SPECTRUM
SYSTEMENTWICKLUNG MICROELECTRONIC GMBH

M2i.20xx
M2i.20xx-exp

fast 8 bit transient recorder,
A/D converter board
for PCI-X, PCI and PCI Express bus

Hardware Manual
Software Driver Manual

English version **November 27, 2008**

SPECTRUM SYSTEMENTWICKLUNG MICROELECTRONIC GMBH · AHRENSFELDER WEG 13-17 · 22927 GROSSHANSDORF · GERMANY
PHONE: +49 (0)4102-6956-0 · FAX: +49 (0)4102-6956-66 · E-MAIL: info@spec.de · INTERNET: http://www.spectrum-instrumentation.com

Introduction

Preface

This manual provides detailed information on the hardware features of your Spectrum instrumentation 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 LabVIEW or MATLAB. These drivers are provided by special order.

For any new information on the board as well as new available options or memory upgrades please contact our website http://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 PCI bus was first introduced in 1995. Nowadays it is the most common platform for PC based instrumentation boards. The very wide range of installations world-wide, especially in the consumer market, makes it a platform of good value. Its successor is the 2004 introduced PCI Express standard. In today's standard PC there are usually two to three slots of both standards available for instrumentation boards. Special industrial PCs offer up to a maximum of 20 slots. The common PCI/PCI-X bus with data rates of up to 133 MHz x 64 bit = 1 GByte/s per bus, is more and more replaced by the PCI Express standard with up to 4 GByte/s data transfer rate per slot. The Spectrum M2i boards are available in two versions, for PCI/PCI-X as well as for PCI Express. The 100% software compatible standards allow to combine both standards in one system with the same driver and software commands.

Within this document the name M2i is used as a synonym for both versions, either PCI/PCI-X or PCI Express. Only passages that differ concerning the bus version of the M2i.xxxx and M2i.xxxx-exp cards are mentioned separately. Also all card drawings will show the PCI/PCI-X version as example if no differences exist compared to the PCI Express version.

General Information

The 4 models of the M2i.20xx series are designed for the fast and high quality data acquisition. Every of the up to four input channels has its own A/D converter and it's own programmable input amplifier.

This allows to record signals with 8 bit resolution without any phase delay between them. The inputs can be selected to one of seven input ranges by software and could be programmed to compensate an input offset of ±400% of the input range. The extremely large on-board memory allows long time recording even with highest sample rates. A FIFO mode is also integrated on the board. This allows to record data continuously and to process it in the PC or to store it to hard disk.

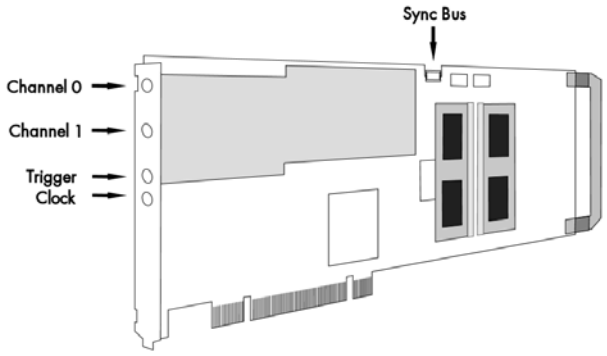
Several boards of the M2i.xxxx series may be connected together by the internal standard synchronisation bus to work with the same time base.

Application examples: Laboratory equipment, Supersonics, LDA/PDA, Radar, Spectroscopy, production test.

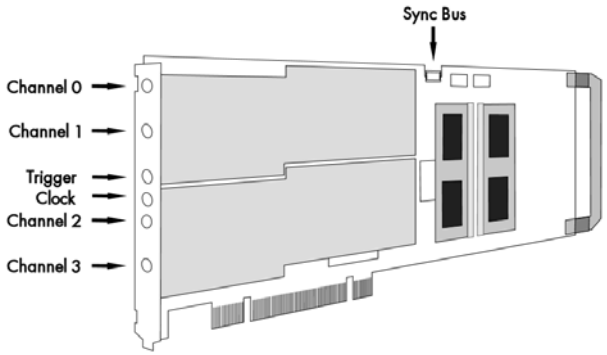
Different models of the M2i.20xx series

The following overview shows the different available models of the M2i.20xx series. They differ in the number of mounted acquistion modules and the number of available channels. You can also see the model dependant location of the output connectors.

- **M2i.2020**
- **M2i.2030**
- **M2i.2020-exp**
- **M2i.2030-exp**



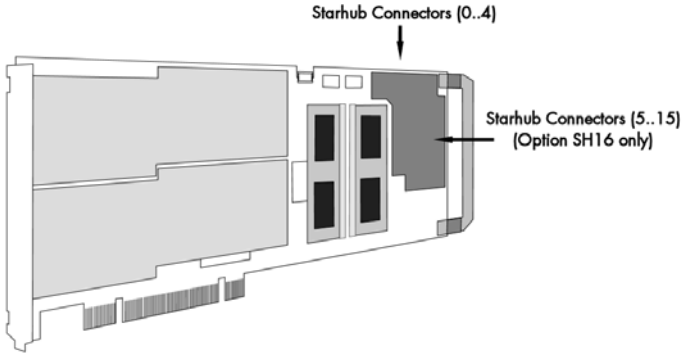
- **M2i.2021**
- **M2i.2031**
- **M2i.2021-exp**
- **M2i.2031-exp**



Additional options

Star-Hub

The star hub piggy-back module allows the synchronisation of up to 16 M2i cards. It is possible to synchronize cards of the same type with each other as well as different types.



Two different versions of the star-hub module are available. A minor one for synchronizing up to five boards of the M2i series, without the need for an additional system slot. The major version (option SH16) allows the synchronization of up to 16 cards with the need for an additional slot.

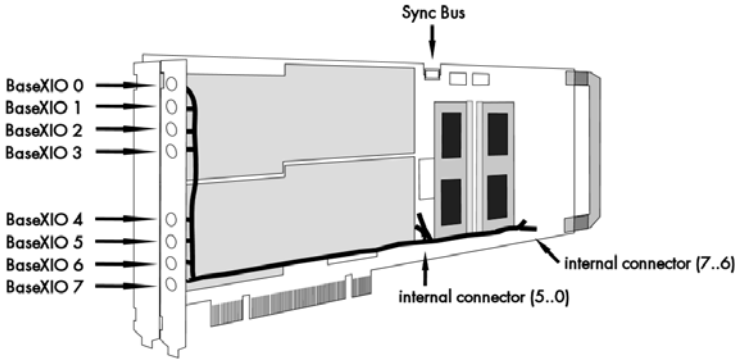
The module acts as a star hub for clock and trigger signals. Each board is connected with a small cable of the same length, even the master board. That minimizes the clock skew between the different cards. The figure shows the piggy-back module mounted on the base board schematically without any cables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

Any of the connected cards can be the clock master and the same or any other card can be the trigger master. All trigger modes that are available on the master card are also available if the synchronization star-hub is used.

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

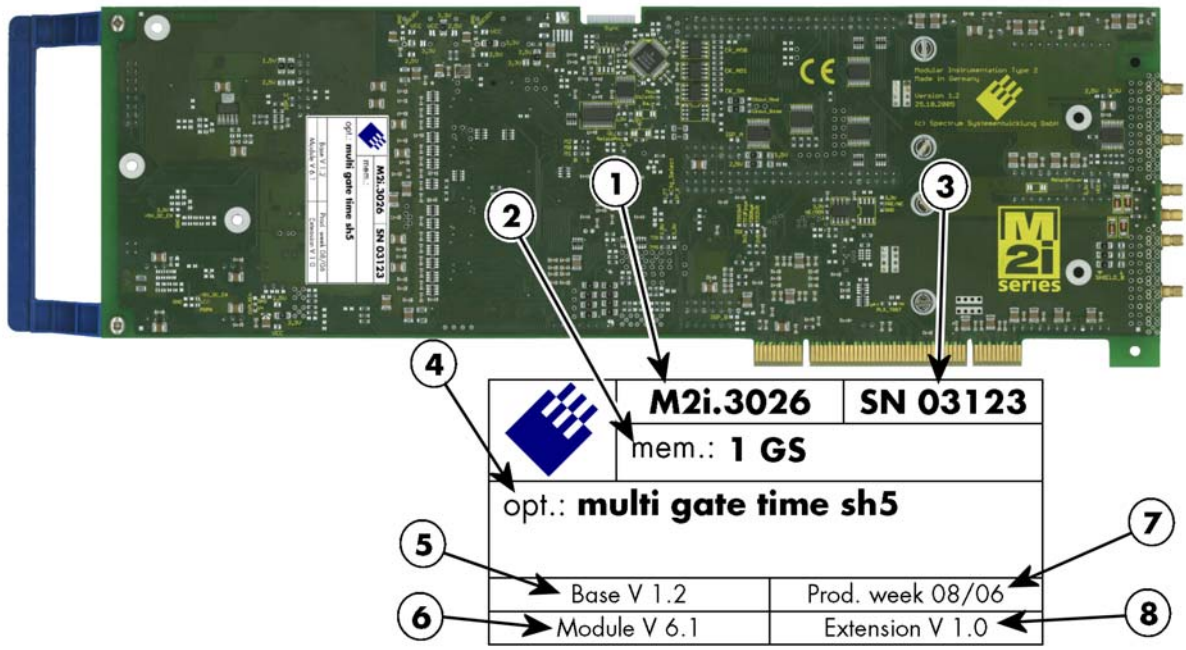
BaseXIO (asynchronous digital I/O)

The option BaseXIO is simple-to-use enhancement to the cards of the M2i series. It is possible to control a wide range of external instruments or other equipment by using the eight lines as asynchronous digital I/O. The BaseXIO option is useful if an external amplifier should be controlled, any kind of signal source must be programmed, if status information from an external machine has to be obtained or different test signals have to be routed to the board.
In addition to the I/O features, these lines are also for special functions. Two of the lines can be used as additional TTL trigger inputs for complex gated conditions, one line can be used as an reference time signal (RefClock) for the timestamp option.



The BaseXIO MMCX connectors are mounted on-board. To gain easier access, these lines are connected to an extra bracket, that holds eight SMB male connectors. For special purposes this option can also be ordered without the extra bracket and instead with internal cables.
The shown option is mounted exemplarily on a board with two modules and with the extra bracket. Of course you can also combine this option as well with a board that is equipped with only one module.

The Spectrum type plate



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 M2i for the PCI-X bus) and the model number.

(2) The size of the on-board installed memory in MSample or GSample. In this example there are 1 GS = 1024 MSample (2 GByte = 2048 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 the options Multiple recording, Gated Sampling, Timestamp and Star-Hub 5 are installed.

(5) The base card version, consisting of the hardware version (the part before the dot) and the firmware version (the part after the dot).

(6) The version of the analog/digital front-end module. Consisting of the hardware version (the part before the dot) and the firmware version (the part after the dot)

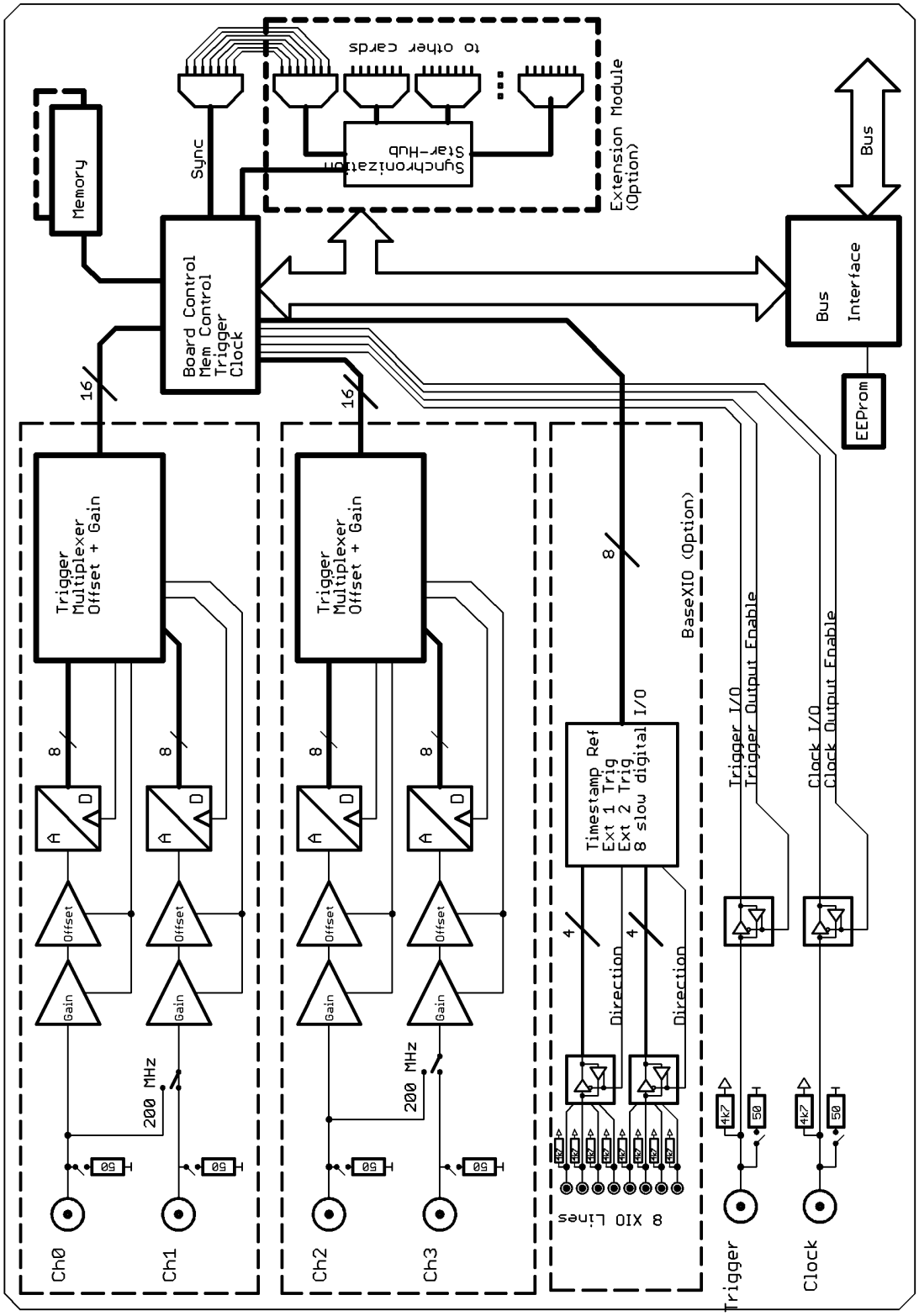
(7) The date of production, consisting of the calendar week and the year.

(8) The version of the extension module if one is installed. Consisting of the hardware version (the part before the dot) and the firmware version (the part after the dot). In our example we have the Star-Hub 5 extension module installed. Therefore the version of the extension module is filled on the type plate. If no extension module is installed this part is left open.

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

Hardware information

Block diagram



Technical Data

Analog Inputs

Resolution	8 bit
Differential non linearity (DNL)	≤ 0.5 LSB (ADC)
Integral non linearity (INL)	≤ 0.5 LSB (ADC)
Offset error	can be calibrated by user
Gain error	< 2% of current value
Programmable input offset	±400% of current input range
Crosstalk 1 MHz signal, 50 Ohm term	< -62 dB between any adjacent channels
Input signal with 50 Ohm termination	max 5 V rms
Analog Input impedance	50 Ohm / 1 MOhm || 25 pF
Over voltage protection (range ≤ ±0.5 V)	±5 V
Over voltage protection (range > ±0.5 V)	±50 V
Connector (analog and trigger/clock)	3 mm SMB male

Trigger

Multi, Gate: re-arming time	<4 Samples
Max Pretrigger at Multi, Gate, FIFO	16352 Samples as sum of all active channels
Trigger accuracy (≤100 MS/s)	1 Sample (internal or external trigger mode)
Trigger accuracy (>100 MS/s)	2 Samples (internal or external trigger mode)
Channel trigger resolution	8 bit
Trigger output delay	One positive edge after internal trigger event
External trigger type	3.3V LVTTL compatible (5V tolerant)
External trigger input	Low ≤ 0.8 V, High ≥ 2.0 V, ≥ 2 clock periods
External trigger maximum voltage	-0.5 V up to +5.7 V (internally clamped to 5.0V, 100 mA max. clamping current)
External trigger output levels	Low ≤ 0.4 V, High ≥ 2.4 V, TTL compatible
External trigger output drive strength	Capable of driving 50 ohm load

Software programmable parameters

Input Ranges	±50 mV, ±100 mV, ±200 mV, ±500 mV, ±1 V, ±2 V, ±5 V
Input impedance	50 Ohm / 1 MOhm
Input Offset	±400% of input range in steps of 1%
Clock mode	Int. PLL, int. quartz, ext. clock, ext. divided, ext. reference clock, sync
Clock impedance	50 Ohm / high impedance (> 4kOhm)
Trigger impedance	50 Ohm / high impedance (> 4kOhm)
Trigger mode	Channel, Extern, SW, Auto, Window, Pulse, Re-Arm, Or/And, Delay
Trigger level	8 bit resolution: 1/256 to 255/256 of input range
Trigger edge	Rising edge, falling edge or both edges
Trigger pulse width	0 to [64k - 1] samples in steps of 1 sample
Trigger delay	0 to [64k - 1] samples in steps of 1 sample
Memory depth	8 up to [installed memory / number of active channels] in steps of 8
Posttrigger	4 up to [8G - 4] samples in steps of 4
Multiple Recording segment size	8 up to[installed memory / 2 / active channels] in steps of 4
Multi / Gated pretrigger	0 up to [16k samples / number of active channels - 32]
ABA clock divider	1 up to [64k - 1] in steps of 1
Synchronization clock divider	2 up to [8k - 2] in steps of 2
Channel selection	Any 1, 2 or 4 channels (see manual for clock limits on the selections)

Clock

Internal clock range (PLL mode)	1 kS/s to max (see table below)
Internal clock accuracy	20 ppm
Internal clock: max. jitter in PLL mode	TBD
Internal clock: max. jitter in quartz mode	TBD
Internal clock setup granularity	≤1% of range (100M, 10M, 1M, 100k,...)
Internal clock setup granularity example	range 1M to 10M: stepsize ≤ 100k
Reference clock: external clock range	≥ 1.0 MHz and ≤ 125.0 MHz
External clock range	1 MS/s to max (see table below)
External clock delay to internal clock	5.4 ns
External clock type	3.3V LVTTL compatible
External clock input	Low ≤ 0.8 V, High ≥ 2.0 V, duty 45% - 55%
External clock maximum voltage	-0.5 V up to +3.8 V (internally clamped to 3.3V, 100 mA max. clamping current)
External clock output levels	Low ≤ 0.4 V, High ≥ 2.4 V, TTL compatible
External clock output drive strength	Capable of driving 50 ohm load

Environmental and Physical details

Dimension (PCB only)	312 mm x 107 mm (full PCI length)
Width (Standard or star-hub 5)	1 full size slot
Width (star-hub 16)	2 full size slots
Weight (depending on options/channels)	290g (2 ch) up to 420g (4 ch + sh)
Warm up time	10 minutes
Operating temperature	0°C - 50°C
Storage temperature	-10°C - 70°C
Humidity	10% to 90%

PCI / PCI-X specific details

PCI / PCI-X bus slot type	32 bit 33/66 MHz
PCI / PCI-X bus slot compatibility	32/64 bit, 33-133 MHz, 3,3 V and 5 V I/O

PCI EXPRESS specific details

PCIe slot type	x1
PCIe slot compatibility	x1/x4/x8/x16*

*Some x16 PCIe slots are for graphic cards only and can not be used for other cards.

Power consumption (max speed)	**PCI / PCI-X**		**PCI EXPRESS**	
	3,3 V	**5 V**	**3,3 V**	**12V**
M2i.20x0 (256 MS memory)	2,2 A	0,5 A	0,4 A	1,0
M2i.20x1 (256 MS memory)	2,8 A	0,8 A	0,4 A	1,2
M2i.2031 (4 GS memory), max power	3,9 A	0,8 A	0,4 A	2,0

Max channels / Star-Hub Option	**SH5**	**SH16**	**SSHS5**	**SSHS16**
M2i.2020/2030	10	32	170	542
M2i.2021/2031	20	64	340	1084

BaseXIO (Option)

BaseXIO Connector (extra bracket)	8 x SMB (8 x MMCX internal)
BaseXIO input	TTL compatible: Low ≤ 0.8 V, High ≥ 2.0 V
BaseXIO input maximum voltage	-0.5 V up to +5.5 V
BaseXIO output levels	TTL compatible: Low ≤ 0.4 V, High ≥ 2.4 V
BaseXIO output drive strength	32 mA maximum current

Certifications, Compliances, Warranty

EMC Immunity	Compliant with CE Mark
EMC Emission	Compliant with CE Mark
Product warranty	2 years starting with the day of delivery
Software and firmware updates	Life-time, free of charge

Dynamic Parameters

	M2i.2020		M2i.2021		M2i.2030		M2i.2031	
max internal clock	50 MS/s		50 MS/s		200 MS/s		200 MS/s	
max external clock	50 MS/s		50 MS/s		100 MS/s		100 MS/s	
-3 dB bandwidth ±50 mV	DC to 25 MHz		DC to 25 MHz		DC to 60 MHz		DC to 60 MHz	
-3 dB bandwidth ≥ ±100 mV	DC to 25 MHz		DC to 25 MHz		DC to 90 MHz		DC to 90 MHz	
Zero noise level (≤ ±100 mV)	≤ 0,6 LSB		≤ 0,9 LSB		≤ 1,5 LSB		≤ 2.0 LSB	
Zero noise level (> ±100 mV)	≤ 0,6 LSB		≤ 0,7 LSB		≤ 1.3 LSB		≤ 1.5 LSB	
Test - sampling rate	50 MS/s		50 MS/s		100 MS/s		100 MS/s	
Test signal frequency	1 MHz	4 MHz	1 MHz	4 MHz	1 MHz	9 MHz	1 MHz	9 MHz
SNR (typ)	47.5 dB	47.5 dB	46.8 dB	46.5 dB	45.3 dB	45.0 dB	45.0 dB	44.5 dB
THD (typ)	-56.0 dB	-55.5 dB	-56.0 dB	-55.5 dB	-51.5 dB	-49.5 dB	-49.5 dB	-49.5 dB
SFDR (typ), excl. harm.	61.3 dB	61.0 dB	60.3 dB	60.1 dB	59.0 dB	57.0 dB	59.0 dB	57.0 dB
ENOB (based on SNR)	7.6 bit	7.6 bit	7.5 bit	7.4 bit	7.2 bit	7.2 bit	7.2 bit	7.2 bit
ENOB (based on SINAD)	7.5 bit	7.5 bit	7.4 bit	7.3 bit	7.1 bit	7.0 bit	7.1 bit	7.0 bit

Dynamic parameters are measured at ± 1 V input range (if no other range is stated) and 50 Ohm 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 of the specified frequency with > 99% amplitude. 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. For a detailed description please see application note 002.

Order Information

PCI/PCI-X

Order no.	Standard mem	1 channel	2 channels	4 channels
M2i.2020	256 MByte	50 MS/s	50 MS/s	
M2i.2021	256 MByte	50 MS/s	50 MS/s	50 MS/s
M2i.2030	256 MByte	200 MS/s	100 MS/s	
M2i.2031	256 MByte	200 MS/s	200 MS/s	100 MS/s

PCI Express

Order no.	Standard mem	1 channel	2 channels	4 channels
M2i.2020-exp	256 MByte	50 MS/s	50 MS/s	
M2i.2021-exp	256 MByte	50 MS/s	50 MS/s	50 MS/s
M2i.2030-exp	256 MByte	200 MS/s	100 MS/s	
M2i.2031-exp	256 MByte	200 MS/s	200 MS/s	100 MS/s

Memory

Order no.	Option
M2i.xxxx-512MB	Memory upgrade to 512 MB of total memory
M2i.xxxx-1GB	Memory upgrade to 1 GB of total memory
M2i.xxxx-2GB	Memory upgrade to 2 GB of total memory
M2i.xxxx-4GB	Memory upgrade to 4 GB of total memory

Options

Order no.	Option
M2i.xxxx-mr	Option Multiple Recording
M2i.xxxx-mgt	Option pack including Multiple Recording, Gated Sampling, Timestamp
M2i.xxxx-mgtab	Option pack including Multiple Recording, Gated Sampling, Timestamp, ABA mode
M2i.xxxx-SH5 (1)	Synchronization Star-Hub for up to 5 cards, only 1 slot width
M2i.xxxx-SH16 (1)	Synchronization Star-Hub for up to 16 cards
M2i.xxxx-SSHM (1)	System-Star-Hub Master for up to 15 cards in the system and up to 17 systems, sync cables included
M2i.xxxx-SSHS5 (1)	System-Star-Hub Slave for up to 5 cards in one system, all sync cables included
M2i.xxxx-SSHS16 (1)	System-Star-Hub Slave for up to 16 cards in one system, all sync cables included
M2i.xxxx-bxio	Option BaseXIO: 8 digital I/O lines usable as asynchronous I/O, timestamp ref-clock and additional external trigger lines, additional bracket with 8 SMB connectors
M2i-upgrade	Upgrade for M2i.xxxx: later installation of option -dig or -bxio

Cables

Order no.	Option
Cab-3f-9m-80	Adapter cable SMB female to BNC male, 80 cm
Cab-3f-9f-80	Adapter cable SMB female to BNC female, 80 cm
Cab-3f-3f-80	Adapter cable SMB female to SMB female, 80 cm
Cab-3f-9m-200	Adapter cable SMB female to BNC male, 200 cm
Cab-3f-9f-200	Adapter cable SMB female to BNC female, 200 cm
Cab-3f-3f-200	Adapter cable SMB female to SMB female, 200 cm
Cab-3f-9f-5	Adapter cable SMB female to BNC female, 5 cm (short cable especially for oscilloscope probes)

Drivers

Order no.	Option
M2i.xxxx-ml	MATLAB driver for all M2i cards
M2i.20xx-lv	LabVIEW driver for all M2i.20xx cards
M2i.20xx-dl	DASYLab driver for all M2i.20xx cards
M2i.20xx-vee	Agilent VEE driver for all M2i.20xx cards

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

Hardware Installation

System Requirements

All Spectrum M2i.xxxx instrumentation cards are compliant to the PCI standard and require in general one free full length slot. This can either be a standard 32 bit PCI legacy slot, a 32 bit or a 64 bit PCI-X slot. Depending on the installed options additional free slots can be necessary.

All Spectrum M2i.xxxx-exp instrumentation cards are compliant to the PCI Express 1.0 standard and require in general one free full length PCI Express slot. This can either be a x1, x4, x8 or x16 slot. Some x16 PCIe slots are for the use of graphic cards only and can not be used for other cards. Depending on the installed options additional free slots can be necessary.

Warnings

ESD Precautions

The boards of the M2i.xxxx series contain electronic components that can be damaged by electrostatic discharge (ESD).

Before installing the board in your system or even before touching it, it is absolutely necessary to bleed of any electrostatic electricity.

Cooling Precautions

The boards of the M2i.xxxx series operate with components having very high power consumption at high speeds. For this reason it is absolutely required to cool this board sufficiently. It is strongly recommended to install an additional cooling fan producing a stream of air across the boards surface. In most cases professional PC-systems are already equipped with sufficient cooling power. In that case please make sure that the air stream is not blocked.

Sources of noise

The boards of the M2i.xxxx series should be placed far away from any noise producing source (like e.g. the power supply). It should especially be avoided to place the board in the slot directly adjacent to another fast board (like the graphics controller).

Installing the board in the system

Installing a single board without any options

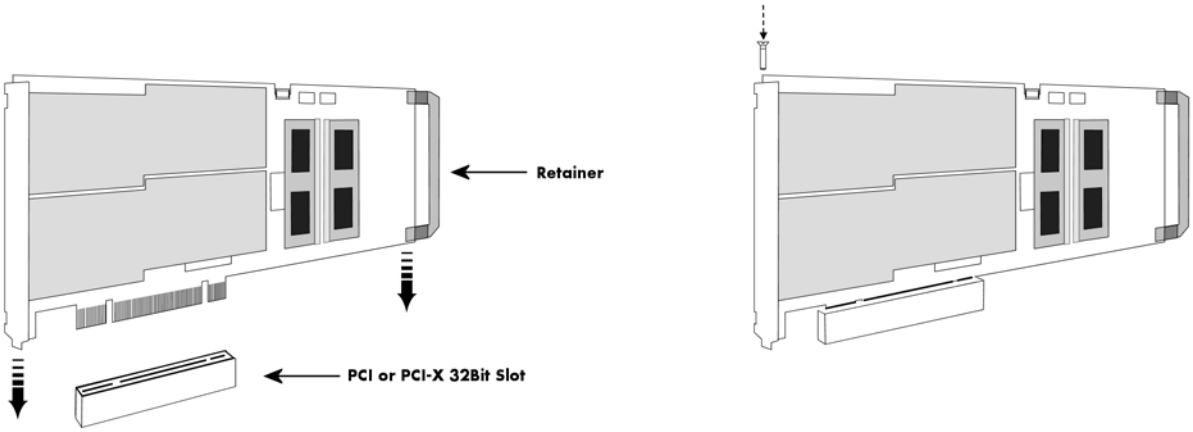
Before installing the board you first need to unscrew and remove the dedicated blind-bracket usually mounted to cover unused slots of your PC. Please keep the screw in reach to fasten your Spectrum card afterwards. All Spectrum cards require a full length PCI, PCI-X slot (either 32Bit or 64Bit) or PCI Express slot (either x1, x4, x8 or x16) with a track at the backside to guide the board by it's retainer. Now insert the board slowly into your computer. This is done best with one hand each at both fronts of the board.

While inserting the board take care not to tilt the retainer in the track.

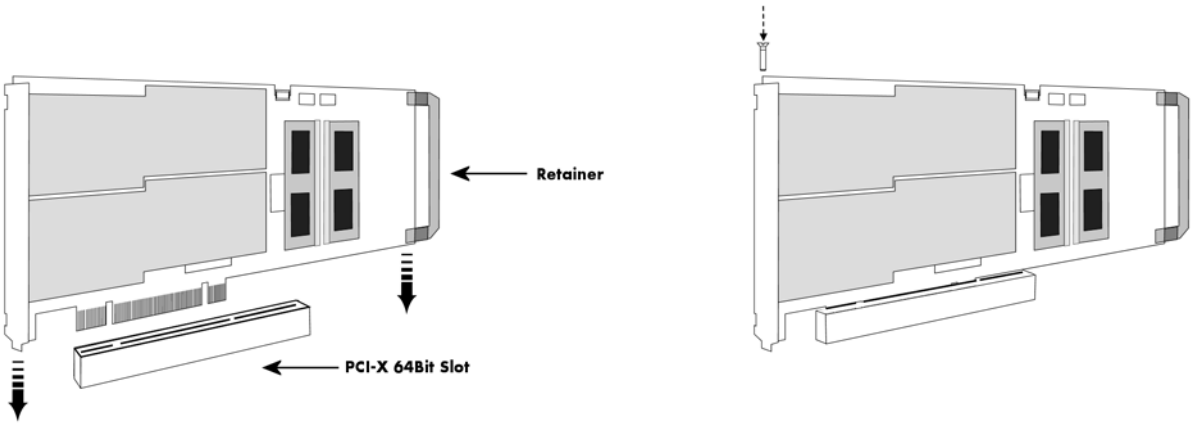
Please be very carefully when inserting the board in the slot, as most of the mainboards are mounted with spacers and therefore might be damaged if they are exposed to high pressure.

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

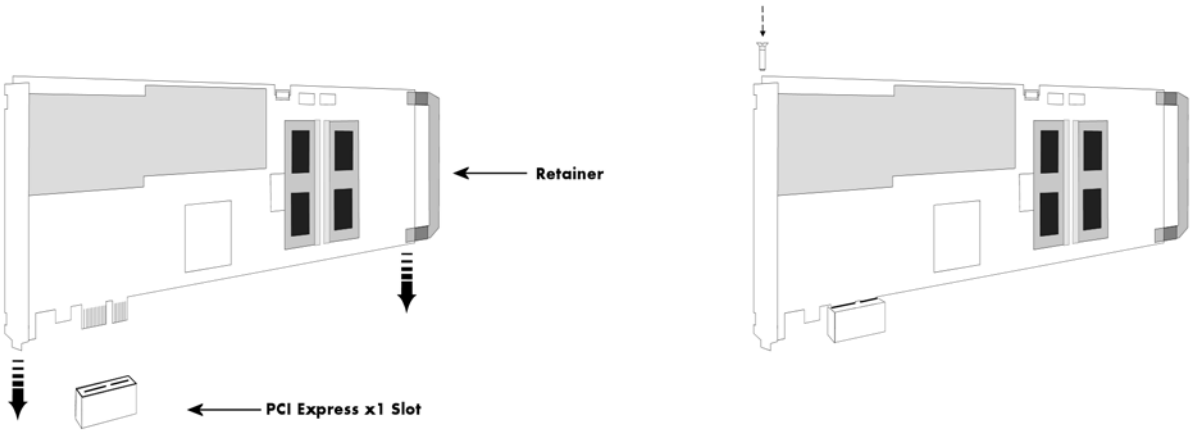
Installing the M2i.xxxx PCI/PCI-X card in a 32 bit PCI/PCI-X slot



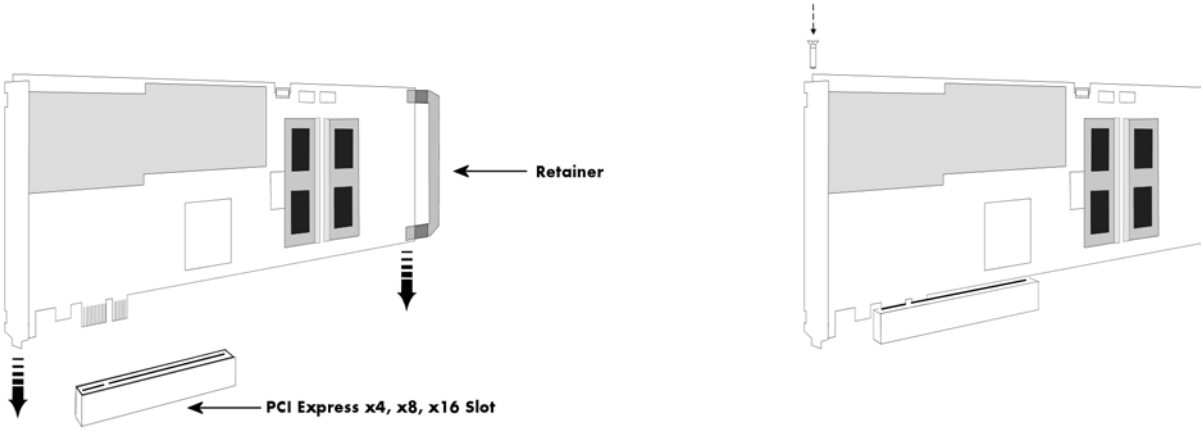
Installing the M2i.xxxx PCI/PCI-X card in a 64 bit PCI/PCI-X slot



Installing the M2i.xxxx-exp PCI Express card in a PCIe x1 slot



Installing the M2i.xxxx-exp PCI Express card in a PCIe x4, x8 or x16 slot



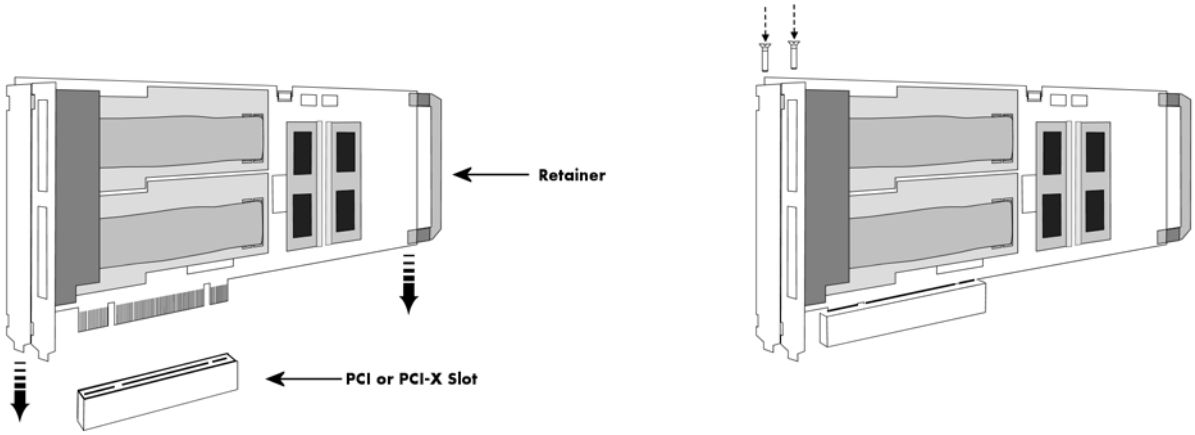
Installing a board with digital inputs/outputs

Before installing the board 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 board and the extra bracket afterwards. All Spectrum boards require a full length PCI slot with a track at the backside to guide the board by it's retainer. Now insert the board and the extra bracket slowly into your computer. This is done best with one hand each at both fronts of the board.

While inserting the board take care not to tilt the retainer in the track.

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

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



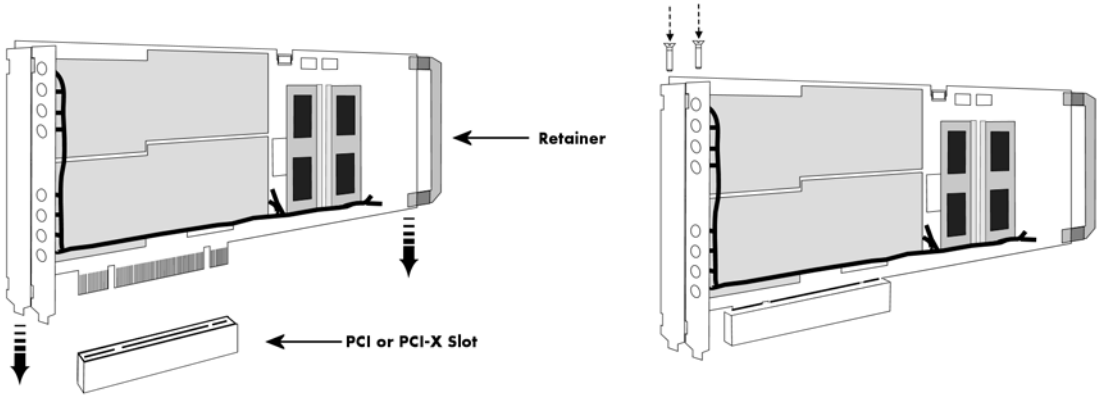
Installing a board with option BaseXIO

Before installing the board 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 board and the extra bracket afterwards. All Spectrum boards require a full length PCI slot with a track at the backside to guide the board by it's retainer. Now insert the board and the extra bracket slowly into your computer. This is done best with one hand each at both fronts of the board.

While inserting the board take care not to tilt the retainer in the track.

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

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



Installing multiple boards synchronized by star-hub

Hooking up the boards

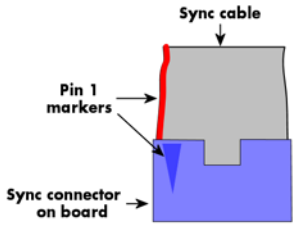
Before mounting several synchronized boards for a multi channel system into the PC you can hook up the cards with their synchronization cables first. If there is enough space in your computer's case (e.g. a big tower case) you can also mount the boards first and hook them up afterwards. Spectrum ships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

Only use the included flat ribbon cables.

All of the cards, including the one that carries the star-hub piggy-back module, must be wired to the star-hub as the figure is showing as an example for three synchronized boards.

It does not matter which of the available connectors on the star-hub module you use for which board. The software driver will detect the types and order of the synchronized boards automatically. The figure shows the three cables mounted on the major SH16 star-hub to achieve a better visibility. When you use the minor SH5 version, only the connectors on the upper side of the star-hub piggy-back module are available (see figure for details on the star-hub connector locations).

As some of the synchronization cables are not secured against wrong plugging you should take care to have the pin 1 markers on the multiple connectors and the cable on the same side, as the figure on the right is showing.



