SPECTRUM
SYSTEMENTWICKLUNG MICROELECTRONIC GMBH

# M4i.66xx-x8

## high-speed 16 bit AWG,
## Arbitrary Waveform Generator
## for PCI Express bus

## Hardware Manual
## Software Driver Manual

**English version** **May 11, 2016**

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 manuals are available on CD or on the Spectrum website.

For any new information on the board as well as new available options or memory upgrades please contact our website 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 Express bus was first introduced in 2004. In today's standard PC there are usually two to six slots available for instrumentation boards. Special industrial PCs offer up to a maximum of 16 slots. The PCI Express Gen2 standard theoretically delivers up to to 8 GByte/s data transfer rate per x16 slot. The Spectrum M4i boards are available as PCI Express x8 (eight lane) Gen2, 3/4 length card.

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

## General Information

The M4i.66xx series offer a wide range of ultra fast 16 bit D/A converter boards for PCI Express (PCIe) bus. Due to the well-planned design these boards are available in several versions and different speed grades. That makes it possible for the user to find an individual solution.

These boards offer one to four channels with a maximum sampling rate of 625 MS/s in addition to models with one or two channels with maximum sampling rates up to 1.25 GS/s. The installed memory of 2 GSample will be used for fast data replay. It can completely be used by the currently active channels. Alternatively the memory can be switched to a FIFO buffer and data will be transferred online from the PC memory or from hard disk.

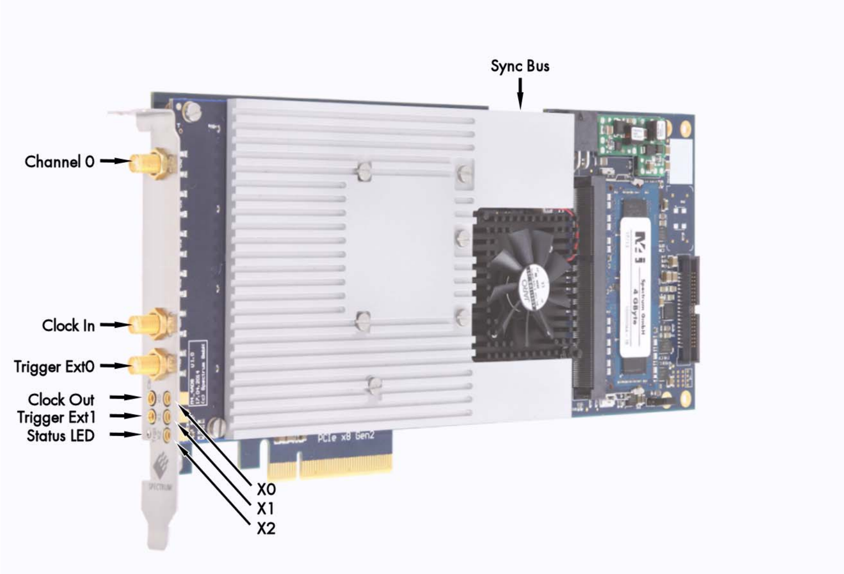
Several boards of the M4i.xxxx series may be connected together by the internal standard synchronisation bus in combination with one of the star-hub options to work with the same time base. That allows to build system with multiple D/A channels or systems with combined A/D and D/A channels.

**Application examples: Automatic test systems, IQ Signal Generation, Stimulus Response Measurements, Noise Generation, Prototype Testing**

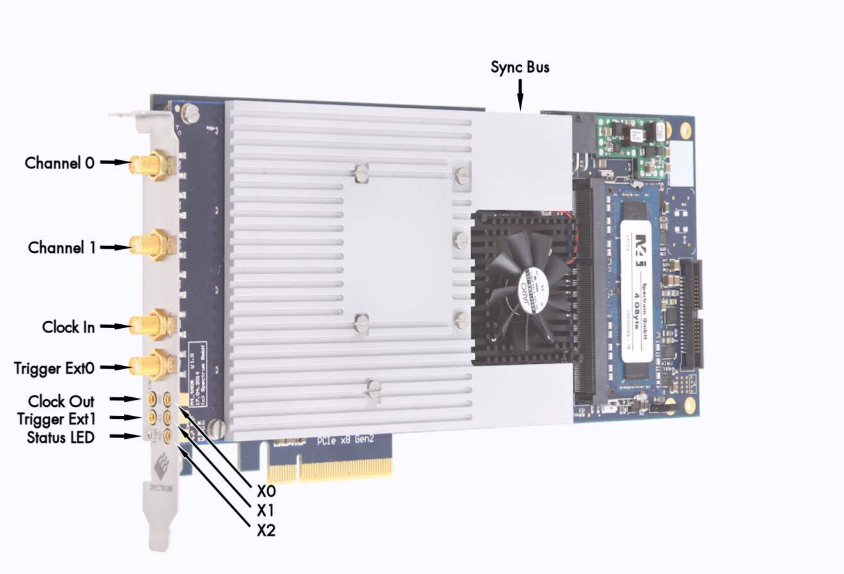
# Different models of the M4i.66xx series

The following overview shows the different available models of the M4i.66xx series. They differ in the number of available channels. You can also see the model dependant location of the input connectors.

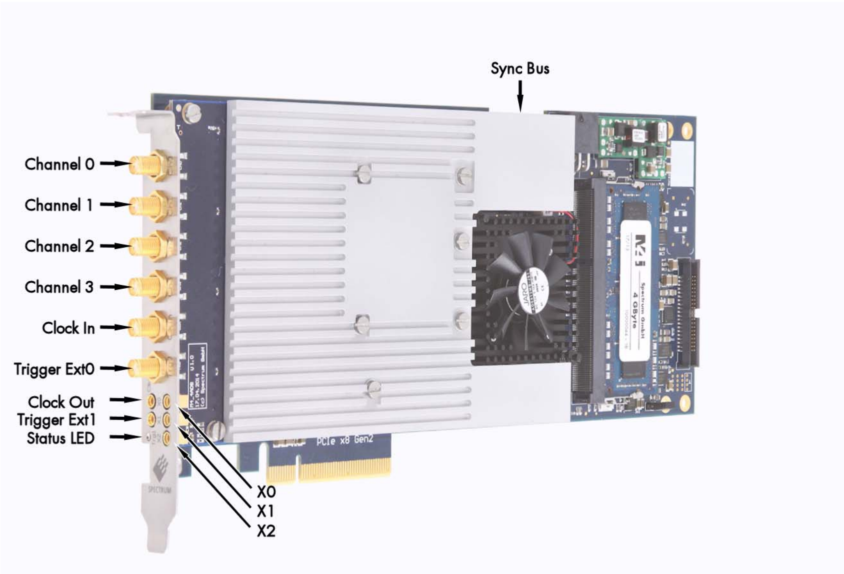
- **M4i.6620-x8**
- **M4i.6630-x8**




- **M4i.6621-x8**
- **M4i.6631-x8**

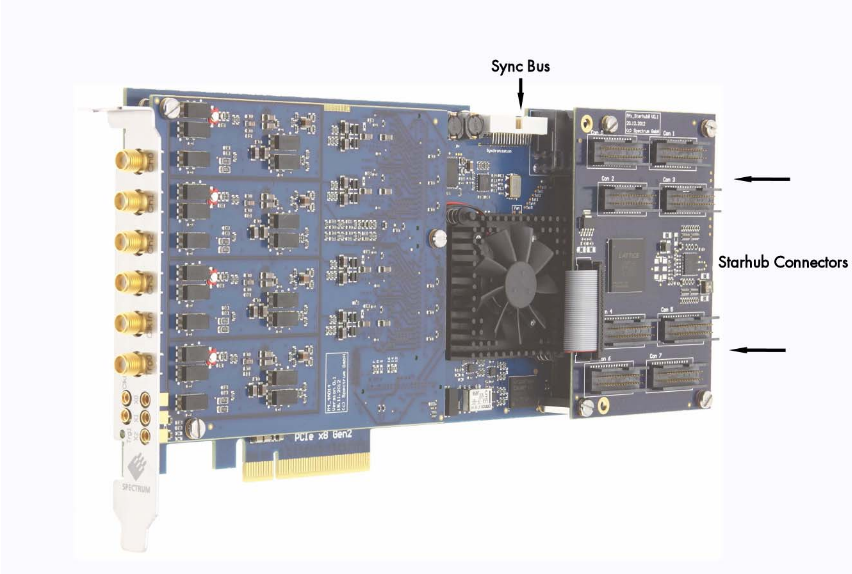



- **M4i.6622-x8**



# Additional options

## Star-Hub




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

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

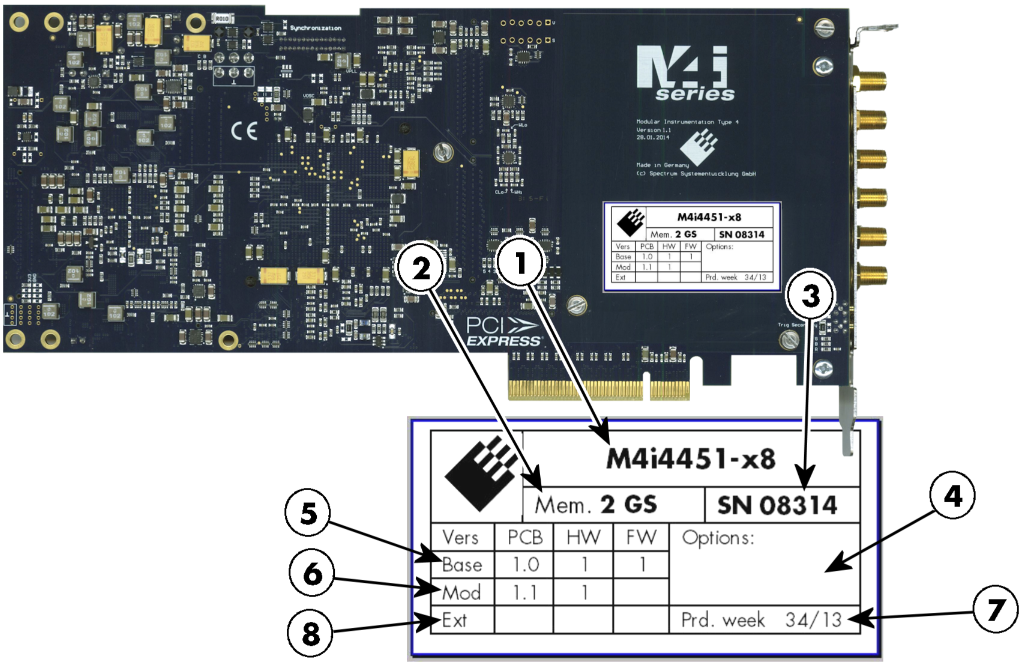
The second version (option SH8ex) is mounted behind the card and extends the M4i base card to a full-length PCI Express card. Therfore it requires the availability of a full-length slot in the system but does not need the space of an additional slot.

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

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

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

# 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 M4i for the PCI Express x8 bus) and the model number.

2. The size of the on-board installed memory in MSample or GSample. In this example there are 2 GS = 2048 MSample (4 GByte = 4096 MByte) installed.

3. The serial number of your Spectrum board. Every board has a unique serial number.

4. A list of the installed options. A complete list of all available options is shown in the order information. In this example no additional options are installed.

5. The base card version, consisting of the printed circuit board (PCB) version, the hardware version and the firmware version.

6. The version of the analog/digital front-end module, consisting of the printed circuit board (PCB) version, the hardware version and the firmware version (if available). If no programmable device is located on the module, the firmware field is left empty.

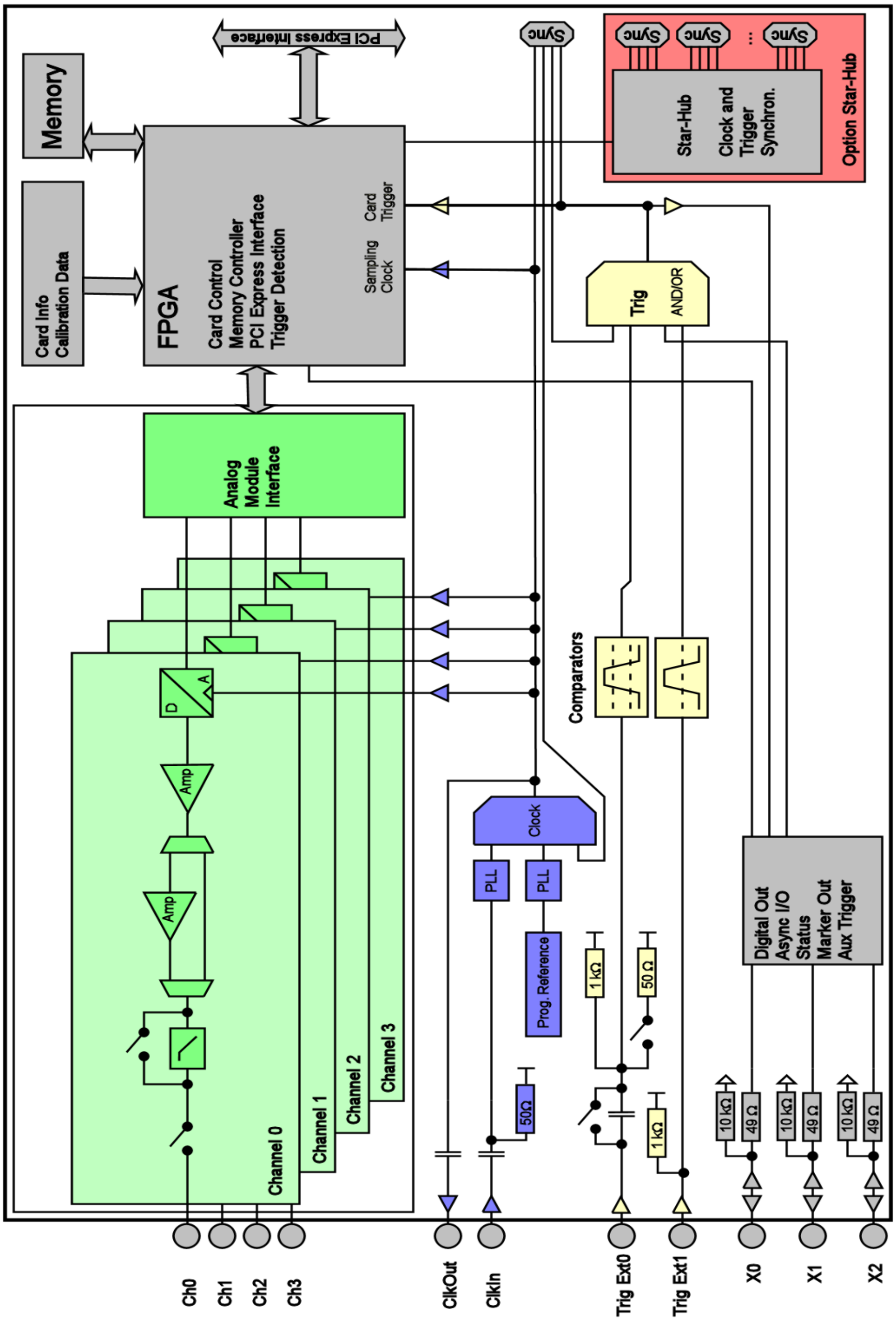
7. The date of production, consisting of the calendar week and the year.

8. The version of the extension module /such as a Starhub) if one is installed, consisting of the printed circuit board (PCB) version, the hardware version and the firmware version. If no extension module is installed this part is left empty.

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

# Hardware information

## Block diagram



# Technical Data

## Analog Outputs

| | | |
|---|---|---|
| Resolution | | 16 bit |
| D/A Interpolation | | no interpolation |
| Output amplitude M4i.663x (1.25 GS/s version) | software programmable | ±80 mV up to ±2 V in 1 mV steps into 50 Ω termination (resulting in ±160 mV up to ±4 V in 2mV steps into high impedance loads) |
| Output amplitude M4i.662x (625 MS/s version) | software programmable | ±80 mV up to ±2.5 V in 1 mV steps into 50 Ω termination (resulting in ±160 mV up to ±5 V in 2mV steps into high impedance loads) |
| Output offset | fixed | 0 V |
| Output Amplifier Path Selection | automatically by driver | Low Power path: ±80 mV to ±480 mV (into 50 Ω)<br>High Power path: ±420 mV to ±2.5 V/±2 V (into 50 Ω) |
| Output Amplifier Setting Hysteresis | automatically by driver | 420 mV to 480 mV (if output is using low power path it will switch to high power path at 480 mV. If output is using high power path it will swicth to low power path at 420 mV) |
| Output amplifier path switching time | | 10 ms (output disabled while switching) |
| Filters | software programmable | bypass with no filter or one fixed filter |
| DAC Differential non linearity (DNL) | DAC only | ±0.5 LSB typical |
| DAC Integral non linearity (INL) | DAC only | ±1.0 LSB typical |
| Output resistance | | 50 Ω |
| Minimum output load | | 0 Ω |
| Max output swing in 50 Ω | | ± 2.5 V for 625 MS/s versions or ± 2 V for 1.25 GS/s version |
| Crosstalk @ 1 MHz signal ±2.5 V into 50 Ω | | TBD |
| Output accuracy | | TBD |

## Trigger

| | | |
|---|---|---|
| Available trigger modes | software programmable | External, Software, Window, Re-Arm, Or/And, Delay |
| Trigger edge | software programmable | Rising edge, falling edge or both edges |
| Trigger delay | software programmable | 0 to (8GSamples - 32) = 8589934560 Samples in steps of 32 samples |
| Multi, Gate: re-arming time | | 40 samples |
| Trigger to Output Delay | M4i.662x series | 244 sample clocks (fixed) |
| Trigger to Output Delay | M4i.663x series | TBD |
| Memory depth | software programmable | 32 up to [installed memory / number of active channels] samples in steps of 32 |
| Multiple Replay segment size | software programmable | 16 up to [installed memory / 2 / active channels] samples in steps of 16 |
| External trigger accuracy | | 1 sample |
| Minimum external trigger pulsewidth | | ≥ 2 samples |

| | | | |
|---|---|---|---|
| External trigger | | **Ext0** | **Ext1** |
| External trigger impedance | software programmable | 50 Ω /1 kΩ | 1 kΩ |
| External trigger coupling | software programmable | AC or DC | fixed DC |
| External trigger type | | Window comparator | Single level comparator |
| External input level | | ±10 V (1 kΩ), ±2.5 V (50 Ω), | ±10 V |
| External trigger sensitivity (minimum required signal swing) | | 2.5% of full scale range | 2.5% of full scale range = 0.5 V |
| External trigger level | software programmable | ±10 V in steps of 1 mV | ±10 V in steps of 1 mV |
| External trigger maximum voltage | | ±30V | ±30 V |
| External trigger bandwidth DC | 50 Ω /1 kΩ | DC to 200 MHz / 150 MHz | DC to 200 MHz |
| External trigger bandwidth AC | 50 Ω | 20 kHz to 200 MHz | n.a. |

