read-out:

Table 141: Spectrum API: base card read-out registers of the fan speed sensors

Register	Value	Direction	Description
SPC_MON_RPM_FAN0	500092	read	Revolution per minute (RPM) of Fan 0 (lower fan, closer to PCIe card edge connector).
SPC_MON_RPM_FAN1	500093	read	Revolution per minute (RPM) of Fan 1.(upper fan, closer to power cable connector).

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.
- Please make sure that fan intake and also the fan outlet on the front-panel is not blocked and free of dust.

33xx temperatures and limits

The following description shows the meaning of each temperature figure on the M5i.33xx series and also gives maximum ratings that should not be exceeded. All temperature figures given in degrees Celsius, the fan speed in RPM:

Table 142: Spectrum API: temperature limits

Sensor Name	Sensor Location	Typical figure at 25°C environment temperature	Maximum temperature
BASE_CTRL	Inside FPGA	55°C ±5°C	80°C
BASE_0	Fan intake	35°C ±5°C	80°C
BASE_1	Power Base	40°C ±5°C	80°C
MODULE_0	Inside ADC	50°C ±5°C	90°C
MODULE_1	Amplifier Front-End	40°C ±5°C	80°C
MODULE_2	Power Front-End	30°C ±5°C	80°C
FAN_0	Power Front-End	2800 ±300 RPM	–
FAN_1	Power Front-End	2800 ±300 RPM	–

Details on M5i cards I/O lines

Multi-Purpose I/O Lines

The four Multi Purpose I/O connectors (X0, X1, X2 and X3) of the M5i cards from Spectrum are protected against over voltage conditions.

For this purpose clamping diodes of the types CD1005 are used in conjunction with a series resistor. All four I/O lines are internally clamped to signal ground and to 3.3V clamping voltage. So when connecting sources with a higher level than the clamping voltage plus the forward voltage of typically 0.6..0.7 V will be the resulting maximum high-level level.

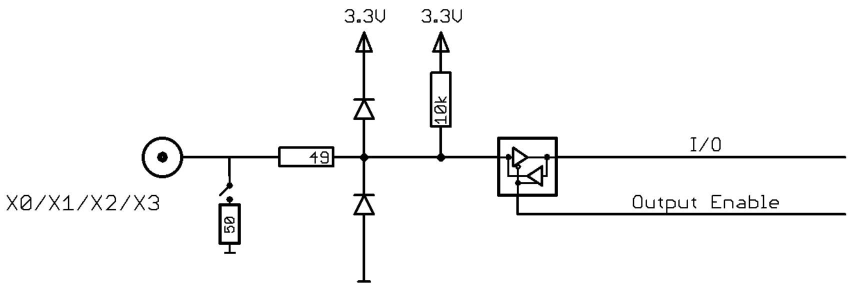


Image 82: electrical structure of multi-purpose I/O lines

The maximum forward current limit for the used CD1005 diodes is 100 mA, which is effectively limited by the used series resistor for logic levels up to 5.0V. To avoid floating levels with unconnected inputs, a pull up resistor of 10 kOhm to 3.3V is used on each line.

Interfacing with clock input

The clock input of the M5i cards is AC-coupled, single-ended PECL type. Due to the internal biasing and a relatively high maximum input voltage swing, it can be directly connected to various logic standards, without the need for external level converters.

Single-ended LVTTL sources

All LVTTL sources, be it 2.5V LVTTL or 3.3V LVTTL must be terminated with a 50 Ohm series resistor to avoid reflections and limit the maximum swing for the M5i card.

Differential (LV)PECL sources

Differential drivers require equal load on both the true and the inverting outputs. Therefore the inverting output should be loaded as shown in the drawing. All PECL drivers require a proper DC path to ground, therefore emitter resistors R_E must be used, whose value depends on the supply voltage of the driving PECL buffer:

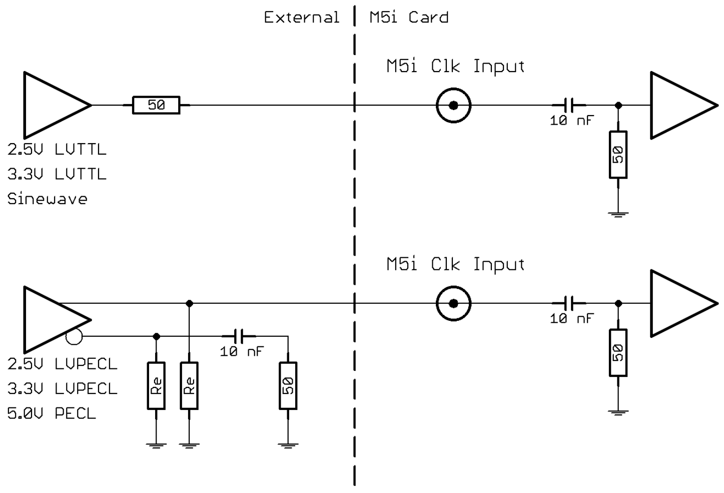


Image 83: electrical structure of clock inputs and potential interfacing circuits

V_{CC} - V_{EE}	2.5 V	3.3 V	5.0 V
R_E	~50 Ohm	~100 Ohm	~200 Ohm

Interfacing with clock output

The clock output of the M5i cards is AC-coupled, single-ended PECL type. The output swing of the M5i clock output is approximately 720 mV_{PP}.

Internal biased single-ended receivers

Because of the AC coupling of the M5i clock output, the signal must be properly re-biased for the receiver. Receivers that provide an internal re-bias only require the signal to be terminated to ground by a 50 Ohm resistor.