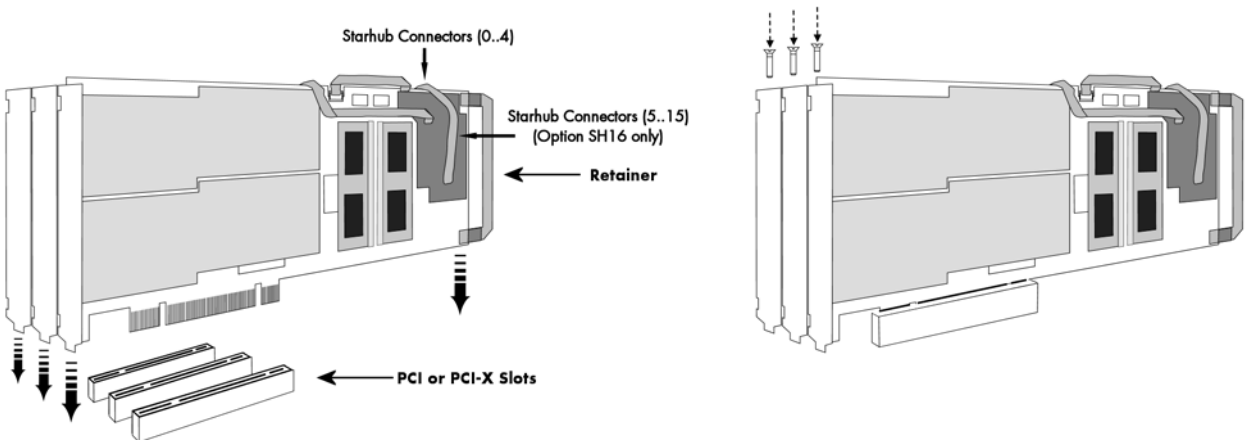
Mounting the wired boards

Before installing the cards you first need to unscrew and remove the dedicated blind-brackets usually mounted to cover unused slots of your PC. Please keep the screws in reach to fasten your Spectrum cards afterwards. All Spectrum boards require a full length PCI slot with a track at the backside to guide the card by it's retainer. Now insert the cards slowly into your computer. This is done best with one hand each at both fronts of the board. Please keep in mind that the board carrying the star-hub piggy-back module might require the width of two slots, when the major SH16 version is used.

While inserting the cards take care not to tilt the retainers in the tracks.

Please be very carefully 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.

After inserting all cards fasten the screws of all brackets carefully, without overdoing. The figure shows an example of three cards with two installed modules each.



Software Driver Installation

Before using the board a driver must be installed that matches the operating system. The installation is done in different ways depending on the used operating system. The driver that is on CD supports all cards of the M2i series. That means that you can use the same driver for all cards of this family.

Interrupt Sharing

This board uses a PCI interrupt for DMA data transfer and for controlling the FIFO mode. The used interrupt line is allocated by the PC BIOS at system start and is normally depending on the selected slot. Because there is only a limited number of interrupt lines available on the PCI bus it can happen that two or more boards must use the same interrupt line. This so called interrupt sharing must be supported by all drivers of the participating equipment.

Most available drivers and also the Spectrum driver for your board can manage interrupt sharing. But there are also some drivers on the market that can only use one interrupt exclusively. If this equipment shares an interrupt with the Spectrum board, the system will hang up if the second driver is loaded (the time is depending on the operating system).

If this happens it is necessary to reconfigure the system in that way that the critical equipment has an exclusive access to an interrupt.

On most systems the BIOS shows a list of all installed PCI boards with their allocated interrupt lines directly after system start. You have to check whether an interrupt line is shared between two boards. Some BIOS allow the manual allocation of interrupt lines. Have a look in your mainboard manual for further information on this topic.

Because normally the interrupt line is fixed for one PCI slot it is simply necessary to use another slot for the critical board to force a new interrupt allocation. You have to search a configuration where all critical boards have only exclusive access to one interrupt.

Depending on the system, using the Spectrum board with a shared interrupt may degrade performance a little. Each interrupt needs to be checked by two drivers. For this reason when using time critical FIFO mode even the Spectrum board should have an exclusively access to one interrupt line.

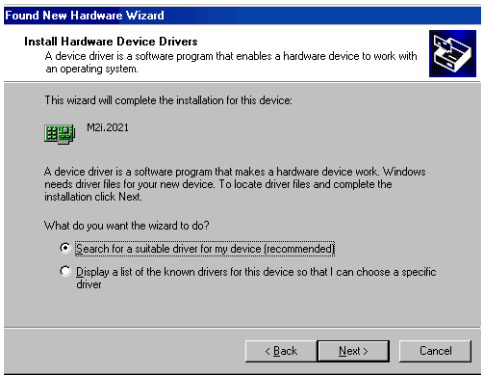
Windows 2000

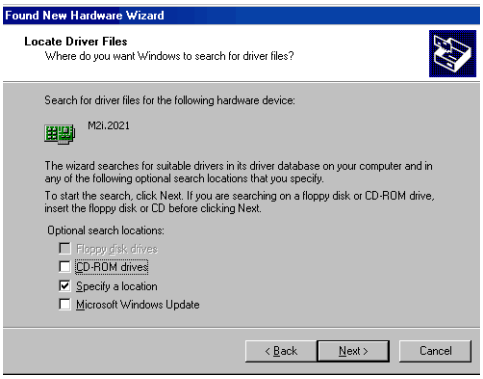
Installation

When installing the board in a Windows 2000 system the Spectrum board will be recognized automatically on the next start-up.

The system offers the direct installation of a driver for the board.

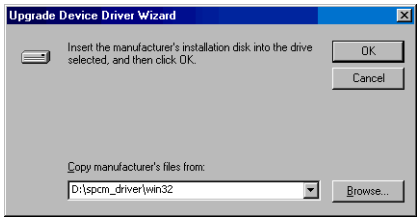
Let Windows search automatically for the best driver for your system.





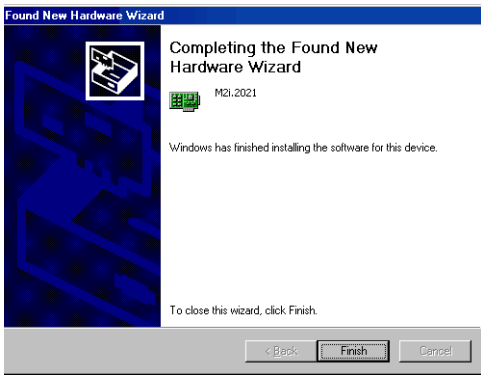
Select the CD that was delivered with the board as installation source. The driver files are located on CD in the directory \spcm_driver\win32.

The hardware assistant shows you the exact board type that has been found like the M2i.2021 in the example.

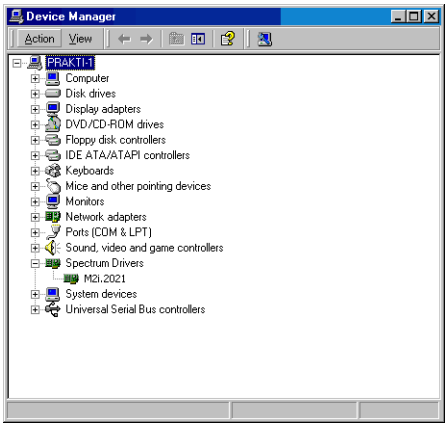


The drivers can be used directly after installation. It is not necessary to restart the system. The installed drivers are linked in the device manager.

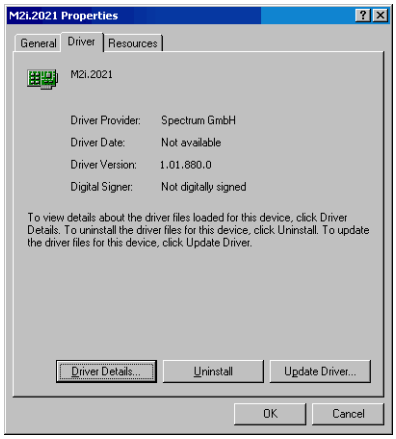
Below you'll see how to examine the driver version and how to update the driver with a newer version.



Version control



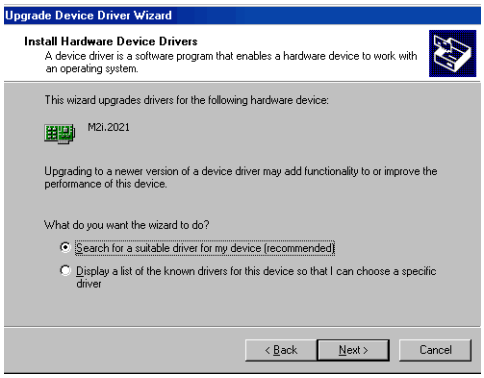
If you want to check which driver version is installed in the system this can be easily done in the device manager. Therefore please start the device manager from the control panel and show the properties of the installed driver.
On the property page Windows 2000 shows the date and the version of the installed driver.
After clicking the driver details button the detailed version information of the driver is shown. This information is also available through the control center.

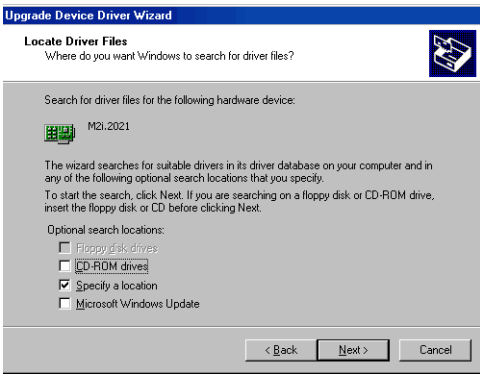


Driver - Update

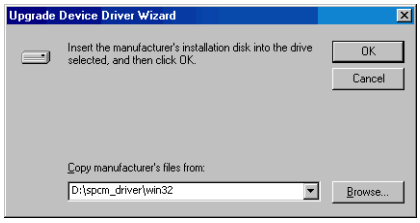
If a new driver version should be installed no Spectrum board is allowed to be in use by any software. So please stop and exit all software that could access the boards.

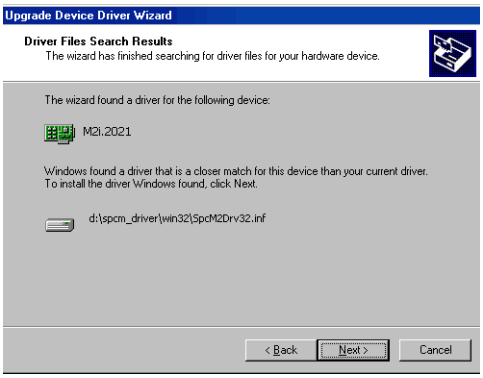
A new driver version is directly installed from the device manager. Therefore please open the properties page of the driver as shown in the section before. As next step click on the update driver button and follow the steps of the driver installation in a similar way to the previous board and driver installation.





Please select the path where the new driver version was unzipped to. If you've got the new driver version on CD please select the \spcm_driver\win32 path on the CD containing the new driver version.





The new driver version can be used directly after installation without restarting the system. Please keep in mind to update the driver of all installed Spectrum boards.

Windows XP 32/64 Bit

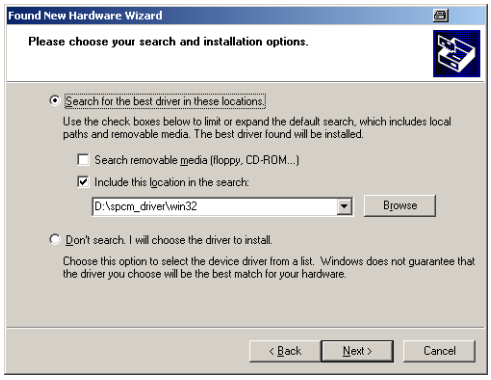
Installation

When installing the board in a Windows XP system the Spectrum board will be recognized automatically on the next start-up.

The system offers the direct installation of a driver for the board.

Do not let Windows automatically search for the best driver, because sometimes the driver will not be found on the CD. Please take the option of choosing a manual installation path instead.

Allow Windows XP to search for the most suitable driver in a specific directory. Select the CD that was delivered with the board as installation source. The driver files are located on CD in the directory \spcm_drv\win32 for Windows XP 32 Bit or \spcm_drv\win64 for Windows XP 64 Bit.



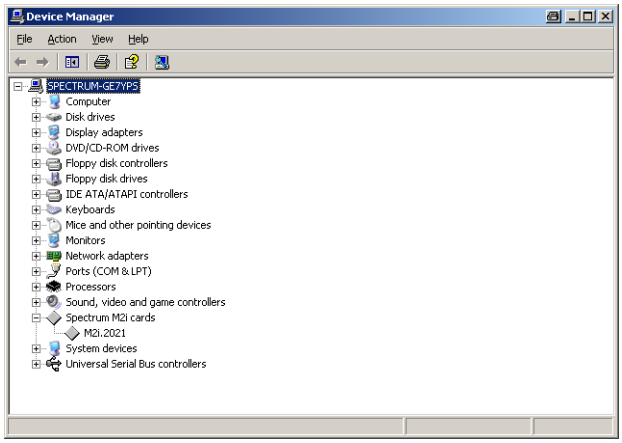
The hardware assistant shows you the exact board type that has been found like the M2i.2021 in the example.

The drivers can be used directly after installation. It is not necessary to restart the system. The installed drivers are linked in the device manager.

Below you'll see how to examine the driver version and how to update the driver with a newer version.

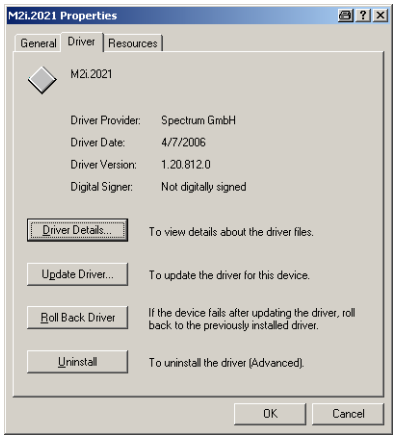
Version control

If you want to check which driver version is installed in the system this can be easily done in the device manager. Therefore please start the device manager from the control panel and show the properties of the installed driver.



On the property page Windows XP shows the date and the version of the installed driver.

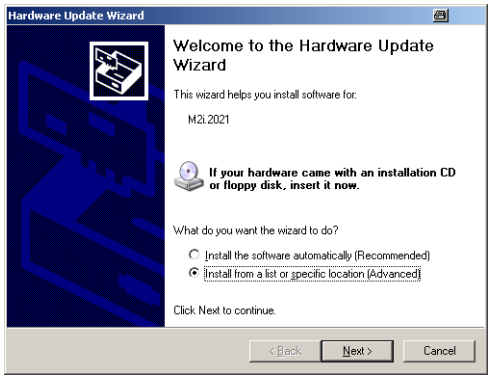
After clicking the driver details button the detailed version information of the driver is shown. This information is also available through the Spectrum Control Center.



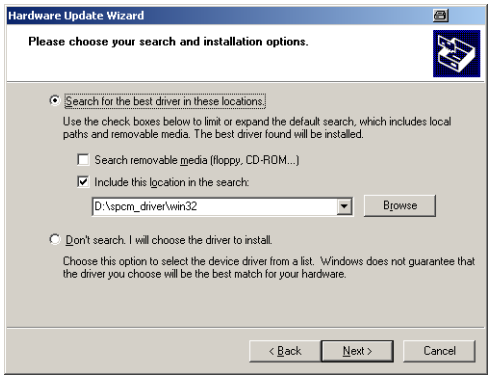
Driver - Update

If a new driver version should be installed no Spectrum board is allowed to be in use by any software. So please stop and exit all software that could access the boards.

A new driver version is directly installed from the device manager. Therefore please open the properties page of the driver as shown in the section before. As next step click on the update driver button and follow the steps of the driver installation in a similar way to the previous board and driver installation.



Please select the path where the new driver version was unzipped to. If you've got the new driver version on CD please select the \spcm_drv\win32 path on the CD containing the new driver version.



The new driver version can be used directly after installation without restarting the system. Please keep in mind to update the driver of all installed Spectrum boards.

Linux

Overview

The Spectrum M2i cards are delivered with Linux drivers suitable for Linux installations based on kernel 2.4 and kernel 2.6, 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 nearly every case necessary to have a directly matching kernel driver to run it on a specific system.

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 following pre-compiled kernel driver modules. This list may have been enhanced in between since printing of the manual. If your specific Linux distribution is not in this list please download the latest drivers from our website.

Distribution	Kernel Version	Processor	Width	Distribution	Kernel Version	Processor	Width
Suse 9.0	2.4.21	single only	32 bit	Redhat 9.0	2.4.20	single and smp	32 bit
Suse 9.1	2.6.4	single and smp	32 bit	Fedora Core 3	2.6.9	single and smp	32 bit
Suse 9.3	2.6.11	single and smp	32 bit	Fedora Core 4	2.6.11	single and smp	32 bit
Suse 10.0	2.6.13	single only	32 bit and 64 bit	Fedora Core 5	2.6.15	single and smp	32 bit and 64 bit
Suse 10.1	2.6.16	single only	32 bit and 64 bit	Fedora Core 6	2.6.18	single and smp	32 bit and 64 bit
Suse 10.2	2.6.18	single and smp	32 bit and 64 bit	Fedora Core 7	2.6.21	single and smp	32 bit and 64 bit
Suse 10.3	2.6.22	single and smp	32 bit and 64 bit	Fedora Core 8	2.6.23	single and smp	32 bit and 64 bit
Suse 11.0	2.6.25	single and smp	32 bit and 64 bit	Fedora Core 9	2.6.25	single and smp	32 bit and 64 bit
Debian Sarge	2.4.27	single	32 bit	Debian Sarge	2.6.8	single	32 bit
				Debian Etch	2.6.18	single and smp	32 bit and 64 bit

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

Standard Driver Installation

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.

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 install the kernel module and some helper scripts to a given directory. If you do not specify a directory it will use your home directory as destination. It is possible to move the installed driver files later to any other directory.

The script will give you a list of matching kernel modules. Therefore it checks for the system width (32 bit or 64 bit) and the processor (single or smp). The script will only show matching kernel modules. Select the kernel module matching your system. The script will then do the following steps:

- copy the selected kernel module to the install directory (spcm.o or spcm.ko)
- copy the helper scripts to the install directory (spcm_start.sh and spc_end.sh)
- copy and rename the matching library to /usr/lib (/usr/lib/libspcm_linux.so)

Udev support

Starting with driver version 1.23 build 1576 (Linux kernel driver version 1.05 build 1572) the driver natively supports udev. 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

Starting the driver can be done with the spcm_start.sh script that has been placed in the install directory. If udev is installed the script will only load the driver. If no udev is installed the start script will load the driver and make the required device nodes /dev/spcm0... for accessing the drivers. Please keep in mind that you need root rights to load the kernel module and to make the device nodes!

Using the dedicated start script makes sure that the device nodes are matching your system setup even if new hardware and drivers have been added in between. Background: when loading the device driver it get's assigned a „major" number that is used to access this driver. All device nodes point to this major number instead of the driver name. The major numbers are assigned first come first served. This means that installing new hardware may result in different major numbers on the next system start.

Get first driver info

After the driver has been loaded successfully some information about the installed boards can be found in the /proc/spcm_cards file. Some basic information from the on-board EEProm is listed for every card.

```
cat /proc/spcm_cards
```

Stop the driver

You may want to unload the driver and clean up all device nodes. This can be done using the spcm_end.sh script that has also been placed in the install directory

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 spcm_end.sh script.

Compilation of kernel driver sources (option)

The driver sources are only available for existing customers on special request and against a signed NDA. 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 make_spcm_linux_kerneldrv.sh

This script will examine the type of system you use and compile the kernel with the correct settings. If using a kernel 2.4 the makefile expects two symbolic links in your system:

- /usr/src/linux pointing to the correct kernel source directory
- /usr/src/linux/.config pointing to the currently used kernel configuration

The compile script will then automatically call the install script and install the just compiled kernel module in your home directory. The rest of the installation procedure is similar as explained above.

Library

The kernel driver module only contains the basic hardware functions. The main part of the driver is located inside a dynamically loadable library that is delivered with the driver. This library is available in 3 different versions:

- spcm_linux_32bit_stdc++5.so - supporting libstdc++.so.5 on 32 bit systems
- 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

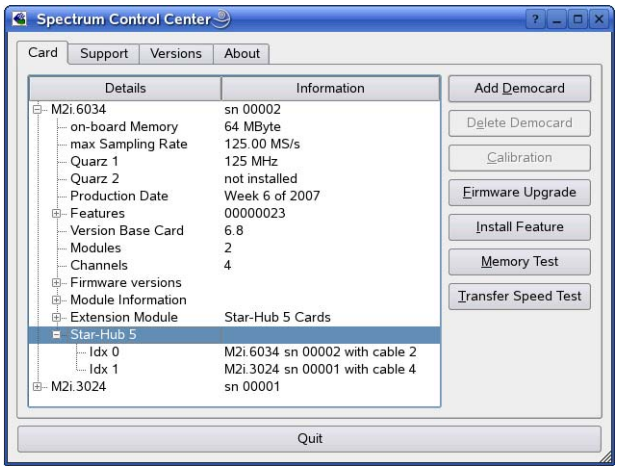
The matching version is installed automatically in the /usr/lib directory by the install script. The library is renamed for easy access to libspcm_linux.so. To access the driver library one must include the library in the compilation:

```
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.

Control Center

The Spectrum Control Center is also available for Linux and need 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
- Qt 4 (version 4.1 or higher)
- expat
- freetype
- fontconfig
- libpng
- libspcm_linux (the Spectrum linux driver library)

Qt

Qt is an universal cross platform class library. The Spectrum Control Center is based on that library. The Qt library is available for newer Linux distributions as pre-compiled modules and can be downloaded separately. To avoid any version conflicts the needed Qt libraries for the Control Center are included in the rpm in the correct version. The libraries are installed automatically under /usr/lib/qt4-spectrum and therefore don't conflict with any pre-installed Qt libraries.

Installation

Use the supplied script install_control_center.sh found in the driver section of the CD by typing as user with root rights inside a terminal window:

```
cd /mnt/cdrom/spcm_driver/linux/spcm_control_center
sh install_control_center.sh
```

The script will install the appropriate package for your distribution, called spcmcontrol-{Version}.rpm or spcmcontrol-{Version}.deb respectively. The Control Center is installed under KDE/Gnome 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 libraries to /usr/lib/qt4-spectrum. Copy the .desktop file to /usr/share/applications. Run ldconfig to update your systems library cache. Finally you can run spcm_control.

Troubleshooting

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

```
spcm_control: error while loading shared libraries: libQtGui.so.4: 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)
libQtCore.so.4 => not found
libz.so.1 => /lib/libz.so.1 (0x402a9000)
libdl.so.2 => /lib/libdl.so.2 (0x402ba000)
libpthread.so.0 => /lib/tls/libpthread.so.0 (0x402be000)
libstdc++.so.5 => /usr/lib/libstdc++.so.5 (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.

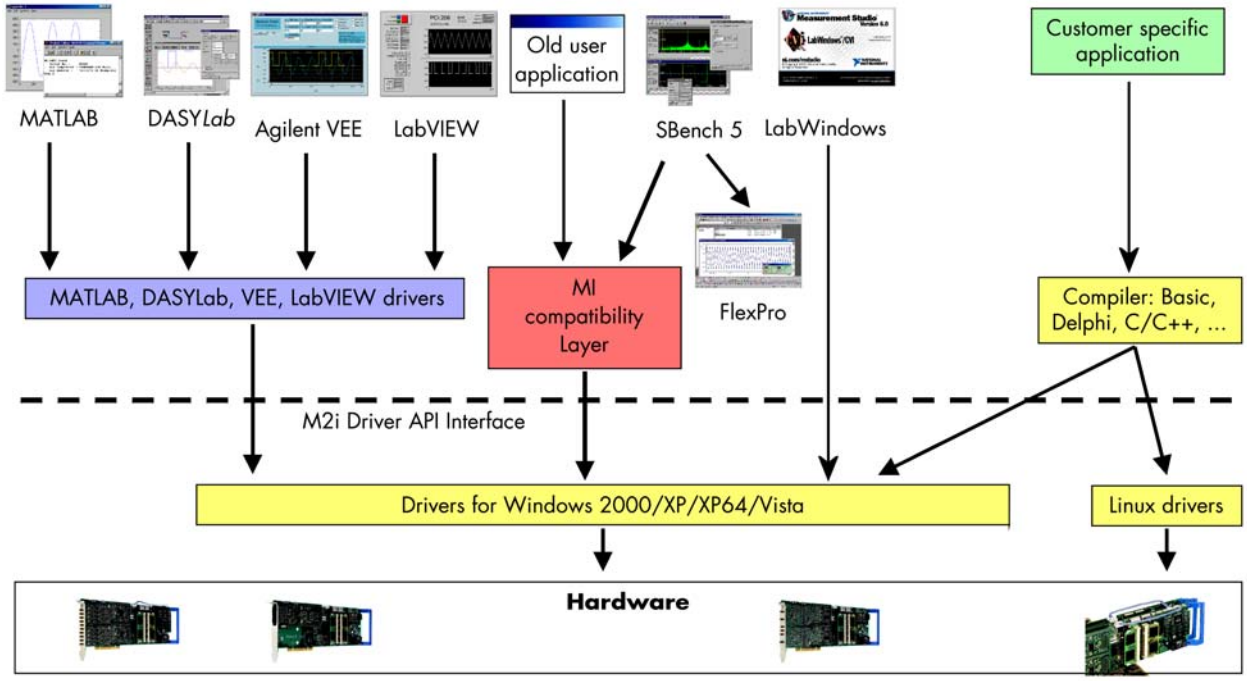
If the libspcm_linux.so is quoted as missing please make sure that you have installed the card driver properly before. If any non-Qt library is stated as missing please install the matching package of your distribution.

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, DASYLab or VEE an additional manual is required that is delivered with the ordered driver option.

Software Overview



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 optional drivers for third-party products like LabVIEW or DASYLab are also based on this API. The special functionality of these drivers is not subject of this document and is described with separate manuals delivered with the particular driver option.

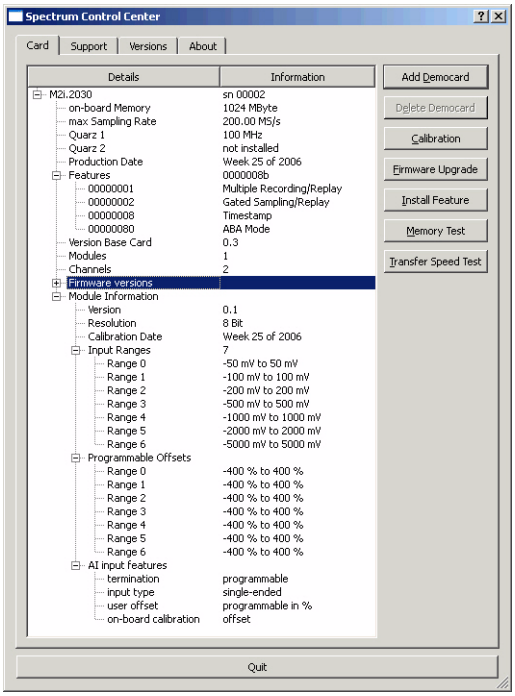
Card Control Center

A special card control center is installed together with the Spectrum M2i driver. This control center is available under Windows as a system setup DLL (*.cpl) and can be accessed directly from the control panel. The different functions of the Spectrum card control center are explained in detail in the following passages.

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 it's 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 by 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.

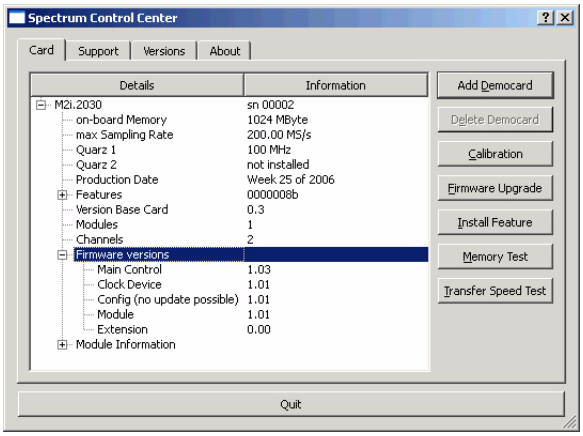


Firmware information

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



Driver information

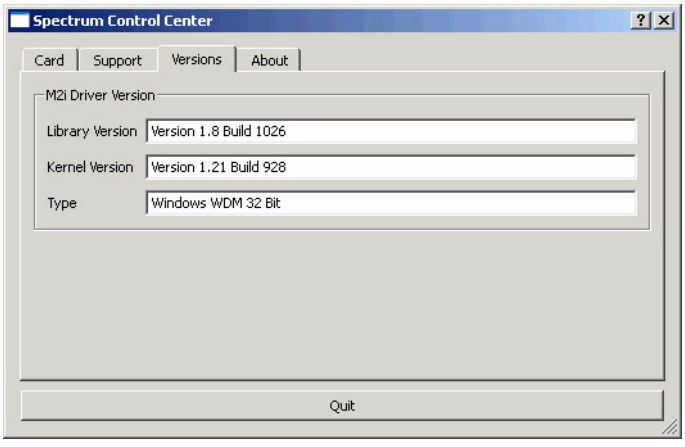
The Spectrum card control center also offers a way to gather information on the installed and used Spectrum driver.

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.

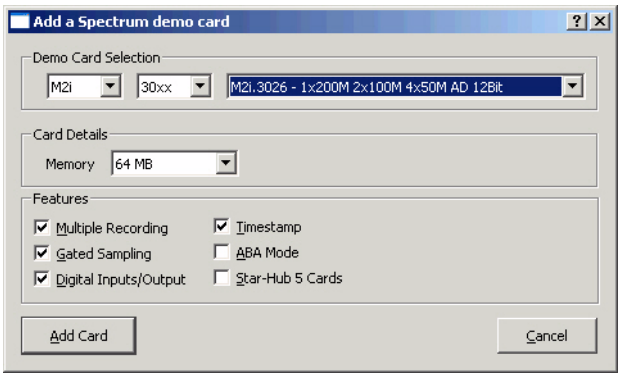
It also gives direct information about the version of the installed Spectrum kernel driver and the library (*.dll under Windows).

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



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 behaviour 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. It is not possible to change any values of an installed demo card afterwards. If you need to update the card e.g. with an additional feature, you have to add a new card with the desired features and might delete the old one.

The card control center is normally installed together with the hardware driver. To use demo cards on a system where no Spectrum cards are installed one needs to copy the control center and the drivers manually. Therefore please copy all the *.dll and *.cpl files from /spcm_driver/win32 to the windows system32 directory (normally /Windows/System32). After a re-boot the card control center is available for use and installation of demo cards.

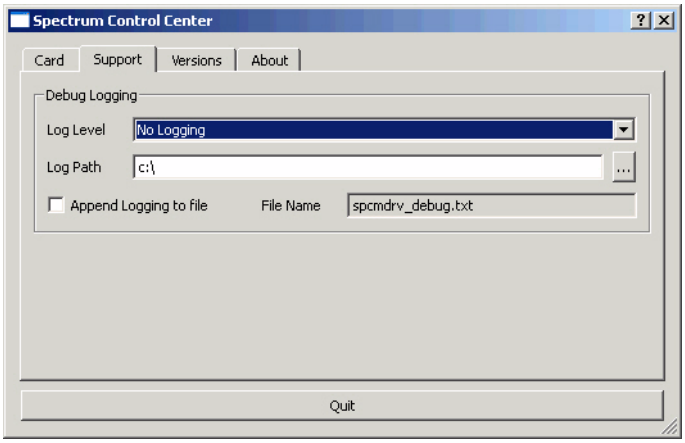
Debug logging for support cases

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.

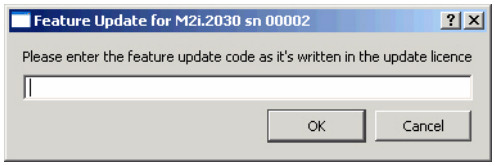
The customer can select different log levels and the path of the generated ASCII text file. One can also decide to delete the previous log file first before creating a new one automatically or to append different logs to one single log file.



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.

Feature upgrade

All optional features of the M2i 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.

Firmware upgrade

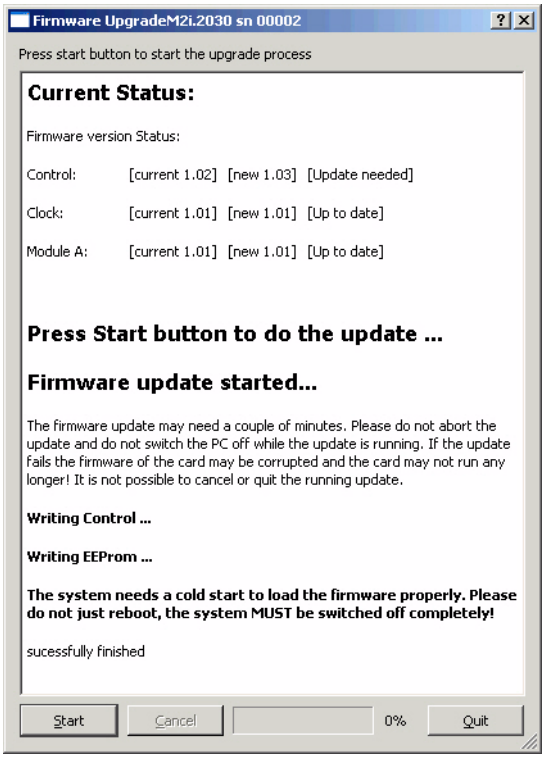
One of the major features of the card control center is the ability to update the cards firmware by an easy-to-use software. The latest firmware revisions can be found in the download section of our homepage under http://www.spectrum-instrumentation.com.

A new firmware version is provided there as an installer, that copies the latest firmware to your system. All files are located in an 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 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 provided with the card. All manuals can also be found on the Spectrum homepage in the literature download section.
- 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 and press the „firmware update" button on the right side.

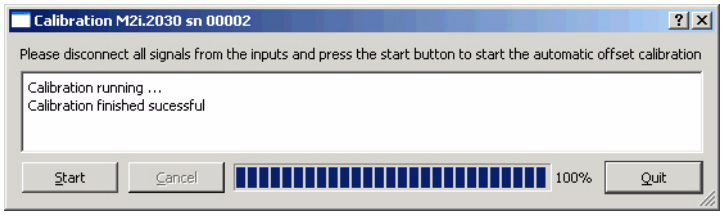
The dialogue then will inform you about the currently installed firmware version for the different devices on the card and the new versions that are available. All devices that will be affected with the update are marked as „update needed". Simply start the update or cancel the operation now, as a running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



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. The re-powering is required to finally activate the new firmware version of your Spectrum card.

Performing card calibration

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.

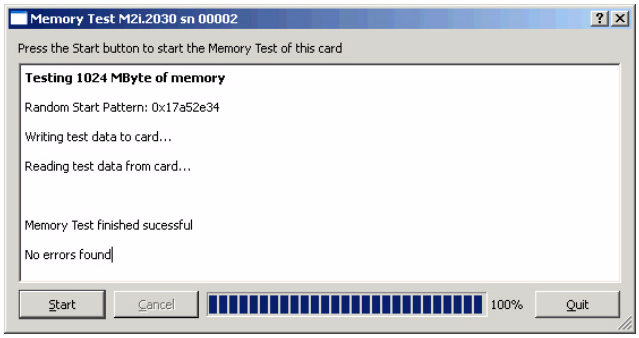


Performing memory test

The complete on-board memory of the Spectrum M2i cards can be tested by the memory test included with the card control center.

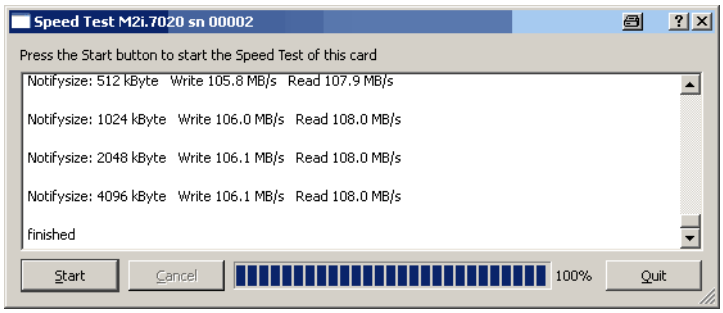
When starting the test, randomized data is generated and written to the on-board memory. After a complete write cycle all the data is read back and compared with the generated pattern.

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



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 above. It is also highly recommended that no other software or time-consuming background threads are running on that system. The speed test program runs the following two tests:



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

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 has 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

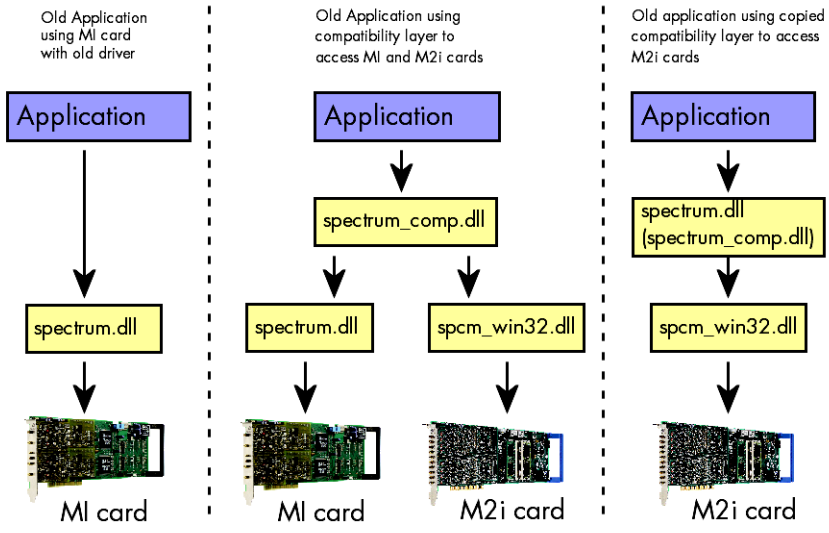
As an example running a card with 2 14 bit analog channels with 28 MHz produces a transfer rate of [2 channels * 2 Bytes/Sample * 28000000] = 112000000 Bytes/second. Taking the above figures measured on a standard 33 MHz PCI slot the system is just capable of reaching this transfer speed: 108.0 MB/s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

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

Compatibility Layer

The installation of the driver also installs a special compatibility DLL (under Windows). This dll allows the use of the M2i cards with software that has been build for the MI cards. The compatibility dll is installed in the Windows system directory under the name spectrum_comp.dll. There are two ways to use the compatibility dll:

Usage modes



- Re-compile the old application software and including the new library spectrum_comp.lib that is delivered with the compatibility DLL. This is the recommended usage. The new compatibility DLL now has control of the older driver for MI, MC and MX drivers as well as of the newer driver for the M2i cards. The newly compiled program is now capable of running with old cards as well as with new cards without any further changes. The compatibility DLL will examine the system and support both card types as they are found. Any driver updates of either the older MI cards or the newer M2i will just update the correct part of the system. SBench 5 uses this mode and is therefore capable of supporting all card types although it was never programmed to support the M2i natively.
- If for any reason a re-compile of the existing program is not possible one can simply rename the compatibility DLL spectrum_comp.dll to spectrum.dll and copy it over the existing spectrum.dll in the Windows system directory. The program won't notice that a different DLL is

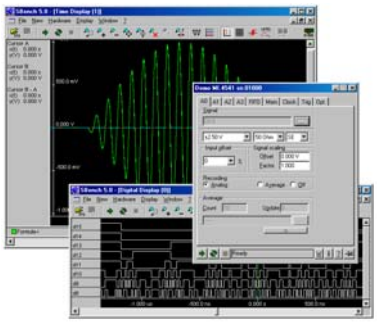
used and uses the newly installed M2i card. Unfortunately a shared access to either MI or M2i is not possible using this method.

Abilities and Limitations of the compatibility DLL

The compatibility layer has been done to help you migrating software for the M2i cards and tries to hide the new hardware to older program as best as possible. However as there are some basic differences between both hardware families not everything can be simulated. The following list should give you an overview of some aspects of the compatibility layer:

- The data transfer is reorganized internally but still uses the same application data buffers. No data is copied for the data transfers. Therefore the transfer speed that one will gain is the full transfer speed of the M2i card series which is between 20% and 130% faster than the one of the MI series.
- As the compatibility layer tries to hide the new driver as much as possible none of the new or improved features are available to older programs. If you need to use a new feature please use the new driver.
- The M2i driver checks the given parameters very carefully while the older driver was sometimes a little lazy and some false commands and driver parameters weren't noticed or were noticed but didn't lock the driver. The M2i will check every register settings at every time and lock the driver if an error occurs. It may be necessary to fix the application code for handling this more strict error checking.
- The compatibility DLL doesn't support all special features that have been added to the MI series over the years as some of them are discontinued in the new hardware. As long as the application program sticks to the main features this won't be a problem.
- The compatibility DLL does not add any delays from the MI series as the M2i series has been optimized for small delays. As an example, the MI cards had a fixed delay from trigger to first sample when using Multiple Recording. The M2i cards now have a programmable pretrigger size. When using the compatibility layer this pretrigger is set to the minimum and data will be visible before the trigger event.
- Although the application software doesn't see a difference between old an new cards there is no chance to synchronize both card types together as the synchronization option uses different connectors, different signals and different timing.