## Clock

| | | |
|---|---|---|
| Clock Modes | software programmable | internal PLL, external reference clock, sync |
| Internal clock accuracy | | ≤ ±20 ppm |
| Internal clock setup granularity | | 8 Hz (internal reference clock only, restrictions apply to external reference clock) |
| Setable Clock speeds | | 50 MHz to max sampling clock |
| Clock Setting Gaps | | 750 to 757 MHz, 1125 to 1145 MHz (no samping clock possible in these gaps) |
| External reference clock range | software programmable | ≥ 10 MHz and ≤ 1.25 GHz |
| External reference clock input impedance | software programmable | 50 Ω fixed |
| External reference clock input coupling | | AC coupling |
| External reference clock input edge | | Rising edge |
| External reference clock input type | | Single-ended, sine wave or square wave |
| External reference clock input swing | | 0.3 V peak-peak up to 3.0 V peak-peak |
| External reference clock input max DC voltage | | ±30 V (with max 3.0 V difference between low and high level) |
| External reference clock input duty cycle requirement | | 45% to 55% |
| External reference clock output type | | Single-ended, 3.3V LVPECL |
| Clock output | sampling clock ≤71.68 MHz | Clock output = sampling clock/4 |
| Clock output | sampling clock >71.68 MHz | Clock output = sampling clock/8 |
| Star-Hub synchronization clock modes | software selectable | Internal clock, external reference clock |

## Sequence Replay Mode (Mode available starting with firmware V1.14)

| | | |
|---|---|---|
| Number of sequence steps | software programmable | 1 up to 4096 (sequence steps can be overloaded at runtime) |
| Number of memory segments | software programmable | 2 up to 64k (segment data can be overloaded at runtime) |
| Loop Count | software programmable | 1 to (1M - 1) loops |
| Sequence Step Commands | software programmable | Loop for #Loops, Next, Loop until Trigger, End Sequence |
| Special Commands | software programmable | Data Overload at runtime, sequence steps overload at runtime, readout current replayed sequence step |

## Multi Purpose I/O lines (front-plate)

| | | |
|---|---|---|
| Number of multi purpose lines | | three, named X0, X1, X2 |
| Input: available signal types | software programmable | Asynchronous Digital-In |
| Input: impedance | | 10 kΩ to 3.3 V |
| Input: maximum voltage level | | -0.5 V to +4.0 V |
| Input: signal levels | | 3.3 V LVTTL |
| Output: available signal types | software programmable | Asynchronous Digital-Out, Synchronous Digital-Out, Trigger Output, Run, Arm, Sample Clock, Marker Output |
| Output: impedance | | 50 Ω |
| Output: signal levels | | 3.3 V LVTTL |
| Output: type | | 3.3V LVTTL, TTL compatible for high impedance loads |
| Output: drive strength | | Capable of driving 50 Ω loads, maximum drive strength ±48 mA |

## Connectors

| | | |
|---|---|---|
| Analog Inputs/Analog Outputs | SMA female (one for each single-ended input) | Cable-Type: Cab-3mA-xx-xx |
| Trigger 0 Input | SMA female | Cable-Type: Cab-3mA-xx-xx |
| Clock Input | SMA female | Cable-Type: Cab-3mA-xx-xx |
| Trigger 1 Input | MMCX female | Cable-Type: Cab-1m-xx-xx |
| Clock Output | MMCX female | Cable-Type: Cab-1m-xx-xx |
| Multi Purpose I/O | MMCX female (3 lines) | Cable-Type: Cab-1m-xx-xx |

## Environmental and Physical Details

| | | |
|---|---|---|
| Dimension (Single Card) | | 241 mm (¾ PCIe length) x 107 mm x 20 mm (single slot width) |
| Dimension (Card with option SH8tm installed) | | 241 mm (¾ PCIe length) x 107 mm x 40 mm (double slot width) |
| Dimension (Card with option SH8ex installed) | | 312 mm (full PCIe length) x 107 mm x 20 mm (single slot width) |
| Width (Standard and option SH8Ex) | | 1 slot |
| Width (option SH8tm installed) | | 2 slots |
| Weight (M4i.44xx series) | maximum | 290 g |
| Weight (M4i.22xx, M4i.66xx, M4i.77xx series) | maximum | 420 g |
| Weight (Option star-hub -sh8ex, -sh8tm) | including 8 sync cables | 130 g |
| Warm up time | | 10 minutes |
| Operating temperature | | 0°C to 50°C |
| Storage temperature | | -10°C to 70°C |
| Humidity | | 10% to 90% |

## PCI Express specific details

| | |
|---|---|
| PCIe slot type | x8 Generation 2 |
| PCIe slot compatibility (physical) | x8/x16 |
| PCIe slot compatibility (electrical) | x1, x4, x8, x16, Generation 1, Generation 2, Generation 3 |

## Certification, Compliance, 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 |

### Power Consumption

| | | PCI EXPRESS 3.3V | 12 V | Total |
|---|---|---|---|---|
| M4i.6620-x8 | Typical values: All channels activated, Sample rate: 625 MSps<br>Output signal: 31,25 MHz sine wave, Ouput level: +/- 1 V into 50 Ω load | 0.2 A | 2.5 A | 31 W |
| M4i.6621-x8 | | 0.2 A | 2.7 A | 33 W |
| M4i.6622-x8 | | 0.2 A | 3.0 A | 36 W |
| M4i.6620-x8 | Typical values: All channels activated, Sample rate: 625 MSps<br>Output signal: 31,25 MHz sine wave, Ouput level: +/- 2.5 V into 50 Ω load | 0.2 A | 2.6 A | 32 W |
| M4i.6621-x8 | | 0.2 A | 2.9 A | 35 W |
| M4i.6622-x8 | | 0.2 A | 3.3 A | 40 W |
| M4i.6630-x8 | Typical values: All channels activated, Sample rate: 1.25 GSps<br>Output signal: 31,25 MHz sine wave, Ouput level: +/- 1 V into 50 Ω load | 0.2 A | 2.7 A | 33 W |
| M4i.6631-x8 | | 0.2 A | 3.0 A | 36 W |
| M4i.6630-x8 | Typical values: All channels activated, Sample rate: 1.25 GSps<br>Output signal: 31,25 MHz sine wave, Ouput level: +/- 2.0 V into 50 Ω load | 0.2 A | 2.9 A | 35 W |
| M4i.6631-x8 | | 0.2 A | 3.3 A | 40 W |

### MTBF

| | |
|---|---|
| MTBF | TBD |

## Bandwidth and Slewrate

| | Filter | Output Amplitude | Mi4.6630-x8<br>M4i.6631-x8 | M4i.6620-x8<br>M4i.6621-x8<br>M4i.6622-x8 |
|---|---|---|---|---|
| Maximum Output Rate | | | 1.25 GS/s | 625 MS/s |
| -3d Bandwidth | no Filter | ±480 mV | 400 MHz | 200 MHz |
| -3d Bandwidth | no Filter | ±1000 mV | 320 MHz | 200 MHz |
| -3d Bandwidth | no Filter | ±2000 mV | 320 MHz | 200 MHz |
| -3d Bandwidth | Filter | all | 65 MHz | 65 MHz |
| Slewrate | no Filter | ±480 mV | 2.25 V/ns | 4.5 V/ns |

## Dynamic Parameters

| | M4i.6620-x8<br>M4i.6621-x8<br>M4i.6622-x8 | | | | | | |
|---|---|---|---|---|---|---|---|
| Test - Samplerate | 625 MS/s | | | 625 MS/s | | 625 MS/s | |
| Output Frequency | 10 MHz | | | 50 MHz | | 50 MHz | |
| Output Level in 50 Ω | ±480 mV | ±1000mV | ±2500mV | ±480 mV | ±2500mV | ±480 mV | ±2500mV |
| Used Filter | none | | | none | | Filter enabled | |
| NSD (typ) | -150 dBm/Hz | -149 dBm/Hz | -149 dBm/Hz | -150 dBm/Hz | -149 dBm/Hz | -150 dBm/Hz | -149 dBm/Hz |
| SNR (typ) | 70.7 dB | 72.4 dB | 63.1 dB | 65.3 dB | 64.4 dB | 67.5 dB | 69.4 dB |
| THD (typ) | -73.3 dB | -70.5 dB | -49.7 dB | -64.1 dB | -39.1 dB | -68.4 dB | -50.4 dB |
| SINAD (typ) | 69.0 dB | 67.7 dB | 49.5 dB | 61.6 dB | 39.1 dB | 64.9 dB | 50.3 dB |
| SFDR (typ), excl harm. | 98 dB | 98 dB | 99 dB | 86 dB | 76 dB | 88 dB | 89 dB |
| ENOB (SINAD) | 11.2 | 11.0 | 8.0 | 10.0 | 6.2 | 10.5 | 8.1 |
| ENOB (SNR) | 11.5 | 11.7 | 10.2 | 10.5 | 10.4 | 10.9 | 11.2 |

| | M4i.6630-x8<br>M4i.6631-x8 | | | | | | |
|---|---|---|---|---|---|---|---|
| Test - Samplerate | 1.25 GS/s | | | 1.25 GS/s | | 1.25 GS/s | |
| Output Frequency | 10 MHz | | | 50 MHz | | 50 MHz | |
| Output Level in 50 Ω | ±480 mV | ±1000mV | ±2000mV | ±480 mV | ±2000mV | ±480 mV | ±2000mV |
| Used Filter | none | | | none | | Filter enabled | |
| NSD (typ) | -150 dBm/Hz | -149 dBm/Hz | -149 dBm/Hz | -150 dBm/Hz | -149 dBm/Hz | -150 dBm/Hz | -149 dBm/Hz |
| SNR (typ) | 70.5 dB | 72.1 dB | 71.4 dB | 65.2 dB | 65.0 dB | 67.2 dB | 68.2 dB |
| THD (typ) | -74.5 dB | -73.5 dB | -59.1 dB | -60.9 dB | -43.9 dB | -67.9 dB | -63.1 dB |
| SINAD (typ) | 69.3 dB | 69.7 dB | 59 dB | 59.5 dB | 43.9 dB | 64.5 dB | 61.9 dB |
| SFDR (typ), excl harm. | 96 dB | 97 dB | 98 dB | 85 dB | 84 dB | 87 dB | 87 dB |
| ENOB (SINAD) | 11.2 | 11.2 | 9.5 | 9.6 | 6.9 | 10.4 | 10.0 |
| ENOB (SNR) | 11.5 | 11.5 | 11.5 | 10.5 | 10.5 | 10.9 | 11.0 |

THD and SFDR are measured at the given output level and 50 Ohm termination with a high resolution M3i.4860/M4i.4450-x8 data acquisition card and are calculated from the spectrum. Noise Spectral Density is measured with built-in calculation from an HP E4401B Spectrum Analyzer. All available D/A channels are activated for the tests. SNR and SFDR figures may differ depending on the quality of the used PC. NSD = Noise Spectral Density, THD = Total Harmonic Distortion, SFDR = Spurious Free Dynamic Range.

## Order Information

The card is delivered with 2 GSample on-board memory and supports standard replay, FIFO replay (streaming), Multiple Replay, Gated Replay, Continuous Replay (Loop), Single-Restart as well as Sequence. Operating system drivers for Windows/Linux 32 bit and 64 bit, examples for C/C++, LabVIEW (Windows), MATLAB (Windows and Linux), LabWindows/CVI, IVI, .NET, Delphi, Visual Basic, Python and a Base license of the oscilloscope software SBench 6 are included.

**Adapter cables are not included. Please order separately!**

### PCI Express x8

| Order no. | Bandwidth | Standard mem | 1 channel | 2 channels | 4 channels |
|---|---|---|---|---|---|
| M4i.6620-x8 | 200 MHz | 2 GSample | 625 MS/s | | |
| M4i.6621-x8 | 200 MHz | 2 GSample | 625 MS/s | 625 MS/s | |
| M4i.6622-x8 | 200 MHz | 2 GSample | 625 MS/s | 625 MS/s | 625 MS/s |
| M4i.6630-x8 | 400 MHz | 2 GSample | 1.25 GS/s | | |
| M4i.6631-x8 | 400 MHz | 2 GSample | 1.25 GS/s | 1.25 GS/s | |

### Options

| Order no. | Option |
|---|---|
| M4i.xxxx-SH8ex (1) | Synchronization Star-Hub for up to 8 cards (extension), only one slot width, extension of the card to full PCI Express length (312 mm). 8 synchronization cables included. |
| M4i.xxxx-SH8tm (1) | Synchronization Star-Hub for up to 8 cards (top mount), two slots width, top mounted on card. 8 synchronization cables included. |
| M4i-upgrade | Upgrade for M4i.xxxx: Later installation of option Star-Hub |
| SPc-RServer | Remote Server Software Package: LAN remote access with discovery function and remote driver access. Runs on Windows and Linux. |

### Standard Cables

| | | Order no. | | | | |
|---|---|---|---|---|---|---|
| for Connections | Length | to BNC male | to BNC female | to SMA male | to SMA female | to SMB female |
| Analog/Clock-In/Trig-In | 80 cm | Cab-3mA-9m-80 | Cab-3mA-9f-80 | | | |
| Analog/Clock-In/Trig-In | 200 cm | Cab-3mA-9m-200 | Cab-3mA-9f-200 | | | |
| Clk-Out/Trig-Out/Extra | 80 cm | Cab-1m-9m-80 | Cab-1m-9f-80 | Cab-1m-3mA-80 | Cab-1m-3fA-80 | Cab-1m-3f-80 |
| Clk-Out/Trig-Out/Extra | 200 cm | Cab-1m-9m-200 | Cab-1m-9f200 | Cab-1m-3mA-200 | Cab-1m-3fA-200 | Cab-1m-3f-200 |
| Information | The standard adapter cables are based on RG174 cables and have a nominal attenuation of 0.3 dB/m at 100 MHz and 0.5 dB/m at 250 MHz. For high speed signals we recommend the low loss cables series CHF | | | | | |

### Low Loss Cables

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

### Software SBench6

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

### Software Options

| Order no. | |
|---|---|
| Spc-RServer | Remote Server Software Package - LAN remote access for M2i/M3i/M4i/M4x cards |

(1) : Just one of the options can be installed on a card at a time.
(2) : Third party product with warranty differing from our export conditions. No volume rebate possible.

# Hardware Installation

## System Requirements

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

## Warnings

### ESD Precautions

The boards of the M4i.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 off any electrostatic electricity.**

### Cooling Precautions

The boards of the M4i.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.

**For all M4i cards it is absolutely mandatory 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 analog acquisition and generator boards of the M4i.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

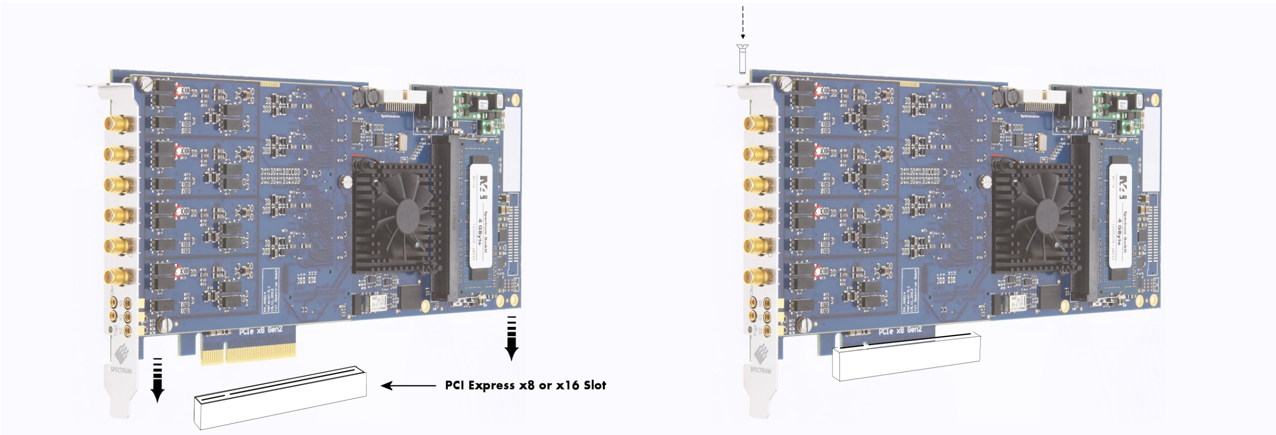
Before installing the board you first need to unscrew and remove the dedicated blind-bracket usually mounted to cover unused slots of your PC. Please keep the screw in reach to fasten your Spectrum card afterwards. All Spectrum M4i cards mechanically require one PCI Express x8 or x16 slot (electrically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

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

**Please be very careful when inserting the 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.

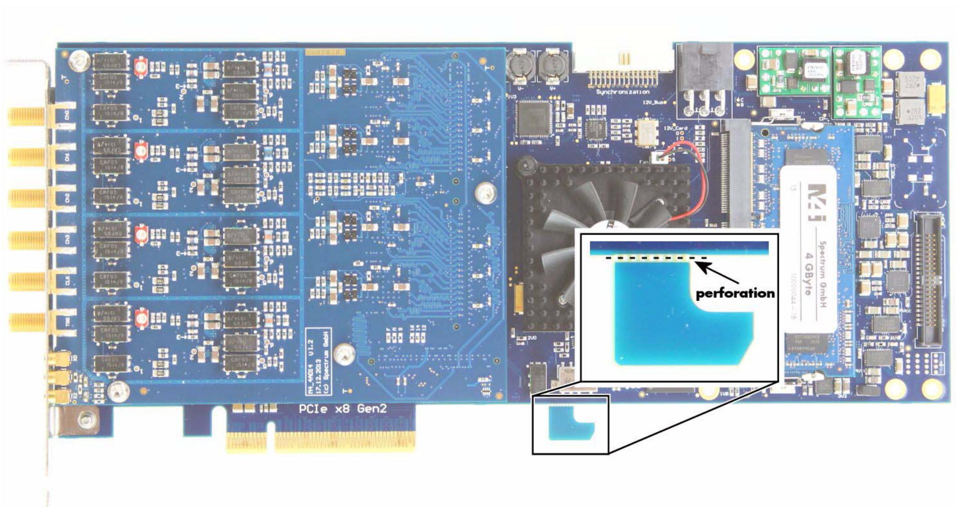
**Installing the M4i.xxxx-x8PCI Express card in a PCIe x8 or x16 slot**




## Additional notes on M4i cards with PCIe x16 slot retention

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

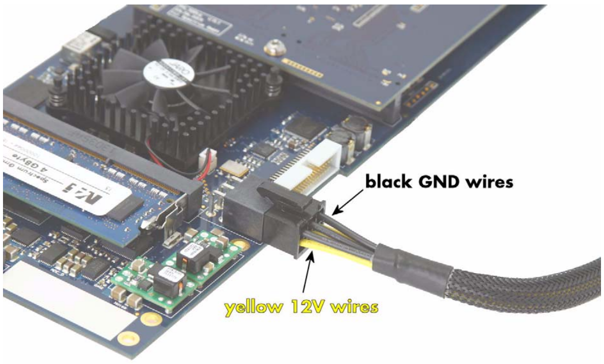
That allows the card to be additionally locked when being installed into a PCIe x16 slot.