Differential (LV)PECL receivers

Differential receivers require proper re-biasing and likely a small minimum difference between the true and the inverting input to avoid ringing with open receiver inputs. Therefore a Thevenin-equivalent can be used, with receiver-type dependent values for R1, R2, R1' and R2'.

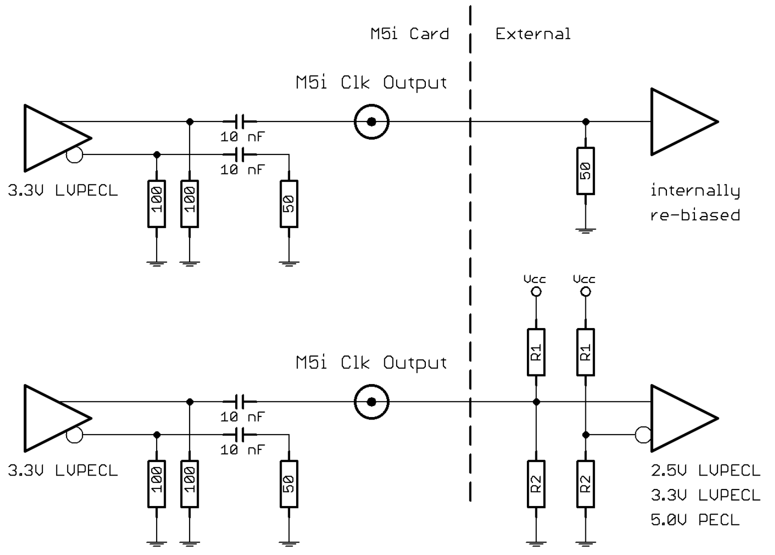


Image 84: electrical structure of clock outputs and potential interfacing circuits

Details on M5i cards status LED

Every M5i card has a two-colour status LED mounted within the multi-purpose I/O connector field on the card bracket.

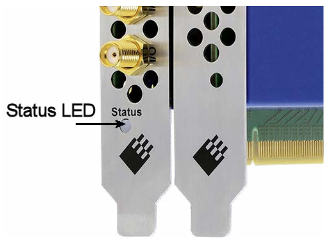


Image 85: location of status LED on front panel

This chapter explains the different colour codes and offers some possible solutions in case of an error condition.

Table 143: card status LED colour and blink coding

Condition	LED colour	Status	Solution
Off	Off	Card not powered	Power on the PC.
Error	Static: red	Power supply error or missing 12V PCIe power cable	Make sure that the 12V power cable is connected. If it is, restart the PC. In case that the error persists, please contact Spectrum support for further assistance.
	LED off	FPGA boot error	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.
	Single yellow pulses with 8 Hz repetition rate	PCI Express link training has not finished	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. 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.
	4 Yellow pulses with 8 Hz repetition rate	PCIe Express is Linked-up but PCIe error	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. 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 x16, Gen 3) 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: 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 16 lanes and/or the card is installed in a PCIe Gen 1 or Gen 2 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 144: 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.

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 slow 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 different 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 depending 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)

	PCI 33 MHz slot		PCI-X 66 MHz slot		PCI Express x1 slot	
Mode	**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)

	Notifysize 16 kByte		Notifysize 64 kByte		Notifysize 512 kByte		Notifysize 2048 kByte		Notifysize 4096 kByte	
Mode	**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 FIFO read**	**Notifysize 8 kByte FIFO read**	**Notifysize 16 kByte FIFO read**	**Notifysize 32 kByte FIFO read**	**Notifysize 64 kByte FIFO read**	**Notifysize 256 kByte FIFO read**	**Notifysize 1024 kByte FIFO read**	**Notifysize 2048 kByte FIFO read**	**Notifysize 4096 kByte 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 FIFO read**	**Notifysize 8 kByte FIFO read**	**Notifysize 16 kByte FIFO read**	**Notifysize 32 kByte FIFO read**	**Notifysize 64 kByte FIFO read**	**Notifysize 256 kByte FIFO read**	**Notifysize 1024 kByte FIFO read**	**Notifysize 2048 kByte FIFO read**	**Notifysize 4096 kByte 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
```

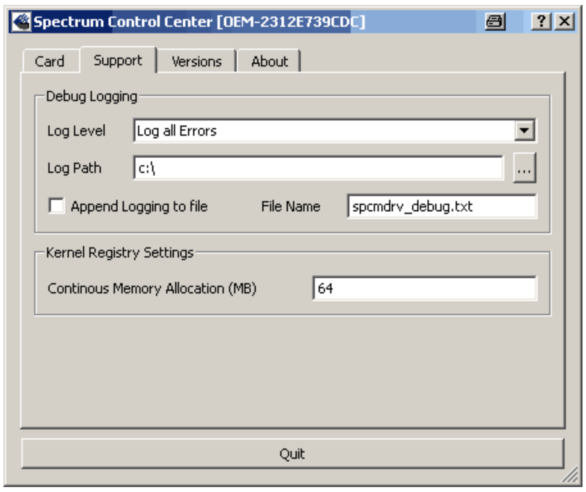
Using a continuous buffer of this size will need root privileges for the using program on most systems!

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.



Debug Logging
Log Level: Log all Errors
Log Path: c:\
☐ Append Logging to file
File Name: spcmdrv_debug.txt

Kernel Registry Settings
Continous Memory Allocation (MB): 64

Image 86: 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);
```

List of Figures

List of Tables

SPECTRUM
INSTRUMENTATION

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.

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