Accessing the cards with SBench 5.x



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. A full version of SBench 5.x is delivered with the card on CD. The program supports all actual acquisition, generator and digital I/O boards from Spectrum. Depending on the used card and the software setup, one could use SBench as a digital storage oscilloscope, a spectrum analyzer, a logic analyzer or simply as a data recording front end. Different export and import formats allow the use of SBench together with a variety of other programs.
As the above overview is showing, SBench 5.x was originally designed for the use with the MI/MC/MX series boards. To make it work with the new M2i series cards, SBench uses the included compatibility layer. Please note, that not all of the new features and functions of the M2i series cards are available under SBench 5.x! On the CD you'll find an install version of SBench in the directory /Install/SBench. There's also a pre-installed program version on CD that can be started directly from CD without installing to hard disk. This file can be found in the /Programs/SBench5 directory. Also on CD is a program description that shows in detail how SBench works and what settings have to be done to use SBench in one of the different modes. The manual is found in the path /Internet/english/swmanuals/SBench. The current version of SBench can be down loaded free of charge directly from the Spectrum website http://www.spectrum-instrumentation.com. Please go to the download section and get the latest version there.

SBench 5 has been designed to run under Windows 2000 and Windows XP, Windows XP64 and Windows Vista. It does not run under Linux. At the moment there is no graphical ready-to-run software for Linux available. Please use the driver examples to examine whether the board is correctly installed under Linux.

C/C++ Driver Interface

C/C++ is the main programming language for which the drivers have been build up. 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 or Basic. 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 CD together with the board. The header files are found in the directory /Driver/header_c. 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.

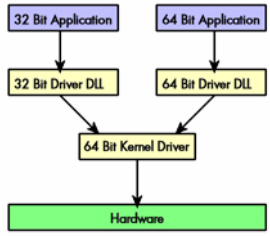
dlltyp.h	Includes the platform specific definitions for data types and function declarations. All data types are based on this definitions. The use of this type definition file allows the use of examples and programs on different platforms without changes to the program source. The header file supports 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 itself 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 shortly all. 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 on 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 and 2005 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 /spcm_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 /spcm_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 Vsual Studio 2005 (*.vcproj) that can be directly loaded 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++ 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 /spcm_examples/c_cpp/c_header. All functions described below are now available in your program.

Borland C++ Builder 32 Bit

Include Driver

The driver files can be easily included in Borland C++ Builder by simply using the library file spcm_win32_bcppb.lib that is delivered together with the drivers. The library file can be found on the CD in the path /spcm_examples/c_cpp/c_header. Please include the library file in your Borland C++ Builder project as shown in the examples. All functions described below are now available in your program.

Examples

The Borland C++ Builder examples share the sources with the Visual C++ examples. Please see above chapter for a more detailed documentation of the examples. In each example directory are project files for Visual C++ as well as Borland C++ Builder.

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 spcm_linux.lib library in your makefile 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, the driver functions must be loaded from the driver dll. Most compiler 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 loaded from the dll using standard Windows functions. There is one example in the example directory /spcm_examples/c_cpp/dll_loading that shows the process.

Example of function loading:

```
hDLL = LoadLibrary ("spcm_win32.dll");
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, the driver functions must be loaded from the driver dll. Most compiler 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 loaded from the dll using standard Windows functions. There is one example in the example directory /spcm_examples/c_cpp/dll_loading that shows the process for 32 bit environments.

Example of function loading:

```
hDLL = LoadLibrary ("spcm_win64.dll");
pfn_spcm_hOpen =  (SPCM_HOPEN*)  GetProcAddress (hDLL, "_spcm_hOpen@4");
pfn_spcm_vClose = (SPCM_VCLOSE*) GetProcAddress (hDLL, "_spcm_vClose@4");
```

National Instruments LabWindows/CVI

Include Drivers

To use the Spectrum driver under LabWindows/CVI it is necessary to first load the functions from the driver dll. Please use the library file spcm_win32_cvi.lib to access the driver functions.

Examples

Examples for LabWindows/CVI can be found on CD in the directory /spcm_examples/cvi. Please mix these examples with the standard C/C++ examples to have access to all functions and modes of the cards.

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

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

Declaration	Type
uint8	8 bit unsigned integer (range from 0 to 255)
uint16	16 bit unsigned integer (range from 0 to 65535)
uint32	32 bit unsigned integer (range from 0 to 4294967296)
uint64	64 bit unsigned integer (full range)

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 results from using incorrect types. Feel free to use this notation form for your programs also-

Declaration	Notation
int8	byName (byte)
int16	nName
int32	lName (long)
int64	llName (long long)
int32*	plName (pointer to long)

Declaration	Notation
uint8	cName (character)
uint16	wName (word)
uint32	dwName (double word)
uint64	qwName (quad word)
char	szName (string with zero termination)

Function spcm_hOpen

This function initializes and opens an installed card supporting the new SpcM driver interface. At the time of printing this manual this are all cards of the M2i cards. The function returns a handle that has to 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 raise and the function returns NULL.

Function spcm_hOpen (char* szDeviceName):

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

Under Linux the device name in the function call need 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 tailing 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 tailing 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

```
drv_handle  hDrv;                   // returns the handle to the opended driver or NULL in case of error
hDrv = spcm_hOpen ("/dev/spcm0");   // string to the driver to open
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_SetParam. These functions sets a register to a defined value or executes 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
```

Example:

```
if (!spcm_dwSetParam_i32 (hDrv, SPC_MEMSIZE, 16384))
    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_dwSpcGetParam

All hardware settings are based on software registers that can be read by one of the functions spcm_dwGetParam. These functions reads 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
```

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

The three functions only differ in the type of the parameters that is 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 splitted in a low double word part and a high double word part. Please be sure to fill both parts with valid information.

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:

Internal register	read/write	Function type	Behaviour
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	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.
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 splitted 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	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 splitted 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.

Function spcm_dwGetContBuf

This function reads out the internal continuous memory buffer if 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 15% - 25%. 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.

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 transer 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
    uint32      dwBrdOffsL,             // low part of offset in board memory
    uint32      dwTransferLenH,         // high part of transfer buffer length
    uint32      dwTransferLenL);        // low part of transfer buffer length

uint32 _stdcall spcm_dwDefTransfer_i64 (// Defines the transer 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
    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 need to use the _i64m function and split offset and length in two 32 bit words.

Example:

```
int16* pnBuffer = new int16[8192];
if (!spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 0, (void*) pnBuffer, 16384))
    printf ("DefTransfer failed\n");
```

The example defines a date 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_vInvalidateBuf

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 this function has no need. When calling spcm_dwDefTransfer any further defined buffer is automatically invalidated.

Function spcm_vInvalidateBuf

```
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.

Function spcm_dwGetErrorInfo

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

Example:

```
char szErrorBuf[ERRORTEXTLEN];
if (spcm_dwSetData_i32 (hDrv, SPC_MEMSIZE, -1))
    {
    spcm_dwGetErrorInfo_i32 (hDrv, NULL, NULL, szErrorBuf);
    printf ("Set of memsize failed with error message: %s\n", szErrorBuf);
    }
```

Borland 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 has 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'

// ----- 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_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';

// ----- 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 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:

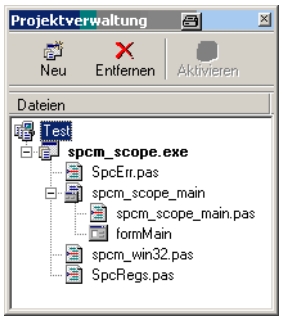
```
const SPC_M2CMD                         = 100;              { write a command }
const     M2CMD_CARD_RESET              = $00000001;        { hardware reset    }
const     M2CMD_CARD_WRITESETUP         = $00000002;        { write setup only }
const     M2CMD_CARD_START              = $00000004;        { 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 given back 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 CD. 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;
```

Examples

Examples for Delphi can be found on CD in the directory /spcm_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

Visual Basic Programming Interface and Examples

Driver interface

The driver interface is located in the sub-directory b_header and contains the following files. The files need to be included in the basic project. Please do not edit any of these files as they're regularly updated if new functions or registers have been included.

file spcm_win32_decl.bas

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

```
' ----- card handling functions -----
Public Declare Function spcm_hOpen Lib "spcm_win32.dll" Alias "_spcm_hOpen@4"
(ByVal szDeviceName As String) As Long

Public Declare Function spcm_vClose Lib "spcm_win32.dll" Alias "_spcm_vClose@4"
(ByVal hDevice As Long) As Long

Public Declare Function spcm_dwGetErrorInfo_i32 Lib "spcm_win32.dll" Alias "_spcm_dwGetErrorInfo_i32@16"
(ByVal hDevice As Long, ByRef lErrorReg, ByRef lErrorValue, ByVal szErrorText As String) As Long

' ----- software register handling -----
Public Declare Function spcm_dwGetParam_i32 Lib "spcm_win32.dll" Alias "_spcm_dwGetParam_i32@12"
(ByVal hDevice As Long, ByVal lRegister As Long, ByRef lValue As Long) As Long

Public Declare Function spcm_dwGetParam_i64m Lib "spcm_win32.dll" Alias "_spcm_dwGetParam_i64m@16"
(ByVal hDevice As Long, ByVal lRegister As Long, ByRef lValueHigh As Long, ByRef lValueLow As Long) As Long

Public Declare Function spcm_dwSetParam_i32 Lib "spcm_win32.dll" Alias "_spcm_dwSetParam_i32@12"
(ByVal hDevice As Long, ByVal lRegister As Long, ByVal lValue As Long) As Long

Public Declare Function spcm_dwSetParam_i64m Lib "spcm_win32.dll" Alias "_spcm_dwSetParam_i64m@16"
(ByVal hDevice As Long, ByVal lRegister As Long, ByVal lValueHigh As Long, ByVal lValueLow As Long) As Long

' ----- data handling -----
Public Declare Function spcm_dwDefTransfer_i64m Lib "spcm_win32.dll" Alias "_spcm_dwDefTransfer_i64m@36"
(ByVal hDevice As Long, ByVal dwBufType As Long, ByVal dwDirection As Long, ByVal dwNotifySize As Long, ByRef
pvDataBuffer As Any, ByVal dwBrdOffsH As Long, ByVal dwBrdOffsL As Long, ByVal dwTransferLenH As Long, ByVal
dwTransferLenL As Long) As Long

Public Declare Function spcm_dwInvalidateBuf Lib "spcm_win32.dll" Alias "_spcm_dwInvalidateBuf@8"
(ByVal hDevice As Long, ByVal lBuffer As Long) As Long
```

file SpcRegs.bas

The SpcRegs.bas 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:

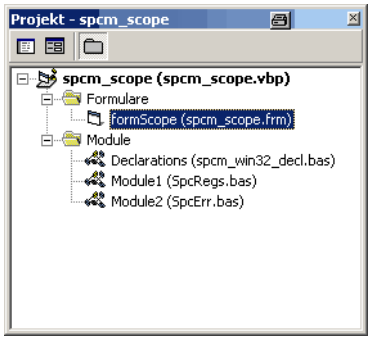
```
Public Const SPC_M2CMD = 100                                ' write a command
Public Const M2CMD_CARD_RESET = &H1&                        ' hardware reset
Public Const M2CMD_CARD_WRITESETUP = &H2&                   ' write setup only
Public Const M2CMD_CARD_START = &H4&                        ' start of card (including writesetup)
Public Const M2CMD_CARD_ENABLETRIGGER = &H8&                ' enable trigger engine
...
```

file SpcErr.bas

The SpeErr.bas file contains all error codes that may be given back 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 CD.



Examples

Examples for Visual Basic can be found on CD in the directory /spcm_examples/basic. The directory contains the above mentioned basic 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 capsulates 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 the DLL SpcmDrv.NET.dll delivered with the .NET examples. Spectrum also delivers the source code of the DLL as a C# project. Theses 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);
```

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);
```

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)
```

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);
```

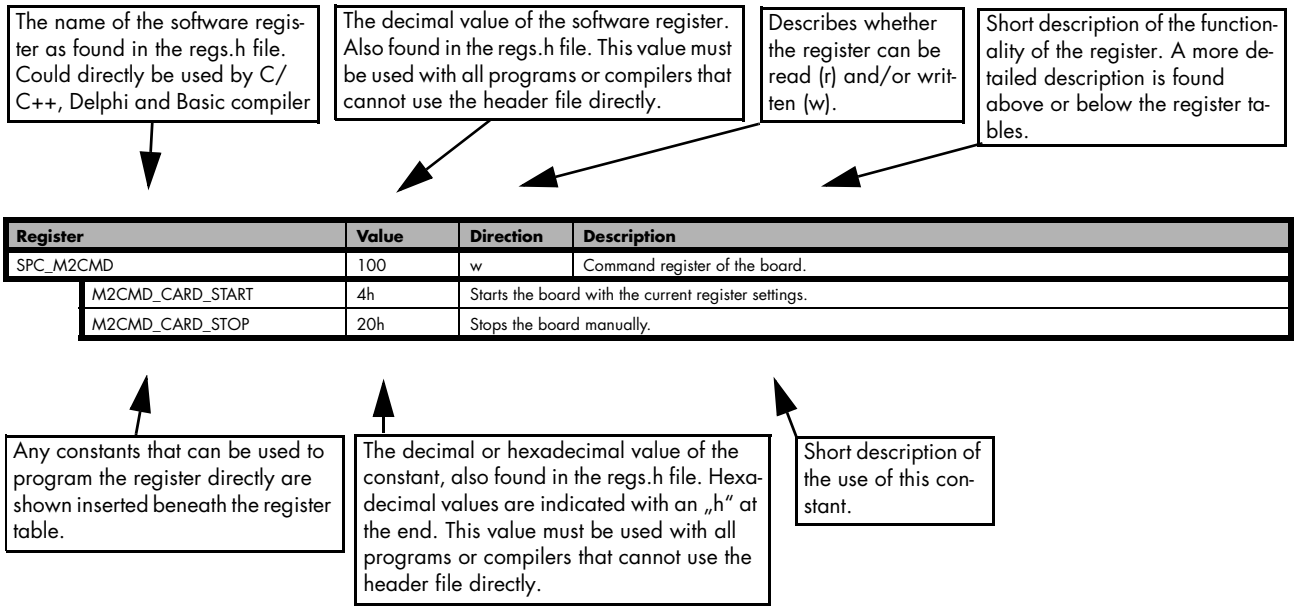
Programming the Board

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:



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	20h	Stops the board manually.	

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 coloured 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
    printf ("Found card M2i.%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;
```

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 behaviour. 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_i32 (hDrv, SPC_SAMPLERATE, 1000000);                    // correct command
spcm_dwSetParam_i32 (hDrv, SPC_MEMSIZE, -345);                          // faulty command
spcm_dwSetParam_i32 (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 parameter 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 it’s 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_i32 (hDrv, SPC_SAMPLERATE, 1000000);                    // correct command
spcm_dwSetParam_i32 (hDrv, SPC_MEMSIZE, -345);                          // faulty command
spcm_dwSetParam_i32 (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.

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

One of the following values is returned, when reading this register. Each card has it's 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.

Card type	Card type as defined in regs.h	Value hexadecimal	Value decimal		Card type	Card type as defined in regs.h	Value hexadecimal	Value decimal
M2i.2020	TYP_M2I2020	32020h	204832		M2i.2030	TYP_M2I2030	32030h	204848
M2i.2021	TYP_M2I2021	32021h	204833		M2i.2031	TYP_M2I2031	32031h	204849
M2i.2020-exp	TYP_M2I2020EXP	42020h	270368		M2i.2030-exp	TYP_M2I2030EXP	42030h	270384
M2i.2021-exp	TYP_M2I2021EXP	42021h	270369		M2i.2031-exp	TYP_M2I2031EXP	42031h	270385

Hardware version

Since all of the M2i boards from Spectrum are modular boards, they consist of one base board and one or two piggy-back front-end modules and eventually of an extension module like the star-hub. Each of these three kinds of hardware has it's own version register. Normally you do not need this information but if you have a support question, please provide the revision together with it.

Register	Value	Direction	Description
SPC_PCIVERSION	2010	read	Base card version: the upper 16 bit show the hardware (PCB) version, the lower 16 bit show the firmware version.
SPC_PCIMODULEVERSION	2012	read	Module version: the upper 16 bit show the hardware (PCB) version, the lower 16 bit show the firmware version.

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

Register	Value	Direction	Description
SPC_PCIEXTVERSION	2011	read	Extension module version: the upper 16 bit show the hardware (PCB) version, the lower 16 bit show the firmware version.

Production date

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

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

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

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.

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.

Register	Value	Direction	Description
SPC_PCISAMPLERATE	2100	read	Maximum sampling rate in Hz as a 32 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 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.

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.

Register		Value	Direction	Description
SPC_PCIFEATURES		2120	read	PCI feature register. Holds the installed features and options as a bitfield. The return 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 option Multiple Recording / Multiple Replay is installed.	
	SPCM_FEAT_GATE	2h	Is set if the option Gated Sampling / Gated Replay is installed.	

SPCM_FEAT_DIGITAL	4h	Is set if the option Digital Inputs / Digital Outputs is installed.
SPCM_FEAT_TIMESTAMP	8h	Is set if the option Timestamp is installed.
SPCM_FEAT_STARHUB5	20h	Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 5 cards.
SPCM_FEAT_STARHUB16	40h	Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 16 cards.
SPCM_FEAT_ABA	80h	Is set if the option ABA mode is installed.
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

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

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

Used type of driver

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

Register	Value	Direction	Description
SPC_GETDRVTYPE	1220	read	Gives information about what type of driver is actually used
DRVTYP_LINUX	1	Linux driver is used	
DRVTYP_WDM	4	Windows WDM driver is used (only Windows 2000/XP/XP64/Vista).	

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.

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.

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.

```
SpcGetParam (hDrv, SPC_GETDRVVERSION, &lLibVersion);
SpcGetParam (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
```

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.

It is recommended, that every custom written program performs a software reset first, to be sure that the driver is in a defined state independent from possible previous setting.

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

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. An 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 card run.
CHANNEL0	1	Activates channel 0	
CHANNEL1	2	Activates channel 1	
CHANNEL2	4	Activates channel 2	
CHANNEL3	8	Activates channel 3	

The channel enable register is set as a bitmap. That means that 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 4 channels:

```
spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0 | CHANNEL1 | CHANNEL2 | CHANNEL3);
```

The following table shows all allowed settings for the channel enable register when your card has a maximum of 2 channels.

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

The following table shows all allowed settings for the channel enable register in case that you have a four channel card.

Channels to activate						
Ch0	Ch1	Ch2	Ch3	Values to program	Value as hex	Value as decimal
X				CHANNEL0	1h	1
	X			CHANNEL1	2h	2
		X		CHANNEL2	4h	4
			X	CHANNEL3	8h	8
X	X			CHANNEL0 | CHANNEL1	3h	3
X		X		CHANNEL0 | CHANNEL2	5h	5
X			X	CHANNEL0 | CHANNEL3	9h	9
	X	X		CHANNEL1 | CHANNEL2	6h	6
	X		X	CHANNEL1 | CHANNEL3	Ah	10
		X	X	CHANNEL2 | CHANNEL3	Ch	12
X	X	X	X	CHANNEL0 | CHANNEL1 | CHANNEL2 | CHANNEL3	Fh	15

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

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 | CHANNEL2);
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: 0x00000005
Number of activated channels with this bitmask: 2
```

Important note on channels 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 actual series. There can be less channels on your actual type of board or bus-system. Please refer to the table(s) above to get the actual number of available channels.

Setting up the inputs

Input ranges

This analog acquisition board uses separate input amplifiers 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. 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 version availble with different input ranges it is recommended to read out the currently available input ranges as shown later in this chapter.

Register	Value	Direction	Description
SPC_AMP0	30010	r/w	Defines the input range of channel0.
SPC_AMP1	30110	r/w	Defines the input range of channel1.
SPC_AMP2	30210	r/w	Defines the input range of channel2.
SPC_AMP3	30310	r/w	Defines the input range of channel3.
	50		± 50 mV calibrated input range for the appropriate channel.
	100		± 100 mV calibrated input range for the appropriate channel.
	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.
	2000		± 2 V calibrated input range for the appropriate channel.
	5000		± 5 V calibrated input range for the appropriate channel.

Universal software that handles different card types can read out how many different input ranges are available on the actual board for each channel. This information can be obtained by using the read-only register shown in the table below.

Register	Value	Direction	Description
SPC_READIRCOUNT	3000	read	Informs about the number of the board's calibrated input ranges.

Additionally one can read out the minimum and the maximum value of each input range as shown in the table below. The number of input ranges is read out with the above shown register.

Register	Value	Direction	Description
SPC_READRANGEMIN0	4000	read	Gives back the minimum value of input range 0 in mV.
SPC_READRANGEMIN1	4001	read	Gives back the minimum value of input range 1 in mV.
SPC_READRANGEMIN2	4002	read	Gives back the minimum value of input range 2 in mV.
...	...	read	...
SPC_READRANGEMAX0	4100	read	Gives back the maximum value of input range 0 in mV.
SPC_READRANGEMAX1	4101	read	Gives back the maximum value of input range 1 in mV.
SPC_READRANGEMAX2	4102	read	Gives back the maximum value of input range 2 in mV.
...	...	r	...

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, 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);
    }
```

Input offset

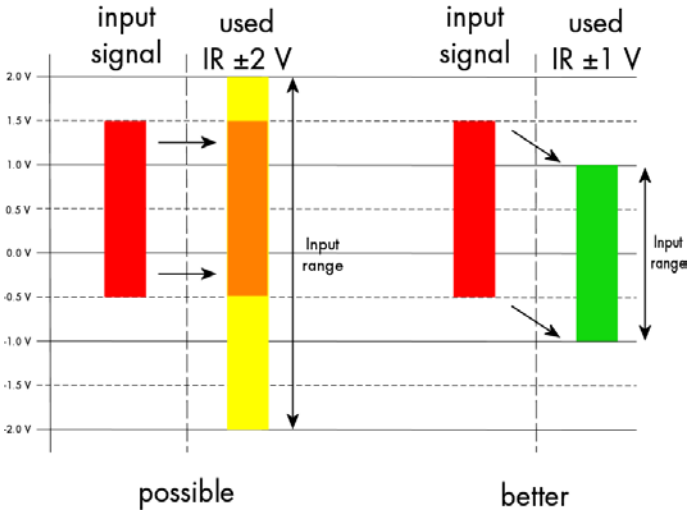
In most cases the external signals will not be symmetrically related to ground. If you want to acquire such asymmetrical signals, it is possible to use the smallest input range that matches the biggest absolute signal amplitude without exceeding the range.

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

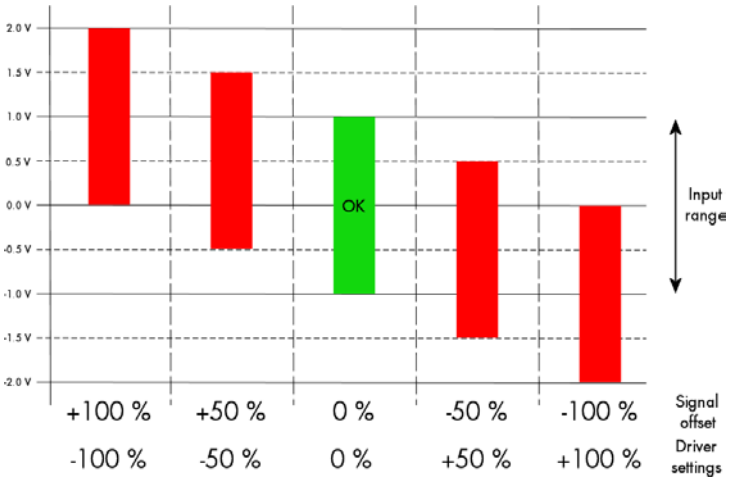


The example in the right figure shows signals with a range of ±1.0 V that have offsets up to ±1.0 V. So related to the desired input range these signals have offsets of ±100 %.

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.



Register	Value	Direction	Description	Offset range
SPC_OFFS0	30000	r/w	Defines the input's offset and therfore shifts the input of channel0.	± 400 % in steps of 1 %
SPC_OFFS1	30100	r/w	Defines the input's offset and therfore shifts the input of channel1.	± 400 % in steps of 1 %
SPC_OFFS2	30200	r/w	Defines the input's offset and therfore shifts the input of channel2.	± 400 % in steps of 1 %
SPC_OFFS3	30300	r/w	Defines the input's offset and therfore shifts the input of channel3.	± 400 % in steps of 1 %

When writing a program that should run with different board families it is useful to just read-out the possible offset than can be programmed. You can use the following read only register to get the possible programmable offset range in percent

Register	Value	Direction	Description
SPC_READOFFSMIN0	4200	read	Minimum programmable offset for input range 0 in percent
SPC_READOFFSMAX0	4100	read	Maximum programmable offset for input range 0 in percent
SPC_READOFFSMIN0	4201	read	Minimum programmable offset for input range 1 in percent
SPC_READOFFSMAX0	4101	read	Maximum programmable offset for input range 1 in percent
..	...	...	...

To give you an example how the registers of the input range and the input offset are to be used, the following example shows a setup to match all of the four signals in the second input offset figure to match the desired input range. Therefore every one of the four channels is set

to the input range of ± 1.0 V. After that the four offset settings are set exactly as the offsets to be compensated, but with the opposite sign. The result is, that all four channels match perfectly to the chosen input range.

```
spcm_dwSetParam_i32 (hDrv, SPC_AMP0 ,    1000); // Set up channel0 to the range of ± 1.0 V
spcm_dwSetParam_i32 (hDrv, SPC_AMP1 ,    1000); // Set up channel1 to the range of ± 1.0 V
spcm_dwSetParam_i32 (hDrv, SPC_AMP2 ,    1000); // Set up channel2 to the range of ± 1.0 V
spcm_dwSetParam_i32 (hDrv, SPC_AMP3 ,    1000); // Set up channel3 to the range of ± 1.0 V

spcm_dwSetParam_i32 (hDrv, SPC_OFFS0,    -100); // Set the input offset to get the signal symmetrically to 0.0 V
spcm_dwSetParam_i32 (hDrv, SPC_OFFS1,     -50);
spcm_dwSetParam_i32 (hDrv, SPC_OFFS2,      50);
spcm_dwSetParam_i32 (hDrv, SPC_OFFS3,     100);
```

Input termination

All inputs of Spectrum's analog boards can be terminated separately with 50 Ohm by software programming. If you do so, please make sure that your signal source is able to deliver the higher output currents. If no termination is used, the inputs have an impedance of 1 Megaohm. The following table shows the corresponding register to set the input termination.

Register	Value	Direction	Description
SPC_50OHM0	30030	read/write	A „1" sets the 50 ohm termination for channel0. A „0" sets the termination to1 MOhm.
SPC_50OHM1	30130	read/write	A „1" sets the 50 ohm termination for channel1. A „0" sets the termination to1 MOhm.
SPC_50OHM2	30230	read/write	A „1" sets the 50 ohm termination for channel2. A „0" sets the termination to1 MOhm.
SPC_50OHM3	30330	read/write	A „1" sets the 50 ohm termination for channel3. A „0" sets the termination to1 MOhm.

Automatic adjustment of the offset settings

All of the channels are calibrated in factory before the board is shipped. These settings 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.

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 has at minimum 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.

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 actual settings to the specified set in the EEPROM. T
		read	Reads out, what kind of settings have been saved last.
ADJ_DEFAULT	0	Default settings can be loaded only. These settings cannot be saved by the user.	
ADJ_USER0	1	User settings 0. This is a valid set for storing user offset settings to.	

If you want to make an offset adjustment on all the channels and store the data to the ADJ_USER0 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 adjustment on all channels
spcm_dwSetParam_i32 (hDrv, SPC_ADJ_SAVE,       ADJ_USER0); // and store values to USER0 set in the EEPROM
```

If teh card has no user settings one can store to the default settings as shown here:

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

When working with a user setting instead of the default ones, you need to restore your user settings with the help of the SPC_ADJ_LOAD register as the following example shows.

```
spcm_dwSetParam_i32 (hDrv, SPC_ADJ_LOAD,       ADJ_USER0); // and load values to USER0 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. If 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 behaviour and there are only a very few differences between them.

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

SPC_REC_STD_SINGLE	1h	Data acquisition to on-board memory for one single trigger event.
SPC_REC_STD_MULTI	2h	Data acquisition to on-board memory for multiple trigger events. Each recorded segment has the same size. Only available if option Multiple Recording is installed. this mode is described in greater detail in a special chapter about the Multiple Recording option.
SPC_REC_STD_GATE	4h	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. This mode is only available if the Gated Sampling option is installed. The mode is described in greater detail in a special chapter about the Gated Sampling option.
SPC_REC_STD_ABA	8h	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. This mode is only available if the ABA mode option is installed. The mode is described in a special chapter about ABA mode option.
SPC_REC_FIFO_SINGLE	10h	Continuous data acquisition for one single trigger event. The on-board memory is used completely as FIFO buffer.
SPC_REC_FIFO_MULTI	20h	Continuous data acquisition for multiple trigger events. Only available if Multiple Recording option is installed.
SPC_REC_FIFO_GATE	40h	Continuous data acquisition using an external gate signal. only available if Gated Sampling option is installed.
SPC_REC_FIFO_ABA	80h	Continuous data acquisition for multiple trigger events together with continuous data acquisition with a slower sampling clock. Only available if ABA mode option is installed

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 splitted into two types of commands: execution commands that fulfil 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 the given commands is not allowed in the current state.

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 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.
M2CMD_CARD_FLUSHFIFO	80h	Used to flush input FIFOs after the card has been stopped while an acquisition was running.

Card wait commands

These commands do not return until either the defined state has been reached what is signalled 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_TIMEOUIT if the specified timeout has expired.

Register	Value	Direction	Description
SPC_TIMEOUT	295130	read/write	Defines the timeout for any following wait command in a milli second 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 command until the timeout is again disabled by writing a zero to this register.

A timeout occuring 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_dwGetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_WAITREADY);
printf ("Card has stopped now!\n");
```

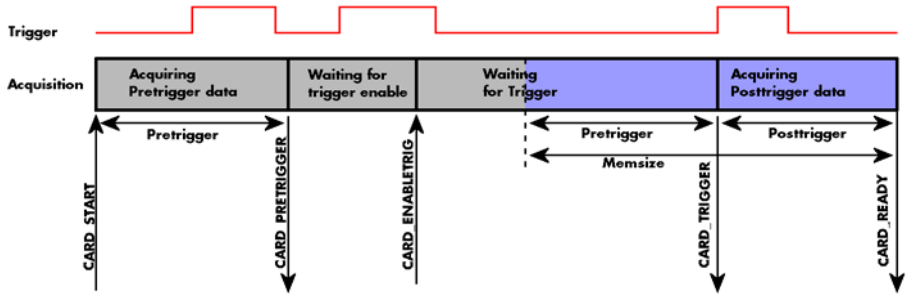
Card Status

In addition to the wait for an interrupt mechanism or completely instead of it one needs also to read out the current card status by reading the SPC_M2STATUS register. The status register is organized as a bitmap showing the status of the card and also of the different data transfers.

Register	Value	Direction	Description
SPC_M2STATUS	110	read only	Reads out the current status information
M2STAT_CARD_PRETRIGGER	1h	Acquisition modes only: the pretrigger area has been filled.	
M2STAT_CARD_TRIGGER	2h	The first trigger has been detected.	
M2STAT_CARD_READY	4h	The card has finished it's run and is ready.	

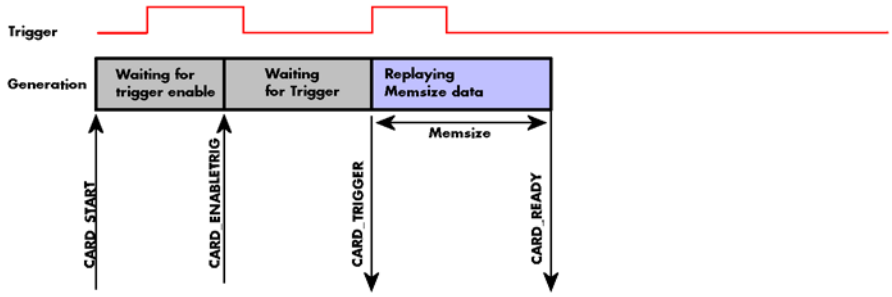
Acquisition cards status overview

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 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 changes to M2STAT_CARD_TRIGGER and the card acquires the programmed posttrigger data. After all post trigger data has been acquired the status changes to M2STAT_CARD_READY and data can be read out:



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 changes to M2STAT_CARD_TRIGGER and the card starts with the data replay. After replay has been finished - depending on the programmed mode - the status changes to M2STAT_CARD_READY and the card stops.



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:

- 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_vInvalidateBuf function.

Definition of the transfer buffer

Before any data transfer can start it is necessary to define the transfer buffer with all it's 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.

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.

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 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.

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.

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.

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:

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.

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:

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

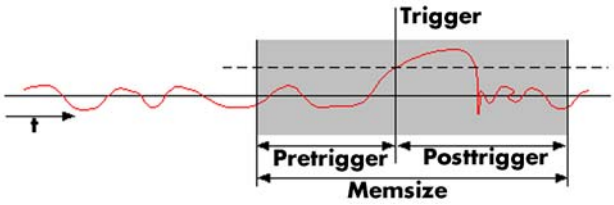
```
void* pvData = (void*) new int8[1024];

// transfer data from PC memory to card memory
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);

// transfer the same data back to PC memory
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);
```

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!

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 continuously ring buffer recording, there are also samples prior to the trigger-event in the memory (pretrigger).



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 up 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.

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 triggerevent has been detected.

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 after the trigger event has been detected.

You can access these settings by the registers 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 triggerevent 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_i32 (hDrv, SPC_MEMSIZE, lMemsize);                 // recording length
spcm_dwSetParam_i32 (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 = (void*) new int8[lMemsize];

// read out the data
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA,  SPCM_DIR_CARDTOPC , 0, pvData, 0, 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.

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

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 behaviour 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 can only be much less.

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

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.

```
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_i32 (hDrv, SPC_PRETRIGGER, 1024);                      // 1 kSample of data before trigger

// in FIFO mode we need to define the buffer before starting the transfer
int8* pbyData = new int8[lBufsizeInSamples];
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 4096, (void*) pbyData, 0, lBufsizeInSamples);

// now we start the acquisition and wait for the first block
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START|M2CMD_CARD_ENABLETRIGGER|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
dwTotalBytes = 0;
while (!dwError)
    {
    // read out the available bytes
    spcm_dwGetParam_i32 (hDrv, SPC_DATA_AVAIL_USER_LEN, &lAvailBytes);
    dwTotalBytes += dwAvailBytes;

    // here is the right position to do something with the data
    printf ("Currently Available: %d, total: %d\n", lAvailBytes, dwTotalBytes);

    // now we free the number of bytes and wait for the next buffer
    spcm_dwSetParam_i32 (hDrv, SPC_DATA_AVAIL_CARD_LEN, lAvailBytes);
    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 samples needs 1 byte 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

Running the card with a sampling rate that is above 100 MS/s switches the cards internally to an interlace mode. In this mode two ADCs are running in parallel using a 180° shifted signal. Due to the fact that two ADCs are running this mode has a little different limitations and is listed separately in the following table.

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 channel	Standard Single	8	Mem	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem	4	4	16k - 32	4	4	Mem/2-4	4	8	Mem/2	4	not used		
	Standard Gate	8	Mem	4	4	16k - 32	4	4	Mem-4	4	not used			not used		
	FIFO Single	not used			4	16k - 32	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	16k - 32	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	16k - 32	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1
1 channel interlace	Standard Single	16	Mem	8	defined by post trigger			8	8G - 8	8	not used			not used		
	Standard Multi/ABA	16	Mem	8	8	16k - 32	8	8	Mem/2-4	8	16	Mem/2	8	not used		
	Standard Gate	16	Mem	8	8	16k - 32	8	8	Mem-8	8	not used			not used		
	FIFO Single	not used			8	16k - 32	8	not used			16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			8	16k - 32	8	8	8G - 8	8	16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Gate	not used			8	16k - 32	8	8	8G - 8	8	not used			0 (∞)	4G - 1	1
2 channels	Standard Single	8	Mem/2	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem/2	4	4	8k - 16	4	4	Mem/4-4	4	8	Mem/4	4	not used		
	Standard Gate	8	Mem/2	4	4	8k - 16	4	4	Mem/2-4	4	not used			not used		
	FIFO Single	not used			4	8k - 16	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	8k - 16	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	8k - 16	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1
2 channels interlace	Standard Single	16	Mem/2	8	defined by post trigger			8	8G - 8	8	not used			not used		
	Standard Multi/ABA	16	Mem/2	8	8	8k - 16	8	8	Mem/4-8	8	16	Mem/4	8	not used		
	Standard Gate	16	Mem/2	8	8	8k - 16	8	8	Mem/2-8	8	not used			not used		
	FIFO Single	not used			8	8k - 16	8	not used			16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			8	8k - 16	8	8	8G - 8	8	16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Gate	not used			8	8k - 16	8	8	8G - 8	8	not used			0 (∞)	4G - 1	1
4 channels	Standard Single	8	Mem/4	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem/4	4	4	4k - 8	4	4	Mem/8-4	4	8	Mem/8	4	not used		
	Standard Gate	8	Mem/4	4	4	4k - 8	4	4	Mem/4-4	4	not used			not used		
	FIFO Single	not used			4	4k - 8	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	4k - 8	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	4k - 8	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1

All figures listed here are given in samples. An entry of [8k - 16] means [8 kSamples - 16] = [8192 - 16] = 8176 samples.

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

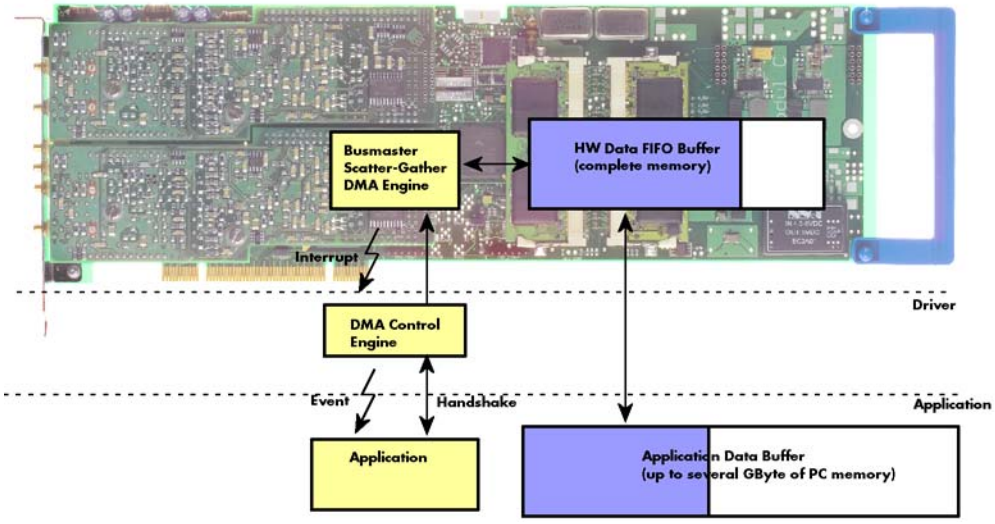
	Installed Memory						
	64 MSample	**128 MSample**	**256 MSample**	**512 MSample**	**1 GSample**	**2 GSample**	**4 GSample**
Mem	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample	2 GSample	4 GSample
Mem / 2	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample	2 GSample
Mem / 4	16 MSample	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample
Mem / 8	8 MSample	16 MSample	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample

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

Buffer handling

To handle the huge amount of data that is possible to acquire with the M2i there is a very reliable two step buffer strategy set up. The on-board memory of the card is 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:



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:

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:

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

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 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 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 avail-

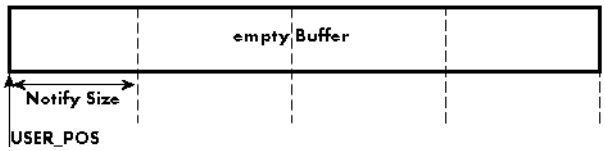
able bytes still stick to the defined notify size!
- If the on-board FIFO buffer has an overrun (card to PC) or an underrun (PC to card) data transfer is stopped. However in case of transfer from card to PC there is still a lot of data in the on-board memory. Therefore the data transfer will continue until all data has been transferred although the status information already shows an overrun.
- Getting best bus transfer performance is done using a „continuous buffer". This mode is explained in the appendix in greater detail.

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 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.

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.

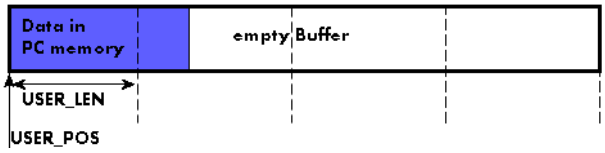
Step 1: Buffer definition

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.



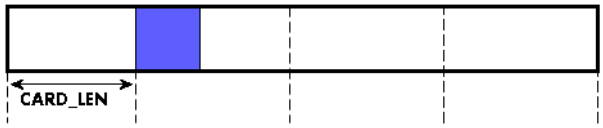
Step 2: Start and first data available

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. Although there is more data already transferred we only get announced to have the notify size of data available. The USER_POS is still zero as we are right at the beginning of the complete transfer.



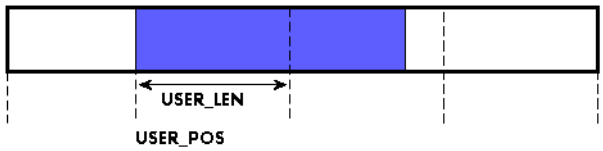
Step 3: set the first data available for card

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 this the amount of data is set available for the card and for the next step.



Step 4: next data available

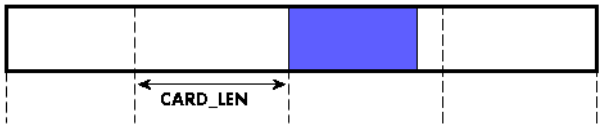
After reaching the next border of the notify size we get the next part of the data buffer to be available. In our example this part of data is again only of one notify size length. The user position will now be at the position [1 x notify size].



Step 5: set data available again

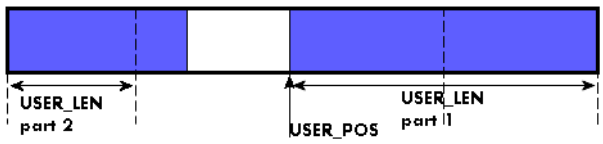
Again after processing the data we set it free for the card use.

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.



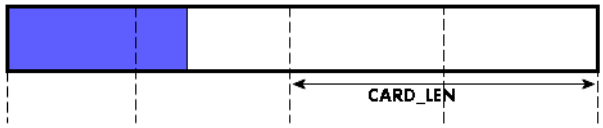
Step 6: roll over the end of buffer

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 got in step 4. 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.

Buffer handling example for transfer from card to PC