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

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

## Providing additional power to M4i.xxxx-x8 cards




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

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

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

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

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

# 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/M3i or M4i series. That means that you can use the same driver for all cards of these families.

**With CD revision 3.00 (June 2010) the CD structure was revised and the root folder of the Windows drivers was changed from „spcm_driver" to „Driver". The screen shots shown in the installation chapters might still show the former version.**

# Windows XP 32/64 Bit

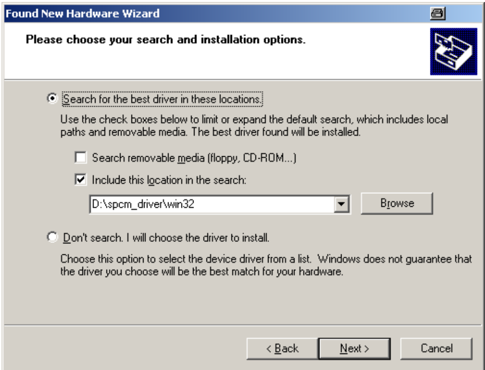
## Installation

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 \Driver\win32 for Windows XP 32 Bit or \Driver\win64 for Windows XP 64 Bit.



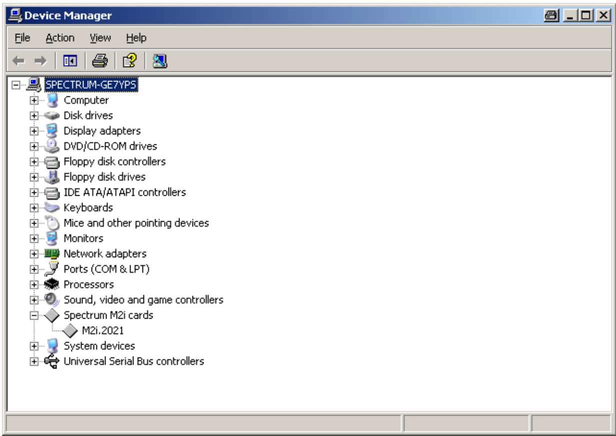

The hardware assistant shows you the exact board type that has been found like the M2i.2021 in the example.

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.

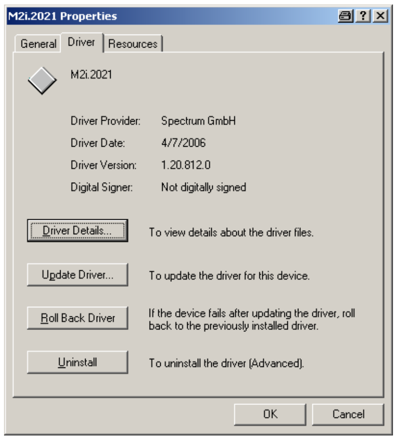

## Version control

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.

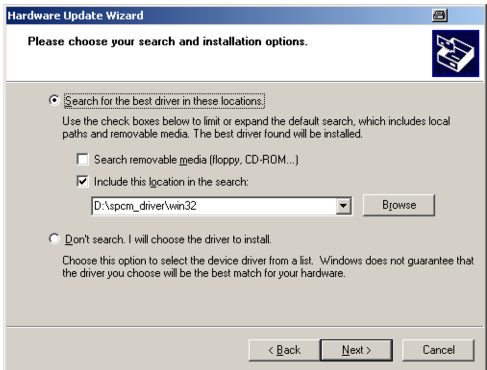



## Driver - Update

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 either the \Driver\win32 or \Driver\win64 path on the CD containing the new driver version.



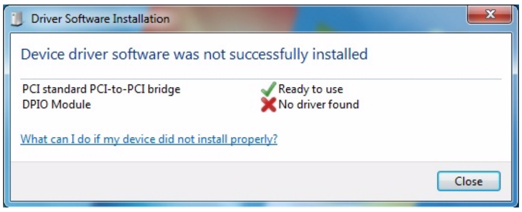

The new driver version can be used directly after installation without restarting the system. Please keep in mind to update the driver of all installed Spectrum boards.

# Windows 7, 32/64 Bit

## Installation

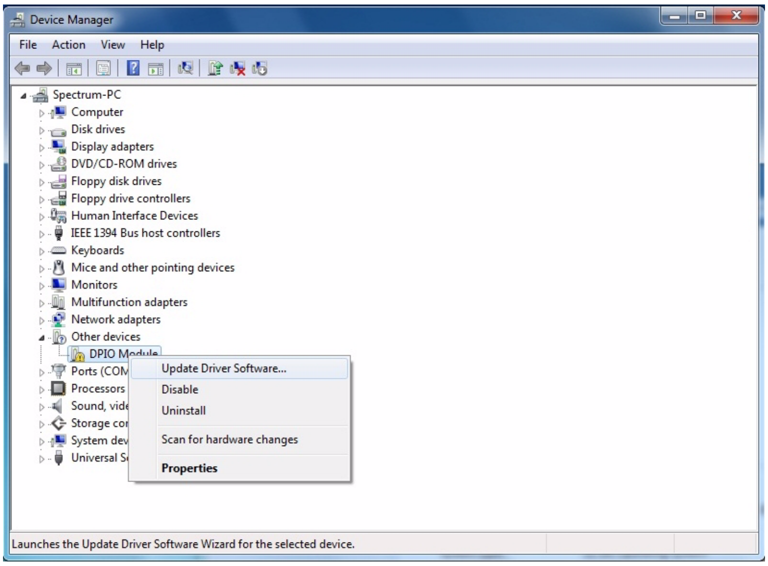
When installing the card in a Windows 7 system, it will be recognized automatically on the next start-up. The system tries at first to automatically search and install the drivers from the Microsoft homepage.

This mechanism will fail at first for the „DPIO Module" device, as shown on the right, because the Spectrum drivers are not available via Microsoft, so simply close the dialog.

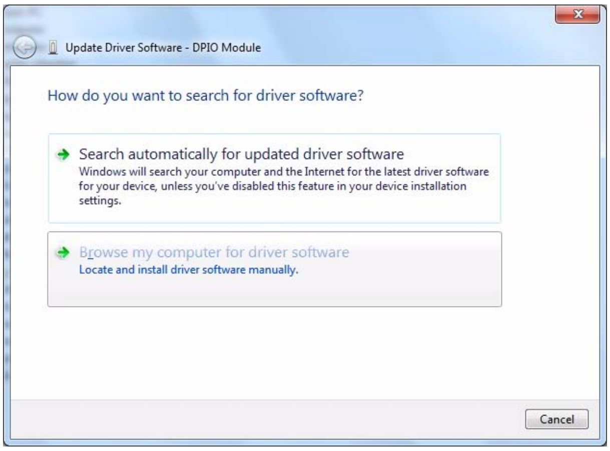



Afterwards open the device manager from the Windows control panel, as shown on the right.

Find the above mentioned „DPIO Module", right-click and select „Update Driver Software..."



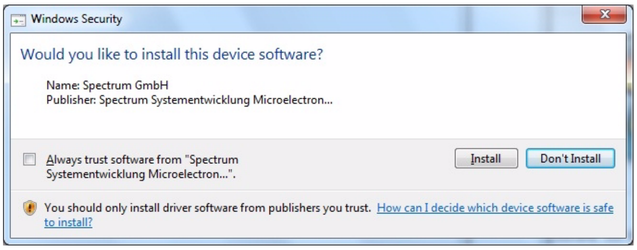

Do not let Windows 7 automatically search the for the best driver, because it will search the internet and not find a proper driver. Please take the option of browsing the computer manually for the driver software instead. Allow Windows 7 to search for the most suitable driver in a specific directory.




Now simply select the root folder of the CD that was delivered with the board as installation source and enable the „Include subfolders" option.

Alternatively you can browse to the installations folders. The driver files are located on CD in the directory
\Driver\win32 for Windows 7 32 Bit
or
\Driver\win64 for Windows 7 64 Bit.

On the upcoming Windows security dialog select install. To prevent Windows 7 to always ask this question for future updates, you can optionally select to always trust software from Spectrum.



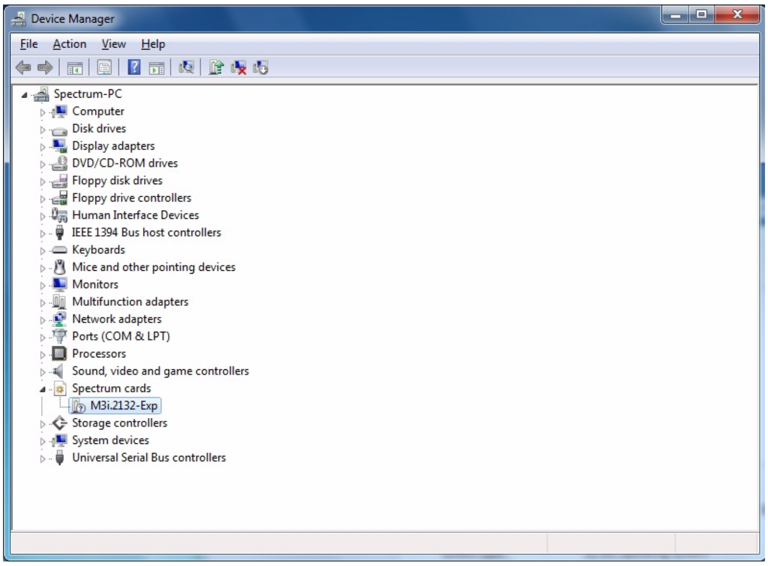

The hardware assistant then shows you the exact board type that has been found like the M3i.2132 in the example.

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.

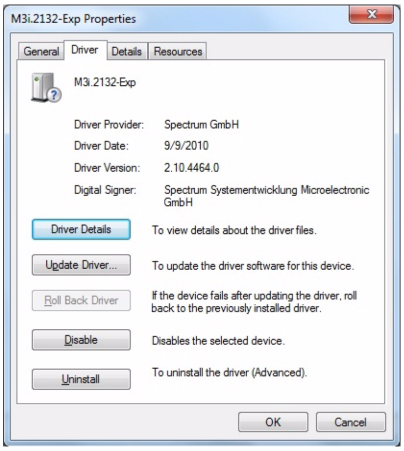

## Version control

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 7 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

The driver update under Windows 7 is exact the same procedure as the initial installation. Please follow the steps above, starting from the device manager, select the Spectrum card to be updated, right-click and select „Update Driver Software…“ and follow the steps above.

# Linux

## Overview

The Spectrum M2i/M3i/M4i cards and digitizerNETBOX/generatorNETBOX products are delivered with Linux drivers suitable for Linux installations based on kernel 2.4, 2.6, 3.x or 4.x, single processor (non-SMP) and SMP systems, 32 bit and 64 bit systems. As each Linux distribution contains different kernel versions and different system setup it is nearly every case necessary to have a directly matching kernel driver for card level products to run it on a specific system. For digitizerNETBOX/generatorNETBOX products the library is suffcient and no kernel driver has to be installed.

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

The Standard delivery contains the 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 9.3 | 2.6.11 | single and smp | 32 bit | Fedora Core 3 | 2.6.9 | single and smp | 32 bit |
| Suse 10.0 | 2.6.13 | single only | 32 bit and 64 bit | Fedora Core 4 | 2.6.11 | single and smp | 32 bit |
| Suse 10.1 | 2.6.16 | single only | 32 bit and 64 bit | Fedora Core 5 | 2.6.15 | single and smp | 32 bit and 64 bit |
| Suse 10.2 | 2.6.18 | single and smp | 32 bit and 64 bit | Fedora Core 6 | 2.6.18 | single and smp | 32 bit and 64 bit |
| Suse 10.3 | 2.6.22 | single and smp | 32 bit and 64 bit | Fedora Core 7 | 2.6.21 | single and smp | 32 bit and 64 bit |
| Suse 11.0 | 2.6.25 | single and smp | 32 bit and 64 bit | Fedora 8 | 2.6.23 | single and smp | 32 bit and 64 bit |
| Suse 11.1 | 2.6.27 | single and smp | 32 bit and 64 bit | Fedora 9 | 2.6.25 | single and smp | 32 bit and 64 bit |
| Suse 11.2 | 2.6.31 | single and smp | 32 bit and 64 bit | Fedora 10 | 2.6.27 | single and smp | 32 bit and 64 bit |
| Suse 11.3 | 2.6.34 | single and smp | 32 bit and 64 bit | Fedora 11 | 2.6.29 | single and smp | 32 bit and 64 bit |
| Suse 11.4 | 2.6.38 | single and smp | 32 bit and 64 bit | Fedora 12 | 2.6.31 | single and smp | 32 bit and 64 bit |
| Suse 12.1 | 3.1 | single and smp | 32 bit and 64 bit | Fedora 13 | 2.6.33.3 | single and smp | 32 bit and 64 bit |
| Suse 12.2 | 3.4.6 | single and smp | 32 bit and 64 bit | Fedora 14 | 2.6.35.6 | single and smp | 32 bit and 64 bit |
| Suse 12.3 | 3.7.0 | single and smp | 32 bit and 64 bit | Fedora 15 | 2.6.38.6 | single and smp | 32 bit and 64 bit |
| Suse 13.1 | 3.11.6 | single and smp | 32 bit and 64 bit | Fedora 16 | 3.1 | single and smp | 32 bit and 64 bit |
| Suse 13.2 | 3.16.6 | single and smp | 32 bit and 64 bit | Fedora 17 | 3.3.4 | single and smp | 32 bit and 64 bit |
| Suse 42.1 | 4.1.12 | single and smp | 64 bit | Fedora 18 | 3.6.10 | single and smp | 32 bit and 64 bit |
| | | | | Fedora 19 | 3.9.5 | single and smp | 32 bit and 64 bit |
| Debian Sarge | 2.4.27 | single | 32 bit | Fedora 20 | 3.11.10 | single and smp | 32 bit and 64 bit |
| Debian Sarge | 2.6.8 | single | 32 bit | Fedora 21 | 3.17.4 | single and smp | 32 bit and 64 bit |
| Debian Etch | 2.6.18 | single and smp | 32 bit and 64 bit | Fedora 22 | 4.0.4 | single and smp | 32 bit and 64 bit |
| Debian Lenny | 2.6.26 | single and smp | 32 bit and 64 bit | Fedora 23 | 4.2.3 | single and smp | 32 bit and 64 bit |
| Debian Squeeze | 2.6.32 | single and smp | 32 bit and 64 bit | | | | |
| Debian Wheezy | 3.2.41 | single and smp | 32 bit and 64 bit | Ubuntu 12.04 LTS | 3.2 | single and smp | 32 bit and 64 bit |
| Debian Jessie | 3.16.7 | single and smp | 32 bit and 64 bit | Ubuntu 14.04 LTS | 3.15.0 | single and smp | 32 bit and 64 bit |
| | | | | Ubuntu 16.04 LTS | 4.4.0 | single and smp | 32 bit and 64 bit |

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

## 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.

**This installation is only needed if you are operating locally installed cards. For remotely installed cards or for digitizerNETBOX/generatorNETBOX products it is only necessary to install the libraries as explained further below.**

### Login as root

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

### Call the install.sh <install_path> script

This script will 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
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 gets 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.

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

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

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

```
sh install_libonly.sh
```

## Library only

The kernel driver module only contains the basic hardware functions that are necessary to access locally installed card level products. The main part of the driver is located inside a dynamically loadable library that is delivered with the driver. This library is available in 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 kernel driver install script for card level products. The library is renamed for easy access to libspcm_linux.so.

For digitizerNETBOX products the library is installed with a separate install script:

```
sh install_libonly.sh
```

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

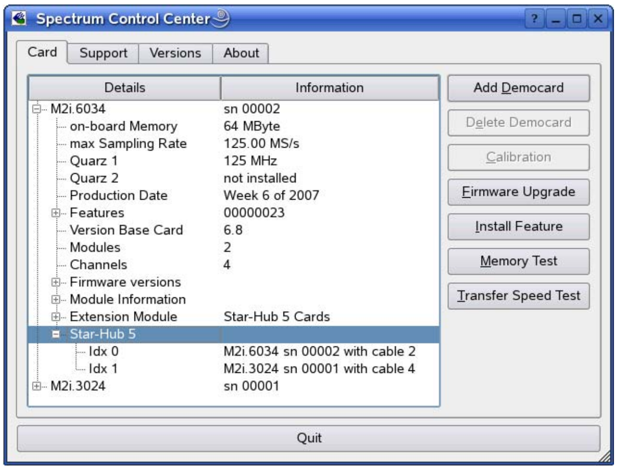
```
gcc -o test_prg -lspcm_linux test.cpp
```

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

## Control Center

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

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




### Installation

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

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

### Manual Installation

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

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

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

**Troubleshooting**

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

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

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

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

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

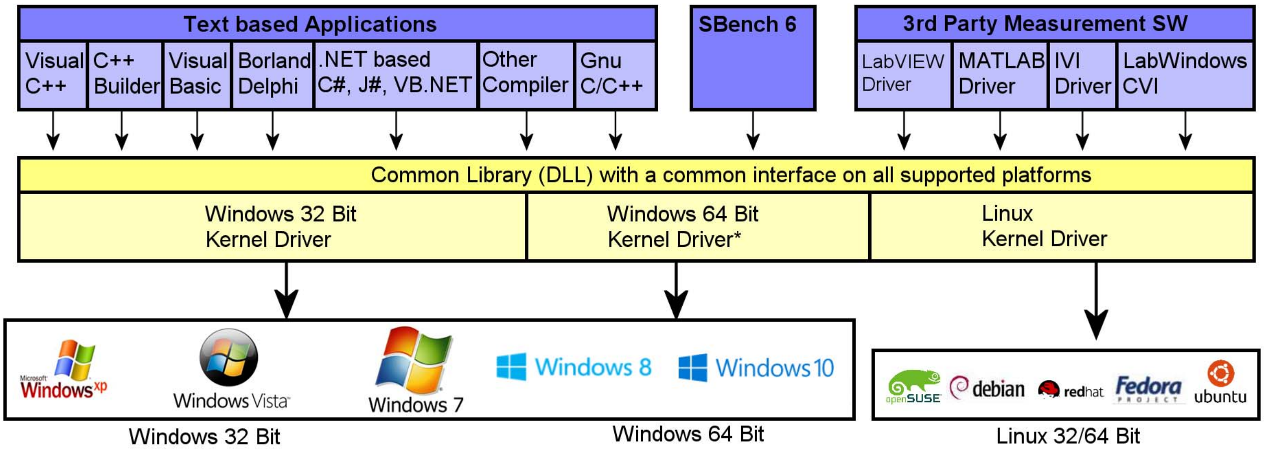
If the libspcm_linux.so is quoted as missing please make sure that you have installed the card driver properly before. If any other library is stated as missing please install the matching package of your distribution.

# Software

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

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

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

## Card Control Center


A special card control center is available on CD and from the internet for all Spectrum M2i/M3i/M4i cards and for all digitizerNETBOX/generatorNETBOX products. Windows user find the Control Center installer on the CD under „Install\win\spcmcontrol_install.exe“.

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

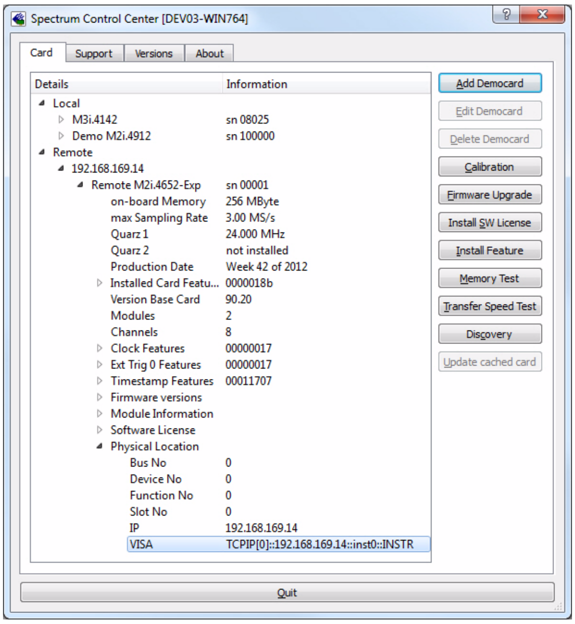
When using a digitizerNETBOX/generatorNETBOX the Card Control Center installers for Windows and Linux are also directly available from the integrated webserver.

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

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

## Discovery of Remote Cards and digitizerNETBOX/generatorNETBOX products

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




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

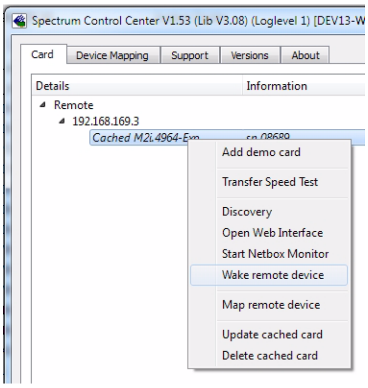
Execute the Discovery function by pressing the „Discovery" button. There is no progress window shown. After the discovery function has been executed the remotely found Spectrum products are listed under the node Remote as separate card level products. Inhere you find all hardware information as shown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

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

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

## Wake On LAN of digitizerNETBOX/generatorNETBOX

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




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

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

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

```
spcm_dwSetParam_64 (NULL, SPC_NETBOX_WAKEONLAN, 0x00032d2048ec);
```

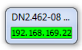
## Netbox Monitor

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

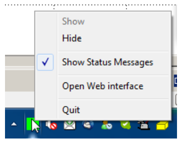
- Starting manually from the Spectrum Control Center using the context menu as shown above
- Starting from command line. The Netbox Monitor program is automatically installed together with the Spectrum Control Center and is located in the selected install folder. Using the command line tool one can place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

```
NetboxMonitor 192.168.169.22
```

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



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




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

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

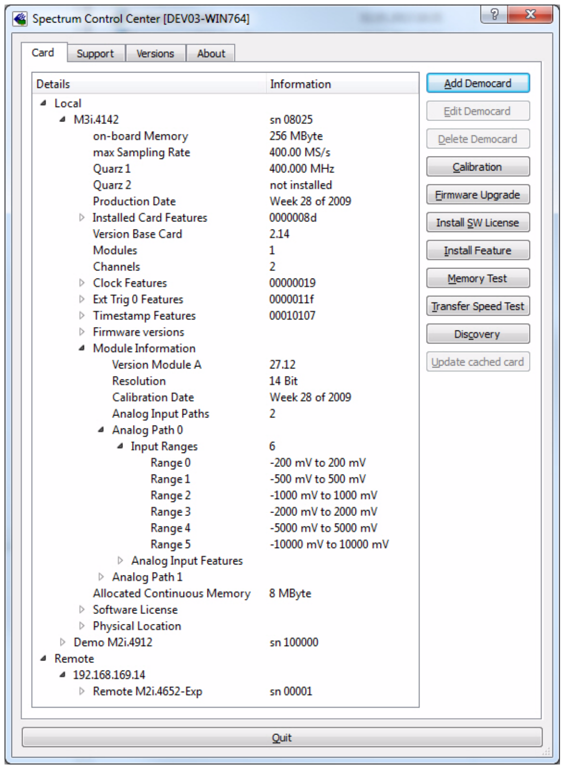
The status colors:

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

## Hardware information

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

- Basic information as the type of card, the production date and its serial number, as well as the installed memory, the hardware revision of the base card, the number of available channels and installed acquisition modules
- Information about the maximum sampling clock and the available quartz clock sources.
- The installed features/options in a sub-tree. The shown card is equipped 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.



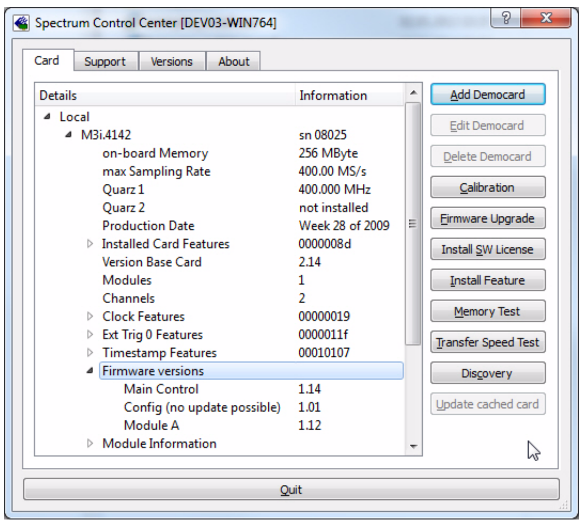

## Firmware information

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.

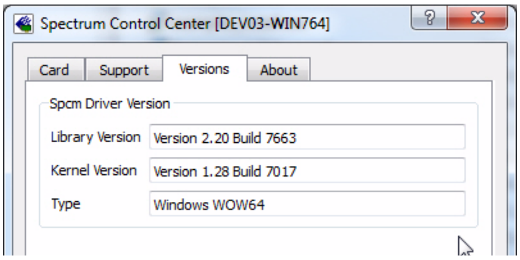
Nearly all of the components firmware can be updated by software. The only exception is the configuration device, which only can receive a factory update.

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.

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



## Driver information




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

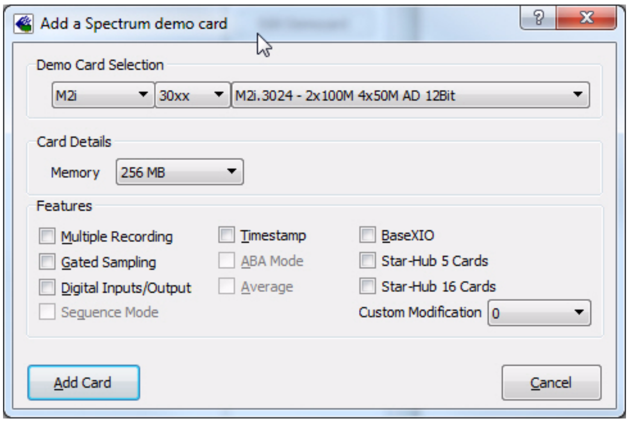
The information on the driver is available through a dedicated tab, as the picture is showing in the example.

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



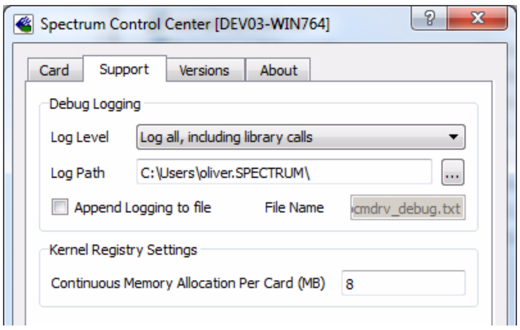

With the help of the card control center one can install demo cards in the system. A demo card is simulated by the Spectrum driver including data production for acquisition cards. As the demo card is simulated on the lowest driver level all software can be tested including SBench, own applications and 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 behavior of a card, especially not any timing concerning trigger, recording length or FIFO mode notification. The demo card will calculate data every time directly after been called and give it to the user application without any more delay. As the calculation routine isn't speed optimized, generating demo data may take more time than acquiring real data and transferring them to the host PC.

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

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

## 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.

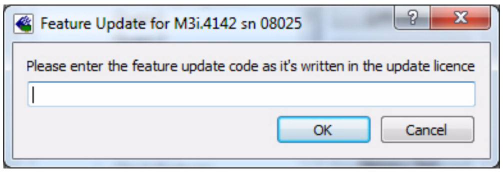
Therefore the card control center provides an easy way to gather all that information automatically.

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.**

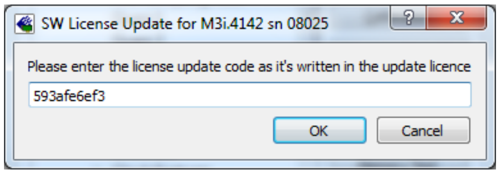
## Feature upgrade




All optional features of the M2i/M3i/M4i 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.

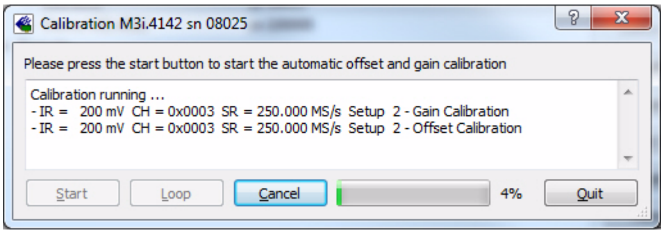
For details on the available options and prices please contact your local Spectrum distributor.

## Software License upgrade



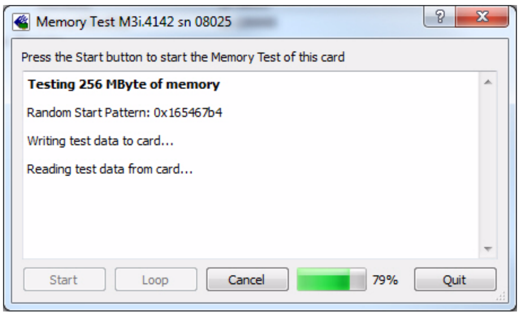

The software license for SBench 6 Professional is installed on the hardware. If ordering a software license for a card that has already been delivered you will get an upgrade code to install that software license. The upgrade code will only match for that particular card with the serial number given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

## 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

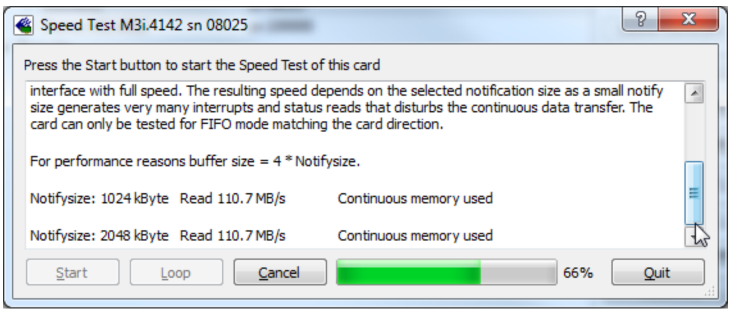



The complete on-board memory of the Spectrum M2i/M3i/M4i cards can be tested by the memory test included with the card control center.

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.

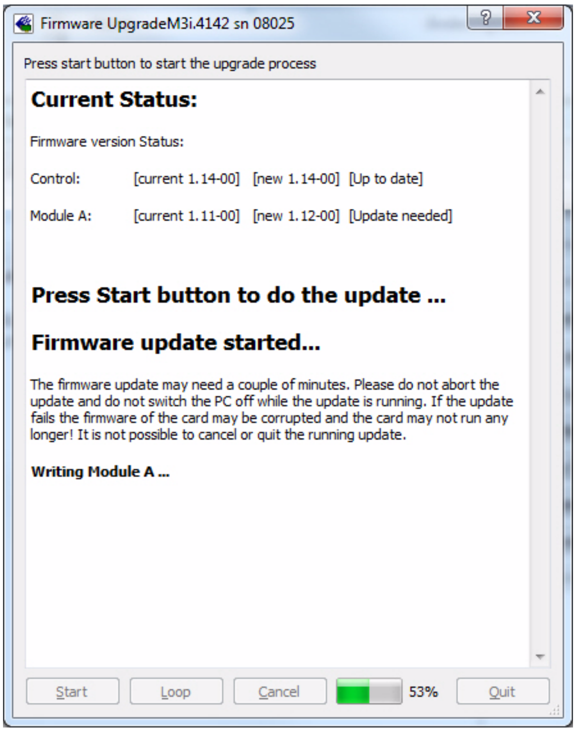
The results are given in MB/s meaning MByte per second. To estimate whether a desired acquisition speed is possible to reach one has to calculate the transfer speed in bytes. There are a few things that has to be put into the calculation:

- 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.

## 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/M3i/M4i card:

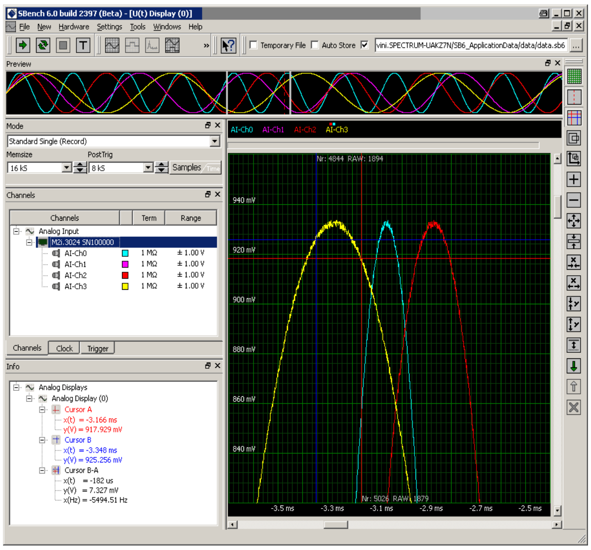
- Download the latest software driver for your operating system provided on the Spectrum homepage.
- Install the new driver as described in the driver install section of your hardware manual provided with the card. All manuals can also be found on the Spectrum homepage in the literature download section.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

The 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.

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

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

# Accessing the hardware with SBench 6



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

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

On the CD you'll find an install version of SBench 6 in the directory /Install/SBench6. The current version of SBench 6 is available free of charge directly from the Spectrum website http://www.spectrum-instrumentation.com. Please go to the download section and get the latest version there. If using the digitizerNETBOX/generatorNETBOX, a SBench 6 version is also available on the webpages of the digitizerNETBOX/generatorNETBOX.

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

# C/C++ Driver Interface

C/C++ is the main programming language for which the drivers have been designed for. Therefore the interface to C/C++ is the best match. All the small examples of the manual showing different parts of the hardware programming are done with C. As the libraries offer a standard interface it is easy to access the libraries also with other programming languages like Delphi 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/c_header. Please don't change them in any way because they are updated with each new driver version to include the new registers and new functionality.

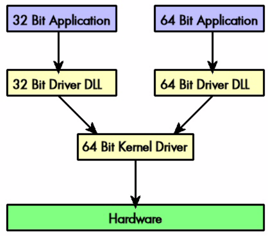
| | |
|---|---|
| dlltyp.h | Includes the platform specific definitions for data types and function declarations. All data types are based on this definitions. The use of this type definition file allows the use of examples and programs on different platforms without changes to the program source. The 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 /examples/c_cpp/c_header. Please include the library file in your Visual C++ project as shown in the examples. All functions described below are now available in your program.

### Examples

Examples can be found on CD in the path /examples/c_cpp. This directory includes a number of different examples that can be used with any card of the same type (e.g. A/D acquisition cards, D/A acquisition cards). You may use these examples as a base for own programming and modify them as you like. The example directories contain a running workspace file for Microsoft Visual C++ 6.0 (*.dsw) as well as project files for Microsoft Visual Studio 2005 (*.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 /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 /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 32 bit 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 /examples/c_cpp/dll_loading that shows the process.

Example of function loading:

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

# Other Windows C/C++ compilers 64 Bit

## Include Driver

To access the driver, the driver functions must be loaded from 64 bit 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 /examples/c_cpp/dll_loading that shows the process for 32 bit environments. The only line that needs to be modified is the one loading the DLL:

Example of function loading:

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

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

## Notation of variables and functions

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

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

## Function spcm_hOpen

This function initializes and opens an installed card supporting the new SpcM driver interface. At the time of printing this manual this are all cards of the M2i/M3i/M4i cards and digitizerNETBOX devices. 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 for local installed cards

```
drv_handle  hDrv;                   // returns the handle to the opended driver or NULL in case of error
hDrv = spcm_hOpen ("/dev/spcm0");   // string to the driver to open
if (!hDrv)
    printf (“open of driver failed\n”);
```

Example for digitizerNETBOX and remote installed cards

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

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

## Function spcm_vClose

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

Function spcm_vClose:

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

Example:

```
spcm_vClose (hDrv);
```

## Function spcm_dwSetParam

All hardware settings are based on software registers that can be set by one of the functions spcm_dwSetParam. These functions 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) != ERR_OK)
    printf ("Error when setting memory size\n");
```

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

## Function spcm_dwGetParam

All hardware settings are based on software registers that can be read by one of the functions spcm_dwGetParam. These functions 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 are used to call them. As some of the registers can exceed the 32 bit integer range (like memory size or post trigger) it is recommended to use the _i64 function to access these registers. However as there are some programs or compilers that don't support 64 bit integer variables there are two functions that are limited to 32 bit integer variables. In case that you do not access registers that exceed 32 bit integer please use the _i32 function. In case that you access a register which exceeds 64 bit value please use the _i64m calling convention. Inhere the 64 bit value is 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 in bytes, in case one has been allocated. If no buffer has been allocated the function returns a size of zero and a NULL pointer. You may use this buffer for data transfers. As the buffer is continuously allocated in memory the data transfer will speed up by 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 needs 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, 0, 16384) != ERR_OK)
    printf ("DefTransfer failed\n");
```

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

### Function spcm_dwInvalidateBuf

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

Function spcm_dwInvalidateBuf

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

### Function spcm_dwGetErrorInfo

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

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

Function spcm_dwGetErrorInfo

```
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:

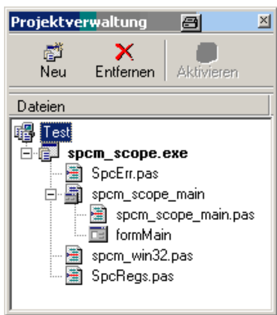
```
const SPC_M2CMD                          = 100;              { write a command }
const     M2CMD_CARD_RESET               = $00000001;        { hardware reset    }
const     M2CMD_CARD_WRITESETUP          = $00000002;        { write setup only }
const     M2CMD_CARD_START               = $00000004;        { 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 returned by the driver.