```
char* pcData = new char[lBufferSizeInBytes];

// we now define the transfer buffer with the minimum notify size of on page = 4 kByte
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA,  SPCM_DIR_CARDTOPC , 4096, (void*) pcData, 0, lBufferSizeInBytes);

do
    {
    // 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_STARTDMA | M2CMD_DATA_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_DATA_AVAIL_USER_LEN, &lAvailBytes);
        spcm_dwGetParam_i32 (hDrv, SPC_DATA_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) >= lBufferSizeInBytes)
            lAvailBytes = lBufferSizeInBytes - lBytePos;

        // our do function get's a pointer to the start of the available data section and the length
        vDoSomething (&pcData[lBytesPos], lAvailBytes);

        // the buffer section is now immediately set available for the card
        spcm_dwSetParam_i32 (hDrv, SPC_DATA_AVAIL_CARD_LEN, lAvailBytes);
        }
    }
while (!dwError); // we loop forever if no error occurs
```

Buffer handling example for transfer from PC to card

```
char* pcData = new char[lBufferSizeInBytes];

// before starting transfer we ned to once fill complete buffer memory with data
vDoGenerateData (&pcData[0], lBufferSizeInBytes);

// we now define the transfer buffer with the minimum notify size of on page = 4 kByte
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA,  SPCM_DIR_PCTOCARD , 4096, (void*) pcData, 0, lBufferSizeInBytes);

do
    {
    // we wait for the next data to be empty. After this call we get at least 4k of data to proceed
    dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_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_DATA_AVAIL_USER_LEN, &lAvailBytes);
        spcm_dwGetParam_i32 (hDrv, SPC_DATA_AVAIL_USER_POS, &lBytePos);

        printf ("We now have %d free 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) >= lBufferSizeInBytes)
            lAvailBytes = lBufferSizeInBytes - lBytePos;

        // our do function get's a pointer to the start of the available data section and the length
        vDoGenerateData (&pcData[lBytesPos], lAvailBytes);

        // the buffer section is now immediately set available for the card again
        spcm_dwSetParam_i32 (hDrv, SPC_DATA_AVAIL_CARD_LEN, lAvailBytes);
        }
    }
while (!dwError); // we loop forever if no error occurs
```

Please keep in mind that you are using a continuous buffer writing/reading that will start again at the zero position if the buffer length is reached. However the DATA_AVAIL_USER_LEN register will give you the complete amount of available bytes even if one part of the free area is at the end of the buffer and the second half at the beginning of the buffer.

Data organisation

Data is organized in a multiplexed way in the transfer buffer. If using 2 modules data of first activated channel of first module comes first, then data of first activated channel of second module, then second activated channel of first module and so on.

Activated Channels	Ch0	Ch1	Ch2	Ch3	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
1 channel			X		C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
1 channel				X	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	D16
2 channels	X	X			A0	B0	A1	B1	A2	B2	A3	B3	A4	B4	A5	B5	A6	B6	A7	B7	A8
2 channels	X		X		A0	C0	A1	C1	A2	C2	A3	C3	A4	C4	A5	C5	A6	C6	A7	C7	A8
2 channels	X			X	A0	D0	A1	D1	A2	D2	A3	D3	A4	D4	A5	D5	A6	D6	A7	D7	A8
2 channels		X	X		B0	C0	B1	C1	B2	C2	B3	C3	B4	C4	B5	C5	B6	C6	B7	C7	B8
2 channels		X		X	B0	D0	B1	D1	B2	D2	B3	D3	B4	D4	B5	D5	B6	D6	B7	D7	B8
2 channels			X	X	C0	D0	C1	D1	C2	D2	C3	D3	C4	D4	C5	D5	C6	D6	C7	D7	C8
4 channels	X	X	X	X	A0	C0	B0	D0	A1	C1	B1	D1	A2	C2	B2	D2	A3	C3	B3	D3	A4

The samples are re-named for better readability. A0 is sample 0 of channel 0, C4 is sample 4 of channel 2, and so on

Sample format

The 8 bit A/D samples are stored in twos complement in one byte. 8 bit resolution means that data is ranging from -128…to…+127.

Bit	Standard Mode
D7	ADx Bit 7 (MSB)
D6	ADx Bit 6
D5	ADx Bit 5
D4	ADx Bit 4
D3	ADx Bit 3
D2	ADx Bit 2
D1	ADx Bit 1
D0	ADx Bit 0 (LSB)

Clock generation

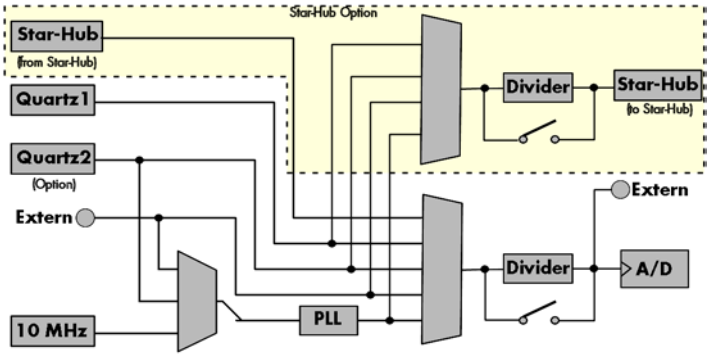
Overview

The different clock modes

The Spectrum M2i 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 M2i 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.



Standard internal sample rate (PLL)

PLL with internal 10 MHz 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 fine resolution. You can find details on the granularity of the clock in PLL mode in the technical data section of this manual.

Quartz1 with or without divider

This mode provides an internal sampling quartz clock with a dedicated divider. It's best suited for applications that need an even lower clock jitter than the PLL produces. The possible sample rates are restricted to the values of the divider. For details on the available divider values please see the according section in this chapter or take a look at the technical data section of this manual.

Quartz2 with or without PLL and/or with or without divider (optional)

This optional second Quartz2 is for special customer needs, either for a special direct sampling clock or as a very precise reference for the PLL. Please feel free to contact Spectrum for your special needs.

External reference clock

PLL with external 1 MHz to 125 MHz reference clock. This provides a very good clock accuracy if a stable external reference clock is used. It also allows the easy synchronization with an external source.

Direct external 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.

External clock with divider

In addition to the direct external clocking it is also possible to use the externally fed in clock and divide it for generating a low-jitter sample rate of a slower speed than the external clock available.

Synchronization clock (optional)

The star-hub option allows the synchronization of up to 16 cards of the M2i series from Spectrum with a minimal phase delay between the different cards. As this clock is also available at the dividers input, cards of the same or slower sampling speeds can be synchronized. For details on the synchronization option please take a look at the dedicated chapter in this manual.

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.

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 PLL with 10 MHz internal reference for sample clock generation	
SPC_CM_QUARTZ1	2	Enables Quartz1 for sample clock generation	
SPC_CM_QUARTZ2	4	Enables optional Quartz2 for sample clock generation	
SPC_CM_EXTERNAL	8	Enables external clock input for direct sample clock generation	
SPC_CM_EXTDIVIDER	16	Enables external clock input for divided 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.

Internally generated sampling rate

Standard internal sampling clock (PLL)

The internal sampling clock is generated in default mode by a PLL and dividers out of an internal precise 10 MHz frequency reference. You can select the clock mode by the dedicated register shown in the table below:

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_INTPLL	1	Enables internal PLL with 10 MHz internal reference for sample clock generation	

In most cases the user does not have to care on 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.

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.

If a sampling rate is generated internally, you can additionally enable the clock output. The clock will be available on the external clock connector and can be used to synchronize external equipment with the board.

Register	Value	Direction	Description
SPC_CLOCKOUT	20110	read/write	Enables clock output on external clock connector.On A/D and D/A cards only possible with internal clocking.

Example on writing and reading internal sampling rate

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_INTPLL);     // Enables internal PLL mode
spcm_dwSetParam_i32 (hDrv, SPC_SAMPLERATE, 1000000);          // Set internal sampling rate to 1 MHz
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKOUT,  1);                 // enable the clock output of that 1 MHz clock
spcm_dwGetParam_i32 (hDrv, SPC_SAMPLERATE, &lSamplerate);     // Read back the programmed sample rate and
printf („Sample rate = %d\n", lSamplerate);                // print it. Output should be „Sample rate = 1000000"
```

Minimum internal sampling rate

The minimum internal sampling rate on all M2i cards is limited to 1 kS/s and the maximum sampling rate depends on the specific type of board. The maximum sampling rates for your type of card are shown in the tables below.

Maximum internal sampling rate in MS/s

activated Channels							
Ch0	Ch1	Ch2	Ch3	M2i.2020	M2i.2021	M2i.2030	M2i.2031
X				50 MS/s	50 MS/s	200 MS/s	200 MS/s
	X			50 MS/s	50 MS/s	100 MS/s	100 MS/s
		X		n.a.	50 MS/s	n.a.	200 MS/s
			X	n.a.	50 MS/s	n.a.	100 MS/s
X	X			50 MS/s	50 MS/s	100 MS/s	100 MS/s
X		X		n.a.	50 MS/s	n.a.	200 MS/s
X			X	n.a.	50 MS/s	n.a.	100 MS/s
	X	X		n.a.	50 MS/s	n.a.	100 MS/s
	X		X	n.a.	50 MS/s	n.a.	100 MS/s
		X	X	n.a.	50 MS/s	n.a.	100 MS/s
X	X	X	X	n.a.	50 MS/s	n.a.	100 MS/s

Using plain Quartz1 without PLL

In some cases it is useful for the application not to have the on-board PLL activated. Although the PLL used on the Spectrum boards is a low-jitter version it still produces more clock jitter than a plain quartz oscillator. For these cases the Spectrum boards have the opportunity to switch off the PLL by software and use a simple clock divider.

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_QUARTZ1	2	Enables Quartz1 for sample clock generation	

The Quartz1 used on the board is similar to the maximum sampling rate the board can achieve. As with internal PLL mode it's also possible to program the clock mode first, set a desired sampling rate with the SPC_SAMPLERATE register and to read it back. The driver will internally set the divider and find the closest matching sampling rate. The result will then again be the best matching sampling rate.

If a sampling rate is generated internally, you can additionally enable the clock output. The clock will be available on the external clock connector and can be used to synchronize external equipment with the board.

Register	Value	Direction	Description
SPC_CLOCKOUT	20110	read/write	Enables clock output on external clock connector.On A/D and D/A cards only possible with internal clocking.

Using plain Quartz2 without PLL (optional)

In some cases it is necessary to use a special frequency for sampling rate generation. For these applications all cards of the M2i series can be equipped with a special customer quartz. Please contact Spectrum for details on available oscillators. If your card is equipped with a second oscillator you can enable it for sampling rate generation with the following register:

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_QUARTZ2	4	Enables optional quartz2 for sample clock generation	

In addition to the direct usage of the second clock oscillator one can program the internal clock divider to use slower sampling rates. As with internal PLL mode it's also possible to program the clock mode first, set a desired sampling rate with the SPC_SAMPLERATE register and to read it back. The result will then again be the best matching sampling rate.

If a sampling rate is generated internally, you can additionally enable the clock output. The clock will be available on the external clock connector and can be used to synchronize external equipment with the board.

Register	Value	Direction	Description
SPC_CLOCKOUT	20110	read/write	Enables clock output on external clock connector. Only possible with internal clocking.

External reference clock

If you have an external clock generator with a extremely stable frequency, you can use it as a reference clock. You can connect it to the external clock connector and the PLL will be fed with this clock instead of the internal reference. The following table shows how to enable the reference clock mode:

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 automatically then 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.

Register	Value	Direction	Description
SPC_REFERENCECLOCK	20140	read/write	Programs the external reference clock in the range from 1 MHz to 125 MHz.
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_i32 (hDrv, SPC_SAMPLERATE,     25000000);        // We want to have 25 MHz as sampling rate
```

Termination of the clock input

If the external connector is used as an input, either for feeding in an external reference clock or for external clocking you can enable a 50 Ohm termination on the board. If the termination is disabled, the impedance is high. Please make sure that your source is capable of driving that current and that it still fulfills the clock input specification as given in the technical data section.

Register	Value	Direction	Description
SPC_CLOCK50OHM	20120	read/write	A „1“ enables the 50 Ohm termination at the external clock connector. Only possible, when using the external connector as an input.

Oversampling

All fast A/D converters have a minimum clock frequency that is defined by the manufacturer of this A/D converter. You find this minimum sampling rate specified in the technical data section as minimum external sampling clock.

When using one of the above mentioned internal clock modes the driver allows you to program sampling clocks that lay far beneath this minimum A/D converter clock. To run the A/D converter properly we use a special oversampling mode where the A/D converter is within it’s 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

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.

The oversampling factor is of interest for three different cases:

- When using clock output the sampling clock at the output connector is the real A/D converter 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.
- As all modern A/D converters have a data pipeline integrated to obtain high speed sampling together with high resolution there is a delay between the trigger and the valid data. Our hardware compensates this delay internally as long as sampling is done synchronous. When oversampling is active this compensation no longer works and data is shifted compared to the trigger position by a couple of samples.
- When using the timestamp option the counter is also running with the real A/D converter clock and not with the programmed slower sampling clock. When interpreting timestamp values it is therefore necessary to check the oversampling factor and take it into account.

External clocking

Direct external clock

An external clock can be fed in on the external clock connector of the board. This can be any clock, that matches the specification of the card. The external clock signal can be used to synchronize the card on a system clock or to feed in an exact matching sampling rate.

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_EXTERNAL	8	Enables external clock input for direct sample clock generation	

The maximum values for the external clock is board dependant and shown in the table below.

Termination of the clock input

If the external connector is used as an input, either for feeding in an external reference clock or for external clocking you can enable a 50 Ohm termination on the board. If the termination is disabled, the impedance is high. Please make sure that your source is capable of driving that current and that it still fulfills the clock input specification as given in the technical data section.

Register	Value	Direction	Description
SPC_CLOCK50OHM	20120	read/write	A „1“ enables the 50 Ohm termination at the external clock connector. Only possible, when using the external connector as an input.

Minimum external sampling rate

The minimum external sampling rate is limited on all boards to 1 MS/s and the maximum sampling rate depends on the specific type of board. The maximum sampling rates for your type of board are shown in the tables below.

Maximum external sampling rate in MS/s

Activated Channels	M2i.2020	M2i.2021	M2i.2030	M2i.2031
1	50 MS/s	50 MS/s	100 MS/s	100 MS/s
2	50 MS/s	50 MS/s	100 MS/s	100 MS/s
4	n.a.	50 MS/s	n.a.	100 MS/s

An external sampling rate above the mentioned maximum can cause damage to the board.

Ranges for external sampling rate

Due to the internal structure of the board it is essential to know for the driver in which clock range the external clock is operating. The external range register must be set according to the clock that is fed in externally.

Register	Value	Direction	Description
SPC_EXTERNRANGE	20130	read/write	Defines the range of the actual fed in external clock. Use one of the below mentioned ranges
EXRANGE_LOW	64	External range for slower clocks	
EXRANGE_HIGH	128	External range for faster clocks	

The range must not be left by more than 5% when the board is running. Remember that the ranges depend on the activated channels as well, so a different board setup for external clocking must always include the related clock ranges.

This table below shows the ranges that are defined by the two range registers mentioned above. The range depends on the activated channels per module. For details about what channels of your specific type of board is located on which module, please take a look at the according introduction chapter. Please be sure to select the correct external range, as otherwise it is possible that the card will not run properly.

	For cards with 8 bit converter resolution		For cards with 12, 14, 16 bit converter resolution	
Activated Channels on one module	EXRANGE_LOW	EXRANGE_HIGH	EXRANGE_LOW	EXRANGE_HIGH
1	< 50.0 MHz	>= 50.0 MHz	< 50.0 MHz	>= 50.0 MHz
2	< 50.0 MHz	>= 50.0 MHz	< 25.0 MHz	>= 25.0 MHz
4	< 25.0 MHz	>= 25.0 MHz	< 12.5 MHz	>= 12.5 MHz
8	< 12.5 MHz	>= 12.5 MHz	< 6.0 MHz	>= 6.0 MHz

How to read this table? If you have a card with a total number of four channels (available on two modules with two channels each), you have an external clock source with 30 MHz and you activate channel 0 and channel 2 (one channel per module), you will have to set the external range to EXRANGE_LOW. If you activate channel 0 and channel 1 on the same card and use the same 30 MHz external clock, you will have to set the external range EXRANGE_HIGH instead. Example:

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_EXTERNAL);        // activate ext. clock (which is e.g. 30 MHz)
spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0 | CHANNEL1);     // activate two channels (asuming that they
                                                                      // are located on one module) you
spcm_dwSetParam_i32 (hDrv, SPC_EXTERNRANGE, EXRANGE_HIGH);            // set external range to EXRANGE_HIGH
```

Further external clock details

- When using the high clock range the external clock has to be stable, needs to be continuously and is not allowed to have gaps or fast changes in frequency.
- When using the high clock range there must be a valid external clock be present before the start command is given.
- The external clock is directly used to feed the converters (on analog boards) or to feed the input registers (on digital boards). Therefore the quality and jitter of this clock may improve or degrade the dynamic performance of the card depending on the quality of the provided clock.
- When using the low clock range the clock needn’t to be continuously and may have gaps. When using a A/D card please keep in mind that most A/D converters need a stable clock and there might be false samples inbetween directly after a gap or after a fast clock fre-

quency change. The quality of the analog samples may also be worse than with a continuous clock.

External clock with divider

In some cases it is necessary to generate a slower frequency for sampling rate generation, than the available external source delivers. For these applications one can use an external clock and divide it.

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_EXTDIVIDER	16	Enables external clock input for divided sample clock generation	

The value for the clock divider must be written to the register shown in the table below:

Register	Value	Direction	Description
SPC_CLOCKDIV	20040	read/write	Register for setting the clock divider. Values up to 8190 in steps of two are allowed.

Please set the external clock range register matching your fed in clock.

Ranges for external sampling rate

Due to the internal structure of the board it is essential to know for the driver in which clock range the external clock is operating. The external range register must be set according to the clock that is fed in externally.

Register	Value	Direction	Description
SPC_EXTERNRANGE	20130	read/write	Defines the range of the actual fed in external clock. Use one of the below mentioned ranges
EXRANGE_LOW	64	External range for slower clocks	
EXRANGE_HIGH	128	External range for faster clocks	

The range must not be left by more than 5% when the board is running. Remember that the ranges depend on the activated channels as well, so a different board setup for external clocking must always include the related clock ranges.

This table below shows the ranges that are defined by the two range registers mentioned above. The range depends on the activated channels per module. For details about what channels of your specific type of board is located on which module, please take a look at the according introduction chapter. Please be sure to select the correct external range, as otherwise it is possible that the card will not run properly.

Activated Channels on one module	EXRANGE_LOW	EXRANGE_HIGH
1	< 50.0 MHz	>= 50.0 MHz
2	< 25.0 MHz	>= 25.0 MHz
3	< 16.0 MHz	>= 16.0 MHz
4	< 12.5 MHz	>= 12.5 MHz
5	< 10.0 MHz	>= 10.0 MHz
6	< 8.0 MHz	>= 8.0 MHz
7	< 7.0 MHz	>= 7.0 MHz
8	< 6.0 MHz	>= 6.0 MHz

How to read this table? If you have a card with a total number of four channels (available on two modules with two channels each), you have an external clock source with 30 MHz and you activate channel 0 and channel 2 (one channel per module), you will have to set the external range to EXRANGE_LOW. If you activate channel 0 and channel 1 on the same card and use the same 30 MHz external clock, you will have to set the external range EXRANGE_HIGH instead. Example:

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_EXTERNAL);        // activate ext. clock (which is e.g. 30 MHz)
spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0 | CHANNEL1);     // activate two channels (asuming that they
                                                                      // are located on one module) you
spcm_dwSetParam_i32 (hDrv, SPC_EXTERNRANGE, EXRANGE_HIGH);            // set external range to EXRANGE_HIGH
```

Further external clock details

- When using the high clock range the external clock has to be stable, needs to be continuously and is not allowed to have gaps or fast changes in frequency.
- When using the high clock range there must be a valid external clock be present before the start command is given.
- The external clock is directly used to feed the converters (on analog boards) or to feed the input registers (on digital boards). Therefore the quality and jitter of this clock may improve or degrade the dynamic performance of the card depending on the quality of the provided clock.
- When using the low clock range the clock needn't to be continuously and may have gaps. But as A/D converters need a stable clock there might be false samples inbetween directly after a gap or after a fast clock frequency change. The quality of the analog samples may

also be worse than with a continuous clock.

Termination of the clock input

If the external connector is used as an input, either for feeding in an external reference clock or for external clocking you can enable a 50 Ohm termination on the board. If the termination is disabled, the impedance is high. Please make sure that your source is capable of driving that current and that it still fulfills the clock input specification as given in the technical data section.

Register	Value	Direction	Description
SPC_CLOCK50OHM	20120	read/write	A „1" enables the 50 Ohm termination at the external clock connector. Only possible, when using the external connector as an input.

Trigger modes and appendant registers

General Description

The trigger modes of the Spectrum M2i series A/D cards are very extensive and give you the possibility to detect nearly any trigger event, you can think of.

You can choose between seven external TTL 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.

Every analog Spectrum board has one dedicated SMB connector mounted in it's bracket for feeding in an external trigger signal or generating a trigger output of an internal trigger event. Due to the fact that only one connector is available for external trigger I/O, it is not possible to forward the fed in external trigger signal to another board. If this is however necessary, you need to split up the external trigger signal before.

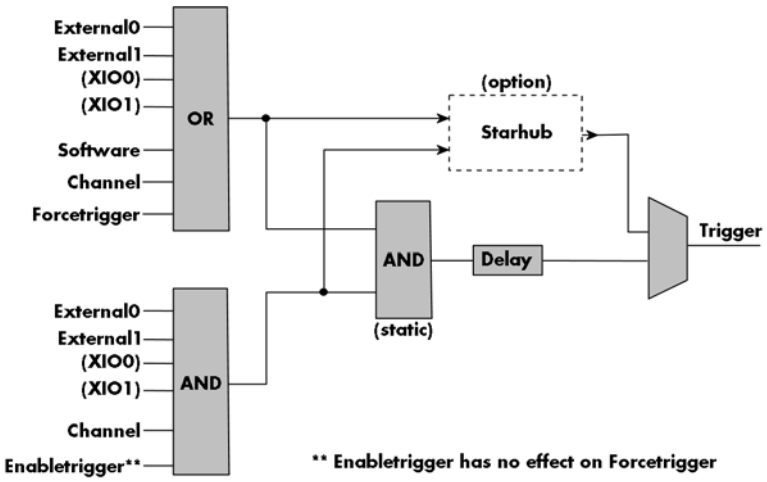
Trigger Engine Overview

To extend trigger facilities of the various trigger sources/modes further on, the trigger engine of the Spectrum M2i series allows the logical combination of different trigger events by an AND-mask and an OR-mask.

The Enable trigger allows the user to enable or disable all trigger sources (including channel trigger and external TTL trigger) with a single software command.

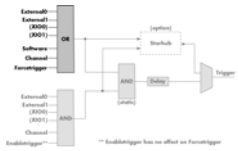
When the card is waiting for a trigger event, either a channel trigger or an external trigger the force-trigger command allows to force a trigger event with a single software command.

Before the trigger event is finally generated, it is wired through a programmable trigger delay.



Trigger masks

Trigger OR mask



The purpose of this passage is to explain the trigger OR mask (see left figure) and all the appendant software registers in detail.

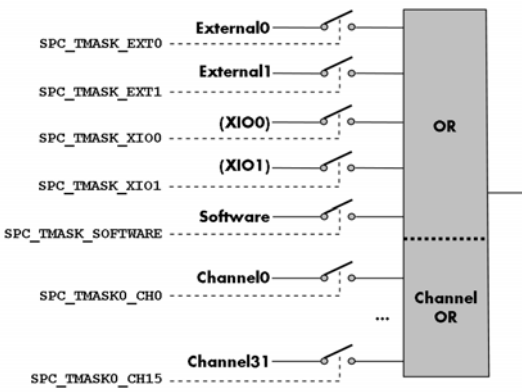
The OR mask shown in the overview before as one object, is separated into two parts: a general OR mask for external TTL trigger and software trigger and a channel OR mask.

Every trigger source of the M2i 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 is realized with the SPC_TRIG_CH_ORMASK0 and the SPC_TRIG_CH_ORMASK1 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.



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

Register	Value	Direction	Description
SPC_TRIG_AVAILORMASK	40400	read	Bitmask, in which all bits of the below mentioned sources for the OR mask are set, if available.
SPC_TRIG_ORMASK	40410	read/write	Defines the events included within the trigger OR mask of the card.
SPC_TMASK_NONE	0	No trigger source selected	
SPC_TMASK_SOFTWARE	1h	Enables the software trigger for the OR mask. The card will trigger immediately after start.	
SPC_TMASK_EXT0	2h	Enables the external trigger0 for the OR mask. The card will trigger when the programmed condition for this input is valid.	
SPC_TMASK_EXT1	4h	Enables the external trigger1 for the OR mask. This input is only available on digital cards. The card will trigger when the programmed condition for this input is valid.	
SPC_TMASK_XIO0	100h	Enables the extra TTL trigger 0 for the OR mask. This input is only available when the option BaseXIO is installed.	
SPC_TMASK_XIO1	200h	Enables the extra TTL trigger 1 for the OR mask. This input is only available when the option BaseXIO is installed.	

The following example shows, how to setup the OR mask, for an external TTL trigger. As an example a simple edge detection has been chosen. The explanation and a detailed description of the different trigger modes for the external TTL trigger inputs will be shown in the dedicated passage within this chapter.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0); // Enable external trigger within the OR mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_POS);  // Setting up external TTL trigger for rising edges
```

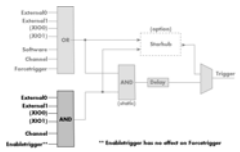
The table below is showing the registers for the channel OR mask and the possible constants that can be written to it.

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILORMASK0	40450	read	Bitmask, in which all bits of the below mentioned sources/channels (0…31) for the channel OR mask are set, if available.
SPC_TRIG_CH_AVAILORMASK1	40451	read	Bitmask, in which all bits of the below mentioned sources/ channels (32…63) for the channel OR mask are set, if available.
SPC_TRIG_CH_ORMASK0	40460	read/write	Includes the analog or digital channels (0…31) within the channel trigger OR mask of the card.
SPC_TRIG_CH_ORMASK1	40461	read/write	Includes the analog or digital channels (32…63) within the channel trigger OR mask of the card.
SPC_TMASK0_CH0	1h	Enables channel0 (channel32) for recognition within the channel OR mask.	
SPC_TMASK0_CH1	2h	Enables channel1 (channel33) for recognition within the channel OR mask.	
SPC_TMASK0_CH2	4h	Enables channel2 (channel34) for recognition within the channel OR mask.	
SPC_TMASK0_CH3	8h	Enables channel3 (channel35) for recognition within the channel OR mask.	
…	…	…	
SPC_TMASK0_CH28	10000000h	Enables channel28 (channel60) for recognition within the channel OR mask.	
SPC_TMASK0_CH29	20000000h	Enables channel29 (channel61 for recognition within the channel OR mask.	
SPC_TMASK0_CH30	40000000h	Enables channel30 (channel62) for recognition within the channel OR mask.	
SPC_TMASK0_CH31	80000000h	Enables channel31 (channel63) for recognition within the channel OR mask.	

The following example shows, how to setup the OR mask, for an external TTL trigger. As an example a simple edge detection has been chosen. The explanation and a detailed description of the different trigger modes for the external TTL trigger inputs will be shown in the dedicated passage within this chapter.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ORMASK0, SPC_TMASK_CH0); // Enable channel0 trigger within the OR mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE,  SPC_TM_POS);    // Setting up external 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.

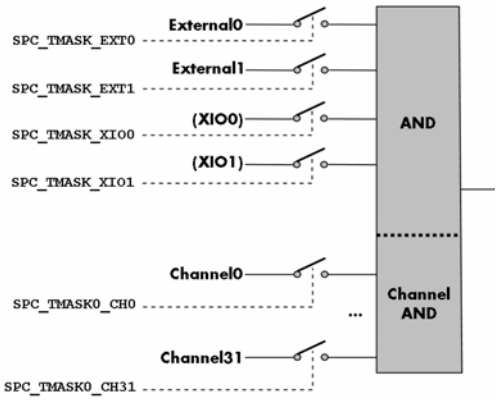
The AND mask shown in the overview before as one object, is separated into two parts: a general AND mask for external TTL trigger and software trigger and a channel AND mask.

Every trigger source of the M2i 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 is realized with the SPC_TRIG_CH_ANDMASK0 and the SPC_TRIG_CH_ANDMASK1 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.

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



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_EXT0	2h	Enables the external trigger0 for the AND mask. The card will trigger when the programmed condition for this input is valid.	
SPC_TMASK_EXT1	4h	Enables the external trigger1 for the AND mask. This input is only available on digital cards. The card will trigger when the programmed condition for this input is valid.	
SPC_TMASK_XIO0	100h	Enables the extra TTL trigger 0 for the OR mask. This input is only available when the option BaseXIO is installed.	
SPC_TMASK_XIO1	200h	Enables the extra TTL trigger 1 for the OR mask. This input is only available when the option BaseXIO is installed.	

The following example shows, how to setup the AND mask, for an external TTL trigger. As an example a simple level detection has been chosen. The explanation and a detailed description of the different trigger modes for the external TTL trigger inputs will be shown in the dedicated passage within this chapter.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ANDMASK, SPC_TMASK_EXT0); // Enable external trigger within the AND mask
spcm_dwSetParam_i32 (hDrv,SPC_TRIG_EXT0_MODE, SPC_TM_HIGH  ); // Setting up external TTL trigger for HIGH level
```

The table below is showing the constants for the channel AND mask and all the constants for the different channels.

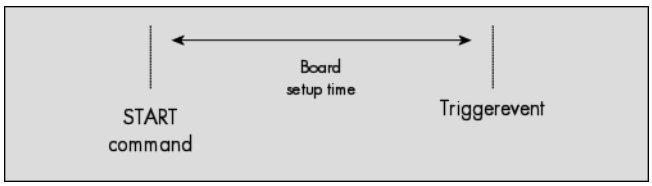
Register	Value	Direction	Description
SPC_TRIG_CH_AVAILANDASK0	40470	read	Bitmask, in which all bits of the below mentioned sources/channels (0…31) for the channel AND mask are set, if available.
SPC_TRIG_CH_AVAILANDMASK1	40471	read	Bitmask, in which all bits of the below mentioned sources/ channels (32…63) for the channel AND mask are set, if available.
SPC_TRIG_CH_ANDMASK0	40480	read/write	Includes the analog or digital channels (0…31) within the channel trigger AND mask of the card.
SPC_TRIG_CH_ANDRMASK1	40481	read/write	Includes the analog or digital channels (32…63) within the channel trigger AND mask of the card.
SPC_TMASK0_CH0	1h	Enables channel0 (channel16) for recognition within the channel AND mask.	
SPC_TMASK0_CH1	2h	Enables channel1 (channel17) for recognition within the channel AND mask.	
SPC_TMASK0_CH2	4h	Enables channel2 (channel18) for recognition within the channel AND mask.	
SPC_TMASK0_CH3	8h	Enables channel3 (channel19) for recognition within the channel AND mask.	
…	…	…	
SPC_TMASK0_CH28	10000000h	Enables channel28 (channel60) for recognition within the channel AND mask.	
SPC_TMASK0_CH29	20000000h	Enables channel29 (channel61 for recognition within the channel AND mask.	
SPC_TMASK0_CH30	40000000h	Enables channel30 (channel62) for recognition within the channel AND mask.	
SPC_TMASK0_CH31	80000000h	Enables channel31 (channel63) for recognition within the channel AND mask.	

The following example shows, how to setup the AND mask, for an external TTL trigger. As an example a simple level detection has been chosen. The explanation and a detailed description of the different trigger modes for the external TTL trigger inputs will be shown in the dedicated passage within this chapter.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ANDMASK0, SPC_TMASK_CH0); // Enable channel0 trigger within the AND mask
spcm_dwSetParam_i32 (hDrv,SPC_TRIG_EXT0_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 starting the board. The only delay results from the time the board needs for its setup.
For enabling the software trigger one simply has to include the software event within the trigger OR mask, as the following table is showing:



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.	

Due to the fact that the software trigger is an internal trigger mode, you can optionally enable the external trigger output to generate a high active trigger signal, which indicates when the data acquisition or replay begins. This can be useful to synchronize external equipment with your Spectrum board.

Register	Value	Direction	Description
SPC_TRIG_OUTPUT	40100	read/write	Defines the data direction of the external trigger connector.
	0	The trigger connector is not used and the line driver is disabled.	
	1	The trigger connector is used as an output that indicates a detected internal trigger event.	

Example for setting up the software trigger:

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_SOFTWARE); // Internal software trigger mode is used
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_OUTPUT, 1                 ); // And the trigger output is enabled
```

Force- and Enable trigger

In addition to the softwaretrigger (free run) it is also possible to force a triggerevent by software while the board is waiting for an internal or external 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.

Register	Value	Direction	Description
SPC_M2CMD	100	write	Command register of the M2i 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_dwSetParem_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_FORCETRIGGER); // Forcetrigger 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 enabled.

Register	Value	Direction	Description
SPC_M2CMD	100	write	Command register of the M2i 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.	

The example shows, how to arm and disarm the card's trigger engine properly:

```
spcm_dwSetParem_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_ENABLETRIGGER); // Trigger engine is armed.
...
spcm_dwSetParem_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_DISABLETRIGGER); // Trigger engine is disarmed.
```

Delay trigger

All of the Spectrum M2i 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. The following table shows the related register and the possible values. A value of 0 disables the extra delay. The resulting delays (due to the

internal structure of the card) can be found in the technical data section of this manual.

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.
0…65535			Defines the additional trigger delay in number of sample clocks.

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

```
spcm_dwSetParem_i32 (hDrv, SPC_TRIG_DELAY, 2000); // A detected trigger event will be
                                                  // delayed for 2000 sample clocks.
```

External TTL trigger

Enabling the external trigger input(s) is done, if you choose one of the following external trigger modes. The dedicated register for that operation is shown below.

Register	Value	Direction	Description
SPC_TRIG_EXT_AVAILMODES	40500	read	Bitmask, in which all bits of the below mentioned modes for the external trigger are set, if available.
SPC_TRIG_EXT0_MODE	40510	read/write	Defines the external TTL trigger mode for the external SMB connector (A/D and D/A boards only). On digital boards this defines the TTL trigger mode for the trigger input of the first module (Mod A).
SPC_TRIG_EXT1_MODE	40511	read/write	Defines the external TTL trigger mode for the trigger input of the second module (digital boards only).
SPC_TRIG_XIO0_MODE	40560	read/write	Defines the trigger mode for the extra TTL input 0. These trigger inputs are only available, when option BaseXIO is installed.
SPC_TRIG_XIO1_MODE	40561	read/write	Defines the trigger mode for the extra TTL input 1. These trigger inputs are only available, when option BaseXIO is installed.
SPC_TM_POS	1h		Sets the trigger mode for external TTL trigger to detect positive edges.
SPC_TM_NEG	2h		Sets the trigger mode for external TTL trigger to detect negative edges
SPC_TM_BOTH	4h		Sets the trigger mode for external TTL trigger to detect positive and negative edges
SPC_TM_HIGH	8h		Sets the trigger mode for external TTL trigger to detect HIGH levels.
SPC_TM_LOW	10h		Sets the trigger mode for external TTL trigger to detect LOW levels.
SPC_TM_POS | SPC_TM_PW_GREATER	4000001h		Sets the trigger mode for external TTL trigger to detect HIGH pulses that are longer than a programmed pulsewidth.
SPC_TM_POS | SPC_TM_PW_SMALLER	2000001h		Sets the trigger mode for external TTL trigger to detect HIGH pulses that are shorter than a programmed pulsewidth.
SPC_TM_NEG | SPC_TM_PW_GREATER	4000002h		Sets the trigger mode for external TTL trigger to detect LOW pulses that are longer than a programmed pulsewidth.
SPC_TM_NEG | SPC_TM_PW_SMALLER	2000002h		Sets the trigger mode for external TTL trigger to detect LOW pulses that are shorter than a programmed pulsewidth.

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

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

If you choose an external trigger mode the SPC_TRIGGEROUT register will be overwritten and the trigger connector will be used as an input any ways.

Register	Value	Direction	Description
SPC_TRIG_OUTPUT	40100	read/write	Enables the trigger output if internal trigger is detected
	X		If external trigger modes are used, this register will have no effect.

As the trigger connector is used as an input, you can decide whether the input is 50 Ohm terminated or not. 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 fulfil the trigger input specification given in the technical data section. If termination is disabled, the input is at high impedance.

Register	Value	Direction	Description
SPC_TRIG_TERM	40110	read/write	A „1“ sets the 50 Ohm termination, if the trigger connector is used as an input for external trigger signals. A „0“ sets the high impedance termination

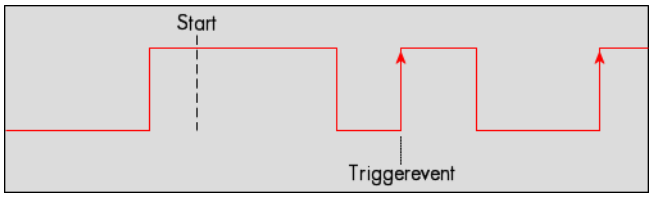
The following short example shows how to set up the board for external positive edge TTL trigger. The trigger input is 50 Ohm terminated. The different modes for external TTL trigger are to be detailed described in the next few passages.

```
spcm_dwSetParam_i32 (hDrv,SPC_TRIG_EXT0_MODE, SPC_TM_POS  );     // Setting up external TTL
                                                                 // trigger to detect rising edges
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_TERM, 1               );     // Enables the 50 Ohm input termination
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0);     // and enable it within the OR mask
```

Edge and level triggers

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 triggerevent will then be detected, if the actual recording/replay has finished and the board is armed and waiting for a trigger again.