### Including the driver files

To use the driver function and all the defined constants it is necessary to include the files into the project as shown in the picture on the right. The project overview is taken from one of the examples delivered on 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 /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:

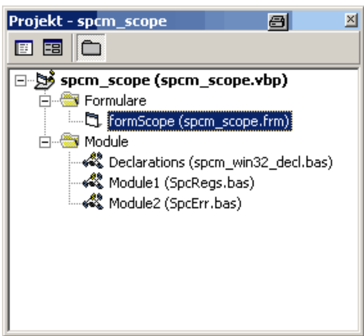
```
Public Const SPC_M2CMD = 100                               ' write a command
Public Const M2CMD_CARD_RESET = &H1&                       ' hardware reset
Public Const M2CMD_CARD_WRITESETUP = &H2&                  ' write setup only
Public Const M2CMD_CARD_START = &H4&                       ' start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

### file SpcErr.bas

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

### Including the driver files

To use the driver function and all the defined constants it is necessary to include the files into the project as shown in the picture on the right. The project overview is taken from one of the examples delivered on CD.



## Examples

Examples for Visual Basic can be found on CD in the directory /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);
```

Example for digitizerNETBOX and remotely installed cards:

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

## Using Managed C++/CLI

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

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

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

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

Example for digitizerNETBOX and remotely installed cards:

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

## Using VB.NET

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

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

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

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

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

Example for digitizerNETBOX and remotely installed cards:

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

## Using J#

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

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

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

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

Example for digitizerNETBOX and remotely installed cards:

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

# Python Programming Interface and Examples

## Driver interface

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

### file pyspcm.py

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

```
# ----- Windows -----
spcmDll = windll.LoadLibrary ("c:\\windows\\system32\\spcm_win32.dll")

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

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

# load spcm_dwGetErrorInfo
spcm_dwGetErrorInfo_i32 = getattr (spcmDll, "_spcm_dwGetErrorInfo_i32@16")
spcm_dwGetErrorInfo_i32.argtype = [drv_handle, ptr32, ptr32, c_char_p]
spcm_dwGetErrorInfo_i32.restype = uint32

# load spcm_dwGetParam_i32
spcm_dwGetParam_i32 = getattr (spcmDll, "_spcm_dwGetParam_i32@12")
spcm_dwGetParam_i32.argtype = [drv_handle, int32, ptr32]
spcm_dwGetParam_i32.restype = uint32

# load spcm_dwGetParam_i64
spcm_dwGetParam_i64 = getattr (spcmDll, "_spcm_dwGetParam_i64@12")
spcm_dwGetParam_i64.argtype = [drv_handle, int32, ptr64]
spcm_dwGetParam_i64.restype = uint32

# load spcm_dwSetParam_i32
spcm_dwSetParam_i32 = getattr (spcmDll, "_spcm_dwSetParam_i32@12")
spcm_dwSetParam_i32.argtype = [drv_handle, int32, int32]
spcm_dwSetParam_i32.restype = uint32

# load spcm_dwSetParam_i64
spcm_dwSetParam_i64 = getattr (spcmDll, "_spcm_dwSetParam_i64@16")
spcm_dwSetParam_i64.argtype = [drv_handle, int32, int64]
spcm_dwSetParam_i64.restype = uint32

# load spcm_dwSetParam_i64m
spcm_dwSetParam_i64m = getattr (spcmDll, "_spcm_dwSetParam_i64m@16")
spcm_dwSetParam_i64m.argtype = [drv_handle, int32, int32, int32]
spcm_dwSetParam_i64m.restype = uint32

# load spcm_dwDefTransfer_i64
spcm_dwDefTransfer_i64 = getattr (spcmDll, "_spcm_dwDefTransfer_i64@36")
spcm_dwDefTransfer_i64.argtype = [drv_handle, uint32, uint32, uint32, c_void_p, uint64, uint64]
spcm_dwDefTransfer_i64.restype = uint32

spcm_dwInvalidateBuf = getattr (spcmDll, "_spcm_dwInvalidateBuf@8")
spcm_dwInvalidateBuf.argtype = [drv_handle, uint32]
spcm_dwInvalidateBuf.restype = uint32


# ----- Linux -----
# use cdll because all driver access functions use cdecl calling convention under linux
spcmDll = cdll.LoadLibrary ("libspcm_linux.so")

# the loading of the driver access functions is similar to windows:

# load spcm_hOpen
spcm_hOpen = getattr (spcmDll, "spcm_hOpen")
spcm_hOpen.argtype = [c_char_p]
spcm_hOpen.restype = drv_handle

# ...
```

**file regs.py**

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

```
SPC_M2CMD = 100l                                     # write a command
M2CMD_CARD_RESET = 0x00000001l                       # hardware reset
M2CMD_CARD_WRITESETUP = 0x00000002l                  # write setup only
M2CMD_CARD_START = 0x00000004l                       # start of card (including writesetup)
M2CMD_CARD_ENABLETRIGGER = 0x00000008l               # enable trigger engine
...
```

**file spcerr.py**

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

## Examples

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

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

# Programming the Board

## Overview

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

## Register tables

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

The name of the software register as found in the regs.h file. Could directly be used by C/C++, Delphi and Basic compiler

The decimal value of the software register. Also found in the regs.h file. This value must be used with all programs or compilers that cannot use the header file directly.

Describes whether the register can be read (r) and/or written (w).

Short description of the functionality of the register. A more detailed description is found above or below the register tables.

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

Any constants that can be used to program the register directly are shown inserted beneath the register table.

The decimal or hexadecimal value of the constant, also found in the regs.h file. Hexadecimal values are indicated with an „h“ at the end. This value must be used with all programs or compilers that cannot use the header file directly.

Short description of the use of this constant.

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

## Programming examples

In this manual a lot of programming examples are used to give you an impression on how the actual mentioned registers can be set within your own program. All of the examples are located in a separated 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 card type
    printf (“Found card M2i/M3i/M4i.%04x in the system\n”, lCardType & TYP_VERSIONMASK);
    spcm_vClose (hDrv);

    return 0;
    }
```

# Initialization

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

Open/Close Example

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

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

... do any work with the driver

spcm_vClose (hDrv);
return 0;
```

# Initialization of Remote Products

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

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

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

...
```

Multiple cards are opened by indexing the remote card number:

```
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST0::INSTR");      // Opens the remote board #0
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST1::INSTR");      // Opens the remote board #1
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INST2:.INSTR");      // Opens the remote board #2
hDrv = spcm_hOpen ("TCPIP::192.168.169.14::INSTR");             // Opens the remote board #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_i64 (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 its own error handler it may be interesting to just read out the error generating register and value.

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

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

spcm_dwSetParam_i64 (hDrv, SPC_SAMPLERATE, 1000000);                    // correct command
spcm_dwSetParam_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 its own card type constant defined in regs.h. Please note that when reading the card information as a hex value, the lower word shows the digits of the card name while the upper word is a indication for the used bus type.

.

| Card type | Card type as defined in regs.h | Value hexadecimal | Value decimal | | Card type | Card type as defined in regs.h | Value hexadecimal | Value decimal |
|---|---|---|---|---|---|---|---|---|
| M4i.6620-x8 | TYP_M4I6620_X8 | 76620h | 484896 | | | | | |
| M4i.6620-x8 | TYP_M4I6621_X8 | 76621h | 484897 | | | | | |
| M4i.6620-x8 | TYP_M4I6622_X8 | 76622h | 484898 | | | | | |

## Hardware and PCB version

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

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

If your board has 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 version, the lower 16 bit show the firmware version. |
| SPC_EXTPCBVERSION | 2017 | read | Extension module PCB version: the lower 16 bit are divided into two 8 bit values containing pre/post decimal point version information. For example a lower 16 bit value of 0106h represents a PCB version V1.6. The upper 16 bit are always zero. |

## 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 (analog cards only)

This register informs you about the date of the last factory calibration. When receiving a new card this date is similar to the delivery date when the production calibration is done. When returning the card to calibration this information is updated. This date is not updated when 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. In the table below you will find every feature that may be installed on a M2i/M3i/M4i card. Please refer to the ordering information to see which of these features are available for your card series.

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_PCIFEATURES | 2120 | read | PCI feature register. Holds the installed features and options as a bitfield. The read value must be masked out with one of the masks below to get information about one certain feature. |
| SPCM_FEAT_MULTI | 1h | | Is set if the feature Multiple Recording / Multiple Replay is available. |
| SPCM_FEAT_GATE | 2h | | Is set if the feature Gated Sampling / Gated Replay is available. |
| SPCM_FEAT_DIGITAL | 4h | | Is set if the feature Digital Inputs / Digital Outputs is available. |
| SPCM_FEAT_TIMESTAMP | 8h | | Is set if the feature Timestamp is available. |
| SPCM_FEAT_STARHUB8_EXTM | 20h | | Is set on the card, that carries the star-hub extension or piggy-back module for synchronizing up to 8 cards (M4i) |
| SPCM_FEAT_STARHUB4 | 20h | | Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 4 cards (M3i) |
| SPCM_FEAT_STARHUB5 | 20h | | Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 5 cards (M2i) |
| SPCM_FEAT_STARHUB8 | 40h | | Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 8 cards (M3i) |
| SPCM_FEAT_STARHUB16 | 40h | | Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 16 cards (M2i) |
| SPCM_FEAT_ABA | 80h | | Is set if the feature ABA mode is available. |
| SPCM_FEAT_BASEXIO | 100h | | Is set if the extra BaseXIO option is installed. The lines can be used for asynchronous digital I/O, extra trigger or timestamp reference signal input |
| SPCM_FEAT_AMPLIFIER_10V | 200h | | Arbitrary Waveform Generators only: card has additional set of calibration values for amplifier card |
| SPCM_FEAT_STARHUBSYSMASTER | 400h | | Is set in the card that carries a System Star-Hub Master card to connect multiple systems (M2i) |
| SPCM_FEAT_DIFFMODE | 800h | | M2i.30xx series only: card has option -diff installed for combining two SE channels to one differential channel |
| SPCM_FEAT_SEQUENCE | 1000h | | Only available for output cards or I/O cards: Replay sequence mode available. |
| SPCM_FEAT_AMPMODULE_10V | 2000h | | Is set on the card that has a special amplifier module for mounted (M2i.60xx/61xx only) |
| SPCM_FEAT_STARHUBSYSSLAVE | 4000h | | Is set in the card that carries a System Star-Hub Slave module to connect with System Star-Hub master systems (M2i) |
| SPCM_FEAT_NETBOX | 8000h | | The card is physically mounted within a digitizerNETBOX. |
| SPCM_FEAT_REMOTESERVER | 10000h | | Support for the Spectrum Remote Server option is installed on this card. |
| SPCM_FEAT_CUSTOMMOD_MASK | F0000000h | | The upper 4 bit of the feature register is used to mark special custom modifications. This is only used if the card has been specially customized. Please refer to the extra documentation for the meaning of the custom modification mark. |

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

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

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

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

### Installed extended Options and Features

Starting with the cards of the M4i series, some cards can optionally have advanced features installed. This can be read out with with the following register:

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_PCIEXTFEATURES | | 2121 | read | PCI extended feature register. Holds the installed extended features and options as a bitfield. The read value must be masked out with one of the masks below to get information about one certain feature. |
| | SPCM_FEAT_EXTFW_SEGSTAT | 1h | Is set if the firmware option „Block Statistics“ is installed on the board, which allows certain statistics to be on-board calculated for data being recorded in segmented memory modes, such as Multiple Recording or ABA. | |
| | SPCM_FEAT_EXTFW_SEGAVERAGE | 2h | Is set if the firmware option „Block Average“ is installed on the board, which allows on-board hardware averaging of data being recorded in segmented memory modes, such as Multiple Recording or ABA. | |

## Miscellaneous Card Information

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

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_MIINST_MODULES | 1100 | read | Number of the installed front-end modules on the card. |
| SPC_MIINST_CHPERMODULE | 1110 | read | Number of channels installed on one front-end module. |
| SPC_MIINST_BYTESPERSAMPLE | 1120 | read | Number of bytes used in memory by one sample. |
| SPC_MIINST_BITSPERSAMPLE | 1125 | read | Resolution of the samples in bits. |
| SPC_MIINST_MAXADCVALUE | 1126 | read | Decimal code of the full scale value. |
| SPC_MIINST_MINEXTCLOCK | 1145 | read | Minimum external clock that can be fed in for direct external clock (if available for card model). |
| SPC_MIINST_MAXEXTCLOCK | 1146 | read | Maximum external clock that can be fed in for direct external clock (if available for card model). |
| SPC_MIINST_MINEXTREFCLOCK | 1148 | read | Minimum external clock that can be fed inas a reference clock. |
| SPC_MIINST_MAXEXTREFCLOCK | 1149 | read | Maximum external clock that can be fed inas a reference clock. |
| SPC_MIINST_ISDEMOCARD | 1175 | read | Returns a value other than zero, if the card is a demo 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 doesn'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_LINUX32 | 1 | Linux 32bit driver is used | |
| | DRVTYP_WDM32 | 4 | Windows WDM 32bit driver is used (XP/Vista/Windows 7/Windows 8/Windows 10). | |
| | DRVTYP_WDM64 | 5 | Windows WDM 64bit driver is used by 64bit application (XP64/Vista/Windows 7/Windows 8/Windows 10). | |
| | DRVTYP_WOW64 | 6 | Windows WDM 64bit driver is used by 32bit application (XP64/Vista/Windows 7/Windows 8/ Windows 10). | |
| | DRVTYP_LINUX64 | 7 | Linux 64bit driver is used | |

### Driver version

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

| 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.

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

This small program will generate an output like this:

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

# 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 Outputs

## 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 1 channels.

| Channels to activate | | | |
|---|---|---|---|
| Ch0 | Values to program | Value as hex | Value as decimal |
| X | CHANNEL0 | 1h | 1 |

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

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

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

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

## Important note on 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 outputs

## Output Enable

The output of each channel can be completely disabled by software command at any time. Disabling the output will cut off the amplifier from the connector with the help of a Relay. Therefore the programmable stoplevel (see below) has no influence if disabling the output. Instead the output is galvanically interrupted and has no defined level any more. If a defined output level is needed the AWG output must be terminated externally.

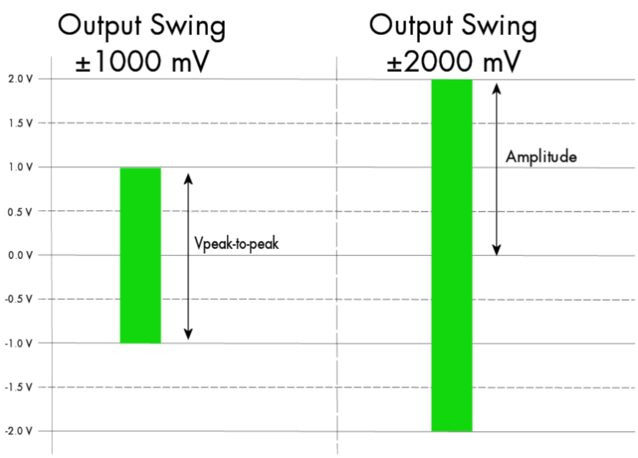
| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_ENABLEOUT0 | 30091 | read/write | Enables (write 1) or Disables (write 0) the output of channel 0 |
| SPC_ENABLEOUT1 | 30191 | read/write | Enables (write 1) or Disables (write 0) the output of channel 1 |
| SPC_ENABLEOUT2 | 30291 | read/write | Enables (write 1) or Disables (write 0) the output of channel 2 |
| SPC_ENABLEOUT3 | 30391 | read/write | Enables (write 1) or Disables (write 0) the output of channel 3 |

## Output Amplifiers

This arbitrary waveform generator board uses separate output amplifiers for each channel. This gives you the possibility to seperately set up the channel outputs to best suit your application

The output amplifiers can easily be set by the corresponding amplitude registers.

The table below shows the available registers to set up the output amplitude for your type of board.




| Register | Value | Direction | Description | Amplitude range |
|---|---|---|---|---|
| SPC_AMP0 | 30010 | read/write | Defines the amplitude of channel0 into 50 Ohm load in mV. | 80 up to 2500 (in mV) |
| SPC_AMP1 | 30110 | read/write | Defines the amplitude of channel1 into 50 Ohm load in mV. | 80 up to 2500 (in mV) |
| SPC_AMP2 | 30210 | read/write | Defines the amplitude of channel2 into 50 Ohm load in mV. | 80 up to 2500 (in mV) |
| SPC_AMP3 | 30310 | read/write | Defines the amplitude of channel3 into 50 Ohm load in mV. | 80 up to 2500 (in mV) |

**The output stage has a 50 Ohm series termination. If not terminating the output with 50 Ohm externally this will result into an output level of double the programmed level. A programmed amplitude of 2500 mV (5000 mV peak-to-peak voltage) will result into an amplitude of 5000 mV (10000 mV peak-to-peak voltage) into high-impedance load !**

## Output Amplitude Setting and Hysteresis

The output amplitude can be changed at any time either while the output is stopped or even while the output is running. The output amplitude is changed on-the-fly with immediate result in the output signal.

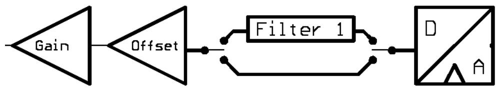
As the output amplifier consist of two different pathes (low power and high power) with slightly different specifications there is a break in the continuous output amplitude change when switching from one output amplifier path to the other as this is done with the help of a relais. If switching from one path to the other the driver will automatically disable the output (zero volt level) for the switching time of 10 ms to avoid disturbed output signal. Please see the technical detail section for the specification of the two different output amplifier path settings.

To prevent the card from switching on and off when operating around the limit between the output amplifiers paths there's a build in hysteresis:

- If output amplifier is already in low power path the output path is switched at the upper border of the hysteresis (480 mV) allowing to use the area between 80 mV and 480 mV with continuous and gap-free change of output amplifier amplitude.
- If output amplifier is already in high power path the output path is switched at the lower border of the hysteresis (420 mV) allowing to use the area between 420 mV and 2500 mV with continuous and gap-free change of output amplifier amplitude.

## Output Filters

Every output of your Spectrum D/A board is equipped with a bypass path and a fixed filter that can be used for signal smoothing.
The filter is located in the signal chain between the output amplification section and the DAC, as shown in the right figure. Depending on your type of board the filter are of differerent filter types and have different cut off frequencies, as shown below. You can choose between the different filters easily by setting the dedicated filter registers. The registers and the possible values are shown in the table below.




| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_FILTER0 | 30080 | read/write | Sets the signal filter of channel0. |
| SPC_FILTER1 | 30180 | read/write | Sets the signal filter of channel1. |
| SPC_FILTER2 | 30280 | read/write | Sets the signal filter of channel2. |
| SPC_FILTER3 | 30380 | read/write | Sets the signal filter of channel3. |
| | 0 | No filter is used on the corresponding channel. | |
| | 1 | Filter 1 is used on the corresponding channel. The type of filter depends on the type of board and is shown below. | |

| Filter | Specifications | M4i.6620-x8<br>M4i.6621-x8<br>M4i.6622-x8 | M4i.6630-x8<br>M4i.6631-x8 |
|---|---|---|---|
| filter 0 | | No filter will be used. | |
| filter 1 | -3 dB bandwidth | 65 MHz | 65 MHz |

## Differential Output

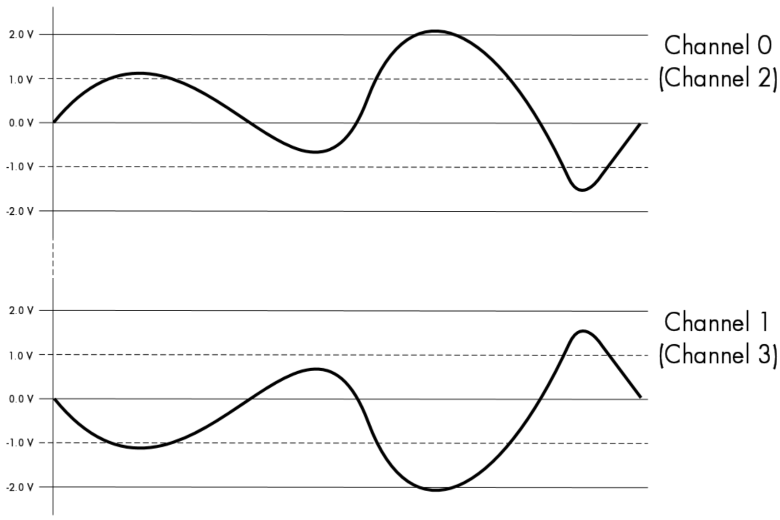
The differntial mode outputs the data on the even channels and the inverted data on the odd channels of one module, as the figure on the right is showing.

As a result you have differential signals, which are more resistant against noise when being transmitted via long cables. Because of the hardware generation, only one data sample in memory is needed for one pair of differential outputs.

The dedicated registers to set up the differential mode are shown below.

If your board has four installed channels you can generate two pairs of differential signals, otherwise one pair is possible.

Differential outputs are not available for all types of boards. Please refer to the table below, which mentions the boards this mode is available on.




| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_DIFF0 | 30040 | read/write | Sets channel 0/1 to differential mode. |
| SPC_DIFF2 | 30240 | read/write | Sets channel 2/3 to differential mode. |

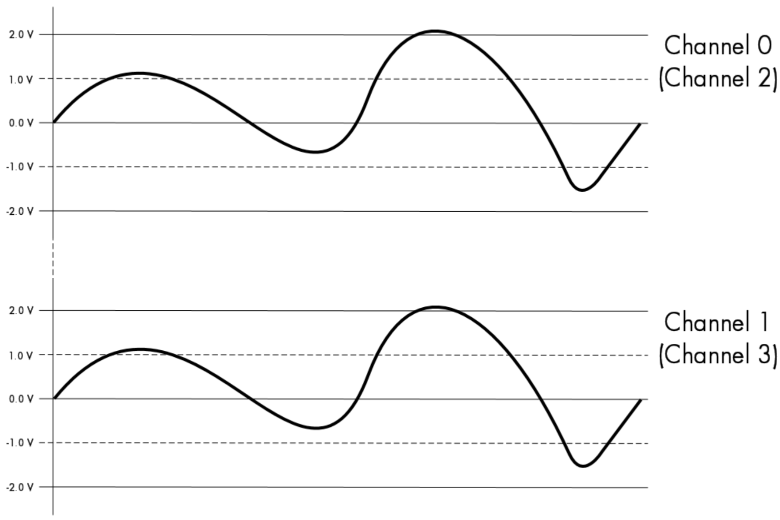
| Mode | M4i.6620 | M4i.6621 | M4i.6622 | M4i.6630 | M4i.6631 |
|---|---|---|---|---|---|
| Differential Output | not available | installed | installed | not available | installed |

## Double Out Mode

The double out mode outputs the data on the even channels and the same data on the odd channels of one module, as the figure on the right is showing. The dedicated registers to set up the differential mode are shown below.

If your board has four installed channels you can generate two pairs of identical signals, otherwise only one pair is possible.

The double out mode is not available for all types of boards. Please refer to the table below, which mentions the boards this mode is available on.




| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_DOUBLEOUT0 | 30041 | read/write | Sets channel 0/1 to double out mode. |
| SPC_DOUBLEOUT2 | 30241 | read/write | Sets channel 2/3 to double out mode. |

| Mode | M4i.6620 | M4i.6621 | M4i.6622 | M4i.6630 | M4i.6631 |
|---|---|---|---|---|---|
| Double out mode | not available | installed | installed | not available | installed |

## Programming the behaviour in pauses and after replay

Usually the used outputs of the analog generation boards are set to zero level after replay. This is in most cases adequate. In some cases it can be necessary to hold the last sample, to output the maximum positive level or maximum negative level after replay. The stoplevel will stay on the defined level until the next output has been made. With the following registers you can define the behaviour after replay:

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_CH0_STOPLEVEL | | 206020 | read/write | Defines the behavior after replay for channel 0 |
| SPC_CH1_STOPLEVEL | | 206021 | read/write | Defines the behavior after replay for channel 1 |
| SPC_CH2_STOPLEVEL | | 206022 | read/write | Defines the behavior after replay for channel 2 |
| SPC_CH3_STOPLEVEL | | 206023 | read/write | Defines the behavior after replay for channel 3 |
| | SPCM_STOPLVL_ZERO | 16 | Defines the analog output to enter zero level (D/A converter is fed with digital zero value) | |
| | SPCM_STOPLVL_LOW | 2 | Defines the analog output to enter maximum negative level (D/A converter is fed with most negative level) | |
| | SPCM_STOPLVL_HIGH | 4 | Defines the analog output to enter maximum positive level (D/A converter is fed with most positive level) | |
| | SPCM_STOPLVL_HOLDLAST | 8 | Holds the last replayed sample on the analog output | |

All outputs that are not activated for replay, will keep the programmed stoplevel also while the replay is in progress.

## Read out of output features

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

Please note that the following table shows all output feature settings that are available throughout all Spectrum generator cards. Some of these features are not installed on your specific hardware.

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_READAOFEATURES | | 3102 | read | Returns a bit map with the available features of the analog output path. The possible return values are listed below. |
| | SPCM_AO_SE | 00000002h | Output is single-ended. If available together with SPC_AO_DIFF: output type is software selectable | |
| | SPCM_AO_DIFF | 00000004h | Output is differential. If available together with SPC_AO_SE: output type is software selectable | |
| | SPCM_AO_PROGFILTER | 00000008h | Software selectable output filters are available. | |
| | SPCM_AO_PROGOFFSET | 00000010h | Output offset is software programmable. | |
| | SPCM_AO_PROGGAIN | 00000020h | Output gain is software programmable. | |
| | SPCM_AO_PROGSTOPLEVEL | 00000040h | The output level between segments of generated data is programmable. | |

| | | |
|---|---|---|
| SPCM_AO_DOUBLEOUT | 00000080h | Double out mode is available allowing to generate cheap copies of even channel data on odd channels outputs for driving multiple loads. |
| SPCM_AO_ENABLEOUT | 00000100h | The output of each channel can be completely disabled by software command at any time. |

# Generation modes

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

## Overview

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

### Setup of the mode

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

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

However as you'll see throughout the detailed documentation of the modes the standard and the FIFO mode are similar in programming and 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. |

#### Replay modes

| Mode | Value | Description |
|---|---|---|
| SPC_REP_STD_SINGLE | 100h | Data generation from on-board memory repeating the complete programmed memory either once, infinite or for a defined number of times after one single trigger event. |
| SPC_REP_STD_MULTI | 200h | Data generation from on-board memory for multiple trigger events. Each generated segment has the same size. This mode is described in greater detail in a special chapter about the Multiple Replay option. |
| SPC_REP_STD_GATE | 400h | Data generation from on-board memory using an external gate signal. Data is only generated as long as the gate signal has a programmed level. The mode is described in greater detail in a special chapter about the Gated Replay option. |
| SPC_REP_STD_SINGLERESTART | 8000h | Data generation from on-board memory. The programmed memory is repeated once after each single trigger event. |
| SPC_REP_STD_SEQUENCE | 40000h | Data generation from on-board memory splitting the memory into several segments and replaying the data using a special sequence memory. The mode is described in greater detail in a special chapter about the Sequence mode. |
| SPC_REP_FIFO_SINGLE | 800h | Continuous data generation after one single trigger event. The on-board memory is used completely as FIFO buffer. |
| SPC_REP_FIFO_MULTI | 1000h | Continuous data generation after multiple trigger events. The on-board memory is used completely as FIFO buffer. |
| SPC_REP_FIFO_GATE | 2000h | Continuous data generation using an external gate signal. The on-board memory is used completely as FIFO buffer. |

## 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 one of 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. |

### Card wait commands

These commands do not return until either the defined state has been reached which 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_TIMEOUT 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 commands until the timeout is disabled again by writing a zero to this register.

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

Example for card control:

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

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

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

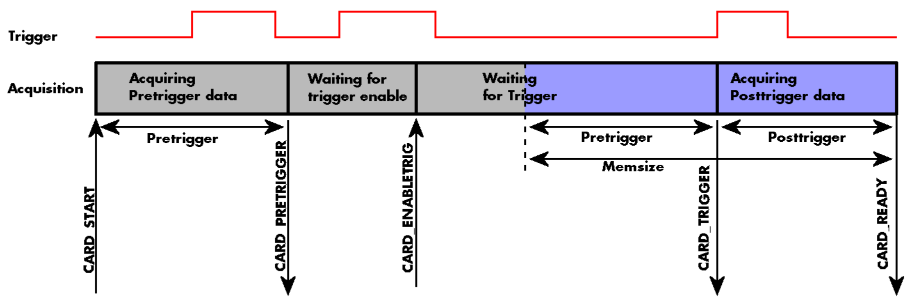
## Card Status

In addition to the wait for an interrupt mechanism or completely instead of it one may also read out the current card status by reading the SPC_M2STATUS register. The status register is organized as a bitmap 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 its run and is ready. | |

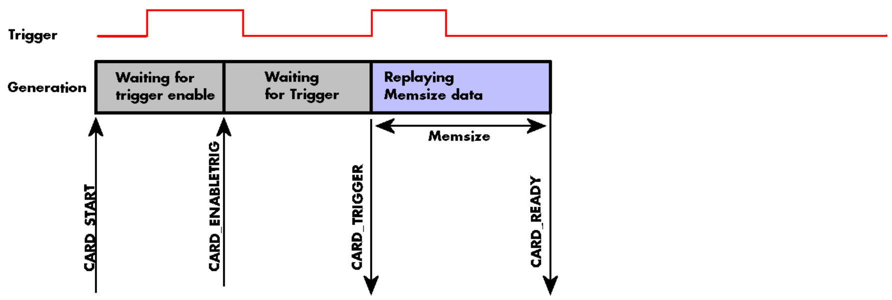
## Acquisition cards status overview

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:

- The memory size register (SPC_MEMSIZE) must be programmed before starting the data transfer.
- Before starting a data transfer the buffer must be defined using the spcm_dwDefTransfer function.
- Each defined buffer is only used once. After transfer has ended the buffer is automatically invalidated.
- If a buffer has to be deleted although the data transfer is in progress or the buffer has at least been defined it is necessary to call the spcm_dwInvalidateBuf function.

### Definition of the transfer buffer

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

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

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

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

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

| | | |
|---|---|---|
| SPCM_DIR_PCTOCARD | 0 | Transfer is done from PC memory to on-board memory of card |
| SPCM_DIR_CARDTOPC | 1 | Transfer is done from card on-board memory to PC memory. |

**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.<br>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);

// explicitely stop DMA tranfer prior to invalidating buffer
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STOPDMA);
delete [] (int8*) pvData;
```

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

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

# Standard Single Replay modes

The standard single modes are the easiest and mostly used modes to generate analog or digital data with a Spectrum arbitrary waveform generation or digital output card. In standard single replay mode the card is working totally independent from the PC, after the card setup is done and the data has been transferred into the on-board memory. The advantage of the Spectrum boards is that regardless to the system usage the card will refresh the outputs with equidistant time intervals.
The data for replay is stored in the on-board memory and is held there for being replayed after the trigger event. This mode allows sample generation at very high refresh rates without the need to transfer the data from the memory of the host system to the card at high speed.

## Card mode

The card mode has to be set to the correct mode SPC_REP_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_REP_STD_SINGLE | 100h | Data generation from on-board memory repeating the complete programmed memory either once, infinite or for a defined number of times after one single trigger event. | |
| | SPC_REP_STD_SINGLERESTART | 8000h | Data generation from on-board memory replaying the complete programmed memory on every detected trigger event. The number of replays can be programmed by loops. | |

## Memory setup

You have to define, how many samples are to be replayed from the on-board memory and how many times the complete memory should be replayed after the trigger event.

**Please note that the memory size must be programmed to the correct value PRIOR to making any data transfer to the card memory. An incorrect memory size value at the time the data transfer is initiated will result in corrupted data and a wrong output.**

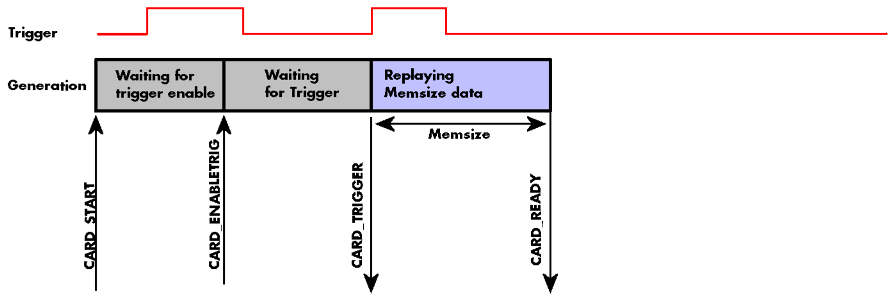
| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_MEMSIZE | 10000 | read/write | Sets the memory size in samples per channel. The memory size setting must be set before transferring data to the card. |
| SPC_LOOPS | 10020 | read/write | Number of times the memory is replayed. If set to zero the generation will run continuously until it is stopped by the user. |

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

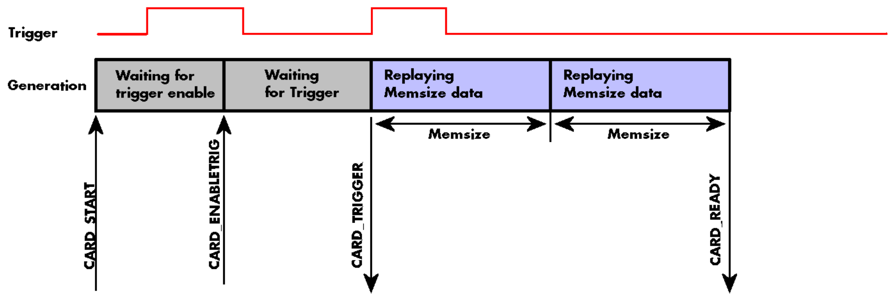
**SPC_REP_STD_SINGLE**

This mode waits for one trigger events and after this it starts to replay the programmed memory either once, a pre-defined number of times on infinitely until explicitly stopped by the user. The SPC_LOOPS register is used to define the number of possible repetitions. Setting this register to 0 the generation will continue until explicitly stopped by the user. Any other value than 0 for SPC_LOOPS will result in the signal being replayed SPC_LOOPS times until the card stopps automatically. For replaying the memory content only once after a trigger the SPC_LOOPS values hence must be set to a value of 1.

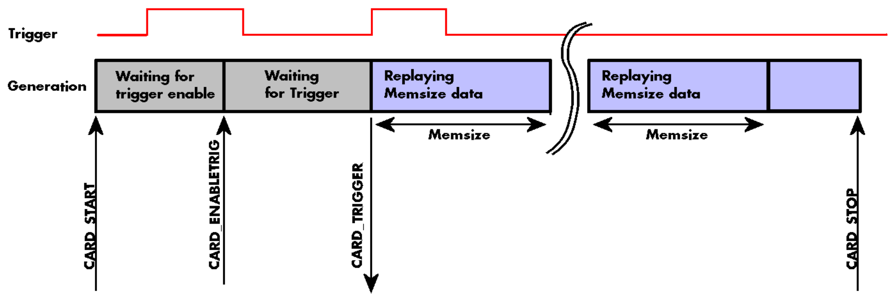
Replay of a data pattern just once




Replay for a defined number of times (2 in the example shown)




Replay continuously until the replay is stopped by the user




## Continuous marker output

If using the continuous output with internal trigger one can activate a marker output on the trigger i/o connectors marking the beginning of each loop. The marker output will generate a TTL marker pulse with the length of ½ of programmed memory

### M2i.60xx/M2i.61xx series details on continuous marker output

The marker output will generate a TTL pulse on the trigger output connector. The pulse length is of ½ of programmed memory up to a maximum trigger pulse width of 256 samples. If memory is larger than 512 samples the trigger pulse width will still be 256 samples. Please be sure to have the trigger output enabled for this function. This function requires driver version ≥ build 1604 and firmware version ≥11.

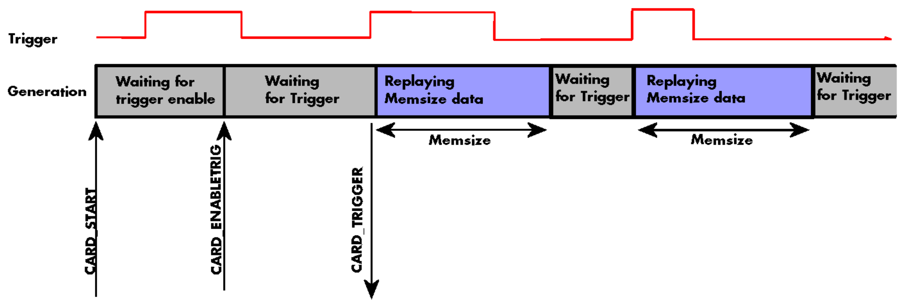
| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_CONTOUTMARK | 200450 | read/write | Writing a 1 enables the marker output on every loop (M2i.60xx/M2i.61xx only) |

### M4i.66xx series details on continuous marker ourput

The marker output will generate a TTL pulse on one of the multi-purpose I/O lines. The pulse length is of ½ of programmed memory. The marker output is enabled using the dedicated multi-purpose I/O line setup that is described later in this manual. Please see the chapter „Multi Purpose I/O Lines" in the trigger section to find more information.

### SPC_REP_STD_SINGLERESTART

This mode behaves like multiple shots of SPC_REP_STD_SINGLE but with a very small re-arming time inbetween. When using this mode the memory content is replayed on every detected trigger event. The SPC_LOOPS parameter defines how long this replay should continue. A value of zero defines the mode to run continuously until stopped by the user.




Between the different replayed pieces the output will go to the programmed stoplevel.

### Overview of settings and resulting modes

This table gives a brief overview on the setup of loops and the resulting behaviour of the output

| | SPC_LOOPS = 0 | SPC_LOOPS = 1 | SPC_LOOPS = N |
|---|---|---|---|
| SPC_REP_STD_SINGLE | Replay starts with the first trigger event and then the programmed data is replayed in a continuous loop until stopped by the user. | The programmed memory content is replayed once after detection of the trigger event. | Replay starts with the first trigger event and then the programmed data is replayed in a continuous loop until the programmed number N of loops has been replayed. Afterward the card stops. |
| SPC_REP_STD_SINGLERESTART | The programmed memory is replayed once on every trigger event. This continues until stopped by the user. | n.a. (similar to SPC_REP_STD_SINGLE) | The programmed memory is replayed once on every trigger event. This continues until the memory is N-times replayed. Afterwards the card stops. |

## Example

The following example shows a simple standard single mode data generation setup with the transfer of data before the card is started. To keep this example simple there is no error checking implemented.

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

spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0);            // only one channel activated
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REP_STD_SINGLE);  // set the standard single replay mode
spcm_dwSetParam_i32 (hDrv, SPC_MEMSIZE, lMemsize);             // replay length
spcm_dwSetParam_i32 (hDrv, SPC_LOOPS, 1);                      // replay memsize once

void* pvData = new int16[lMemsize];                            // create a data buffer
vCalculate_or_Load_Data (pvData);                              // pvData must now be filled with data

// transfer the data to the on-board memory
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA,  SPCM_DIR_PCTOCARD , 0, pvData, 0, 2 * lMemsize);
spcm_dwSetParam_i32    (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

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

# FIFO Single replay mode

The FIFO single mode does a continuous data replay using the on-board memory as a FIFO buffer and transferring data continuously from PC memory. One can generate the data on-line or load data continuously from disk.

## Card mode

The card mode has to be set to the correct mode SPC_REP_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_REP_FIFO_SINGLE | 800h | Continuous data replay from PC memory. Complete on-board memory is used as FIFO buffer. | |

## Length of FIFO mode

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_SEGMENTSIZE | 10010 | read/write | Length of segments to replay. |
| SPC_LOOPS | 10020 | read/write | Number of segments to replay 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 replayed can be calculated by [SPC_LOOPS * SPC_SEGMENTSIZE]. Please stick to the below mentioned limitations of these registers.

## Difference to standard single mode

The standard modes and the FIFO modes do not differ very much from the programming point of view. In fact one can even use the FIFO mode to get the same behaviour as the standard mode. The buffer handling that is shown in the next chapter is the same for both modes.

### Length of replay.

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

## Example (FIFO replay)

The following example shows a simple FIFO single mode data replay setup with the data calculation placed somewhere else. To keep this example simple there is no error checking implemented. Please see in this example that data has to be calculated and transferred prior to the start of the output. The card start and the DMA transfer start cannot be done simultaneously.