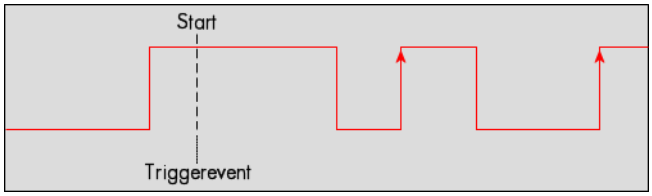
Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board.
SPC_TM_POS	1h		Sets the trigger mode for external TTL trigger to detect positive edges.

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

```
SpcSetParam (hDrv, SPC_TRIGGERMODE, TM_TTLPOS); // Setting up external TTL trigger to detect positive edges
```

HIGH level TTL trigger

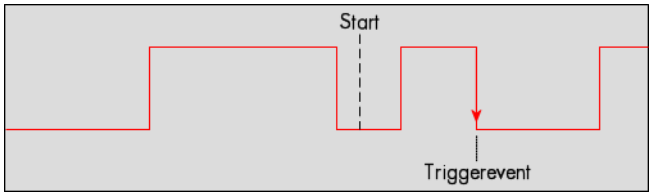
This mode is for detecting the HIGH levels of an external TTL signal. The board will trigger on the first HIGH level that is detected after starting the board. If this condition is fulfilled when the board is started, a trigger event will be detected.
The next triggerevent will then be detected, if the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



Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board.
SPC_TM_HIGH	8h		Sets the trigger mode for external TTL trigger to detect HIGH levels.

Negative TTL trigger

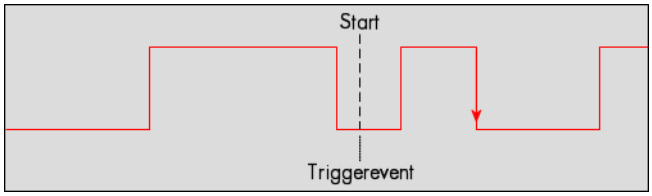
This mode is for detecting the falling edges of an external TTL signal. The board will trigger on the first falling edge that is detected after starting the board. The next triggerevent will then be detected, if the actual recording/replay has finished and the board is armed and waiting for a trigger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



Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board.
SPC_TM_NEG	2h		Sets the trigger mode for external TTL trigger to detect negative edges.

LOW level TTL trigger

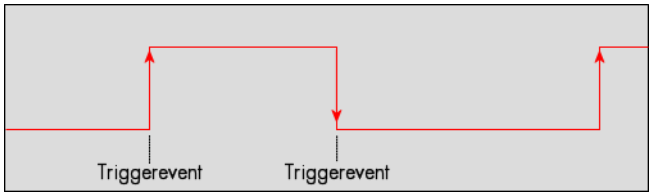
This mode is for detecting the LOW levels of an external TTL signal. The board will trigger on the first LOW level that is detected after starting the board. If this condition is fulfilled when the board is started, a trigger event will be detected.
The next triggerevent will then be detected, if the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



Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board.
SPC_TM_LOW	10h		Sets the trigger mode for external TTL trigger to detect LOW levels.

Positive and negative 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 triggerevent will then be detected, if the actual recording/replay has finished and the board is armed and waiting for a trigger again.

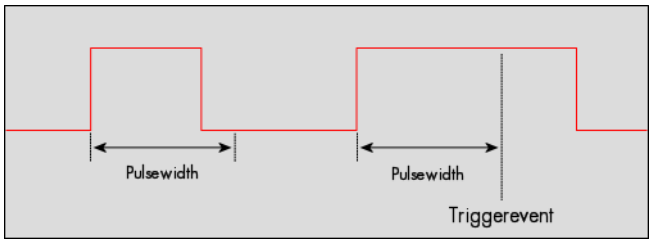


Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board.
SPC_TM_BOTH	4h		Sets the trigger mode for external TTL trigger to detect positive and negative edges.

Pulsewidth triggers

TTL pulsewidth trigger for long HIGH pulses

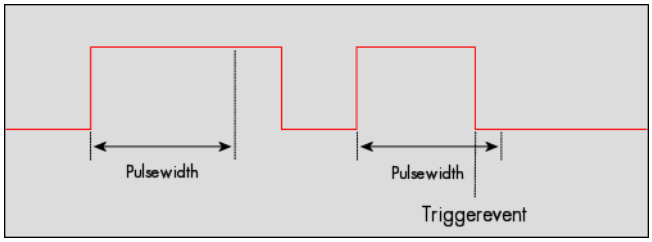
This mode is for detecting HIGH pulses of an external TTL signal that are longer than a programmed pulsewidth. If the pulse is shorter than the programmed pulsewidth, no trigger will be detected. The board will trigger on the first pulse matching the trigger condition after starting the board. The next triggerevent 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_EXT0_PULSEWIDTH	44210	read/write	Sets the pulsewidth in samples.	2 up to 65535
SPC_TRIG_EXT0_MODE	40510	read/write	(SPC_TM_POS | SPC_TM_PW_GREATER)	4000001h

TTL pulsewidth trigger for short HIGH pulses

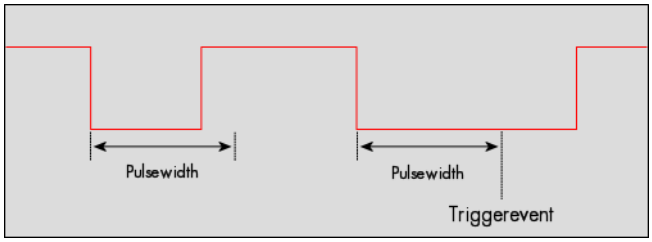
This mode is for detecting HIGH pulses of an external TTL signal that are shorter than a programmed pulsewidth. If the pulse is longer than the programmed pulsewidth, no trigger will be detected. The board will trigger on the first pulse matching the trigger condition after starting the board. The next triggerevent 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_EXT0_PULSEWIDTH	44210	read/write	Sets the pulsewidth in samples.	2 up to 65535
SPC_TRIG_EXT0_MODE	40510	read/write	(SPC_TM_POS | SPC_TM_PW_SMALLER)	2000001h

TTL pulsewidth trigger for long LOW pulses

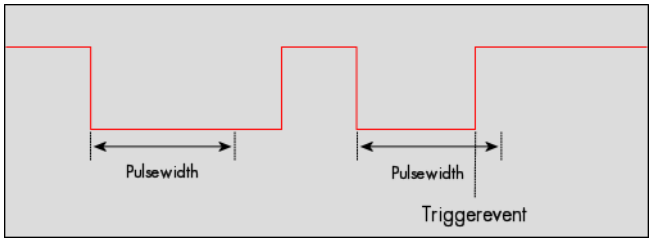
This mode is for detecting LOW pulses of an external TTL signal that are longer than a programmed pulsewidth. If the pulse is shorter than the programmed pulsewidth, no trigger will be detected. The board will trigger on the first pulse matching the trigger condition after starting the board. The next triggerevent 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_EXT0_PULSEWIDTH	44210	read/write	Sets the pulsewidth in samples.	2 up to 65535
SPC_TRIG_EXT0_MODE	40510	read/write	(SPC_TM_NEG | SPC_TM_PW_GREATER)	4000002h

TTL pulsewidth trigger for short LOW pulses

This mode is for detecting LOW pulses of an external TTL signal that are shorter than a programmed pulsewidth. If the pulse is longer than the programmed pulsewidth, no trigger will be detected. The board will trigger on the first pulse matching the trigger condition after starting the board. The next triggerevent 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_EXT0_PULSEWIDTH	44210	read/write	Sets the pulsewidth in samples.	2 up to 65535
SPC_TRIG_EXT0_MODE	40510	read/write	(SPC_TM_NEG | SPC_TM_PW_SMALLER)	2000002h

The following example shows, how to setup the card for using external TTL pulse width trigger:

```
spcm_dwSetParam_i32 (hDrv,SPC_TRIG_EXT0_MODE, SPC_TM_NEG | SPC_TM_PW_GREATER); // Setting up external TTL
                                                                               // trigger to detect low pulses
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_PULSEWIDTH ,                  50); // that are longer than 50 samples.
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK,                SPC_TMASK_EXT0); // and enable it within the OR mask
```

To find out, what maximum pulsewidth (in samples) is available, please read out the register shown in the table below:

Register	Value	Direction	Description
SPC_TRIG_EXT_AVAILPULSEWIDTH	44200	read	Contains the maximum possible value, for the external trigger pulsewidth counter.

Channel Trigger

Overview of the channel trigger registers

The channel trigger modes are the most common modes, compared to external equipment like oscilloscopes. The 19 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):.

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, a error message will be returned.

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_TRIG_CH2_MODE	40612	read/write	Sets the trigger mode for channel 2. Channel 2 must be enabled in the channel OR/AND mask.
SPC_TRIG_CH3_MODE	40613	read/write	Sets the trigger mode for channel 3. Channel 3 must be enabled in the channel OR/AND mask.
SPC_TM_NONE	0h	Channel is not used for trigger detection. This is as with the trigger masks another possibility for disabling channels.	
SPC_TM_POS	1h	Enables the trigger detection for positive edges	
SPC_TM_NEG	2h	Enables the trigger detection for negative edges	
SPC_TM_BOTH	4h	Enables the trigger detection for positive and negative edges	
SPC_TM_HIGH	8h	Enables the trigger detection for HIGH levels	
SPC_TM_LOW	10h	Enables the trigger detection for LOW levels	
SPC_TM_POS | SPC_TM_PW_GREATER	4000001h	Enables the pulsewidth trigger detection for long positive pulses	
SPC_TM_NEG | SPC_TM_PW_GREATER	4000002h	Enables the pulsewidth trigger detection for long negative pulses	
SPC_TM_POS | SPC_TM_PW_SMALLER	2000001h	Enables the pulsewidth trigger detection for short positive pulses	
SPC_TM_NEG | SPC_TM_PW_SMALLER	2000002h	Enables the pulsewidth trigger detection for short negative pulses	
SPC_TM_STEEPPOS | SPC_TM_PW_GREATER	4000800h	Enables the steepness trigger detection for flat positive pulses	
SPC_TM_STEEPNEG | SPC_TM_PW_GREATER	4001000h	Enables the steepness trigger detection for flat negative pulses	
SPC_TM_STEEPPOS | SPC_TM_PW_SMALLER	2000800h	Enables the steepness trigger detection for steep positive pulses	
SPC_TM_STEEPNEG | SPC_TM_PW_SMALLER	2000800h	Enables the steepness trigger detection for steep negative pulses	
SPC_TM_WINENTER	20h	Enables the window trigger for entering signals	
SPC_TM_WINLEAVE	40h	Enables the window trigger for leaving signals	
SPC_TM_INWIN	80h	Enables the window trigger for inner signals	
SPC_TM_OUTSIDEWIN	100h	Enables the window trigger for outer signals	
SPC_TM_WINENTER | SPC_TM_PW_GREATER	4000020h	Enables the window trigger for long inner signals	
SPC_TM_WINLEAVE | SPC_TM_PW_GREATER	4000040h	Enables the window trigger for long outer signals	
SPC_TM_WINENTER | SPC_TM_PW_SMALLER	2000020h	Enables the window trigger for short inner signals	
SPC_TM_WINLEAVE | SPC_TM_PW_SMALLER	2000040h	Enables the window trigger for short outer signals	

If you want to set up a four 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 TM_CHANNEL and the TM_CHOR 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_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 channel 0 to positive edge
```

If you want to set up a four channel board to detect a trigger event on either a positive edge on channel 1 or a negative edge on channel 3 you would have to set up your board as the following example shows.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ORMASK0, SPC_TMASK0_CH1 | SPC_TMASK0_CH3);   // Enable channel 1 + 3
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH1_MODE, SPC_TM_POS );    // Set triggermode of channel 1 to positive edge
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH3_MODE, SPC_TM_NEG );    // Set triggermode of channel 3 to negative edge
```

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 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.

8 bit resolution for the trigger levels:

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	-127 to +127
SPC_TRIG_CH1_LEVEL0	42201	read/write	Trigger level 0 channel 1: main trigger level / upper level if 2 levels used	-127 to +127
SPC_TRIG_CH2_LEVEL0	42202	read/write	Trigger level 0 channel 2: main trigger level / upper level if 2 levels used	-127 to +127
SPC_TRIG_CH3_LEVEL0	42203	read/write	Trigger level 0 channel 3: main trigger level / upper level if 2 levels used	-127 to +127
SPC_TRIG_CH0_LEVEL1	42300	read/write	Trigger level 1 channel 0: auxiliary trigger level / lower level if 2 levels used	-127 to +127
SPC_TRIG_CH1_LEVEL1	42301	read/write	Trigger level 1 channel 1: auxiliary trigger level / lower level if 2 levels used	-127 to +127
SPC_TRIG_CH2_LEVEL1	42302	read/write	Trigger level 1 channel 2: auxiliary trigger level / lower level if 2 levels used	-127 to +127
SPC_TRIG_CH3_LEVEL1	42303	read/write	Trigger level 1 channel 3: auxiliary trigger level / lower level if 2 levels used	-127 to +127

Trigger level representation depending on selected input range

	Input ranges						
Triggerlevel	**±50 mV**	**±100 mV**	**±200 mV**	**±500 mV**	**±1 V**	**±2 V**	**±5 V**
127	49,6 mV	99,2 mV	198,4 mV	496,1 mV	992,2 mV	1.984,4 mV	4.960,9 mV
126	49,2 mV	98,4 mV	196,9 mV	492,2 mV	984,4 mV	1.968,8 mV	4.921,9 mV
…							
64	25,0 mV	50,0 mV	100,0 mV	250,0 mV	500,0 mV	1.000,0 mV	2.500,0 mV
…							
2	0,8 mV	1,6 mV	3,1 mV	7,8 mV	15,6 mV	31,3 mV	78,1 mV
1	0,4 mV	0,8 mV	1,6 mV	3,9 mV	7,8 mV	15,6 mV	39,1 mV
0	0,0 mV	0,0 mV	0,0 mV	0,0 mV	0,0 mV	0,0 mV	0,0 mV
-1	-0,4 mV	-0,8 mV	-1,6 mV	-3,9 mV	-7,8 mV	-15,6 mV	-39,1 mV
-2	-0,8 mV	-1,6 mV	-3,1 mV	-7,8 mV	-15,6 mV	-31,3 mV	-78,1 mV
…							
-64	-25,0 mV	-50,0 mV	-100,0 mV	-250,0 mV	-500,0 mV	-1.000,0 mV	-2.500,0 mV
…							
-126	-49,2 mV	-98,4 mV	-196,9 mV	-492,2 mV	-984,4 mV	-1.968,8 mV	-4.921,9 mV
-127	-49,6 mV	-99,2 mV	-198,4 mV	-496,1 mV	-992,2 mV	-1.984,4 mV	-4.960,0 mV
Stepsize	**0.4 mV**	**0.8 mV**	**1.6 mV**	**3.9 mV**	**7.8 mV**	**15,6 mV**	**39.1 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 12.5 mV, which is the resulting trigger level.

```
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, 40);            // Sets triggerlevel to 62.5 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:

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

In case of a board that uses 8 bits for trigger detection the returned value would be 255, 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.

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

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.

$$\text{Trigger step width} = \frac{+1000\ \text{mV} - (-1000\ \text{mV})}{255 + 1}$$

Pulsewidth counter

Some of the trigger modes need an additional pulsewidth counter that is measuring the size of a pulse. All the trigger modes running with pulse width counters are able to detect a trigger event that is shorter than the programmed pulsewidth or that is longer than the programmed pulsewidth. Please see the detailed trigger mode description for further details.

To find out what maximum pulsewidth (in samples) is available for all the channel trigger modes it is possible to read out the maximum programmable pulsewidth counter using the register shown in the table below:

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILPULSEWIDTH	44100	r	Contains the maximum possible value, for the channel trigger pulsewidth counter.

Each channel trigger has it's own pulsewidth register:

Register	Value	Direction	Description	Range
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples for ch 0 trigger modes using pulsewidth counters	2 to 65535
SPC_TRIG_CH1_PULSEWIDTH	44102	read/write	Sets the pulsewidth in samples for ch 1 trigger modes using pulsewidth counters	2 to 65535
SPC_TRIG_CH2_PULSEWIDTH	44103	read/write	Sets the pulsewidth in samples for ch 2 trigger modes using pulsewidth counters	2 to 65535
SPC_TRIG_CH3_PULSEWIDTH	44104	read/write	Sets the pulsewidth in samples for ch 3 trigger modes using pulsewidth counters	2 to 65535

Please keep in mind that your card only has one channel pulsewidth counter available in hardware. It is not possible to use more than one channel trigger source when activating a pulsewidth trigger mode. The driver will then report an error.

Detailed description of the channel trigger modes

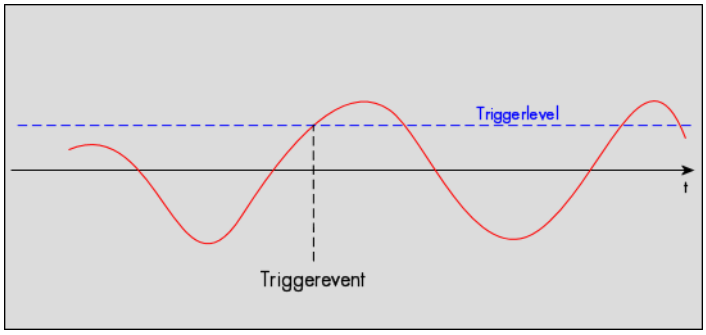
For all channel trigger modes, the OR mask must contain the corresponding input channels (channel 0 taken as example here):.

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.

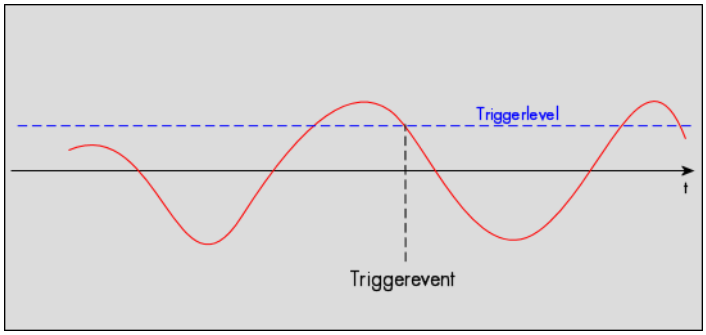


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS	1h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

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.

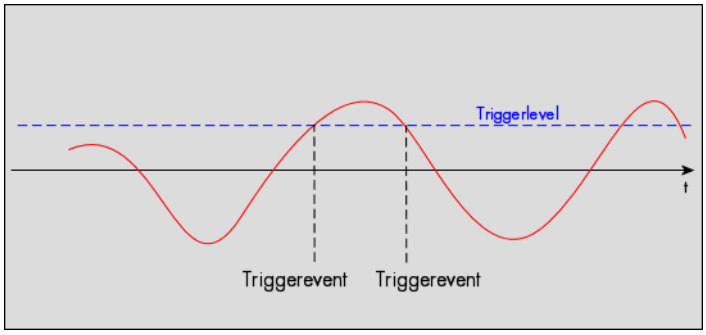


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG	2h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

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.

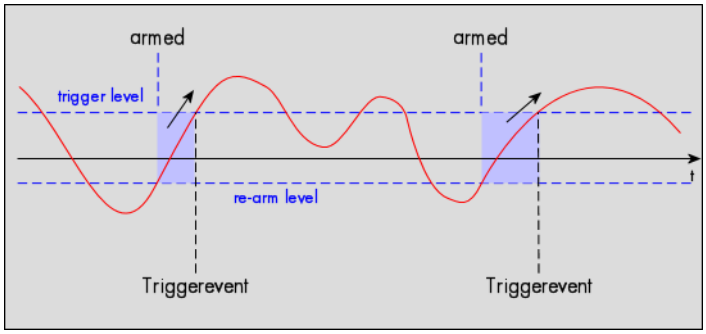


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_BOTH	4h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

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.

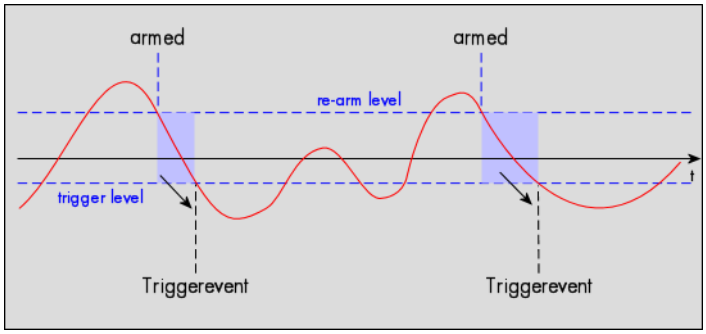


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS | SPC_TM_REARM	01000001h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the re-arm level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

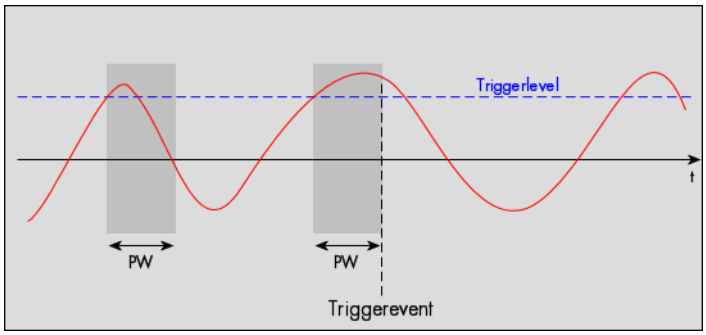


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG | SPC_TM_REARM	01000002h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Defines the re-arm level relatively to the channels's input range	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the desired trigger level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

Channel pulsewidth trigger for long positive pulses

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 to higher values (rising edge) the pulsewidth counter is started. If the signal crosses the trigger level again in the opposite direction within the the programmed pulsewidth time, no trigger will be detected. If the pulsewidth counter reaches the programmed amount of samples, without the signal crossing the trigger level in the opposite direction, the triggerevent will be detected.

The pulsewidth trigger modes for long pulses can be used to prevent the board from triggering on wrong (short) edges in noisy signals.

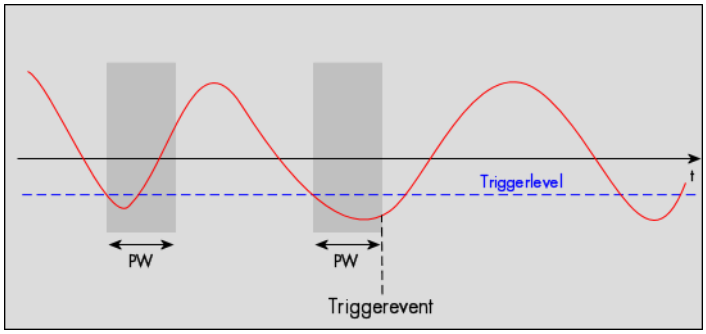


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS | SPC_TM_PW_GREATER	04000001h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel pulsewidth trigger for long negative pulses

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 to lower values (falling edge) the pulsewidth counter is started. If the signal crosses the trigger level again in the opposite direction within the the programmed pulsewidth time, no trigger will be detected. If the pulsewidth counter reaches the programmed amount of samples, without the signal crossing the trigger level in the opposite direction, the triggerevent will be detected.

The pulsewidth trigger modes for long pulses can be used to prevent the board from triggering on wrong (short) edges in noisy signals.

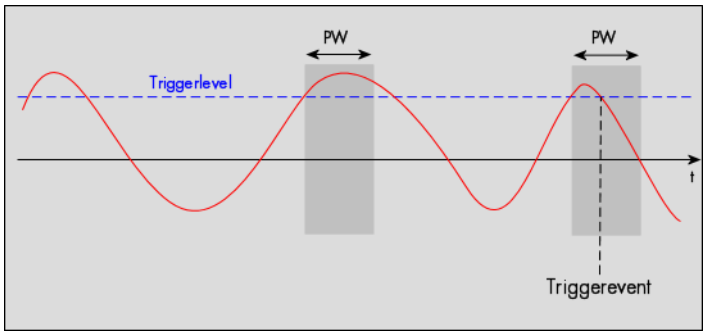


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG | SPC_TM_PW_GREATER	04000002h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel pulsewidth trigger for short positive pulses

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 to higher values (rising edge) the pulsewidth counter is started. If the pulsewidth counter reaches the programmed amount of samples, no trigger will be detected.

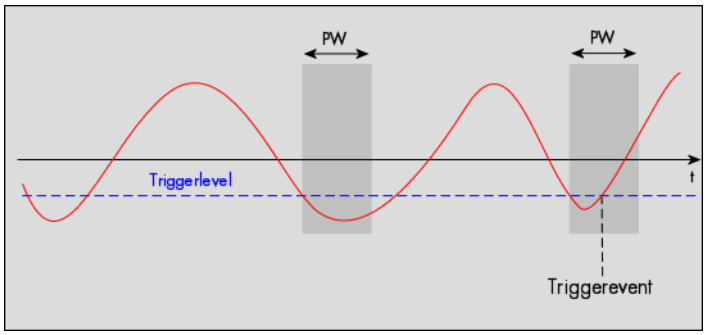
If the signal does cross the trigger level again within the the programmed pulsewidth time, a triggerevent will be detected.



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS | SPC_TM_PW_SMALLER	02000001h
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_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel pulsewidth trigger for short negative pulses

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 to lower values (falling edge) the pulsewidth counter is started. If the pulsewidth counter reaches the programmed amount of samples, no trigger will be detected.
If the signal does cross the trigger level again within the the programmed pulsewidth time, a triggerevent will be detected.

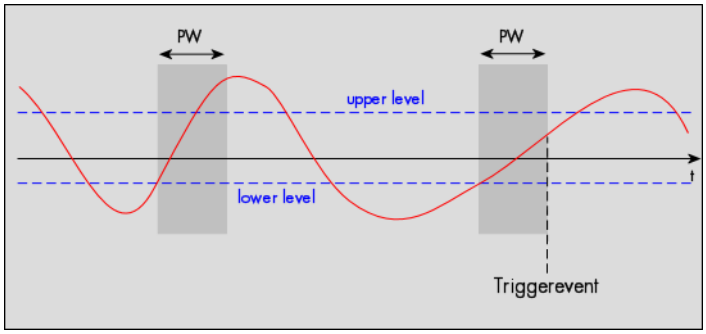


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG | SPC_TM_PW_SMALLER	02000002h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel steepness trigger for flat positive pulses

The analog input is continuously sampled with the selected sample rate. If the programmed lower level is crossed by the channel's signal from lower to higher values (rising edge) the pulsewidth counter is started. If the signal does cross the upper level within the the programmed pulsewidth time, no trigger will be detected.

If the pulsewidth counter reaches the programmed amount of samples a triggerevent will be detected.

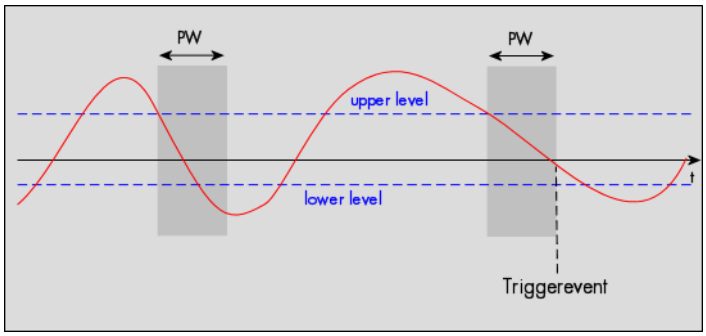


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_STEEPPOS | SPC_TM_PW_GREATER	04000800h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel steepness trigger for flat negative pulses

The analog input is continuously sampled with the selected sample rate. If the programmed upper level is crossed by the channel's signal from higher to lower values (falling edge) the pulsewidth counter is started. If the signal does cross the lower level within the the programmed pulsewidth time, no trigger will be detected.

If the pulsewidth counter reaches the programmed amount of samples a triggerevent will be detected.

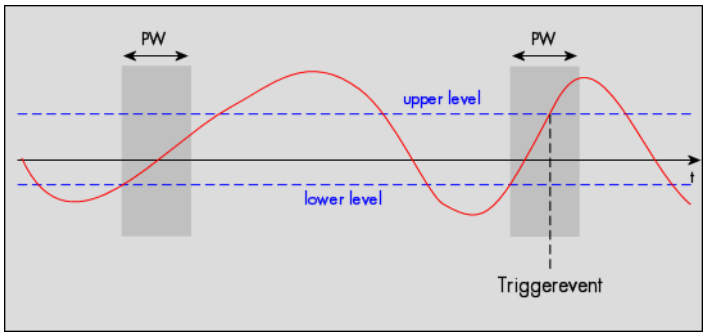


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_STEEPNEG | SPC_TM_PW_GREATER	04001000h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel steepness trigger for steep positive pulses

The analog input is continuously sampled with the selected sample rate. If the programmed lower level is crossed by the channel's signal from lower to higher values (rising edge) the pulsewidth counter is started. If the pulsewidth counter reaches the programmed amount of samples without the signal crossing the higher level, no trigger will be detected.

If the signal does cross the upper level within the the programmed pulsewidth time, a triggerevent will be detected.

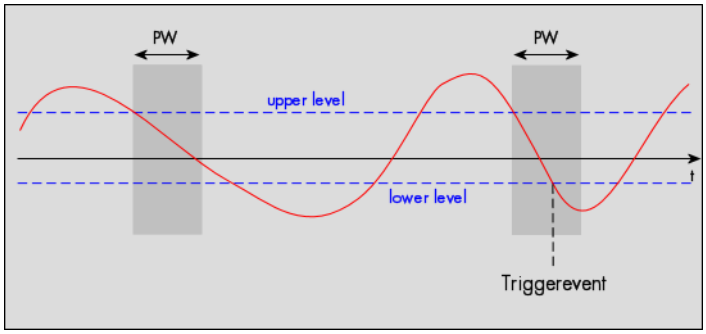


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_STEEPPOS | SPC_TM_PW_SMALLER	02000800h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel steepness trigger for steep negative pulses

The analog input is continuously sampled with the selected sample rate. If the programmed upper level is crossed by the channel's signal from higher to lower values (falling edge) the pulsewidth counter is started. If the pulsewidth counter reaches the programmed amount of samples without the signal crossing the lower level, no trigger will be detected.

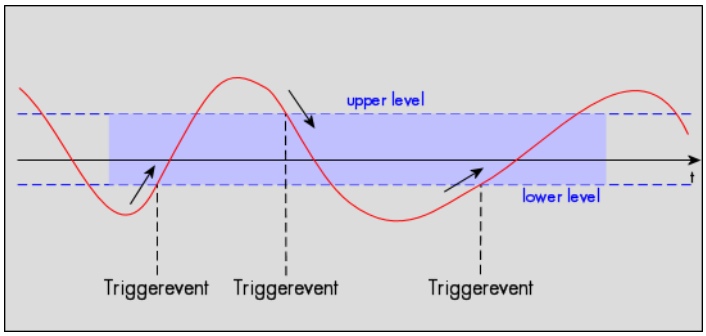
If the signal does cross the lower level within the the programmed pulsewidth time, a triggerevent will be detected.



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_STEEPNEG | SPC_TM_PW_SMALLER	02001000h
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
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel window trigger for entering signals

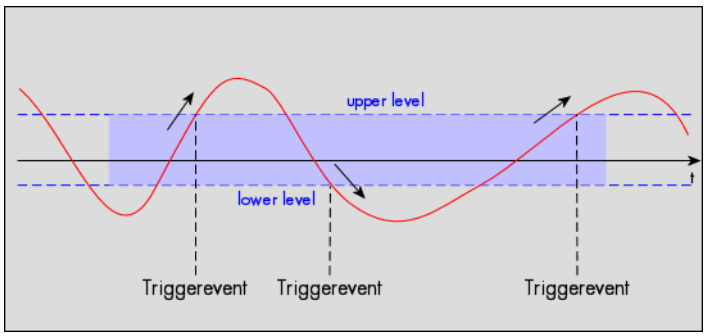
The analog input is continuously sampled with the selected sample rate. The upper and the lower level define a window. Every time the signal enters the window from the outside, a triggerevent will be detected.



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.

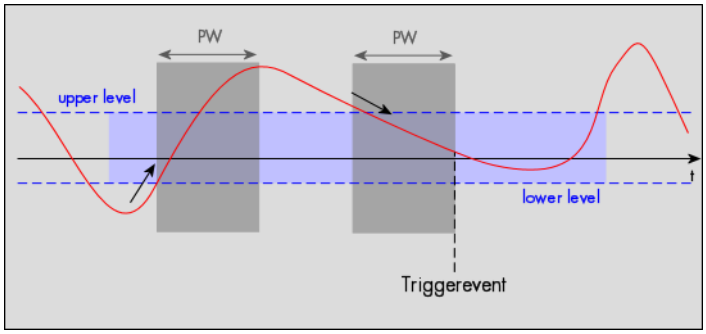


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_WINLEAVE	00000040h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively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

Channel window trigger for long inner signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower levels define a window. Every time the signal enters the window from the outside, the pulsewidth counter is started. If the signal leaves the window before the pulsewidth counter has stopped, no trigger will be detected.

If the pulsewidth counter stops and the signal is still inside the window, the triggerevent will be detected.

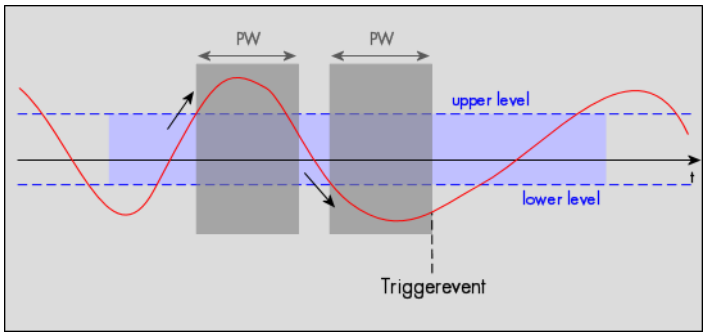


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_WINENTER | SPC_TM_PW_GREATER	04000020h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel window trigger for long outer signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower levels define a window. Every time the signal leaves the window from the inside, the pulsewidth counter is started. If the signal enters the window before the pulsewidth counter has stopped, no trigger will be detected.

If the pulsewidth counter stops and the signal is still outside the window, the triggerevent will be detected.

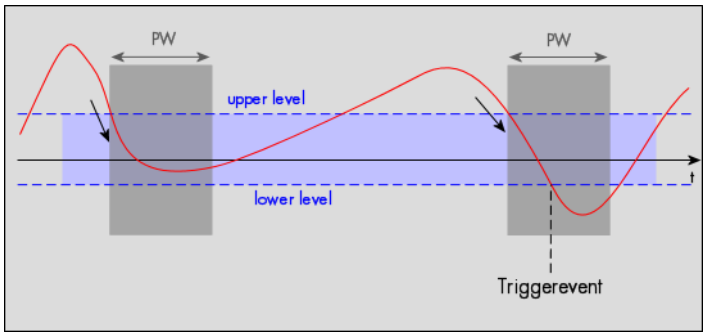


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_WINLEAVE | SPC_TM_PW_GREATER	04000040h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel window trigger for short inner signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower levels define a window. Every time the signal enters the window from the outside, the pulsewidth counter is started. If the pulsewidth counter stops and the signal is still inside the window, no trigger will be detected.

If the signal leaves the window before the pulsewidth counter has stopped, the triggerevent will be detected.

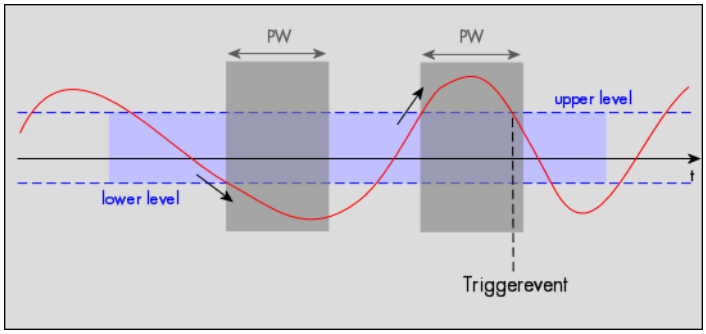


Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_WINENTER | SPC_TM_PW_SMALLER	02000020h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel window trigger for short outer signals

The analog input is continuously sampled with the selected sampling rate. The upper and the lower levels define a window. Every time the signal leaves the window from the inside, the pulsewidth counter is started. If the pulsewidth counter stops and the signal is still outside the window, no trigger will be detected.

If the signal enters the window before the pulsewidth counter has stopped, the trigger event will be detected.



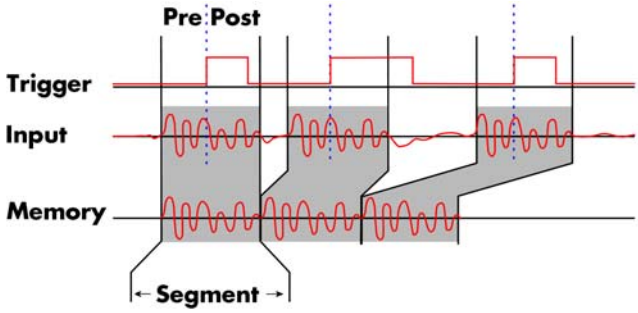
Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_WINLEAVE | SPC_TM_PW_SMALLER	02000040h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Option Multiple Recording/Replay

The option Multiple Recording/Replay allows the acquisition/ generation of data blocks with multiple trigger events without re-starting 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) or replay data of one segment (replay 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.

Register	Value	Direction	Description
SPC_POSTRIGGER	10100	read/write	Acquistion only: defines the number of samples to be recorded after the trigger event.
SPC_SEGMENTSIZE	10010	read/write	Size of one Multiple Recording/Replay segment: the total number of samples to be recorded/ replayed after detection of one trigger event.

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.

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.

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

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 transfered on the PCI bus as gaps of no interest did 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.

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:

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 samples needs 1 byte 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

Running the card with a sampling rate that is above 100 MS/s switches the cards internally to an interlace mode. In this mode two ADCs are running in parallel using a 180° shifted signal. Due to the fact that two ADCs are running this mode has a little different limitations and is listed separately in the following table.

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 channel	Standard Single	8	Mem	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem	4	4	16k - 32	4	4	Mem/2-4	4	8	Mem/2	4	not used		
	Standard Gate	8	Mem	4	4	16k - 32	4	4	Mem-4	4	not used			not used		
	FIFO Single	not used			4	16k - 32	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	16k - 32	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	16k - 32	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1
1 channel interlace	Standard Single	16	Mem	8	defined by post trigger			8	8G - 8	8	not used			not used		
	Standard Multi/ABA	16	Mem	8	8	16k - 32	8	8	Mem/2-4	8	16	Mem/2	8	not used		
	Standard Gate	16	Mem	8	8	16k - 32	8	8	Mem-8	8	not used			not used		
	FIFO Single	not used			8	16k - 32	8	not used			16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			8	16k - 32	8	8	8G - 8	8	16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Gate	not used			8	16k - 32	8	8	8G - 8	8	not used			0 (∞)	4G - 1	1
2 channels	Standard Single	8	Mem/2	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem/2	4	4	8k - 16	4	4	Mem/4-4	4	8	Mem/4	4	not used		
	Standard Gate	8	Mem/2	4	4	8k - 16	4	4	Mem/2-4	4	not used			not used		
	FIFO Single	not used			4	8k - 16	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	8k - 16	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	8k - 16	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1
2 channels interlace	Standard Single	16	Mem/2	8	defined by post trigger			8	8G - 8	8	not used			not used		
	Standard Multi/ABA	16	Mem/2	8	8	8k - 16	8	8	Mem/4-8	8	16	Mem/4	8	not used		
	Standard Gate	16	Mem/2	8	8	8k - 16	8	8	Mem/2-8	8	not used			not used		
	FIFO Single	not used			8	8k - 16	8	not used			16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			8	8k - 16	8	8	8G - 8	8	16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Gate	not used			8	8k - 16	8	8	8G - 8	8	not used			0 (∞)	4G - 1	1
4 channels	Standard Single	8	Mem/4	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem/4	4	4	4k - 8	4	4	Mem/8-4	4	8	Mem/8	4	not used		
	Standard Gate	8	Mem/4	4	4	4k - 8	4	4	Mem/4-4	4	not used			not used		
	FIFO Single	not used			4	4k - 8	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	4k - 8	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	4k - 8	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1

All figures listed here are given in samples. An entry of [8k - 16] means [8 kSamples - 16] = [8192 - 16] = 8176 samples.

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

	Installed Memory						
	64 MSample	**128 MSample**	**256 MSample**	**512 MSample**	**1 GSample**	**2 GSample**	**4 GSample**
Mem	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample	2 GSample	4 GSample
Mem / 2	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample	2 GSample
Mem / 4	16 MSample	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample
Mem / 8	8 MSample	16 MSample	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample

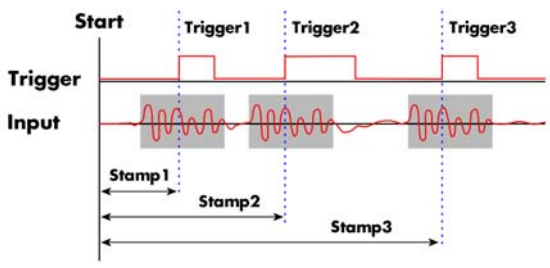
Please keep in mind that this table shows all values at once. Only the absolute maximum and minimum values are shown. There might be additional limitations. Which of these values is programmed depends on the used mode. Please read the detailed documentation of the mode.

Multiple Recording and Timestamps

Multiple Recording is well matching with the timestamp option. If timestamp recording is activated each trigger event and therefore each Multiple Recording segment will get timestamped as shown in the drawing on the right.

Please keep in mind that the trigger events are timestamped, not the beginning of the acquisition. The first sample that is available is at the time position of [Timestamp - Pretrigger].

The programming details of the timestamp option is explained in an extra chapter.



Trigger Modes

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

Trigger Output

When using internal trigger recognition and enabling the trigger output there is a trigger pulse generated for each acquired segment. The trigger output goes to high level after recoginition of the internal trigger event and goes back again to low level if the acquisition of this segment has been finished. To give compatibility to older hardware and to give maxmimum flexibility there is a special register to change that behaviour.

Register	Value	Direction	Description
SPC_LONGTRIG_OUTPUT	200830	read/write	Defines the trigger pulse output as explained below
	0 (default)	The trigger pulse is generated on every trigger event and stays high until acquisition of segment has finished	
	1	The trigger pulse is generated on the first trigger event and stays high until the end of the complete acquisition	

Programming examples

The following example shows how to set up the card for Multiple Recording in standard mode.

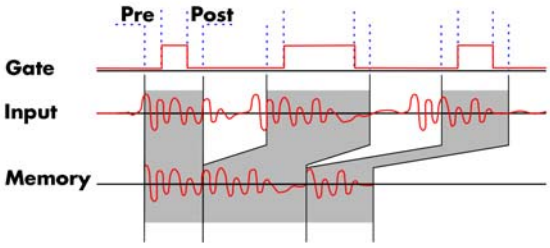
```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_MULTI);  // Enables Standard Multiple Recording