```
spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0);                    // only one channel activated
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REP_FIFO_SINGLE);         // set the FIFO single replay mode

// starting with firmware version V9 we can program the hardware buffer size to reduce the latency
spcm_dwGetParam_i32 (hDrv, SPC_PCIVERSION, &lVersion);
if ((lVersion & 0xffff) >= 9)
    {
    spcm_dwSetParam_i32 (stCard.hDrv, SPC_DATA_OUTBUFSIZE, 65536);
    spcm_dwSetParam_i32 (stCard.hDrv, SPC_M2CMD, M2CMD_CARD_WRITESETUP);
    }

// in FIFO mode we need to define the buffer before starting the transfer
int16* pnData = new int16[lBufsizeInSamples];
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_PCTOCARD, 4096, (void*) pnData, 0, 2 * lBufsizeInSamples);

// before start we once have to fill some data in for the start of the output
vCalcOrLoadData (&pnData[0], 2 * lBufsizeInSamples);
spcm_dwSetParam_i32 (hDrv, SPC_DATA_AVAIL_CARD_LEN, 2 * lBufsizeInSamples);
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

// now the first <notifysize> bytes have been transferred to card and we start the output
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);

// we replay data in a loop. As we defined a notify size of 4k we'll get the data in >=4k chuncks
dwTotalBytes = 2 * lBufsizeInSamples;
while (!dwError)
    {
    // read out the available bytes that are free again
    spcm_dwGetParam_i32 (hDrv, SPC_DATA_AVAIL_USER_LEN, &lAvailBytes);
    spcm_dwGetParam_i32 (hDrv, SPC_DATA_AVAIL_USER_POS, &lUserPosInBytes);

    // be sure not to make a rollover and limit the data to be processed
    if ((lUserPosInBytes + lAvailBytes) > (2 * lBufsizeInSamples))
        lAvailBytes = (2 * lBufsizeInSamples) - lUserPosInBytes;
    dwTotalBytes += lAvailBytes;

    // generate some new data
    vCalcOrLoadData (&pnData[lUserPosInBytes / 2], lAvailBytes);
    printf ("Currently Available: %d, total: %d\n", lAvailBytes, dwTotalBytes);

    // now we mark the number of bytes that we just generated for replay and wait for the next free buffer
    spcm_dwSetParam_i32 (hDrv, SPC_DATA_AVAIL_CARD_LEN, lAvailBytes);
    dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_WAITDMA);
    }
```

# Limits of segment size, 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 2 bytes of memory to be stored.

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 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 | | | Segment size SPC_SEGMENTSIZE | | | Loops SPC_LOOPS | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Min | Max | Step | Min | Max | Step | Min | Max | Step |
| 1 channel | Standard Single | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Single Restart | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Standard Multi | 32 | Mem | 32 | 16 | Mem/2 | 16 | | not used | |
| | Standard Gate | 32 | Mem | 32 | | not used | | | not used | |
| | FIFO Single | | not used | | 16 | 8G - 16 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Multi | | not used | | 16 | Mem/2 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Gate | | not used | | | not used | | 0 (∞) | 4G - 1 | 1 |
| 2 channels | Standard Single | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Single Restart | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Standard Multi | 32 | Mem | 32 | 16 | Mem/4 | 16 | | not used | |
| | Standard Gate | 32 | Mem | 32 | | not used | | | not used | |
| | FIFO Single | | not used | | 16 | 8G - 16 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Multi | | not used | | 16 | Mem/4 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Gate | | not used | | | not used | | 0 (∞) | 4G - 1 | 1 |
| 4 channels | Standard Single | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Single Restart | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Standard Multi | 32 | Mem | 32 | 16 | Mem/8 | 16 | | not used | |
| | Standard Gate | 32 | Mem | 32 | | not used | | | not used | |
| | FIFO Single | | not used | | 16 | 8G - 16 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Multi | | not used | | 16 | Mem/8 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Gate | | not used | | | 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.

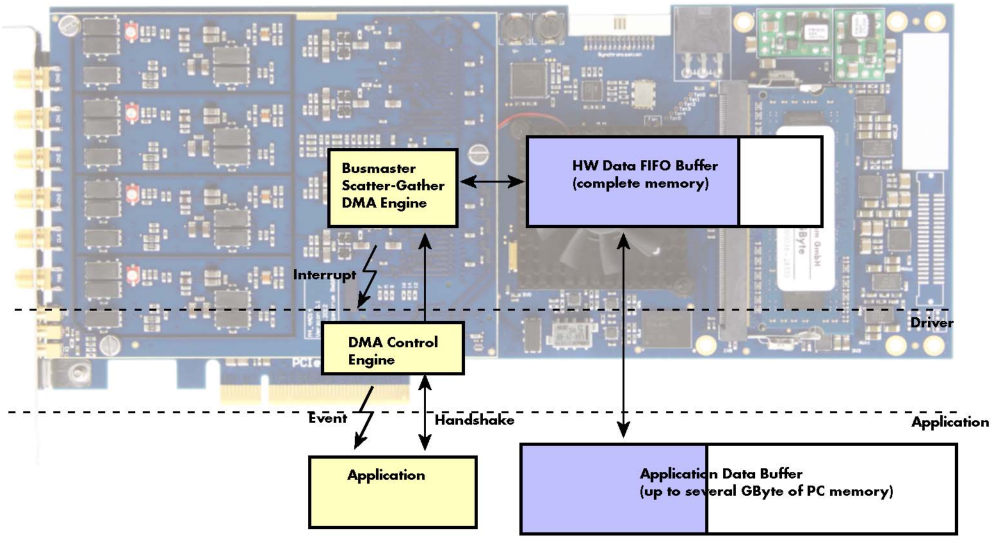
The given memory and memory / divider figures depend on the installed on-board memory as listed below:

| | **Installed Memory 2 GSample** |
|---|---|
| Mem | 2 GSample |
| Mem / 2 | 1 GSample |
| Mem / 4 | 512 MSample |
| Mem / 8 | 256 MSample |

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

# Buffer handling

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




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 |
| Any direction | SPC_FILLSIZEPROMILLE | read | The register holds the current fill size of the on-board memory (FIFO buffer) in promille (1/1000) of the full on-board memory. Please note that the hardware reports the fill size only in 1/16 parts of the full memory. The reported fill size is therefore only shown in 1000/16 = 63 promille steps. |

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

**The counter that is holding the user buffer available bytes (SPC_DATA_AVAIL_USER_LEN) is 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

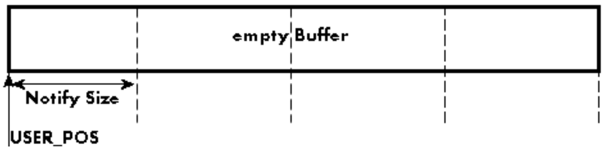
one can simply request the current status whenever the program has time to do so. When using this polling mode the announced available bytes still stick to the defined notify size!
- If the on-board FIFO buffer has an overrun (card to PC) or an underrun (PC to card) data transfer is stopped. However in case of transfer from card to PC there is still a lot of data in the on-board memory. Therefore the data transfer will continue until all data has been transferred although the status information already shows an overrun.
- 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.

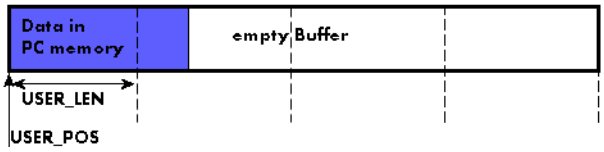
**Step 1: Buffer definition**

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.



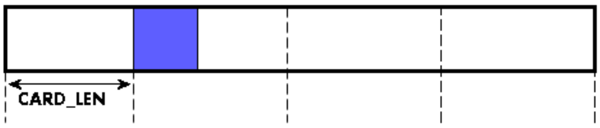

**Step 2: Start and first data available**

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.



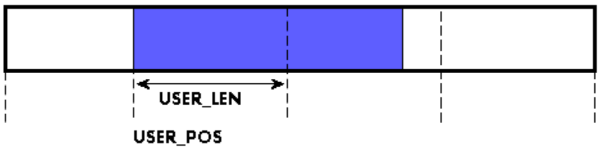

**Step 3: set the first data available for card**

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**

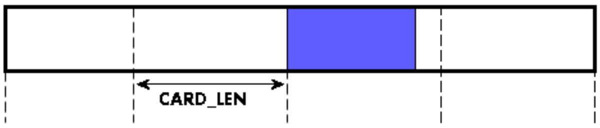
After reaching the next border of the notify size we get the next part of the data buffer to be available. In our example this part of data is again only of one notify size length. The user position will now be at the position [1 x notify size].




**Step 5: set data available again**

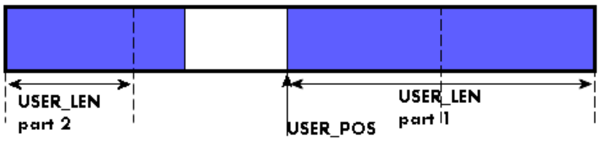
Again after processing the data we set it free for the card use.

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.



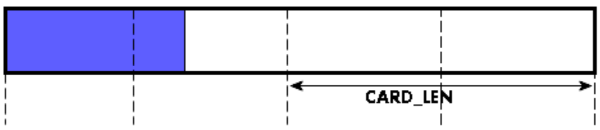

**Step 6: roll over the end of buffer**

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);

// we start the DMA transfer
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA);

do
    {
    if (!dwError)
        {
        // we wait for the next data to be available. Afte this call we get at least 4k of data to proceed
        dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_WAITDMA);

        // if there was no error we can proceed and read out the available bytes that are free again
        spcm_dwGetParam_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 gets 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
```

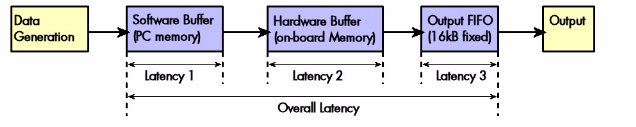
## Buffer handling example for transfer from PC to card

```
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);

// before start we once have to fill some data in for the start of the output
spcm_dwSetParam_i32 (hDrv, SPC_DATA_AVAIL_CARD_LEN, lBufferSizeInBytes);
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

do
    {
    if (!dwError)
        {
        // if there was no error we can proceed and read out the current amount of available data
        spcm_dwGetParam_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 gets a pointer to the start of the available data section and the length
        vDoGenerateData (&pcData[lBytesPos], lAvailBytes);

        // now we mark the number of bytes that we just generated for replay
        // and wait for the next free buffer
        spcm_dwSetParam_i32 (hDrv, SPC_DATA_AVAIL_CARD_LEN, lAvailBytes);
        dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_WAITDMA);
        }
    }
while (!dwError); // we loop forever if no error occurs
```

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

# Output latency

The card is designed to have a most stable and reliable continuous output in FIFO mode. Therefore as default the complete on-board memory is used for buffering data. This however means that you have quite a large latency when changing output data dynamically in reaction of - for example - some external events.




To have a smaller output latency when using dynamically changing data it is recommended that you use smaller buffers. The size of the software buffer is programmed as described above. The size of the hardware buffer can be programmed using a special register:

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_DATA_OUTBUFSIZE | 209 | read/write | Programms the used hardware buffer size for output direction. The default value is the complete standard on-board memory (which is 4 GByte). The output buffer size can be programmed in steps of factor two of the minimum size of TBD. Resulting in allowed settings of TBD, ... up to the installed on-board memory size |

The size of the output FIFO is fixed to TBD kByte and cannot be changed. If using a hardware buffer of TBD kByte and a software buffer of TBD kByte also the total size of buffered data is TBD kByte. Please see the following table for some example output latency calculations.

| Configuration | Sampling rate | Software Buffer | | Hardware Buffer | | Output FIFO | | Overall |
|---|---|---|---|---|---|---|---|---|
| | | Size | Latency | Size | Latency | Size | Latency | Latency |
| 1 x 16 Bit Channel | 625 MS/s | | | | | | | |
| ... | ... | | | | | | | |
| ... | ... | | | | | | | |
| 1 x 16 Bit Channel | 100 MS/s | | | | | | | |
| ... | ... | | | | | | | |
| ... | ... | | | | | | | |

**Please keep in mind that lowering the output buffer size also means that the risk of a buffer underrun gets higher as less data is buffered on the hardware side. Therefore please be careful with selecting the correct hardware buffer size and do not make it smaller than absolutely necessary.**

# Data organization

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

| 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 | B0 | C0 | D0 | A1 | B1 | C1 | D1 | A2 | B2 | C2 | D2 | A3 | B3 | C3 | D3 | A4 |

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

# Sample format

The 16 bit D/A samples are stored in twos complement in a 16 bit data word. 16 bit resolution means that data is ranging from -32768…to…+32767.

| Bit | Standard Mode |
|---|---|
| D15 | DAx Bit 15 (MSB) |
| D14 | DAx Bit 14 |
| D13 | DAx Bit 13 |
| D12 | DAx Bit 12 |
| D11 | DAx Bit 11 |
| D10 | DAx Bit 10 |
| D9 | DAx Bit 9 |
| D8 | DAx Bit 8 |
| D7 | DAx Bit 7 |
| D6 | DAx Bit 6 |
| D5 | DAx Bit 5 |
| D4 | DAx Bit 4 |
| D3 | DAx Bit 3 |
| D2 | DAx Bit 2 |
| D1 | DAx Bit 1 |
| D0 | DAx Bit 0 (LSB) |

# Hardware data conversion

The data conversion modes allow the conversion of input data in hardware. This is especially usefull when replaying previously recorded data of acquisition cards with either 14 bit, 12 bit or even 8 bit resolution. The conversion takes place in hardware and therefore avoids a possible time consuming shift in the user application software.

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_AVAILDATACONVERSION | | 201401 | read | Bitmask, in which all bits of the below mentioned data conversion modes are set, if available. |
| SPC_DATACONVERSION | | 201400 | read/write | Defines the used hardware data conversion mode or reads out the actual selected one. |
| | SPCM_DC_NONE | 0h | No hardware data conversion will be done. | |
| | SPCM_DC_12BIT_TO_16BIT | 8h | 12 bit input data is assumed and samples will be logically shifted upwards to use available 16 bit resolution. | |
| | SPCM_DC_15BIT_TO_16BIT | 10h | 15 bit input data is assumed and samples will be logically shifted upwards to use available 16 bit resolution. | |
| | SPCM_DC_14BIT_TO_16BIT | 20h | 14 bit input data is assumed and samples will be logically shifted upwards to use available 16 bit resolution. | |

# Clock generation

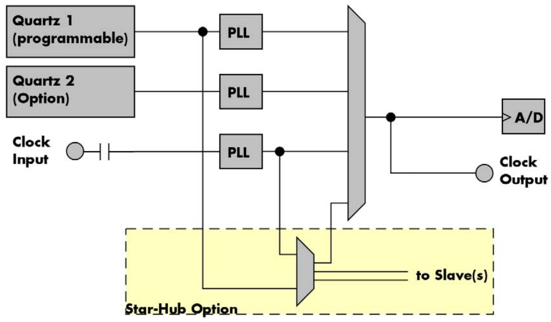
## Overview

### The different clock modes

The Spectrum M3i/M4i 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 M3i/M4i 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 (programmable reference quartz 1)

This is the easiest and most common way to generate a sample rate with no need for additional external clock signals. The sample rate has a very fine resolution, low jitter and a high accuracy. The Quartz 1 is a high quality software programmable clock device acting as a reference to the internal PLL. The specification is found in the technical data section of this manual.