spcm_dwSetParam_i32 (hDrv, SPC_SEGMENTSIZE,   1024);          // Set the segment size to 1024 samples
spcm_dwSetParam_i32 (hDrv, SPC_POSTTRIGGER,    768);        // Set the posttrigger to 768 samples and therefore
                                                            // the pretrigger will be 256 samples
spcm_dwSetParam_i32 (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 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 Multiple Recording in FIFO mode.

```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_FIFO_MULTI); // Enables FIFO Multiple Recording

spcm_dwSetParam_i32 (hDrv, SPC_SEGMENTSIZE,   2048);          // Set the segment size to 2048 samples
spcm_dwSetParam_i32 (hDrv, SPC_POSTTRIGGER,   1920);        // Set the posttrigger to 1920 samples and therefore
                                                            // the pretrigger will be 128 samples
spcm_dwSetParam_i32 (hDrv, SPC_LOOPS           256);          // 256 segments will be recorded

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

Option Gated Sampling/Replay

The option Gated Sampling/Replay allows the data acquisition/data generation controlled by an external or an internal gate signal. Data will only be recorded if the programmed gate condition is true. When using the Gated Sampling acquisition mode it is in addition also possible to program a pre- and/or posttrigger for recording samples prior to and/or after the valid gate.



This chapter will explain all the necessary software register to set up the card for Gated Sampling/Replay properly.

The section on the allowed trigger modes deals with detailed description on the different trigger events and the resulting gates.

When using Gated Sampling/Replay the maximum pretrigger is limited as shown in the technical data section. When the programmed value exceeds that limit, the driver will return the error ERR_PRETRIGGERLEN.

Register	Value	Direction	Description
SPC_PRETRIGGER	10030	read/write	Defines the number of samples to be record prior to the gate start.
SPC_POSTTRIGGER	10100	read/write	Defines the number of samples to be record after the gate end.

Acquisition modes

Standard Mode

Data will be recorded as long as the gate signal fulfils the programmed gate condition. At the end of the gate interval the recording will be stopped and the card will pause until another gates signal appears. If the total amount of data to acquire has been reached, the card stops immediately. For that reason the last gate segment is ended by the expiring memory size counter and not by the gate end signal. The total amount of samples to be recorded can be defined by the memsize register. The table below shows the register for enabling Gated Sampling. For detailed information on how to setup and start the standard acquisition mode please refer to the according chapter earlier in this manual.

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode
SPC_REC_STD_GATE	4	Enables Gated Sampling for standard acquisition.	

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

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

FIFO Mode

The Gated Sampling in FIFO Mode is similar to the Gated Sampling in Standard Mode. In contrast to the Standard Mode you cannot program a certain total amount of samples to be recorded, but two other end conditions can be set instead. The acquisition can either run until the user stops it by software (infinite recording), or until a programmed number of gates has been recorded. The data is read continuously by the driver. These data is online available for further data processing by the user program. The advantage of Gated Sampling in FIFO mode is that you can stream data online to the host system with a lower average data rate than in conventional FIFO mode without Gated Sampling. You can make real-time data processing or store a huge amount of data to the hard disk. The table below shows the dedicated register for enabling Gated Sampling in FIFO mode. For detailed information how to setup and start the card in FIFO mode please refer to the according chapter earlier in this manual.

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode
SPC_REC_FIFO_GATE	64	Enables Gated Sampling for FIFO acquisition.	

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

Register	Value	Direction	Description
SPC_LOOPS	10020	read/write	Defines the number of gates to be recorded
0		Recording will be infinite until the user stops it.	
1 ... [4G - 1]		Defines the total gates to be recorded.	

Limits of pre trigger, post trigger, memory size

The maximum memory size parameter is only limited by the number of activated channels and by the amount of installed memory. Please keep in mind that each samples needs 1 byte 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

Running the card with a sampling rate that is above 100 MS/s switches the cards internally to an interlace mode. In this mode two ADCs are running in parallel using a 180° shifted signal. Due to the fact that two ADCs are running this mode has a little different limitations and is listed separately in the following table.

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 channel	Standard Single	8	Mem	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem	4	4	16k - 32	4	4	Mem/2-4	4	8	Mem/2	4	not used		
	Standard Gate	8	Mem	4	4	16k - 32	4	4	Mem-4	4	not used			not used		
	FIFO Single	not used			4	16k - 32	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	16k - 32	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	16k - 32	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1
1 channel interlace	Standard Single	16	Mem	8	defined by post trigger			8	8G - 8	8	not used			not used		
	Standard Multi/ABA	16	Mem	8	8	16k - 32	8	8	Mem/2-4	8	16	Mem/2	8	not used		
	Standard Gate	16	Mem	8	8	16k - 32	8	8	Mem-8	8	not used			not used		
	FIFO Single	not used			8	16k - 32	8	not used			16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			8	16k - 32	8	8	8G - 8	8	16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Gate	not used			8	16k - 32	8	8	8G - 8	8	not used			0 (∞)	4G - 1	1
2 channels	Standard Single	8	Mem/2	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem/2	4	4	8k - 16	4	4	Mem/4-4	4	8	Mem/4	4	not used		
	Standard Gate	8	Mem/2	4	4	8k - 16	4	4	Mem/2-4	4	not used			not used		
	FIFO Single	not used			4	8k - 16	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	8k - 16	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	8k - 16	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1
2 channels interlace	Standard Single	16	Mem/2	8	defined by post trigger			8	8G - 8	8	not used			not used		
	Standard Multi/ABA	16	Mem/2	8	8	8k - 16	8	8	Mem/4-8	8	16	Mem/4	8	not used		
	Standard Gate	16	Mem/2	8	8	8k - 16	8	8	Mem/2-8	8	not used			not used		
	FIFO Single	not used			8	8k - 16	8	not used			16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			8	8k - 16	8	8	8G - 8	8	16	8G - 8	8	0 (∞)	4G - 1	1
	FIFO Gate	not used			8	8k - 16	8	8	8G - 8	8	not used			0 (∞)	4G - 1	1
4 channels	Standard Single	8	Mem/4	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem/4	4	4	4k - 8	4	4	Mem/8-4	4	8	Mem/8	4	not used		
	Standard Gate	8	Mem/4	4	4	4k - 8	4	4	Mem/4-4	4	not used			not used		
	FIFO Single	not used			4	4k - 8	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	4k - 8	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	4k - 8	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1

All figures listed here are given in samples. An entry of [8k - 16] means [8 kSamples - 16] = [8192 - 16] = 8176 samples.

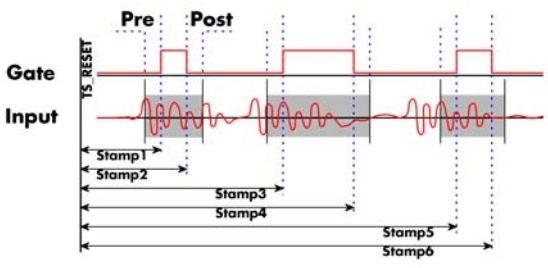
The given memory and memory / divider figures depend on the installed on-board memory as listed below:

	Installed Memory						
	64 MSample	**128 MSample**	**256 MSample**	**512 MSample**	**1 GSample**	**2 GSample**	**4 GSample**
Mem	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample	2 GSample	4 GSample
Mem / 2	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample	2 GSample
Mem / 4	16 MSample	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample
Mem / 8	8 MSample	16 MSample	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample

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

Gated Sampling and Timestamps

Gated Sampling and the timestamp option fit very good together. If timestamp recording is activated each gate will get timestamped as shown in the drawing on the right. As you can see both, beginning and end of the gate interval are timestamped. Each gate segment will therefore produce two timestamps showing start of the gate interval and end of the gate interval. By taking both timestamps into account one can read out the time position of each gate as well as the length in samples. There is no other way to examine the length of each gate segment than reading out the timestamps.



Please keep in mind that the gate signals are timestamped, not the beginning and end of the acquisition. The first sample that is available is at the time po-

sition of [Timestamp1 - Pretrigger]. The last sample of the gate segment is at the position [Timestamp2 + Posttrigger]. The length of the gate segment is [Timestamp2 - Timestamp1 + Pretrigger + Posttrigger]. When using the standard gate mode the end of recording is defined by the expiring memsize counter. Therefore there is no end of gate timestamp for the last gate segment!

The programming details of the timestamp option is explained in an extra chapter.

Trigger

Trigger Output

When using internal trigger recognition and enabling the trigger output there is a trigger pulse generated for each acquired segment. The trigger output goes to high level after recoginition of the internal trigger event and goes back again to low level if the acquisition of this segment has finished. To give compatibility to older hardware and to give maxmimum flexibility there is a special register to change that behaviour.

Register	Value	Direction	Description
SPC_LONGTRIG_OUTPUT	200830	read/write	Defines the trigger pulse output as explained below
	0 (default)		The trigger pulse is generated on every trigger event and stays high until acquisition of segment has finished
	1		The trigger pulse is generated on the first trigger event and stays high until the end of the complete acquisition

Edge and level triggers

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

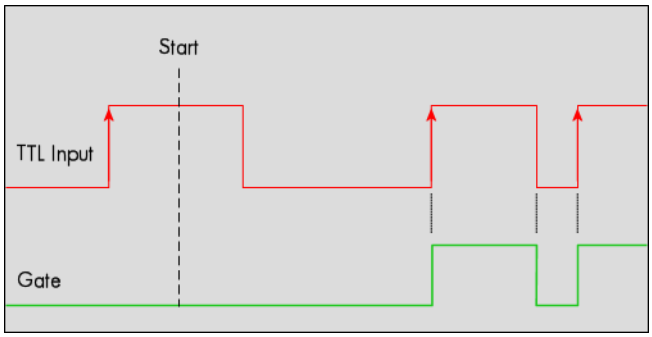
Register	Value	Direction	Description
SPC_TRIG_ORMASK	40410	read/write	Defines the OR mask for the different trigger sources.
SPC_TMASK_EXT0	2h		Enable external trigger input for the OR mask

Positive TTL single edge trigger

This mode is for detecting the rising edges of an external TTL signal. The gate will start on rising edges that are detected after starting the board.

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

With the next falling edge the gate will be stopped.



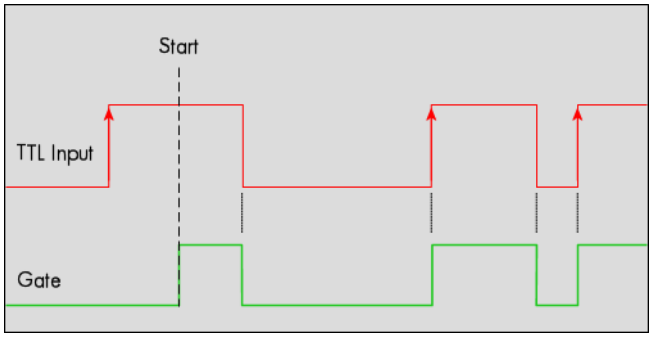
Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board
SPC_TM_POS	1h		Sets the trigger mode for external TTL trigger to detect positive edges

HIGH TTL level trigger

This mode is for detecting the high levels of an external TTL signal. The gate will start on high levels that are detected after starting the board acquisition/generation.

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

With the next low level the gate will be stopped.



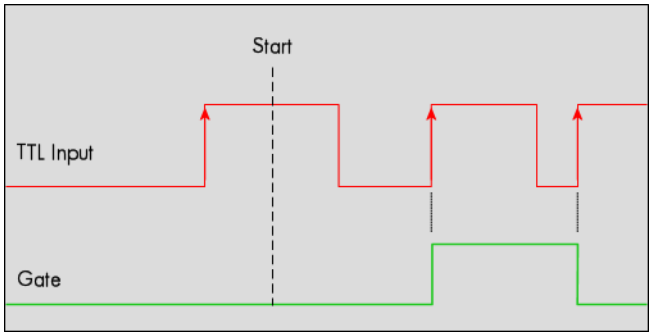
Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board
SPC_TM_HIGH	8h		Sets the trigger mode for external TTL trigger to detect high levels.

Positive TTL double edge trigger

This mode is for detecting the rising edges of an external TTL signal. The gate will start on the first rising edge that is detected after starting the board.

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

The gate will stop on the second rising edge that is detected.



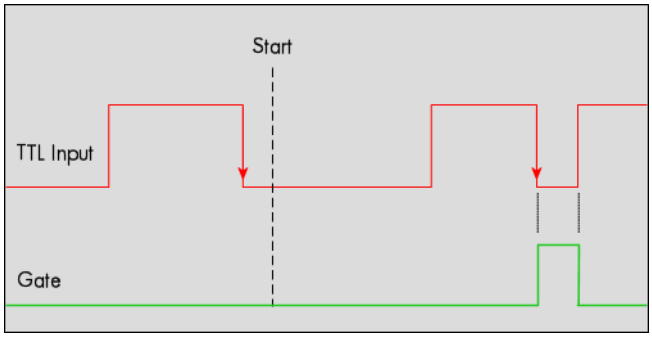
Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board
SPC_TM_POS | SPC_TM_DOUBLEEDGE	08000001h	Sets the gate mode for external TTL trigger to start and stop on positive edges.	

Negative TTL single edge trigger

This mode is for detecting the falling edges of an external TTL signal. The gate will start on falling edges that are detected after starting the board.

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

With the next rising edge the gate will be stopped.



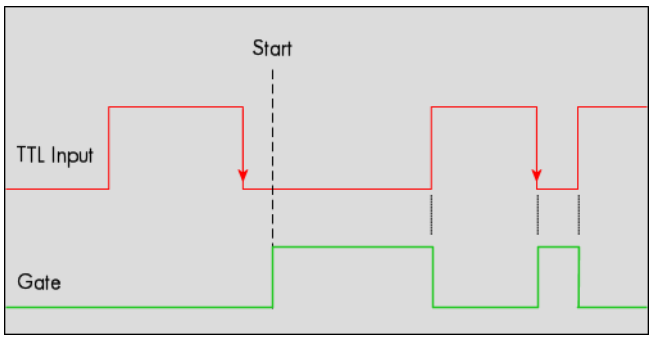
Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board
SPC_TM_NEG	2h	Sets the trigger mode for external TTL trigger to detect negative edges.	

LOW TTL level trigger

This mode is for detecting the low levels of an external TTL signal. The gate will start on low levels that are detected after starting the board.

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

With the next high level the gate will be stopped.



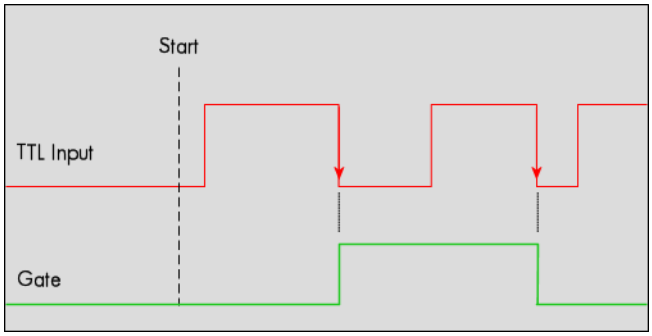
Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board
SPC_TM_LOW	10h	Sets the trigger mode for external TTL trigger to detect low levels.	

Negative TTL double edge trigger

This mode is for detecting the falling edges of an external TTL signal. The gate will start on the first falling edge that is detected after starting the board.

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

The gate will stop on the second falling edge that is detected.



Register	Value	Direction	Description
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the external trigger mode for the board
SPC_TM_NEG | SPC_TM_DOUBLEEDGE	08000002h	Sets the gate mode for external TTL trigger to start and stop on negative edges	

Pulsewidth triggers

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

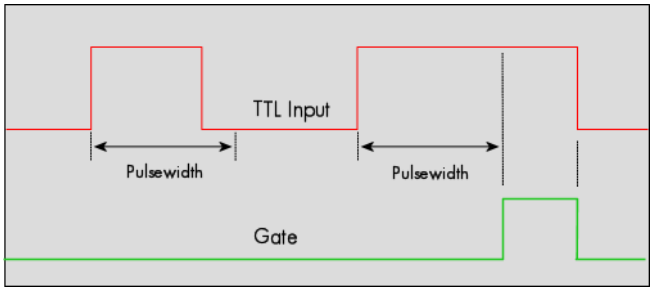
Register	Value	Direction	Description
SPC_TRIG_ORMASK	40410	read/write	Defines the OR mask for the different trigger sources.
SPC_TMASK_EXT0	2h	Enable external trigger input for the OR mask	

TTL pulsewidth trigger for long HIGH pulses

This mode is for detecting a rising edge of an external TTL signal followed by a HIGH pulse that are longer than a programmed pulsewidth. If the pulse is shorter than the programmed pulse-width, no trigger will be detected.

The gate will start on the first pulse matching the trigger condition after starting the board.

The gate will stop with the next falling edge.



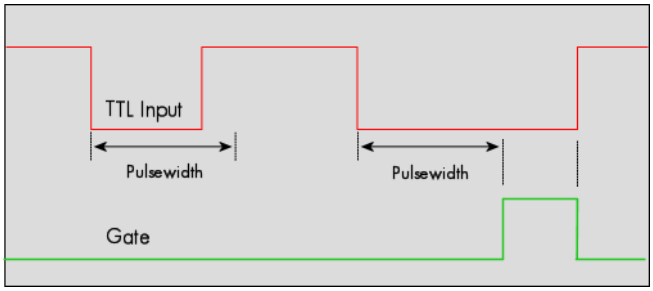
Register	Value	Direction	Description
SPC_TRIG_EXT0_PULSEWIDTH	44210	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the trigger mode for the board.
(SPC_TM_POS | SPC_TM_PW_GREATER)	4000001h	Sets the trigger mode for external TTL trigger to detect HIGH pulses that are longer than a programmed pulsewidth.	

TTL pulsewidth trigger for long LOW pulses

This mode is for detecting a falling edge of an external TTL signal followed by a LOW pulse that are longer than a programmed pulsewidth. If the pulse is shorter than the programmed pulse-width, no trigger will be detected.

The gate will start on the first pulse matching the trigger condition after starting the board.

The gate will stop with the next rising edge.



Register	Value	Direction	Description
SPC_TRIG_EXT0_PULSEWIDTH	44210	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.
SPC_TRIG_EXT0_MODE	40510	read/write	Sets the trigger mode for the board.
(SPC_TM_NEG | SPC_TM_PW_GREATER)	4000002h	Sets the trigger mode for external TTL trigger to detect LOW pulses that are longer than a programmed pulsewidth.	

```
spcm_dwSetParam_i32 (hDrv,SPC_TRIG_EXT0_MODE, SPC_TM_NEG | SPC_TM_PW_GREATER); // Setting up external TTL
                                                                               // trigger to detect low pulses
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_PULSEWIDTH ,                  50); // that are longer than 50 samples.
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK,                SPC_TMASK_EXT0); // and enable it within the OR mask
```

Channel triggers modes

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

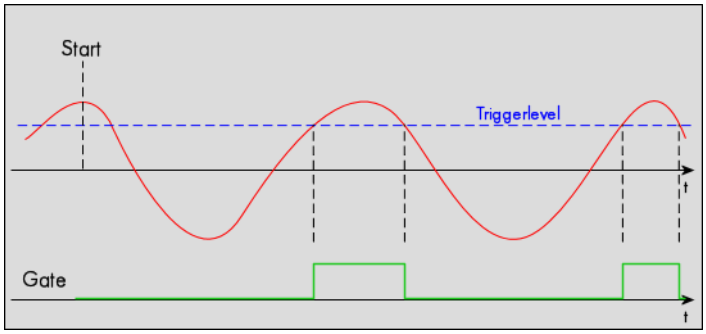
Register	Value	Direction	Description
SPC_TRIG_CH_ORMASK0	40460	read/write	Defines the OR mask for the channel trigger sources.
SPC_TMASK0_CH0	1h		Enables channel0 input for the channel OR mask

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) the gate starts.

When the signal crosses the programmed trigger 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.



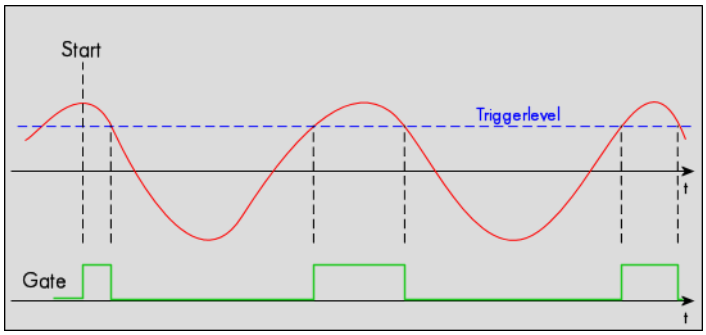
Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS	1h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

Channel trigger HIGH level

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 trigger level the gate will stop.

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



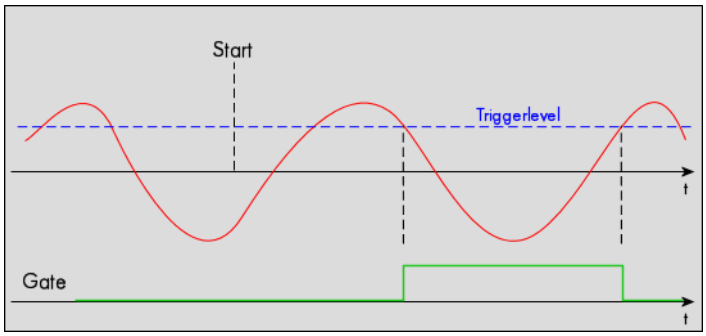
Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_HIGH	8h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

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 higher values to lower values (falling edge) the gate starts.

When the signal crosses the programmed trigger 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.



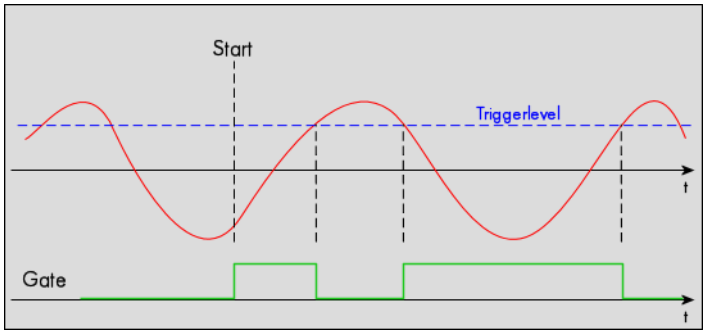
Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG	2h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

Channel trigger LOW level

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 trigger level the gate will stop.

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



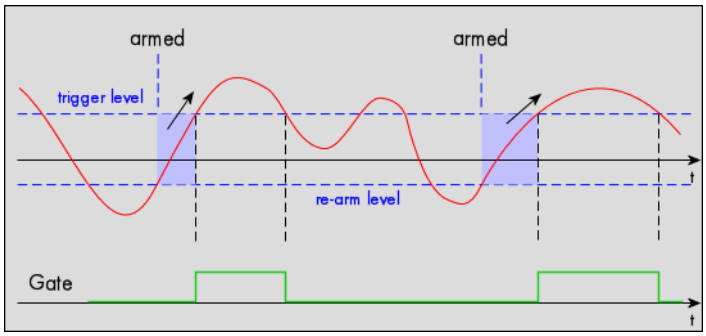
Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_LOW	10h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant

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 gate starts and the trigger engine will be disarmed.

If the programmed trigger 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.

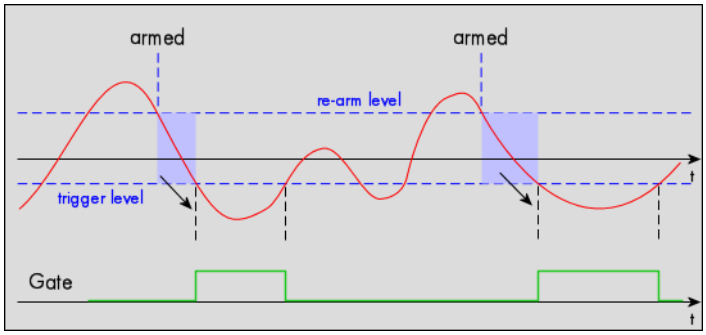
Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS | SPC_TM_REARM	01000001h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the desired trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the re-arm level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

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



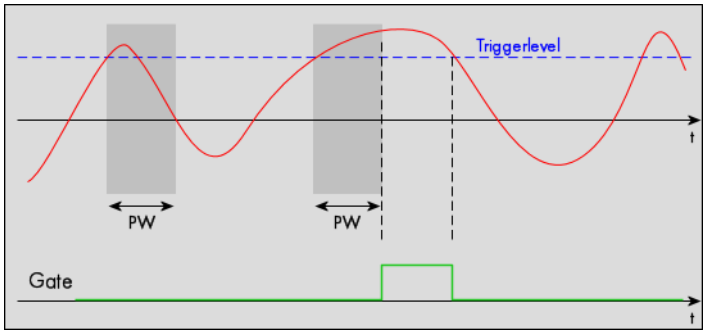
A new trigger event is only detected, if the trigger engine is armed again. The re-arm trigger modes can be used to prevent the board from triggering on wrong edges in noisy signals.

Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG | SPC_TM_REARM	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	Defines the re-arm level relatively to the channels's input range	board dependant

Channel pulsewidth trigger for long positive pulses

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 to higher values (rising edge) the pulsewidth counter is started. If the signal crosses the trigger level again in the opposite direction within the the programmed pulsewidth time, no trigger will be detected. If the pulsewidth counter reaches the programmed amount of samples, without the signal crossing the trigger level in the opposite direction, the gate will start.

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



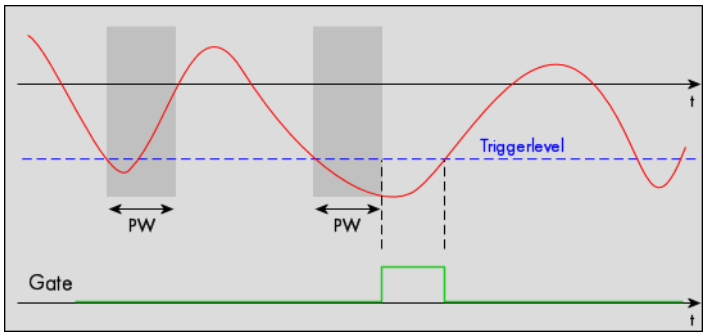
The pulsewidth trigger modes for long pulses can be used to prevent the board from triggering on wrong (short) edges in noisy signals.

Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_POS | SPC_TM_PW_GREATER	04000001h
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_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel pulsewidth trigger for long negative pulses

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 to lower values (falling edge) the pulsewidth counter is started. If the signal crosses the trigger level again in the opposite direction within the the programmed pulsewidth time, no trigger will be detected. If the pulsewidth counter reaches the programmed amount of samples, without the signal crossing the trigger level in the opposite direction, the gate will start.

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



The pulsewidth trigger modes for long pulses can be used to prevent the board from triggering on wrong (short) edges in noisy signals.

Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_NEG | SPC_TM_PW_GREATER	04000002h
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_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

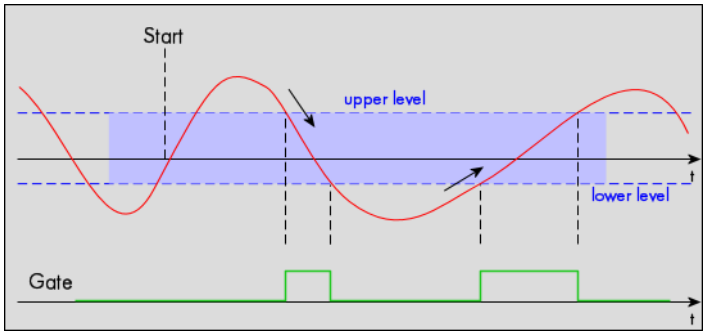
Channel window trigger for entering signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower level define a window.

When the signal enters the window from the outside to the inside, the gate will start.

When the signal leaves the window from the inside to the outside, the gate will stop.

As this mode is purely edge-triggered, the signal outside the window 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_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

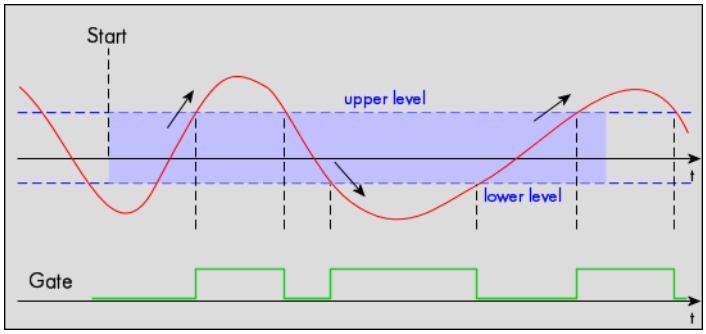
Channel window trigger for leaving signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower level define a window.

When the signal leaves the window from the inside to the outside, the gate will start.

When the signal enters the window from the outside to the inside, the gate will stop.

As this mode is purely edge-triggered, the signal within the window 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_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

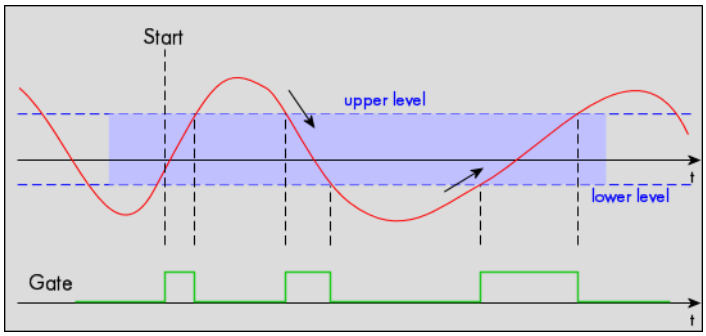
Channel window trigger for inner signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower level define a window.

When the signal enters the window from the outside to the inside, the gate will start.

When the signal leaves the window from the inside to the outside, the gate will stop.

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



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_INWIN	00000080h
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

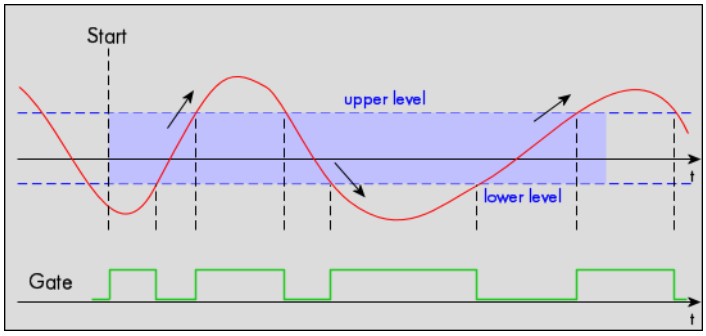
Channel window trigger for outer signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower level define a window.

When the signal leaves the window from the inside to the outside, the gate will start.

When the signal enters the window from the outside to the inside, the gate will stop.

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



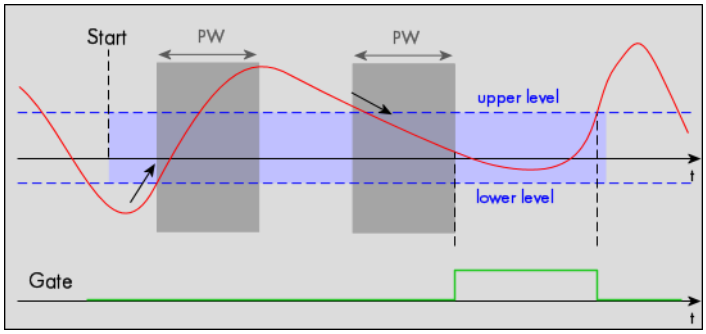
Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_OUTSIDEWIN	00000100h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant

Channel window trigger for long inner signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower levels define a window. Every time the signal enters the window from the outside, the pulsewidth counter is started. If the signal leaves the window before the pulsewidth counter has stopped, no trigger will be detected.

When the pulsewidth counter stops and the signal is still inside the window, the gate will start.

When the signal leaves the window from the inside to the outside, the gate will stop.



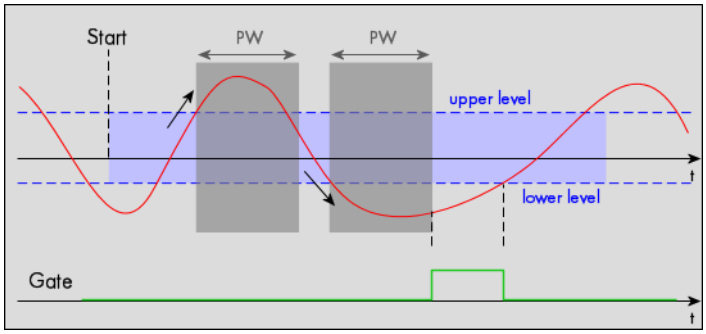
Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_WINENTER | SPC_TM_PW_GREATER	04000020h
SPC_TRIG_CH0_LEVEL0	42200	read/write	Set it to the upper trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_LEVEL1	42300	read/write	Set it to the lower trigger level relatively to the channel's input range.	board dependant
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Channel window trigger for long outer signals

The analog input is continuously sampled with the selected sample rate. The upper and the lower levels define a window. Every time the signal leaves the window from the inside, the pulsewidth counter is started. If the signal enters the window before the pulsewidth counter has stopped, no trigger will be detected.

When the pulsewidth counter stops and the signal is still outside the window, the gate will start.

When the signal enters the window from the outside to the inside, the gate will stop.



Register	Value	Direction	set to	Value
SPC_TRIG_CH0_MODE	40610	read/write	SPC_TM_WINLEAVE | SPC_TM_PW_GREATER	04000040h
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
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples. Values from 2 to 65535 are allowed.	2 to 65535

Programming examples

The following examples shows how to set up the card for Gated Sampling in standard mode for Gated Sampling in FIFO mode.

```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_GATE);  // Enables Standard Gated Sampling

spcm_dwSetParam_i32 (hDrv, PRETRIGGER,   256);               // Set the pretrigger to 256 samples
spcm_dwSetParam_i32 (hDrv, POSTTRIGGER, 2048);               // Set the posttrigger to 2048 samples
spcm_dwSetParam_i32 (hDrv, SPC_MEMSIZE, 8192);            // Set the total memsize for recording to 8192 samples

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
```

```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_FIFO_GATE); // Enables FIFO Gated Sampling

spcm_dwSetParam_i32 (hDrv, PRETRIGGER,   128);               // Set the pretrigger to 128 samples
spcm_dwSetParam_i32 (hDrv, POSTTRIGGER,  512);               // Set the posttrigger to 512 samples
spcm_dwSetParam_i32 (hDrv, SPC_LOOP,    1024);               // 1024 gates will be recorded

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

Option 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 (or at the start and at the end of a gate interval when using Gated Sampling mode) the current counter value is stored in an extra FIFO memory.

This function is designed as an enhancement to the Multiple Recording and the Gated Sampling mode and is also used together with the ABA mode option but can also be used without these options with plain single acquisitions. If Gated Sampling mode is used, then both the start and end of a recorded segment are timestamped.

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 and the oversampling factor. Please have a look at the clock chapter to see how to read out the sampling rate and the oversampling factor

$$t = \frac{\text{Timestamp}}{\text{Sampling rate * Oversampling}}$$

If you want to know the time between two timestamps, you can simply calculate this by the formula on the right.

$$\Delta t = \frac{\text{Timestamp}_{n+1} - \text{Timestamp}_{n}}{\text{Sampling rate * Oversampling}}$$

The following registers can be used for the timestamp option:

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	Set's a timeout in milli seconds for waiting of an reference clock edge
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. If reference clock mode is used this command waits for the edge the timeout time.	
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_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	4096	Enables the trigger synchronous acquisition of the BaseXIO inputs with every stored timestamp in the upper byte.	
SPC_TSXIOACQ_DISABLE	0	The timestamp is filled up with leading zeros as a sign extension for positive values.	

Example for setting timestamp mode:

The timestamp mode consists of one of the mode constants and one of the counter constants:

```
// setting timestamp mode to standard using internal clocking
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TSMODE_STANDARD | SPC_TSCNT_INTERNAL);

// setting timestamp mode to start reset mode using internal clocking
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TSMODE_STARTRESET | SPC_TSCNT_INTERNAL);

// 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);
```

Limits

The timestamp counter is running with the sampling clock on the base card. Some card types (like 2030 and 3025) use an interlace mode to double the sampling speed. In this case the timestamp counter is only running with the non-interlaced sampling rate. Therefore the maximum counting frequency of the timestamp option is limited to 125 MS/s.

Timestamp modes

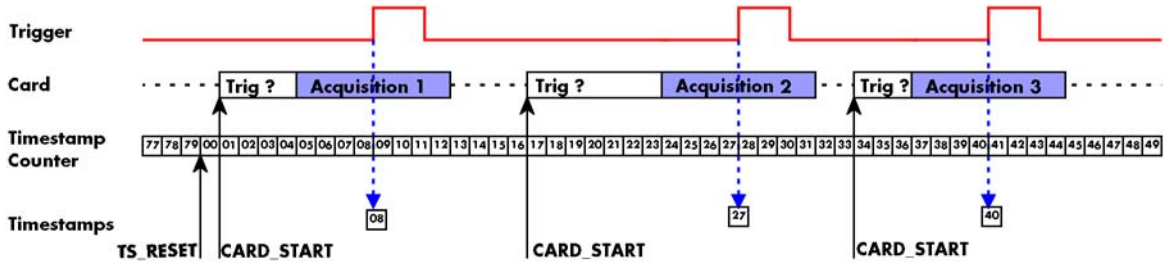
The timestamp command register selects which of the following modes should be used for generating timestamps. Independent of the used mode each timestamp is every time 64 bit wide and is generated with the currently used sampling rate. As some A/D acquisition cards need to use an oversampling factor to go beneath the minimum ADC sampling clock there might be a difference between the programmed sampling rate and the sampling rate that is used to count the timestamp counter. The currently used sampling rate and oversampling counter can be read out with the following register:

Register	Value	Direction	Description
SPC_SAMPLERATE	20000	read	Read out the internal sample rate that is currently used.
SPC_OVERSAMPLINGFACTOR	200123	read only	Returns the oversampling factor for further calculations. If oversampling isn't active a 1 is returned.

There is no oversampling factor if using full digital acquisition cards.

Standard mode

In standard mode the timestamp counter is set to zero once by writing the TS_RESET commando 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 Multiple Recording or Gated Sampling).



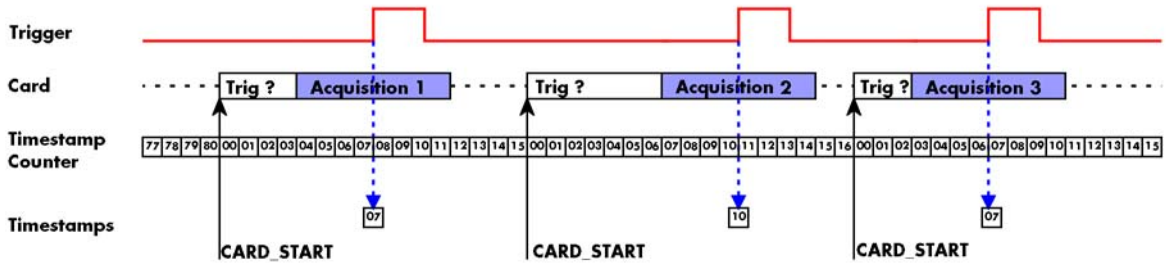
The following table shows the valid values that can be written to the timestamp command register for this mode:

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)



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

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	

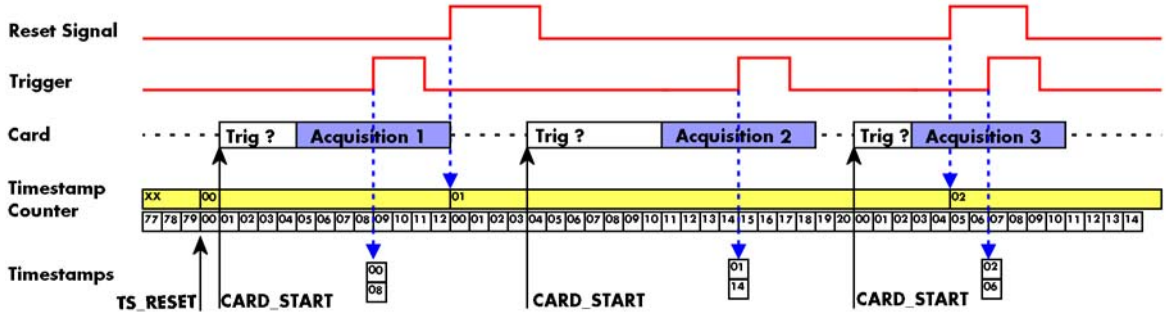
Refclock mode (needs BaseXIO option)

The counter is split in a HIGH and a LOW part and an additional external signal, that affects both parts of the counter, need to be fed in externally. The external reference clock signal will reset the LOW part of the counter and increase the HIGH part of the counter. The upper 32 bit of the timestamp value hold the number of the clock edges that have occurred on the external reference clock signal, the lower 32 bit will hold the timestamp counter with 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 currently sampling rate.

Please keep in mind that as this mode uses an additional external signal and can therefore only be used when the option BaseXIO is installed on the card. Otherwise there is no additional reference clock input available and this mode has no functionality.

The counting is initialized with the timestamp reset command. Both counter parts will then be set to zero.



The following table shows the valid values that can be written to the timestamp command register for this 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	Set's a timeout in milli seconds for waiting of an 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. If reference clock mode is used this command waits for the edge the timeout time.	
SPC_TSMODE_STANDARD	2h	Standard mode, counter is reset by explicit reset command.	
SPC_TSMODE_STARTRESET	4h	Counter is reset on every card start, all timestamps are in relation to card start.	
SPC_TSCNT_REFCLOCKPOS	200h	Counter is split, upper part is running with external reference clock positive edge, lower part is running with sampling clock	
SPC_TSCNT_REFCLOCKNEG	400h	Counter is split, upper part is running with external reference clock negative edge, lower part is running with sampling clock	

To synchronize the external reference clock signal with the PC clock it is possible to perform a timestamp reset command which waits a specified time for the occurrence of the external clock edge. As soon as the clock edge is found the function stores the current PC time and date which can be used to get the absolute time. As the timestamp reference clock can also be used with other clocks that don't need to be synchronized with the PC clock the waiting time can be programmed using the SPC_TIMESTAMP_TIMEOUT register.

Example for initialization of timestamp reference clock and synchronization of a seconds signal with the PC clock:

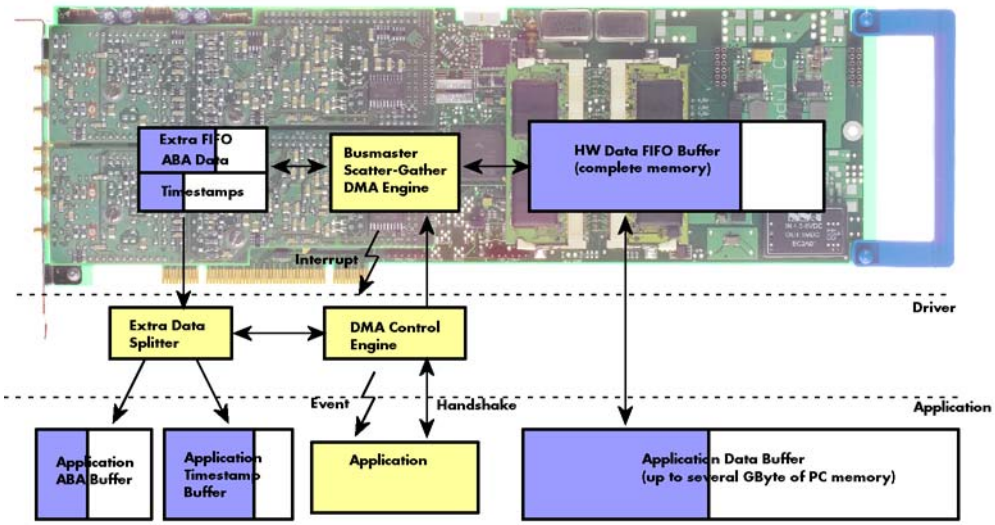
```
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TSMODE_STANDARD | SPC_TSCNT_REFCLOCKPOS);
spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_TIMEOUT, 1500);
if (ERR_TIMEOUT == spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_CMD, SPC_TS_RESET))
    printf ("Synchronization with external clock signal failed\n");

// 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);

// 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, (lSyncData >> 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 two completely independent busmaster DMA engines in hardware allowing the simultaneous transfer of both timestamp and sample data.

As seen in the picture the extra FIFO is holding ABA and timestamp data as the same time. Nevertheless it is not necessary to care for the shared FIFO as the extra FIFO data is splitted inside the driver in the both data parts.



The only part that is similar for both kinds of data transfer is the handling of the DMA engine. This is 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

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:.

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 it's 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_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.

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:

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)

```
char* pcData = new char[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_TIMESTAMP_AVAIL_USER_LEN, &lAvailBytes);
        spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_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 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_TIMESTAMP_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

The Polling mode needs driver version V1.25 and firmware version V11 to run. Please update your system to the newest versions to run this mode.

If the extra data is quite slow and the delay caused by the notify size on DMA transfers is inacceptable 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

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)

```
char* pcData = new char[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 pur polling loop
 do
    {
    spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_USER_LEN, &lAvailBytes);
    spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_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_TIMESTAMP_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_TIMESTAMP_AVAIL_USER_LEN, &lBytes);	spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_USER_LEN, &lBytes);
Min available bytes	programmed notify size	4 bytes
Current position?	spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_USER_LEN, &lBytes);	spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_USER_LEN, &lBytes);
Free buffer for card	spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_CARD_LEN, lBytes);	spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_CARD_LEN, lBytes);

Data format

Each timestamp is 56 bit long and internally mapped to 64 bit (8 bytes). The 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 occures 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 identically for all recordings with the same setup.

Standard data format

When internally mapping the timestamp from 56 bit to a 64 bit value the leading 8 bits are filled up with zeros (as a sign extension for positive values), to have the stamps ready for calculations as a unsigned 64 bit wide integer value.

Timestamp Mode	8th byte	7th byte	6th byte	5th byte	4th byte	3rd byte	2nd byte	1st byte
Standard/StartReset	0h	56 bit wide Timestamp						
Refclock mode	0h	24 bit wide Refclock edge counter (seconds counter)			32bit wide sample counter			

Extended BaseXIO data format

Sometimes it is usefull 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 byte of every 64 bit timestamp value is not (as in standard data mode) filled up with leading zeros, but with the values of the BaseXIO digital inputs. The following table shows the resulting 64 bit timestamps.

Timestamp Mode	8th byte	7th byte	6th byte	5th byte	4th byte	3rd byte	2nd byte	1st byte
Standard / StartReset	XIO7...XIO0	56 bit wide Timestamp						
Refclock mode	XIO7...XIO0	24 bit wide Refclock edge counter (seconds counter)			32bit wide sample counter			

This special sampling option requires the option BaseXIO to be installed. All enhanced timestamps are not longer integer 64 values. Before using these stamps for calculations (such as difference between two stamps)

one has to mask out the leading byte of the stamps first.

Selecting the timestamp data format

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 tis organized a bitfield, the data format selection is available for all possible timestamp modes.

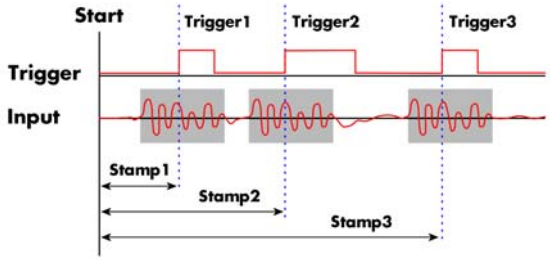
Register	Value	Direction	Description
SPC_TIMESTAMP_CMD	47100	r/w	
SPC_TSXIOACQ_ENABLE	4096	Enables the trigger synchronous acquisition of the BaseXIO inputs with every stored timestamp in the upper byte.	
SPC_TSXIOACQ_DISABLE	0	The timestamp is filled up with leading zeros as a sign extension for positive values.	

Combination of Multiple Recording and Gated Sampling with Timestamps

This topic should give you a brief overview how the timestamp option interacts with the both options Multiple Recording and Gated Sampling 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.

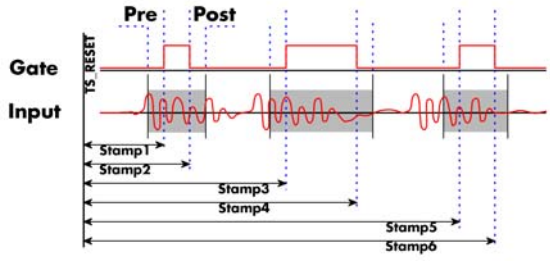
Example Multiple Recording and Timestamps

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);  // Enables Standard Multiple Recording
spcm_dwSetParam_i32 (hDrv, SPC_SEGMENTSIZE,   1024);          // Segment size is 1 kSample, Posttrigger is 768
spcm_dwSetParam_i32 (hDrv, SPC_POSTTRIGGER,    768);          // samples and pretrigger therefore 256 samples.
spcm_dwSetParam_i32 (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 acquistion, each timestamp is 64 bit = 8 bytes
int64* pllStamps = new int64[4];
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_TIMESTAMP, SPCM_DIR_CARDTOPC, 0, (void*) pllStamps, 0, 4 * 8);
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
int32 lSamplerate, lOver, i;
spcm_dwGetParam_i32 (hDrv, SPC_SAMPLERATE, &lSamplerate);
spcm_dwGetParam_i32 (hDrv, SPC_OVERSAMPLINGFACTOR, &lOver);
for (i = 0; i < 4; i++)
    printf ("#%d: %I64d samples = %.3f ms\n", i, pllStamps[i], 1000.0 * pllStamps[i] / lSamplerate / lOver);
```

Gated Sampling and Timestamps

Gated Sampling and the timestamp option fit very good together. If timestamp recording is activated each gate will get timestamped as shown in the drawing on the right. As you can see both, beginning and end of the gate interval are timestamped. Each gate segment will therefore produce two timestamps showing start of the gate interval and end of the gate interval. By taking both timestamps into account one can read out the time position of each gate as well as the length in samples. There is no other way to examine the length of each gate segment than reading out the timestamps.



Please keep in mind that the gate signals are timestamped, not the beginning and end of the acquisition. The first sample that is available is at the time position of [Timestamp1 - Pretrigger]. The last sample of the gate segment is at the position [Timestamp2 + Posttrigger]. The length of the gate segment is [Timestamp2 - Timestamp1 + Pretrigger + Posttrigger]. When using the standard gate mode the end of recording is defined by the expiring memsize counter. Therefore there is no end of gate timestamp for the last gate segment!

The programming details of the timestamp option is explained in an extra chapter.

Example Gated Sampling and Timestamps

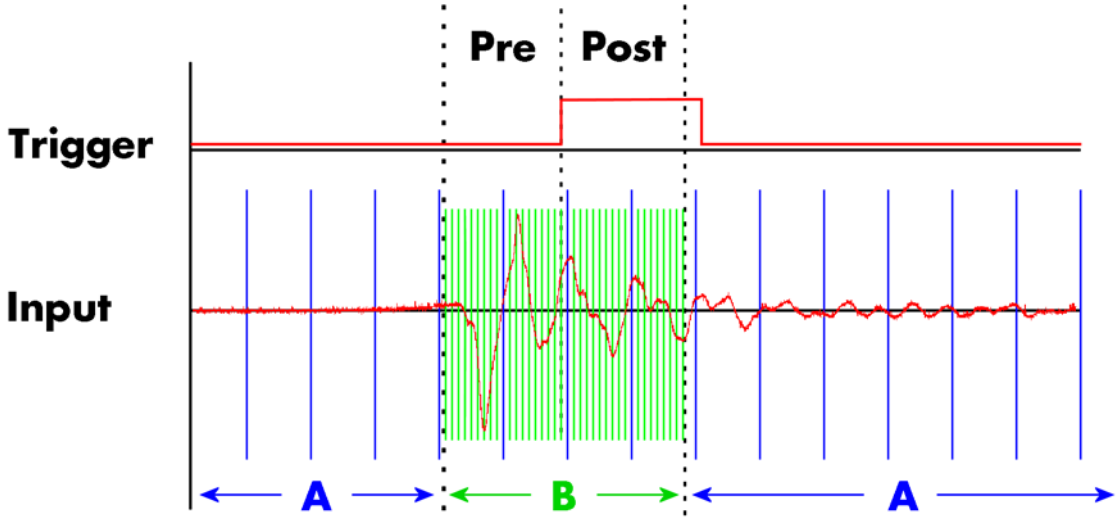
The following example shows the setup of the Gated Sampling mode together with activated timestamps recording and a short display of the the acquired timestamps. The example doesn't care for the acquired data itself and doesn't check for error:

```
// setup of the Gated Sampling mode
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_GATE);   // Enables Standard Gated Sampling
spcm_dwSetParam_i32 (hDrv, SPC_PRETRIGGER,      32);          // 32 samples to acquire before gate start
spcm_dwSetParam_i32 (hDrv, SPC_POSTTRIGGER,     32);          // 32 samples to acquire before gate end
spcm_dwSetParam_i32 (hDrv, SPC_MEMSIZE,       4096);          // 4 kSamples in total acquired

// 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 acquistion, each timestamp is 64 bit = 8 bytes
// as we don't know the number of gate intervals we define the buffer quite large
int64* pllStamps = new int64[1000];
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_TIMESTAMP, SPCM_DIR_CARDTOPC, 0, (void*) pllStamps, 0, 1000 * 8);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_EXTRA_STARTDMA);

// we wait for the end of timestamps transfer and read out the number of timestamps that have been acquired
int32 lAvailTimestampBytes;
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_EXTRA_WAITDMA);
spcm_dwSetParam_i32 (hDrv, SPC_TS_AVAIL_USER_LEN, &lAvailTimestampBytes);

// as we now have the timestamps we just print them and calculate the time in milli seconds
int32 lSamplerate, lOver, i;
spcm_dwGetParam_i32 (hDrv, SPC_SAMPLERATE, &lSamplerate);
spcm_dwGetParam_i32 (hDrv, SPC_OVERSAMPLINGFACTOR, &lOver);

// each 1st timestamp is the starting position of the gate segment, each 2nd the end position
for (i = 0; (i < (lAvailTimestampBytes / 8)) && (i < 1000); i++)
    if ((i % 2) == 0)
        printf ("#%d: %I64d samples = %.3f ms", i, pllStamps[i], 1000.0 * pllStamps[i] / lSamplerate / lOver);
    else
        printf ("(Len = %I64d samples)\n", (pllStamps[i] - pllStamps[i - 1] + 64));
```

Option ABA mode (dual timebase)

General information

The ABA option allows the acquisition of data with a dual timebase. In case of trigger event the inputs are sampled very fast with the programmed sampling rate. This part is similar to the Multiple Recording option. But instead of having no data in between the segments one has the opportunity to continuously sample the inputs with a slower sampling rate the whole time. Combining this with the recording of the timestamps gives you a complete acquisition with a dual timebase as shown in the drawing.



As seen in the drawing the area around the trigger event is sampled between pretrigger and posttrigger with full sampling speed (area B of the acquisition). Outside of this area B the input is sampled with the slower ABA clock (area A of the acquisition). As changing sampling clock on the fly is not possible there is no real change in the sampling speed but area A runs continuously with a slow sampling speed without stopping when the fast sampling takes place. As a result one get's a continuous slow sampled acquisition (area A) with some fast sampled parts (area B)

The ABA mode is available for standard recording as well as for FIFO recording. In case of FIFO recording ABA and the acquisition of the fast sampled segments will run continuously until it is stopped by the user.

A second possible application for the ABA mode is the use of the ABA data for slow monitoring of the inputs while waiting for an acquisition. In that case one wouldn't record the timestamps but simply monitor the current values by acquiring ABA data.

The ABA mode needs a second clock base. As explained above the acquisition is not changing the sampling clock but runs the slower acquisition with a divided clock. The divider value can be programmed with the following register

Register	Value	Direction	Description
SPC_SEGMENTSIZE	10010	read/write	Size of one Multiple Recording segment: the number of samples to be record after each trigger event.
SPC_POSTRIGGER	10030	read/write	Defines the number of samples to be record after each trigger event.
SPC_ABADIVIDER	10040	read/write	Programs the divider which is used to sample slow ABA data between 2 and 65535

The resulting ABA clock is then calculated by sampling rate / ABA divider.

Each segment 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 ABA mode or 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.

Standard Mode

With every detected trigger event one data block is filled with data. The length of one ABA segment is set by the value of the segmentsize register. 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 standard ABA mode. For detailed information on how to setup and start the standard acquisition mode please refer to the according chapter earlier in this manual.

Register		Value	Direction	Description
SPC_CARDMODE		9500	read/write	Defines the used operating mode
	SPC_REC_STD_ABA	8h		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. This mode is only available if the ABA mode option is installed. The mode is described in a special chapter about ABA mode option.

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.

FIFO Mode

The ABA FIFO Mode is similar to the Multiple Recording FIFO 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 Single FIFO mode example earlier in this manual. These blocks are online available for further data processing by the user program. This mode significantly reduces the average data transfer rate on the PCI bus. This enables you to use faster sample rates then you would be able to in FIFO mode without ABA.

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode
SPC_REC_FIFO_ABA	80h	Continuous data acquisition for multiple trigger events together with continuous data acquisition with a slower sampling clock. Only available if ABA mode option is installed	

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

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 samples needs 1 byte 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

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 channel	Standard Single	8	Mem	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem	4	4	16k - 32	4	4	Mem/2-4	4	8	Mem/2	4	not used		
	Standard Gate	8	Mem	4	4	16k - 32	4	4	Mem-4	4	not used			not used		
	FIFO Single	not used			4	16k - 32	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	16k - 32	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	16k - 32	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1
2 channels	Standard Single	8	Mem/2	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem/2	4	4	8k - 16	4	4	Mem/4-4	4	8	Mem/4	4	not used		
	Standard Gate	8	Mem/2	4	4	8k - 16	4	4	Mem/2-4	4	not used			not used		
	FIFO Single	not used			4	8k - 16	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	8k - 16	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	8k - 16	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1
4 channels	Standard Single	8	Mem/4	4	defined by post trigger			4	8G - 4	4	not used			not used		
	Standard Multi/ABA	8	Mem/4	4	4	4k - 8	4	4	Mem/8-4	4	8	Mem/8	4	not used		
	Standard Gate	8	Mem/4	4	4	4k - 8	4	4	Mem/4-4	4	not used			not used		
	FIFO Single	not used			4	4k - 8	4	not used			8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Multi/ABA	not used			4	4k - 8	4	4	8G - 4	4	8	8G - 4	4	0 (∞)	4G - 1	1
	FIFO Gate	not used			4	4k - 8	4	4	8G - 4	4	not used			0 (∞)	4G - 1	1

All figures listed here are given in samples. An entry of [8k - 16] means [8 kSamples - 16] = [8192 - 16] = 8176 samples.

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

	Installed Memory						
	64 MSample	**128 MSample**	**256 MSample**	**512 MSample**	**1 GSample**	**2 GSample**	**4 GSample**
Mem	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample	2 GSample	4 GSample
Mem / 2	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample	2 GSample
Mem / 4	16 MSample	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample	1 GSample
Mem / 8	8 MSample	16 MSample	32 MSample	64 MSample	128 MSample	256 MSample	512 MSample

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

Example for setting ABA mode:

The following example will program the standard ABA mode, will set the fast sampling rate to 100 MHz and acquire 2k segments with 1k pretrigger and 1k posttrigger on every rising edge of the trigger input. Meanwhile the inputs are sampled continuously with the ABA mode with a ABA divider set to 5000 resulting in a slow sampling clock for the A area of 100 MHz / 5000 = 20 kHz:

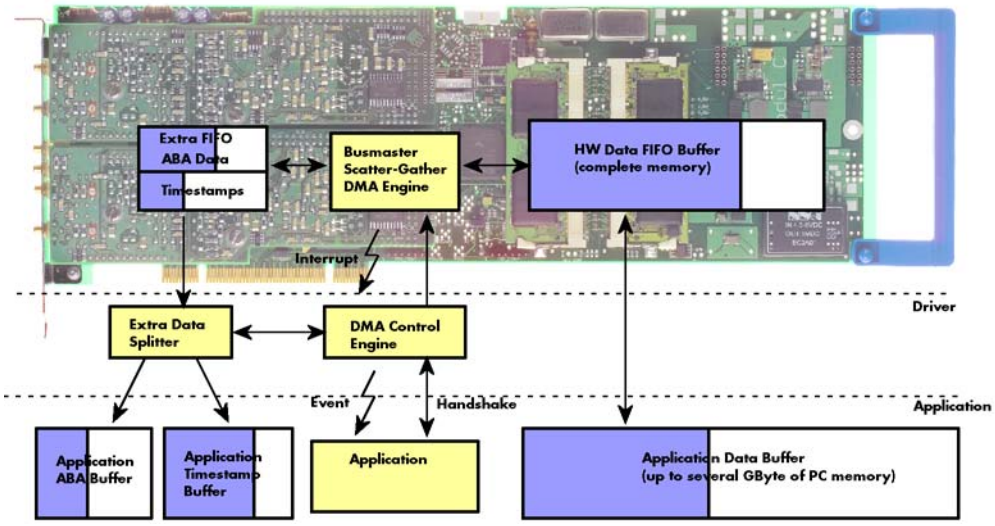
```
// setting the fast sampling clock as internal 100 MHz
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hDrv, SPC_SAMPLERATE, 100000000);

// enable the ABA mode and set the ABA divider to 5000 -> 100 MHz / 5000 = 20 kHz
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_ABA);
spcm_dwSetParam_i32 (hDrv, SPC_ABADIVIDER, 5000);

// define the segmentsize, pre and posttrigger and the total amount of data to acquire
spcm_dwSetParam_i32 (hDrv, SPC_MEMSIZE, 16384);
spcm_dwSetParam_i32 (hDrv, SPC_SEGMENTSIZE, 2048);
spcm_dwSetParam_i32 (hDrv, SPC_POSTTRIGGER, 1024);

// set the trigger mode to external with positive edge
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0);
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_POS);
```

Reading out ABA data

General

The slow „A" data is stored in an extra FIFO that is located in hardware on the card. This extra FIFO can read out slow „A" data using DMA transfer similar to the DMA transfer of the main sample data DMA transfer. The card has two completely independent busmaster DMA engines in hardware allowing the simultaneous transfer of both „A" and sample data. The sample data itself is read out as explained before using the standard DMA routine.

As seen in the picture the extra FIFO is holding ABA and timestamp data as the same time. Nevertheless it is not necessary to care for the shared FIFO as the extra FIFO data is splitted inside the driver in the both data parts.



The only part that is similar for both kinds of data transfer is the handling of the DMA engine. This is 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

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:.

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 it's 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_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.

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:

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)

```
char* pcData = new char[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_TIMESTAMP_AVAIL_USER_LEN, &lAvailBytes);
        spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_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 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_TIMESTAMP_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

The Polling mode needs driver version V1.25 and firmware version V11 to run. Please update your system to the newest versions to run this mode.

If the extra data is quite slow and the delay caused by the notify size on DMA transfers is inacceptable 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

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)

```
char* pcData = new char[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 pur polling loop
 do
    {
    spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_USER_LEN, &lAvailBytes);
    spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_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_TIMESTAMP_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_TIMESTAMP_AVAIL_USER_LEN, &lBytes);	spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_USER_LEN, &lBytes);
Min available bytes	programmed notify size	4 bytes
Current position?	spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_USER_LEN, &lBytes);	spcm_dwGetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_USER_LEN, &lBytes);
Free buffer for card	spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_CARD_LEN, lBytes);	spcm_dwSetParam_i32 (hDrv, SPC_TIMESTAMP_AVAIL_CARD_LEN, lBytes);

Option BaseXIO

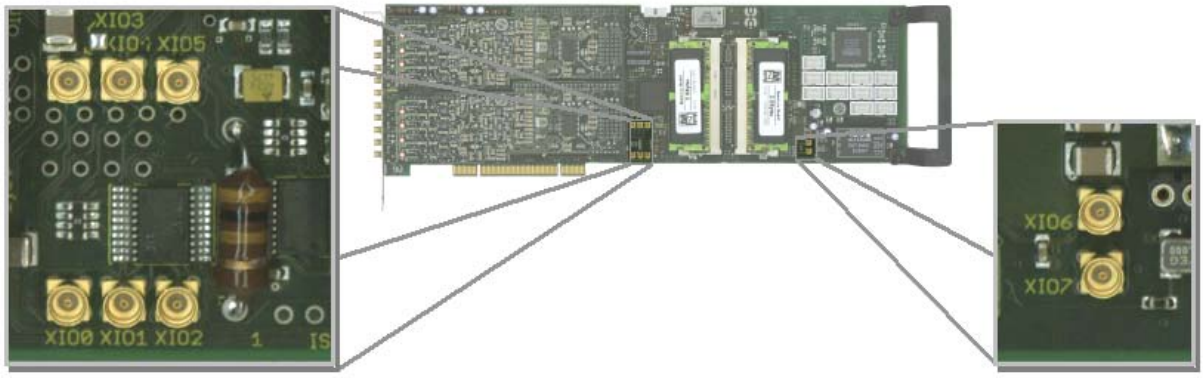
Introduction

With this simple-to-use versatile enhancement it is possible to control a wide range of external instruments or other equipment. Therefore you have up to eight asynchronous digital I/Os available. When using the BaseXIO lines as digital I/O, they are completely independent from the board's function, data direction or sampling rate and directly controlled by software (asynchronous I/Os).

Using the option BaseXIO this way is useful if external equipment should be digitally controlled or any kind of signal source must be programmed. It also can be used if status information from an external machine has to be obtained or different test signals have to be routed to the board. In addition to the asynchronous I/O function, some of these lines can have special purposes such as secondary TTL trigger lines or as a RefClock seconds signal for the timestamp option.

The eight MMCX coaxial connectors are directly mounted on the M2i base card. When plugged internally with right-angle MMCX connectors, this options does not require any additional system slot. By default this option is delivered with a readily plugged additional bracket equipped with SMB connectors, to have access to the lines from outside the system to easily connect with external equipment.

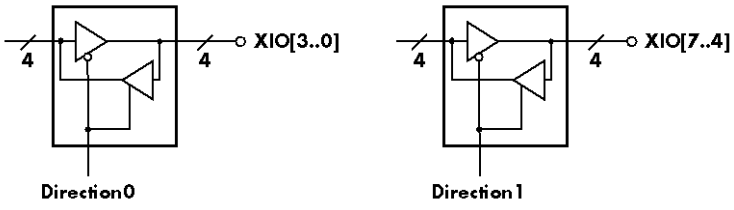
The internal connectors are mounted on two locations on the base card. The picture below shows the location of the MMCX connectors on the card, the details of the connectors on the extra bracket are shown in the introductional part of this manual.



Different functions

Asynchronous Digital I/O

This way of operating the option BaseXIO allows to asynchronously sample the data on the inputs or to generate asynchronous pattern on the outputs. The eight available lines consist of two groups of buffers each driving or receiving 4 bits of digital data as the drawing is showing.



The data direction of each group can be individually programmed to be either input or output.

As a result three different combinations are possible when using BaseXIO as pure digital I/O:

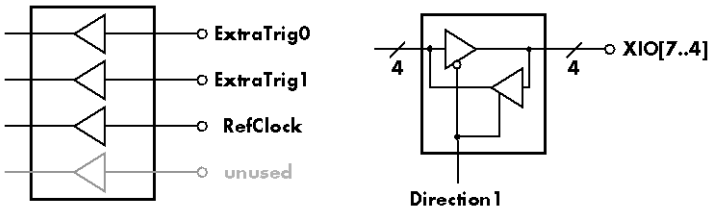
- 8 asynchronous digital inputs
- 8 asynchronous digital outputs
- mixed mode with 4 inputs and 4 outputs

The table below shows the direction register and the possible values. To combine the values you can easily OR them bitwise.

Register	Value	Direction	Description
SPC_XIO_DIRECTION	47100	r/w	Defines groupwise the direction of the digital I/O lines. Values can be combined by a bitwise OR.
XD_CH0_INPUT	0	Sets the direction of the lower group (bit D3…D0) to input.	
XD_CH1_INPUT	0	Sets the direction of the upper group (bit D7…D4) to input.	
XD_CH0_OUTPUT	1	Sets the direction of the lower group (bit D3…D0) to output.	
XD_CH1_OUTPUT	2	Sets the direction of the upper group (bit D7…D4) to output.	

Special Input Functions

This way of operating the option BaseXIO requires the lower of the above mentioned group of four lines (XIO3…XIO0) to be set as input. The upper group can be programmed to be either input or output.



The four lower input bits then can have additional functions besides working as asynchronous digital inputs:

- XIO0: additional TTL trigger ExtraTrig0
- XIO1: additional TTL trigger ExtraTrig1
- XIO2: RefClock for timestamp option
- XIO3: no special feature yet

All of the above mentioned special features are explained in detail in the relating section of this manual.

When using one or more of the inputs whith their special features, it is still possible to sample them asynchronously as descibed in the section before. So as an example when using bit 0 as an additional TTL trigger input the remaining three lines of the input group can still be used as asynchronous digital inputs. When reading the data of the inputs all bits are sampled, even those that are used for special purposes. In these cased the user might mask the read out digital data manually, to not recieve unwanted lines.

The table below shows the direction register for the remaining upper group and the possible values. To combine the values for both groups you can easily OR them bitwise.

Register	Value	Direction	Description
SPC_XIO_DIRECTION	47100	read/write	Defines the direction of the remaining digital I/O lines.
XD_CH0_INPUT	0	The direction of the lower group (bit D3…D0) must be set to input, when using the special features.	
XD_CH1_INPUT	0	Sets the direction of the upper group (bit D7…D4) to input.	
XD_CH1_OUTPUT	2	Sets the direction of the upper group (bit D7…D4) to output.	

Transfer Data

The outputs can be written or read by a single 32 bit register. If the register is read, the actual pin data will be sampled. Therefore reading the lines declared as outputs gives back the generated pattern. The single bits of the digital I/O lines correspond with the number of the bit of the 32 bit register. Values written to the three upper bytes will be ignored.

Register	Value	Direction	Description
SPC_XIO_DIGITALIO	47110	r	Reads the data directly from the pins of all digital I/O lines either if they are declared as inputs or outputs.
SPC_XIO_DIGITALIO	47110	w	Writes the data to all digital I/O lines that are declared as outputs. Bytes that are declared as inputs will ignore the written data.

Programming Example

The following example shows, how to program the lower group to be input and the upper group to be output, and how to write and read and interpret/mask the digital data:

```
// Define direction: set Ch0 as Input and Ch1 as output
spcm_dwSetParam_i32 (hDrv, SPC_XIO_DIRECTION, XD_CH0_INPUT | XD_CH1_OUTPUT);

spcm_dwSetParam_i32 (hDrv, SPC_XIO_DIGITALIO, 0xA0);   // Set all even output bits HIGH, all odd to LOW
                                                       // The write to the inputs will be ignored
spcm_dwGetParam_i32 (hDrv, SPC_XIO_DIGITALIO, &lData); // Read back the digital data (incl. outputs)
                                                       // Bits 7…4 will be the output value 0xA
lData = lData & (uint32) 0x0F                          // Mask out the output bits to have inputs only
```

Special Sampling Feature

When using the option BaseXIO in combination with the timestamp option one can enable a special auto sampling option, that samples the eight BaseXIO lines synchronously with each trigger event. This feature is independent of the BaseXIO line settings. For details, please refer to the timestamp chapter in this manual.

This special sampling feature requires the option Timestamp to be installed.

Electrical specifications

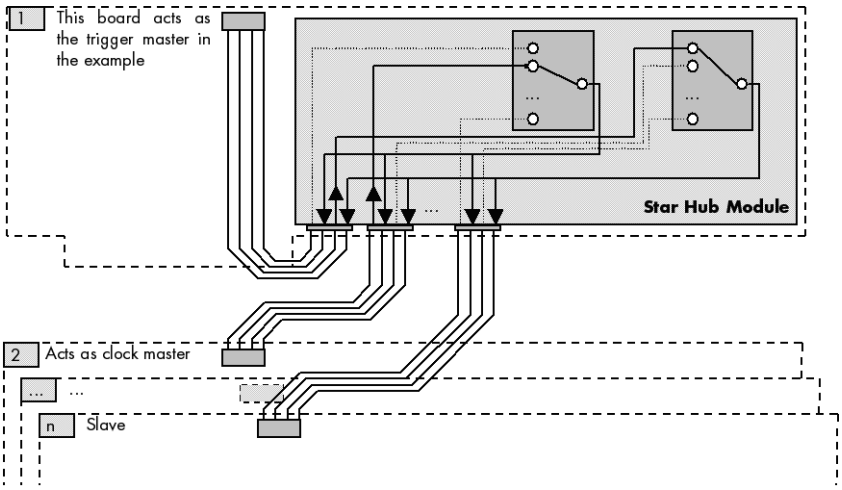
The electrical specifications of the BaseXIO inputs and outputs can be found either in the technical data section of this manual or in the datasheet.

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. It is even possible to interconnect multiple systems using the system star-hubs described further below.

The Star-Hub option allows to synchronize several cards of the M2i series that are mounted within one host system (PC). Two different versions are available: a small version with 5 connectors (option SH5) for synchronizing up to five cards and a big version with 16 connectors (option SH16) for synchronizing up to 16 cards.

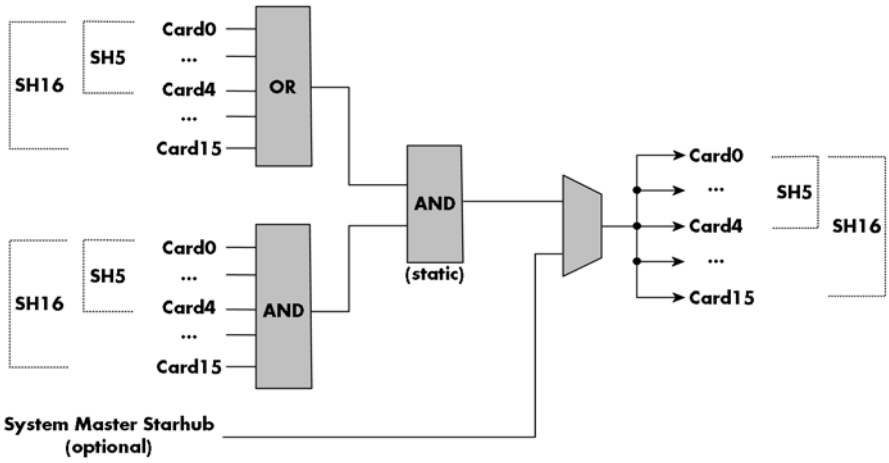


Both versions are implemented as a piggy - back module that is mounted to one of the cards. For details on how to install several cards including the one carrying the Star-Hub module, please refer to the section on hardware installation.

Either which of the two available Star-Hub options is used, there will be no phase delay between the sampling clocks of the synchronized cards and either no delay between the trigger events, if all synchronized cards run with the same sampling rate. Any one of the synchronized cards can be used as a clock master and besides any card can be part of the trigger generation.

Star-Hub trigger engine

The trigger bus between an M2i card and the Star-Hub option consists of three lines. Two of them send the trigger information from the card's trigger engine to the Star-Hub and one line receives the resulting trigger from 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.

Two lines are used to send the trigger from the card to the Star-Hub to provide the possibility to use the same OR/AND conjunctions for the resulting synchronization trigger like on a card that runs on it's own.

By this separation all OR masks of all synchronized cards are therefore extended to one big OR mask, while all AND masks of the synchronized cards are extended to one overall AND mask. This allows to combine the various trigger sources of all synchronized cards with AND and OR conditions and so to create highly complex trigger conditions that will certainly suit your application's needs.

For details on the card's trigger engine and the usage of the OR/AND trigger masks please refer to the relating section of this manual.

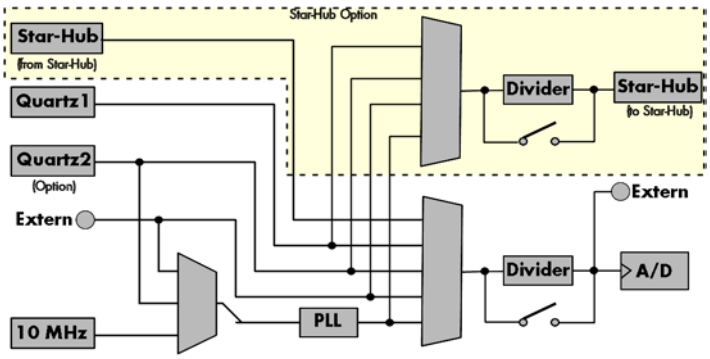
As an option it is also possible to synchronize multiple host systems each containing one Star-Hub module. These system slaves then will simply listen on the trigger line from the system master and distribute it to the connected cards. As this multi-system synchronization comes with some limits on certain settings and also needs some special attention on synchronizing the application software as well, it is therefore described in a separate section later in this manual.

Star-Hub clock engine

One of the cards can be the clock master for the complete system. This can be any card of the system even one card that does not contain the Star-Hub. As shown in the drawing on the right the clock master can use any of it's clock sources to be broadcasted 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 running with the same speed.

Each slave card can use an additional divider on the received Star-Hub clock. This allows to synchronize fast and slow cards in one system.



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 it's own handle which has some extra commands for synchronization set-up. 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]);
```

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:

Register	Value	Direction	Description
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_CARDIDX15	49015	read	Index of card that is connected to star-hub logical index 15 (mask 0x8000)

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 except 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 except 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 and Clock

The synchronization setup only requires two additional registers to enable the cards that are synchronized in the next run and to select a clock master for the next run.

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.

Register	Value	Direction	Description
SPC_SYNC_CLKMASK	49220	read/write	Mask of the card that is the clock master, only one bit is allowed to be set

One of the enabled cards must be selected to be the clock master for the complete system. If you intend to run cards with different clock speeds the clock master must have the highest clock as all other cards will derive their clock by dividing the master clock. The locally selected clock source from the clock master is routed throughout the complete synchronized system.

When using external clock please be sure that the external clock stays within all limits of all synchronized cards. Please take special care regarding the minimum and maximum frequencies as offending these may damage components on the cards!

In our example we synchronize all four cards and select card number 2 to be the clock master:

```
spcm_dwSetParam_i32 (hSync, SPC_SYNC_ENABLEMASK, 0x000F); // all 4 cards are masked
spcm_dwSetParam_i32 (hSync, SPC_SYNC_CLKMASK,    0x0004); // card 2 is selected as clock master

// set the clock master to 1 MS/s internal clock
spcm_dwSetParam_i32 (hCard[2], SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hCard[2], SPC_SAMPLEATE, MEGA(1));

// set all the slaves to run synchronously with 1 MS/s
spcm_dwSetParam_i32 (hCard[0], SPC_SAMPLEATE, MEGA(1));
spcm_dwSetParam_i32 (hCard[1], SPC_SAMPLEATE, MEGA(1));
spcm_dwSetParam_i32 (hCard[3], SPC_SAMPLEATE, MEGA(1));
```

When running the slave cards with a divided clock it is simply necessary to write the desired sampling rate to this card. The synchronization will automatically calculate the matching divider and set up all details internally:

```
// set the clock master to 1 MS/s internal clock
spcm_dwSetParam_i32 (hCard[2], SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hCard[2], SPC_SAMPLEATE, MEGA(1));

// set all the slaves to run with 100 kS/s only
spcm_dwSetParam_i32 (hCard[0], SPC_SAMPLEATE, KILO(100));
spcm_dwSetParam_i32 (hCard[1], SPC_SAMPLEATE, KILO(100));
spcm_dwSetParam_i32 (hCard[3], SPC_SAMPLEATE, KILO(100));
```

The slaves can only run with a sampling rate divided from the master clock using a divider up to 8190 in steps of two. Values that are not matching will be calculated to the nearest matching value on start of the synchronization.

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 (except trigger delay) 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:

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.
M2CMD_CARD_FLUSHFIFO	80h		Used to flush input FIFOs after the card has been stopped while an acquisition was running.

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
spcm_dwSetParam_i32 (hSync, SPC_SYNC_CLKMASK,    0x0004); // card 2 is selected as clock master

// 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_SAMPLEATE, MEGA(1));
    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 sampling rates or 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.

Excluding cards from trigger synchronization

When synchronizing cards with the Star-Hub option it is possible and most likely to synchronize clock and trigger. For some applications it can be useful to synchronize the sampling clock only for one or multiple cards. This can be useful, when acquisition cards are synchronized together with one or multiple generation cards. When these cards are used to feed a DUT (device under test) with signals and the result/ reaction is to be recorded, it is often necessary that the generation is in progress before the acquisition can begin.

For such applications it is possible to exclude one or multiple of the synchronized cards from receiving the Star-Hub trigger:

Register	Value	Direction	Description
SPC_SYNC_NOTRIGSYNCMASK	49210	read/write	Bitmask that defines which of the connected cards is using it's own trigger engine as trigger source instead of using the synchronization trigger. If set to 1, a card only uses the synchronization clock, when set to 0 the card uses also the synchronization trigger. By default this mask is set to 0 for all cards.

The following example shows, how to exclude certain cards from receiving the synchronization trigger:

```
spcm_dwSetParam_i32 (hSync, SPC_SYNC_NOTRIGSYNCMASK, 0x00000005);  // Exclude cards 0 and 2 from sync trigger
```

By default all cards that are enabled for synchronization are set to take part in clock and trigger synchronization.

SH-Direct: using the Star-Hub clock directly without synchronization

Starting with driver version 1.26 build 1754 it is possible to use the clock from the star-hub just like an external clock and running one or more cards totally independent of the synchronized card. The mode is p.e. useful if one has one or more output cards that run continuously in a loop and are synchronized with star-hub and in addition to this one or more acquisition cards should make multiple acquisitions but using the same clock.

It is also possible to run the „slave" cards with a divied clock. Therefore please program a desired divided sampling rate in the SPC_SAMPLERATE register (example: running the star-hub card with 10 MS/s and the independent cards with 1 MS/s). The sampling rate is automatically adjusted by the driver to the next matching value.

What is necessary?

- All cards need to be connected to the star-hub
- The card(s) that should run independently can not hold the star-hub
- The card(s) with the star-hub must be setup to synchronization even if it's only one card
- The synchronized card(s) have to be started prior to the card(s) that run with the direct star-hub clock

Setup

At first all cards that should run synchronized with the star-hub are set-up exactly as explained before. The card(s) that should run independently and use the star-hub clock need to use the following clock mode:

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_SHDIRECT	128	Uses the clock from the star-hub as if this was an external clock	

Example

In this example we have one generator card with the star-hub mounted running in a continuous loop and one acquisition card running independently using the SH-Direct clock.

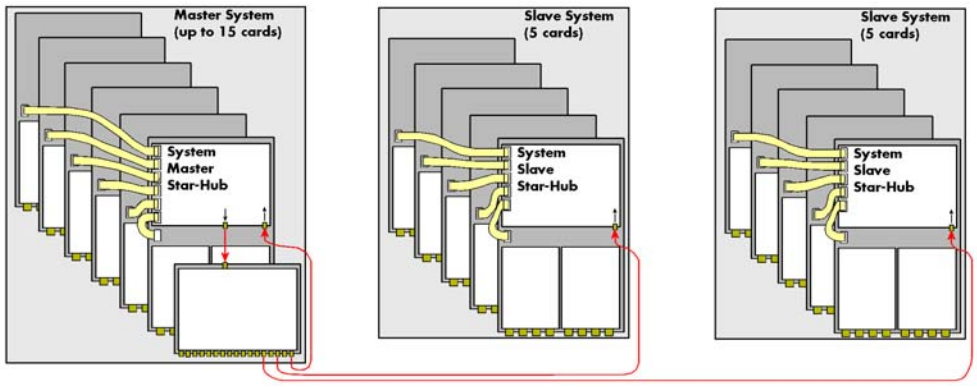
```
// setup of the generator card
spcm_dwSetParam_i32 (hCard[0], SPC_CARDMODE, SPC_REP_STD_CONTINUOUS);
spcm_dwSetParam_i32 (hCard[0], SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hCard[0], SPC_SAMPLEATE, MEGA(1));
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_ORMASK, SPC_TM_SOFTWARE);

spcm_dwSetParam_i32 (hSync, SPC_SYNC_ENABLEMASK, 0x0001); // card 0 is the generator card
spcm_dwSetParam_i32 (hSync, SPC_SYNC_CLKMASK,    0x0001); // ...

// Setup of the acquisition card (waiting for external trigger)
spcm_dwSetParam_i32 (hCard[1], SPC_CARDMODE, SPC_REC_STD_SINGLE);
spcm_dwSetParam_i32 (hCard[1], SPC_CLOCKMODE, SPC_CM_SHDIRECT);
spcm_dwSetParam_i32 (hCard[1], SPC_SAMPLEATE, MEGA(1));
spcm_dwSetParam_i32 (hCard[1], SPC_TRIG_ORMASK, SPC_TMASK_EXT0);
spcm_dwSetParam_i32 (hCard[1], SPC_TRIG_EXT0_MODE, SPC_TM_POS);

// now start the generator card (sync!) first and then the acquisition card 2 times
spcm_dwSetParam_i32 (hSync, SPC_TIMEOUT, 1000);
spcm_dwSetParam_i32 (hSync, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);
spcm_dwSetParam_i32 (hCard[1], SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_CARD_WAITREADY);
spcm_dwSetParam_i32 (hCard[1], SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_CARD_WAITREADY);
```

System Star-Hub

For the synchronization of several systems which each other, special system star-hubs are available. Besides their capability to synchronize systems which each other they can also work as complete standard star-hubs as explained above.

Overview

Two different versions are available: a master system star-hub and a slave system-star-hub. When using the system synchronization feature the slave systems simply act as slaves only receiving clock and trigger information. The master system must generate these clock and trigger information and routes them to all slave systems. All cables are made of equal length minimizing any phase delay between the different channels.



An installed master system can be extended by further systems at any time until the maximum number of systems is reached. Each of the slave systems as well as the master system can be extended by further cards until the maximum number of cards per system is reached.

Programming

For system star-hubs is not yet implemented...

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.

error name	value (hex)	value (dec.)	error description
ERR_OK	0h	0	Execution OK, no error.
ERR_INIT	1h	1	An error occurred when initialising the given card. Either the card has already been opened by another process or an hardware error occurred.
ERR_TYP	3h	3	Initialisation 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 reinstallation 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_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_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_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_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 actual setup (e.g. Only channel 0 is accessible in interlace mode)
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.

error name	value (hex)	value (dec.)	error description
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_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 http:// www.spectrum-instrumentation.com and install this one.
ERR_FIFOHWOVERRUN	301h	769	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. If acquisition speed is smaller than the theoretical bus transfer speed please check the application buffer and try to improve the handling of this one.
ERR_FIFOFINISHED	302h	770	FIFO transfer has been finished, programmed data length has been transferred completely.
ERR_STARHUB	320h	800	The auto routing function of the star-hub initialisation 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.

Continuous buffer for increased data transfer rate

The continuous 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 physical 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.

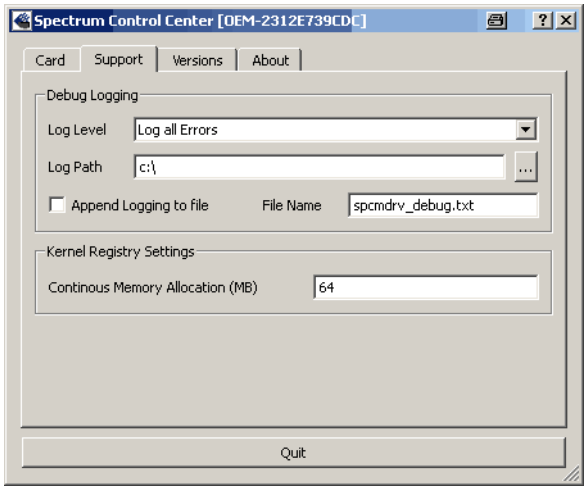
Next we'll see some measuring results of the data transfer rate with/without continuous buffer. You will find more results on different motherboards and systems in the application note number 6 „Bus Transfer Speed Details"

Bus Transfer Speed Details (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%

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 386 MByte of continuous buffer on a system with 4 GByte memory installed.

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.

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 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 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_i32 (hDrv, SPC_MEMSIZE, lMemsize);                    // recording length
spcm_dwSetParam_i32 (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 == 0)
    pvData = new int16[lMemsize];

// 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);

if (qwContBufLen == 0)
    delete (pvData);
```

Register overview

These tables give you an overview on all available registers for your card type and shortly explains them. Most of these registers are explained in greater detail throughout the manual. Please use this register tables as a reference for programming.

Card information registers

Register	Value	Direction	Description
SPC_PCITYP	2000	read	Type of card. Upper 16 bit show the series (M2i is 0x00030000), lower 16 bit show the type as hex value
SPC_PCIVERSION	2010	read	Base card version: the upper 16 bit show the hardware (PCB) version, the lower 16 bit show the firmware version.
SPC_PCIEXTVERSION	2011	read	Extension module version: the upper 16 bit show the hardware (PCB) version, the lower 16 bit show the firmware version.
SPC_PCIMODULEVERSION	2012	read	Module version: the upper 16 bit show the hardware (PCB) version, the lower 16 bit show the firmware version.
SPC_PCIDATE	2020	read	Production date: week in bit 31 to 16, year in bit 15 to 0
SPC_CALIBDATE	2025	read	Last calibration date: week in bit 31 to16, year in bit 15 to 0
SPC_PCISERIALNO	2030	read	Serial number of the board
SPC_PCISAMPLERATE	2100	read	Maximum sampling rate in Hz as a 32 bit integer value
SPC_PCIMEMSIZE	2110	read _i64	Installed memory in bytes as a 64 bit integer value
SPC_PCIFEATURES	2120	read	PCI feature register. Holds the installed features and options as a bitfield. All possible features are shown in the below list
SPCM_FEAT_MULTI	1h	Is set if the option Multiple Recording / Multiple Replay is installed.	
SPCM_FEAT_GATE	2h	Is set if the option Gated Sampling / Gated Replay is installed.	
SPCM_FEAT_DIGITAL	4h	Is set if the option Digital Inputs / Digital Outputs is installed.	
SPCM_FEAT_TIMESTAMP	8h	Is set if the option Timestamp is installed.	
SPCM_FEAT_STARHUB5	20h	Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 5 cards.	
SPCM_FEAT_STARHUB16	40h	Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 16 cards.	
SPCM_FEAT_ABA	80h	Is set if the option ABA mode is installed.	
SPCM_FEAT_BASEXIO	100h	Is set if the extra XIO connectors are used for asynchronous digital I/O.	

Register	Value	Direction	Description
SPC_FNCTYPE	2001	read	Returns the general function of the card as listed below
SPCM_TYPE_AI	1h	The card is an analog input card (transient recorder, digitizer)	
SPCM_TYPE_AO	2h	The card is an analog output card (arbitrary waveform generator)	
SPCM_TYPE_DI	4h	The card is an digital input card	
SPCM_TYPE_DO	8h	The card is an digital output card (pattern generator)	
SPCM_TYPE_DIO	10h	The card is an digital input/output card	

Register	Value	Direction	Description
SPC_MIINST_MODULES	1100	read	Returns the number of modules installed on the card (not counting extension)
SPC_MIINST_CHPERMODULE	1110	read	Returns the number of channels that are available on each module
SPC_MIINST_BYTESPERSAMPLE	1120	read	Returns the number of bytes that is needed for one sample
SPC_MIINST_BITSPERSAMPLE	1125	read	Returns the number of bits that one sample has (A/D and D/A: resolution of converter)
SPC_MIINST_MINADCLOCK	1130	read	Returns the minimum A/D or D/A clock in Hz that the converter can cope. This is the minimum allowed external clock. For internal clocking, any sampling rate beneath this border will be using oversampling.
SPC_MIINST_MAXADCLOCK	1140	read	Returns the maximum sampling clock in Hz.
SPC_MIINST_QUARZ	1150	read	returns frequency of the installed standard quartz in Hz.
SPC_MIINST_QUARZ2	1151	read	Returns frequency of the installed optional queers 2 in Hz.
SPC_MIINST_ISDEMOCARD	1175	read	Returns a value > zero if card is a demo card
SPC_GETDRVVERSION	1200	read	Gives information about the driver library version. Bit 24 to 31 hold the mayor version, bit 16 to 23 the minor version and bit 0 to 15 the build number.
SPC_GETKERNELVERSION	1210	read	Gives information about the kernel driver version. Bit 24 to 31 hold the mayor version, bit 16 to 23 the minor version and bit 0 to 15 the build number.
SPC_GETDRVTYPE	1220	read	Gives information about what type of driver is actually used
DRVTYP_LINUX	1	Linux driver is used	
DRVTYP_WDM	4	Windows WDM driver is used (only Windows 2000/XP/XP64/Vista).	

Standard card setup and 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.	
M2CMD_CARD_FLUSHFIFO	80h	Used to flush input FIFOs after the card has been stopped while an acquisition was running.	
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 fro 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 an generation mode receiving this command means that the output has stopped.	
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.	
M2CMD_DATA_POLL	80000h	Polls data without using DMA. It is not possible to mix DMA and polling mode.	
M2CMD_EXTRA_STARTDMA	100000h	Starts the DMA transfer for an already defined buffer for ABA data and timestamp data (shared buffer).	
M2CMD_EXTRA_WAITDMA	200000h	Waits until the data transfer of ABA and timestamp data 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_EXTRA_STOPDMA	400000h	Stops a running DMA transfer. Data is invalid afterwards.	
M2CMD_EXTRA_POLL	800000h	Polls data without using DMA. It is not possible to mix DMA and polling mode.	

Register	Value	Direction	Description
SPC_M2STATUS	110	read only	Reads out the current status information
M2STAT_CARD_PRETRIGGER	1h	Acquisition modes only: the pretrigger area has been filled.	
M2STAT_CARD_TRIGGER	2h	The first trigger has been detected.	
M2STAT_CARD_READY	4h	The card has finished it's run and is ready.	
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.	
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 for either ABA data or timestamp data 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 of either ABA data FIFO or timestamp data FIFO.	
M2STAT_EXTRA_ERROR	8000h	An internal error occurred while doing data transfer.	

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
SPC_ABA_AVAIL_USER_LEN	210	read	Returns the number of currently to the user available bytes inside a ABA data transfer.
SPC_ABA_AVAIL_USER_POS	211	read	Returns the position as byte index where the currently available ABA samples start.
SPC_ABA_AVAIL_CARD_LEN	212	write	Writes the number of bytes that the card can now use for sample ABA transfer again
SPC_TS_AVAIL_USER_LEN	220	read	Returns the number of currently to the user available bytes inside a timestamp data transfer.
SPC_TS_AVAIL_USER_POS	221	read	Returns the position as byte index where the currently available timestamp samples start.
SPC_TS_AVAIL_CARD_LEN	222	write	Writes the number of bytes that the card can now use for sample timestamp transfer again
SPC_MEMSIZE	10000	read/write	Defines the used memory size in samples per channel for all standard modes, not used in FIFO.
SPC_SEGMENTSIZE	10010	read/write	Multiple Recording: size of one segment, number of samples to be record after each trigger event. Multiple Replay: size of one segment, number of samples to be generated after each trigger event
SPC_LOOPS	10020	read/write	Defines the number of segments to be recorded/replayed in FIFO mode, a zero for endless
SPC_PRETRIGGER	10030	read/write	Gated Sampling: Defines the number of samples to be recorded prior to the gate start. Gated Replay: no function
SPC_ABADIVIDER	10040	read/write	Acquisition: programs the divider which is used to sample slow ABA data if ABA option is installed Replay: no function
SPC_POSTTRIGGER	10100	read/write	Acquisition: sets the number of samples to be recorded after the trigger event has been detected. Replay: no function.

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.
SPC_REC_STD_SINGLE	1h	Data acquisition to on-board memory for one single trigger event.	
SPC_REC_STD_MULTI	2h	Data acquisition to on-board memory for multiple trigger events. Each recorded segment has the same size. Only available if option Multiple Recording is installed. this mode is described in greater detail in a special chapter about the Multiple Recording option.	
SPC_REC_STD_GATE	4h	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. This mode is only available if the Gated Sampling option is installed. The mode is described in greater detail in a special chapter about the Gated Sampling option.	
SPC_REC_STD_ABA	8h	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. This mode is only available if the ABA mode option is installed. The mode is described in a special chapter about ABA mode option.	
SPC_REC_FIFO_SINGLE	10h	Continuous data acquisition for one single trigger event. The on-board memory is used completely as FIFO buffer.	
SPC_REC_FIFO_MULTI	20h	Continuous data acquisition for multiple trigger events. Only available if Multiple Recording option is installed.	
SPC_REC_FIFO_GATE	40h	Continuous data acquisition using an external gate signal. only available if Gated Sampling option is installed.	
SPC_REC_FIFO_ABA	80h	Continuous data acquisition for multiple trigger events together with continuous data acquisition with a slower sampling clock. Only available if ABA mode option is installed	

Register	Value	Direction	Description
SPC_CHENABLE	11000	read/write	Sets the channel enable information for the next card run.
SPC_CHCOUNT	11001	read	Reads back the number of currently activated channels.
SPC_FILLSIZEPROMILLE	200910	read	Returns the current fill size of the data FIFO in promille, granularity is 1000/16
SPC_TIMEOUT	295130	read/write	Defines the timeout for any following wait command in a milli second resolution. Writing a zero to this register disables the timeout.

Clock Settings

Register	Value	Direction	Description
SPC_SAMPLERATE	20000	read/write	Defines the sampling rate in Hz for internal sample rate generation. Read access return the currently selected sampling rate that best matches the setup.
SPC_OVERSAMPLINGFACTOR	200123	read only	Returns the oversampling factor for further calculations. If oversampling isn't active a 1 is returned.
SPC_CLOCKDIV	20040	read/write	Register for setting the internal clock divider. Values up to 8190 in steps of two are allowed.
SPC_CLOCKOUT	20110	read/write	Enables clock output on external clock connector. Only possible with internal clocking.
SPC_CLOCK50OHM	20120	read/write	A „1" enables the 50 Ohm termination at the external clock connector. Only possible, when using the external connector as an input.
SPC_EXTERNRANGE	20130	read/write	Defines the range of the actual fed in external clock. Use EXRANGE_LOW or EXRANGE_HIGH
SPC_REFERENCECLOCK	20140	read/write	Programs the external reference clock in the range from 1 MHz to 125 MHz.
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 PLL with 10 MHz internal reference for sample clock generation	
SPC_CM_QUARTZ1	2	Enables Quartz1 for sample clock generation	
SPC_CM_QUARTZ2	4	Enables optional Quartz2 for sample clock generation	
SPC_CM_EXTERNAL	8	Enables external clock input for direct sample clock generation	
SPC_CM_EXTDIVIDER	16	Enables external clock input for divided sample clock generation	
SPC_CM_EXTREFCLOCK	32	Enables internal PLL with external reference for sample clock generation	

Trigger Settings

Register	Value	Direction	Description
SPC_TRIG_AVAILORMASK	40400	r	Bitmask, in which all bits of the below mentioned sources for the OR mask are set, if available.
SPC_TRIG_ORMASK	40410	r/w	Defines the events included within the trigger OR mask of the card.
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 trigger0 for the OR mask. The card will trigger when the programmed condition for this input is valid.	
SPC_TMASK_EXT1	4h	Enables the external trigger1 for the OR mask. This input is only available on digital cards. The card will trigger when the programmed condition for this input is valid.	
SPC_TMASK_XIO0	100h	Enables the extra TTL trigger 0 for the OR mask. This input is only available when the option BaseXIO is installed.	
SPC_TMASK_XIO1	200h	Enables the extra TTL trigger 1 for the OR mask. This input is only available when the option BaseXIO is installed.	

Register	Value	Direction	Description
SPC_TRIG_AVAILANDMASK	40420	r	Bitmask, in which all bits of the below mentioned sources for the AND mask are set, if available.
SPC_TRIG_ANDMASK	40430	r/w	Defines the events included within the trigger AND mask of the card.
SPC_TMASK_EXT0	2h	Enables the external trigger0 for the AND mask. The card will trigger when the programmed condition for this input is valid.	
SPC_TMASK_EXT1	4h	Enables the external trigger1 for the AND mask. This input is only available on digital cards. The card will trigger when the programmed condition for this input is valid.	
SPC_TMASK_XIO0	100h	Enables the extra TTL trigger 0 for the OR mask. This input is only available when the option BaseXIO is installed.	
SPC_TMASK_XIO1	200h	Enables the extra TTL trigger 1 for the OR mask. This input is only available when the option BaseXIO is installed.	

Register	Value	Direction	Description
SPC_TRIG_EXT_AVAILMODES	40500	read	Bitmask, in which all bits of the below mentioned modes for the external trigger are set, if available.
SPC_TRIG_EXT0_MODE	40510	read/write	Defines the external TTL trigger mode for the external SMB connector (A/D and D/A boards only). On digital boards this defines the TTL trigger mode for the trigger input of the first module (Mod A).
SPC_TRIG_EXT1_MODE	40511	read/write	Defines the external TTL trigger mode for the trigger input of the second module (digital boards only).
SPC_TRIG_XIO0_MODE	40560	read/write	Defines the trigger mode for the extra TTL input 0. These trigger inputs are only available, when option BaseXIO is installed.
SPC_TRIG_XIO1_MODE	40561	read/write	Defines the trigger mode for the extra TTL input 1. These trigger inputs are only available, when option BaseXIO is installed.
SPC_TM_POS	1h	Sets the trigger mode for external TTL trigger to detect positive edges.	
SPC_TM_NEG	2h	Sets the trigger mode for external TTL trigger to detect negative edges	
SPC_TM_BOTH	4h	Sets the trigger mode for external TTL trigger to detect positive and negative edges	
SPC_TM_HIGH	8h	Sets the trigger mode for external TTL trigger to detect HIGH levels.	
SPC_TM_LOW	10h	Sets the trigger mode for external TTL trigger to detect LOW levels.	
SPC_TM_POS | SPC_TM_PW_GREATER	4000001h	Sets the trigger mode for external TTL trigger to detect HIGH pulses that are longer than a programmed pulsewidth.	
SPC_TM_POS | SPC_TM_PW_SMALLER	2000001h	Sets the trigger mode for external TTL trigger to detect HIGH pulses that are shorter than a programmed pulsewidth.	
SPC_TM_NEG | SPC_TM_PW_GREATER	4000002h	Sets the trigger mode for external TTL trigger to detect LOW pulses that are longer than a programmed pulsewidth.	
SPC_TM_NEG | SPC_TM_PW_SMALLER	2000002h	Sets the trigger mode for external TTL trigger to detect LOW pulses that are shorter than a programmed pulsewidth.	

Register	Value	Direction	Description
SPC_TRIG_OUTPUT	40100	read/write	Enables the trigger output if internal trigger is detected
SPC_TRIG_TERM	40110	read/write	A „1“ sets the 50 Ohm termination, if the trigger connector is used as an input for external trigger signals. A „0“ sets the 1 MOhm termination
SPC_TRIG_EXT_AVAILPULSEWIDTH	44200	read	Contains the maximum possible value, for the external trigger pulsewidth counter.
SPC_TRIG_EXT0_PULSEWIDTH	44210	read/write	Sets the pulsewidth for external trigger in samples.

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.

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILORMASK0	40450	read	Bitmask, in which all bits of the below mentioned sources/channels (0…31) for the channel OR mask are set, if available.
SPC_TRIG_CH_AVAILORMASK1	40451	read	Bitmask, in which all bits of the below mentioned sources/ channels (32…63) for the channel OR mask are set, if available.
SPC_TRIG_CH_ORMASK0	40460	read/write	Includes the analog or digital channels (0…31) within the channel trigger OR mask of the card.
SPC_TRIG_CH_ORMASK1	40461	read/write	Includes the analog or digital channels (32…63) within the channel trigger OR mask of the card.
SPC_TMASK0_CH0	1h	Enables channel0 (channel16) for recognition within the channel OR mask.	
SPC_TMASK0_CH1	2h	Enables channel1 (channel17) for recognition within the channel OR mask.	
SPC_TMASK0_CH2	4h	Enables channel2 (channel18) for recognition within the channel OR mask.	
SPC_TMASK0_CH3	8h	Enables channel3 (channel19) for recognition within the channel OR mask.	
…	…	…	

SPC_TMASK0_CH28	10000000h	Enables channel28 (channel60) for recognition within the channel OR mask.
SPC_TMASK0_CH29	20000000h	Enables channel29 (channel61 for recognition within the channel OR mask.
SPC_TMASK0_CH30	40000000h	Enables channel30 (channel62) for recognition within the channel OR mask.
SPC_TMASK0_CH31	80000000h	Enables channel31 (channel63) for recognition within the channel OR mask.

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILANDASK0	40470	read	Bitmask, in which all bits of the below mentioned sources/channels (0…31) for the channel AND mask are set, if available.
SPC_TRIG_CH_AVAILANDMASK1	40471	read	Bitmask, in which all bits of the below mentioned sources/ channels (32…63) for the channel AND mask are set, if available.
SPC_TRIG_CH_ANDMASK0	40480	read/write	Includes the analog or digital channels (0…31) within the channel trigger AND mask of the card.
SPC_TRIG_CH_ANDRMASK1	40481	read/write	Includes the analog or digital channels (32…63) within the channel trigger AND mask of the card.

SPC_TMASK0_CH0	1h	Enables channel0 (channel16) for recognition within the channel AND mask.
SPC_TMASK0_CH1	2h	Enables channel1 (channel17) for recognition within the channel AND mask.
SPC_TMASK0_CH2	4h	Enables channel2 (channel18) for recognition within the channel AND mask.
SPC_TMASK0_CH3	8h	Enables channel3 (channel19) for recognition within the channel AND mask.
…	…	…
SPC_TMASK0_CH28	10000000h	Enables channel28 (channel60) for recognition within the channel AND mask.
SPC_TMASK0_CH29	20000000h	Enables channel29 (channel61 for recognition within the channel AND mask.
SPC_TMASK0_CH30	40000000h	Enables channel30 (channel62) for recognition within the channel AND mask.
SPC_TMASK0_CH31	80000000h	Enables channel31 (channel63) for recognition within the channel AND mask.

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 channel0.
SPC_TRIG_CH1_MODE	40611	read/write	Sets the trigger mode for channel1.
SPC_TRIG_CH2_MODE	40612	read/write	Sets the trigger mode for channel2.
SPC_TRIG_CH3_MODE	40613	read/write	Sets the trigger mode for channel3.

SPC_TM_NONE	0h	Channel is not used for trigger detection. This is as with the trigger masks another possibility for disabling channels.
SPC_TM_POS	1h	Enables the trigger detection for positive edges
SPC_TM_NEG	2h	Enables the trigger detection for negative edges
SPC_TM_BOTH	4h	Enables the trigger detection for positive and negative edges
SPC_TM_HIGH	8h	Enables the trigger detection for HIGH levels
SPC_TM_LOW	10h	Enables the trigger detection for LOW levels
SPC_TM_POS | SPC_TM_PW_GREATER	4000001h	Enables the pulsewidth trigger detection for long positive pulses
SPC_TM_NEG | SPC_TM_PW_GREATER	4000002h	Enables the pulsewidth trigger detection for long negative pulses
SPC_TM_POS | SPC_TM_PW_SMALLER	2000001h	Enables the pulsewidth trigger detection for short positive pulses
SPC_TM_NEG | SPC_TM_PW_SMALLER	2000002h	Enables the pulsewidth trigger detection for short negative pulses
SPC_TM_STEEPPOS | SPC_TM_PW_GREATER	4000800h	Enables the steepness trigger detection for flat positive pulses
SPC_TM_STEEPNEG | SPC_TM_PW_GREATER	4001000h	Enables the steepness trigger detection for flat negative pulses
SPC_TM_STEEPPOS | SPC_TM_PW_SMALLER	2000800h	Enables the steepness trigger detection for steep positive pulses
SPC_TM_STEEPNEG | SPC_TM_PW_SMALLER	2000800h	Enables the steepness trigger detection for steep negative pulses
SPC_TM_WINENTER	20h	Enables the window trigger for entering signals
SPC_TM_WINLEAVE	40h	Enables the window trigger for leaving signals
SPC_TM_INWIN	80h	Enables the window trigger for inner signals
SPC_TM_OUTSIDEWIN	100h	Enables the window trigger for outer signals
SPC_TM_SPIKE	200h	Enables the spike trigger mode. This mode is not availavle on all M2i boards.
SPC_TM_WINENTER | SPC_TM_PW_GREATER	4000020h	Enables the window trigger for long inner signals
SPC_TM_WINLEAVE | SPC_TM_PW_GREATER	4000040h	Enables the window trigger for long outer signals
SPC_TM_WINENTER | SPC_TM_PW_SMALLER	2000020h	Enables the window trigger for short inner signals
SPC_TM_WINLEAVE | SPC_TM_PW_SMALLER	2000040h	Enables the window trigger for short outer signals

Register	Value	Direction	Description
SPC_TRIG_CH0_LEVEL0	42200	read/write	Defines the upper level (trigger level) for channel 0
SPC_TRIG_CH1_LEVEL0	42201	read/write	Defines the upper level (trigger level) for channel 1
SPC_TRIG_CH2_LEVEL0	42202	read/write	Defines the upper level (trigger level) for channel 2
SPC_TRIG_CH3_LEVEL0	42203	read/write	Defines the upper level (trigger level) for channel 3
SPC_TRIG_CH0_LEVEL1	42300	read/write	Defines the lower level (trigger level) for channel 0
SPC_TRIG_CH1_LEVEL1	42301	read/write	Defines the lower level (trigger level) for channel 1
SPC_TRIG_CH2_LEVEL1	42302	read/write	Defines the lower level (trigger level) for channel 2
SPC_TRIG_CH3_LEVEL1	42303	read/write	Defines the lower level (trigger level) for channel 3
SPC_TRIG_CH0_PULSEWIDTH	44101	read/write	Sets the pulsewidth in samples for ch 0 trigger modes using pulsewidth counters
SPC_TRIG_CH1_PULSEWIDTH	44102	read/write	Sets the pulsewidth in samples for ch 1 trigger modes using pulsewidth counters
SPC_TRIG_CH2_PULSEWIDTH	44103	read/write	Sets the pulsewidth in samples for ch 2 trigger modes using pulsewidth counters
SPC_TRIG_CH3_PULSEWIDTH	44104	read/write	Sets the pulsewidth in samples for ch 3 trigger modes using pulsewidth counters

Registers for timestamp option

Register	Value	Direction	Description
SPC_TIMESTAMP_STARTTIME	47030	read/write	Return the reset time when using reference clock mode
SPC_TIMESTAMP_STARTDATE	47031	read/write	Return the reset date when using reference clock mode
SPC_TIMESTAMP_TIMEOUT	47045	read/write	Set's a timeout in milli seconds for waiting of an reference clock edge
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. If reference clock mode is used this command waits for the edge the timeout time.	
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_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	4096	Enables the trigger synchronous acquisition of the BaseXIO inputs with every stored timestamp in the upper byte.	
SPC_TSXIOACQ_DISABLE	0	The timestamp is filled up with leading zeros as a sign extension for positive values.	

Registers for BaseXIO option

Register	Value	Direction	Description
SPC_XIO_DIRECTION	47100	r/w	Defines groupwise the direction of the digital I/O lines. Values can be combined by a bitwise OR.
XD_CH0_INPUT	0	Sets the direction of the lower group (bit D3…D0) to input.	
XD_CH1_INPUT	0	Sets the direction of the upper group (bit D7…D4) to input.	
XD_CH0_OUTPUT	1	Sets the direction of the lower group (bit D3…D0) to output.	
XD_CH1_OUTPUT	2	Sets the direction of the upper group (bit D7…D4) to output.	

Register	Value	Direction	Description
SPC_XIO_DIGITALIO	47110	r	Reads the data directly from the pins of all digital I/O lines either if they are declared as inputs or outputs.
SPC_XIO_DIGITALIO	47110	w	Writes the data to all digital I/O lines that are declared as outputs. Bytes that are declared as inputs will ignore the written data.

Registers for analog acquisition cards

Register	Value	Direction	Description
SPC_READIRCOUNT	3000	read	Informs about the number of the board's calibrated input ranges.
SPC_READMAXOFFSET	3100	read	Reads out the maximum input programmable offset in ±percent.
SPC_READAIFEATURES	3101	read	Returns a bitmap with available features of the analog input as listed below
SPCM_AI_TERM	1h	Input channel has programmable terminaton (50 ohm)	
SPCM_AI_SE	2h	Input channel can be programmed to single ended (if SPCM_AI_DIFF is not set, the channel is fixed to single ended)	
SPCM_AI_DIFF	4h	Input channel can be programmed to differential (if SPCM_AI_SE is not set, the channel is fixed to differential)	
SPCM_AI_OFFSPERCENT	8h	Input offset of channel is programmed in percent of current input range	
SPCM_AI_OFFSMV	10h	Input offset of the channel can be programmed in mV as absolute offset	
SPCM_AI_AUTOCALOFFS	1000h	Automatic offset calibration in hardware available	
SPCM_AI_AUTOCALGAIN	2000h	Automatic gain calibration in hardware available	
SPCM_AI_AUTOCALOFFSNOIN	4000h	Automatic offset calibration in hardware available with open inputs (no signal is allowed to be connected)	

Register	Value	Direction	Description
SPC_READRANGEMIN0	4000	read	Gives back the minimum value of input range 0 in mV.
SPC_READRANGEMIN1	4001	read	Gives back the minimum value of input range 1 in mV.
SPC_READRANGEMIN2	4002	read	Gives back the minimum value of input range 2 in mV.
...	...	read	...
SPC_READRANGEMAX0	4100	read	Gives back the maximum value of input range 0 in mV.
SPC_READRANGEMAX1	4101	read	Gives back the maximum value of input range 1 in mV.
SPC_READRANGEMAX2	4102	read	Gives back the maximum value of input range 2 in mV.
...	...	r	...

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 actual settings to the specified set in the EEPROM. T
		read	Reads out, what kind of settings have been saved last.
SPC_ADJ_AUTOADJ	50020	write	Performs the automatic offset compensation in the driver either for all input ranges or only the actual.
SPC_RELAISWAITTIME	200700	read/write	Wait time in ms for relays settling before the start of the board. Default value is 50 ms.

Register	Value	Direction	Description
SPC_OFFS0	30000	read/write	Defines the input's offset and therefore shifts the input of channel 0.
SPC_AMP0	30010	read/write	Defines the input range of channel 0 as mV. ±1 V range will be programmed as 1000
SPC_OFFS1	30100	read/write	Defines the input's offset and therefore shifts the input of channel 1.
SPC_AMP1	30110	read/write	Defines the input range of channel 1 as mV. ±1 V range will be programmed as 1000
SPC_OFFS2	30200	read/write	Defines the input's offset and therefore shifts the input of channel 2.
SPC_AMP2	30210	read/write	Defines the input range of channel 2 as mV. ±1 V range will be programmed as 1000
SPC_OFFS3	30300	read/write	Defines the input's offset and therefore shifts the input of channel 3.
SPC_AMP3	30310	read/write	Defines the input range of channel 3 as mV. ±1 V range will be programmed as 1000

Register	Value	Direction	Description
SPC_50OHM0	30030	read/write	A „1" sets the 50 ohm termination for channel 0. A „0" sets the termination to 1 MOhm.
SPC_50OHM1	30130	read/write	A „1" sets the 50 ohm termination for channel 1. A „0" sets the termination to 1 MOhm.
SPC_50OHM2	30230	read/write	A „1" sets the 50 ohm termination for channel 2. A „0" sets the termination to 1 MOhm.
SPC_50OHM3	30330	read/write	A „1" sets the 50 ohm termination for channel 3. A „0" sets the termination to 1 MOhm.

Details on M2i cards clock and trigger I/O section

The SMB clock and trigger I/O connectors of the M2i cards from Spectrum are protected against over voltage conditions.

For this purpose clamping diodes of the types 1N4148 are used. Both I/O lines are internally clamped to signal ground and to a specific clamping voltage named Vt* for the trigger and Vc* for the clock line. 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.

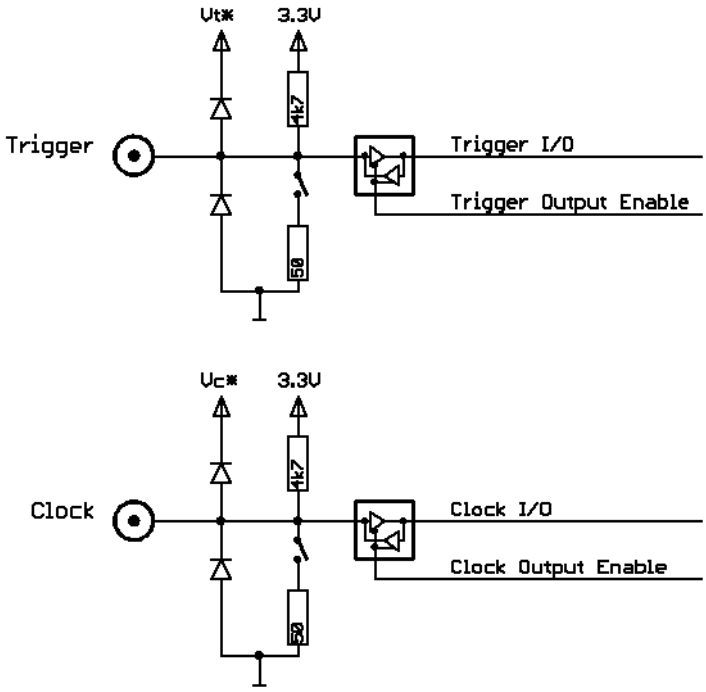
The maximum forward current limit for the used 1N4148 diodes is 100 mA.

When connecting a high performance clock or trigger source with the cards clock or trigger inputs, with logic high levels above the clamping voltage please make sure to not exceed the current limit of the clamping diodes.

This can most easily be ensured, when using the cards 50 Ohm termination and a series resistor of 33 Ohm up to 47 Ohm on the clock or trigger source.

To avoid floating levels with unconnected inputs, a pull up resistor of 4.7 kOhm to 3,3V on both lines is used.

The following table shows the values for the both clamping voltages Vt* and Vc*:



Card series	Base Hardware Version	Vt*	Vc*
M2i.xxxx	≤ V20	3.3 V	3.3 V
M2i.xxxx	> V20	5.0 V	3.3 V
M2i.xxxx-exp	> V20	5.0 V	3.3 V

For details on how to read out the base hardware version from the driver or where to find that information on the cards type plate please look up the relating sections in this manual.