#### Quartz2 with PLL (option)

This optional second Quartz 2 is for special customer needs, either for a special direct sampling clock or as a very precise reference for the PLL. Please feel free to contact Spectrum for your special needs. The Quarz 2 clock footprint can be equipped with a wide variety of clock sources that are available on the market.

#### External Clock (reference clock)

Any clock can be fed in that matches the specification of the board. The external clock signal can be used to synchronize the board on a system clock or to feed in an exact matching sample rate. The external clock is divided/multiplied using a PLL allowing a wide range of external clock modes.

#### Synchronization clock (option Star-Hub)

The star-hub option allows the synchronization of up to 8 cards of the M3i/M4i series from Spectrum with a minimal phase delay between the different cards. The clock is distributed from the master card to all connected cards. As a source it is possible to either use the programmable Quarz 1 clock or the external clock input of the master card. For details on the synchronization option please take a look at the dedicated chapter later 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 programmable high precision Quartz 1 for sample clock generation | |
| | SPC_CM_QUARTZ2 | 4 | Enables optional Quartz 2 as reference for sample clock generation | |
| | SPC_CM_EXTREFCLOCK | 32 | Enables internal PLL with external reference for sample clock generation | |

The different clock modes and all other related or required register settings are described on the following pages.

# Details on the different clock modes

## Standard internal sampling clock (PLL)

The internal sampling clock is generated in default mode by a programmable high precision quartz. You need to select the clock mode by the dedicated register shown in the table below:

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_CLOCKMODE | | 20200 | read/write | Defines the used clock mode |
| | SPC_CM_INTPLL | 1 | Enables internal programmable high precision Quartz 1 for sample clock generation | |

The user does not have to care 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. |

Independent of the used clock source it is possible to enable the clock output. The clock will be available on the external clock output connector and can be used to synchronize external equipment with the board.

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_CLOCKOUT | 20110 | read/write | Writing a „1" enables clock output on external clock output connector. Writing a „0" disables the clock output (tristate) |

Example on writing and reading internal sampling rate

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_INTPLL); // Enables internal programmable quarz 1
spcm_dwSetParam_i64 (hDrv, SPC_SAMPLERATE,  62500000);    // Set internal sampling rate to 62.5 MHz
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKOUT,  1);             // enable the clock output of the card
spcm_dwGetParam_i64 (hDrv, SPC_SAMPLERATE, &lSamplerate); // Read back the programmed sample rate and print
printf („Sample rate = %d\n", lSamplerate);               // it. Output should be „Sample rate = 62500000"
```

### Minimum internal sampling rate

The minimum and the maximum internal sampling rates depend on the specific type of board. Both values can be found in the technical data section of this manual.

## Using Quartz2 with PLL (optional)

In some cases it is necessary to use a special high precision frequency for sampling rate generation. For these applications all cards of the M3i/M4i 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 | |

The quartz 2 clock is routed through a PLL to allow the generation of sampling rates based on this reference clock. 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.

Independent of the used clock source it is possible to enable the clock output. The clock will be available on the external clock output connector and can be used to synchronize external equipment with the board.

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_CLOCKOUT | 20110 | read/write | Writing a „1" enables clock output on external clock output connector. Writing a „0" disables the clock output (tristate) |

## External clock (reference clock)

The external clock input of the M3i/M4i series is fed through a PLL to the clock system. Therefore the input will act as a reference clock input thus allowing to either use a copy of the external clock or to generate any sampling clock within the allowed range from the reference clock.

Please note the limited setup granularity in comparison to the internal sampling clock generation. Details are found in the technical data section.

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_CLOCKMODE | | 20200 | read/write | Defines the used clock mode |
| | SPC_CM_EXTREFCLOCK | 32 | Enables internal PLL with external reference for sample clock generation | |

Due to the fact that the driver needs to know the external fed in frequency for an exact calculation of the sampling rate you must set the SPC_REFERENCECLOCK register accordingly as shown in the table below. The driver then automatically sets the PLL to achieve the desired sampling rate. Please be aware that the PLL has some internal limits and not all desired sampling rates may be reached with every reference clock.

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_REFERENCECLOCK | | 20140 | read/write | Programs the external reference clock in the range from 10 MHz to 1 GHz. |
| | External sampling rate in Hz as an integer value | | You need to set up this register exactly to the frequency of the external fed in clock. | |

Example of reference clock:

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_EXTREFCLOCK);   // Set to reference clock mode
spcm_dwSetParam_i32 (hDrv, SPC_REFERENCECLOCK, 10000000);        // Reference clock that is fed in is 10 MHz
spcm_dwSetParam_i64 (hDrv, SPC_SAMPLERATE,     65200000);        // We want to have 62.5 MHz as sampling rate
```

**PLL Locking Error**

The external clock signal is routed to a PLL to generate any sampling clock from this external clock. Due to the internal structure of the card the PLL is even used if a copy of the clock fed in externally is used for sampling (SPC_REFERENCECLOCK = SPC_SAMPLERATE). The PLL needs a stable and defined external clock with no gaps and no variation in the frequency. The external clock must be present when issuing the start command. It is not possible to start the card with external clock activated and no external clock available.

When starting the card all settings are written to hardware and the PLL is programmed to generate the desired sampling clock. If there has been any change to the clock setting the PLL then tries to lock on the external clock signal to generate the sampling clock. This locking will normally need 10 to 20 ms until the sampling clock is stable. Some clock settings may also need 200 ms to lock the PLL. This waiting time is automatically added at card start.

However if the PLL can not lock on the external clock either because there is no clock available or it hasn't sufficient signal levels or the clock is not stable the driver will return with an error code ERR_CLOCKNOTLOCKED. In that case it is necessary to check the external clock connection. Please see the example below:

```
// settings done to external clock like shown above.
if (spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER) == ERR_CLOCKNOTLOCKED)
    {
    printf („External clock not locked. Please check connection\n");
    return -1;
    }
```

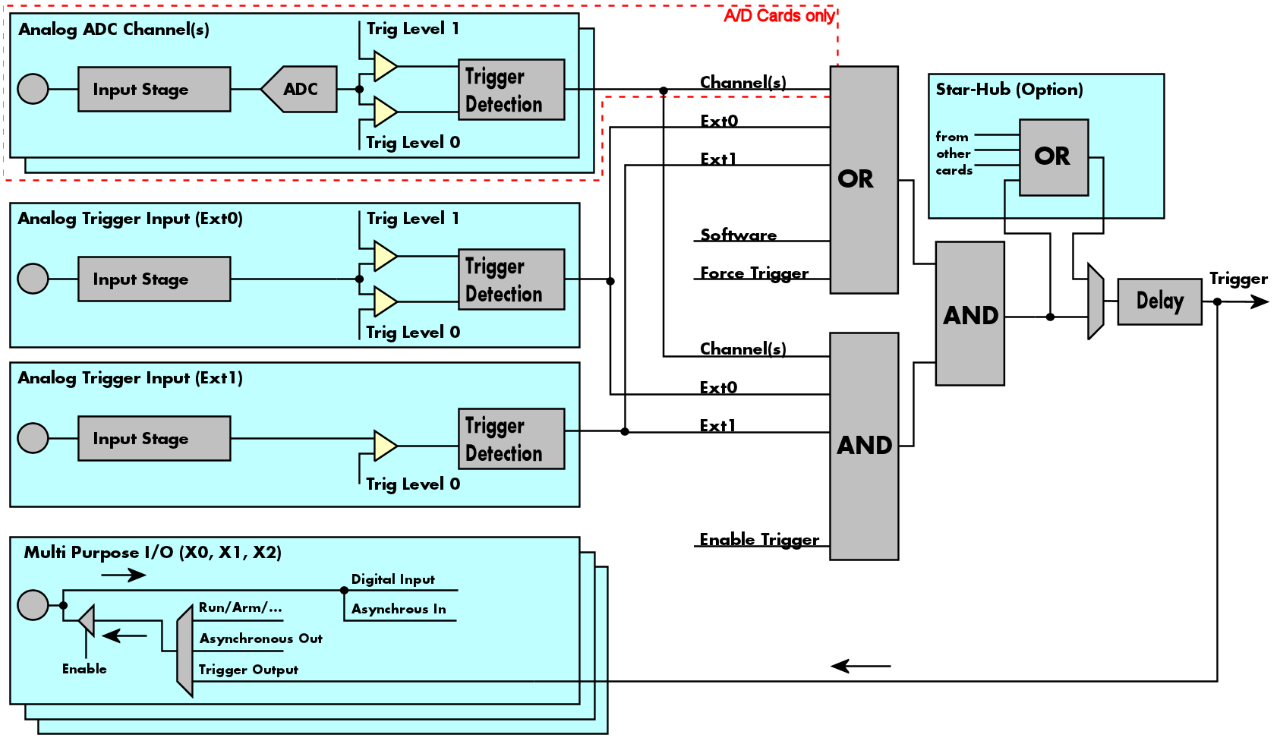
# Trigger modes and appendant registers

## General Description

The trigger modes of the Spectrum M4i series A/D and D/A cards are very extensive and give you the possibility to detect nearly any trigger event you can think of.

You can choose between more than 10 external trigger modes and up to 20 internal trigger modes (on analog acquisition cards) including software and channel trigger, depending on your type of board. Many of the channel trigger modes can be independently set for each input channel (on A/D boards only) resulting in a even bigger variety of modes. This chapter is about to explain all of the different trigger modes and setting up the card's registers for the desired mode.

## Trigger Engine Overview




The trigger engine of the M4i card series allows to combine several different trigger sources with OR and AND combination, with a trigger delay or even with an OR combination across several cards when using the Star-Hub option. The above drawing gives a complete overview of the trigger engine and shows all possible features that are available.

On A/D cards each analog input channel has two trigger level comparators to detect edges as well as windowed triggers. All card types have a total of three different additional external trigger sources. One main trigger source which also has two analog level comparators also allowing to use edge and windowed trigger detection and two multi purpose in/outputs that can be software programmed to either additional trigger inputs or trigger outputs or to some extended status signals.

The Enable trigger allows the user to enable or disable all trigger sources (including channel trigger on A/D cards and external trigger) with a single software command. The enable trigger command will not work on force trigger.

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

Before the trigger event is finally generated, it is wired through a programmable trigger delay. This trigger delay will also work when used in a synchronized system thus allowing each card to individually delay its trigger recognition.

# Multi Purpose I/O Lines


The M4i series has three multi purpose I/O lines that can be used for a wide variety of functions to help the interconnection with external equipment. The functionality of these multi purpose I/O lines can be software programmed and each of these lines can either be used for input or output.

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

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

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

## Programming the behaviour

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

| Register | Value | Direction | Description |
|---|---|---|---|
| SPCM_X0_AVAILMODES | 47210 | read | Bitmask with all bits of the below mentioned modes showing the available modes for (X0) |
| SPCM_X1_AVAILMODES | 47211 | read | Bitmask with all bits of the below mentioned modes showing the available modes for (X1) |
| SPCM_X2_AVAILMODES | 47212 | read | Bitmask with all bits of the below mentioned modes showing the available modes for (X2) |
| SPCM_X0_MODE | 47200 | read/write | Defines the mode for (X0). Only one mode selection is possible to be set at a time |
| SPCM_X1_MODE | 47201 | read/write | Defines the mode for (X1). Only one mode selection is possible to be set at a time |
| SPCM_X2_MODE | 47202 | read/write | Defines the mode for (X2). Only one mode selection is possible to be set at a time |
| SPCM_XMODE_DISABLE | 00000000h | | No mode selected. Output is tristate (default setup) |
| SPCM_XMODE_ASYNCIN | 00000001h | | Connector is programmed for asynchronous input. Use SPCM_XX_ASYNCIO to read data asynchronous as shown in next chapter. |
| SPCM_XMODE_ASYNCOUT | 00000002h | | Connector is programmed for asynchronous output. Use SPCM_XX_ASYNCIO to write data asynchronous as shown in next chapter. |
| SPCM_XMODE_DIGIN | 00000004h | | Connector is programmed for digital input. For each analog channel, one digital channel X0/X1/X2 is integrated into the ADC data stream. Depending on the ADC resolution of your card the resulting merged samples can have different formats. Please check the data format chapter to see more details. Please note that automatic sign extension of analog data is ineffective as soon as one digital input line is activated and the software must properly mask out the digital bits. |
| SPCM_XMODE_TRIGOUT | 00000020h | | Connector is programmed as trigger output and shows the trigger detection. The trigger output goes HIGH as soon as the trigger is recognized. After end of acquisition it is LOW again. In Multiple Recording/Gated Sampling/ABA mode it goes LOW after the acquisition of the current segment stops. In standard FIFO mode the trigger output is HIGH until FIFO mode is stopped. |
| SPCM_XMODE_DIGIN2BIT | 00000080h | | Connector is programmed for digital input. For each analog channel, two digital channels X0/X1/X2 are integrated into the ADC data stream. Depending on the ADC resolution of your card the resulting merged samples can have different formats. Please check the data format chapter to see more details. Please note that automatic sign extension of analog data is ineffective as soon as one digital input line is activated and the software must properly mask out the digital bits. |
| SPCM_XMODE_RUNSTATE | 00000100h | | Connector shows the current run state of the card. If acquisition/output is running the signal is HIGH. If card has stopped the signal is LOW. |
| SPCM_XMODE_ARMSTATE | 00000200h | | Connector shows the current ARM state of the card. If the card is armed and ready to receive a trigger the signal is HIGH. If the card isn't running or the card is still acquiring pretrigger data or the trigger has been detected the signal is LOW. |
| SPCM_XMODE_REFCLKOUT | 00001000h | | Connector reflects the internal generated PLL reference clock in the range of 10 to 62.5 MHz. |
| SPCM_XMODE_CONTOUTMARK | 00002000h | | Generator Cards only: outputs a HIGh pulse as continuous marker signal for continuous replay mode. The marker signal length is ½ of the programmed memory size. |

**Please note that a change to the SPCM_X0_MODE, SPCM_X1_MODE or SPCM_X2_MODE will only be updated with the next call to either the M2CMD_CARD_START or M2CMD_CARD_WRITESETUP register. For further details please see the relating chapter on the M2CMD_CARD registers.**

## Using asynchronous I/O

To use asynchronous I/O on the multi purpose I/O lines it is first necessary to switch these lines to the desired asynchronous mode by programming the above explained mode registers. As a special feature asynchronous input can also be read if the mode is set to trigger input or digital input.

| Register | Value | Direction | Description |
|---|---|---|---|
| SPCM_XX_ASYNCIO | 47220 | read/write | Connector X0 is linked to bit 0 of the register, connector X1 is linked to bit 1 while connector X2 is linked to bit 2of this register. Data is written/read immediately without any relation to the currently used sampling rate or mode. If a line is programmed to output, reading this line asynchronously will return the current output level. |

Example of asynchronous write and read. We write a high pulse on output X1 and wait for a high level answer on input X0:

```
spcm_dwSetParam_i32 (hDrv, SPCM_X0_MODE, SPCM_XMODE_ASYNCIN);    // X0 set to asynchronous input
spcm_dwSetParam_i32 (hDrv, SPCM_X1_MODE, SPCM_XMODE_ASYNCOUT);   // X1 set to asynchronous output
spcm_dwSetParam_i32 (hDrv, SPCM_X2_MODE, SPCM_XMODE_TRIGOUT);    // X2 set to trigger output

spcm_dwSetParam_i32 (hDrv, SPCM_XX_ASYNCIO, 0);                  // programming a high pulse on output
spcm_dwSetParam_i32 (hDrv, SPCM_XX_ASYNCIO, 2);
spcm_dwSetParam_i32 (hDrv, SPCM_XX_ASYNCIO, 0);

do {
    spcm_dwGetParam_i32 (hDrv, SPCM_XX_ASYNCIO, &lAsyncIn);      // read input in a loop
} while ((lAsyncIn & 1) == 0)                                    // until X0 is going to high level
```

## Special behaviour of trigger output

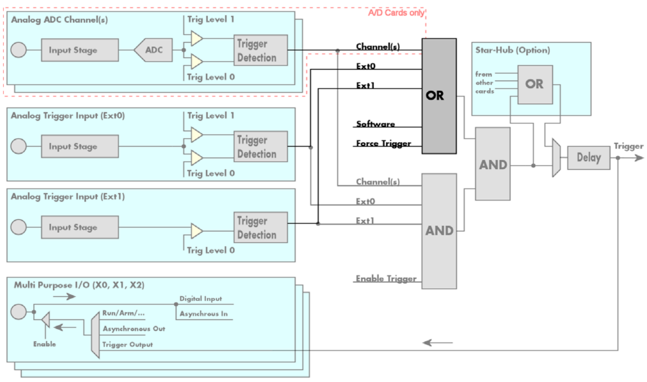
As the driver of the M4i series is the same as the driver for the M2i/M3i series and some old software may rely on register structure of the M2i/M3i card series there is a special compatible trigger output register that will work according to the M2i/M3i series style. It is not recommended to use this register unless you're writing software for multiple card series:

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_OUTPUT | 40100 | read/write | M2i style trigger output programming. Write a „1" to enable:<br>- X2 trigger output (SPCM_X2_MODE = SPCM_XMODE_TRIGOUT)<br>- X1 arm state (SPCM_X1_MODE = SPCM_XMODE_ARMSTATE).<br>- X0 run state (SPCM_X0_MODE = SPCM_XMODE_RUNSTATE).<br><br>Write a „0" to disable both outputs:<br>- SPCM_X0_MODE = SPCM_X1_MODE = SPCM_X2_MODE = SPCM_XMODE_DISABLE |

The SPC_TRIG_OUTPUT register overrides the multi purpose I/O settings done by SPCM_X0_MODE, SPCM_X1_MODE and SPCM_X2_MODE and vice versa. Please do not use both methods in one program.

# 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.

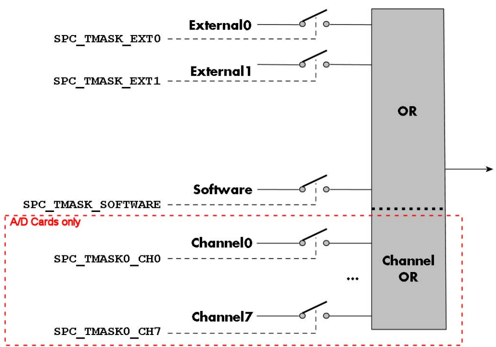
The OR mask shown in the overview before as one object, is separated into two parts: a general OR mask for main external trigger (external analog window trigger), the secondary external trigger (external analog comparator trigger) and software trigger and a channel OR mask.

Every trigger source of the M4i series cards is wired to one of the above mentioned OR masks. The user then can program which trigger source will be recognized, and which one won't.

This selection for the general mask is realized with the SPC_TRIG_ORMASK register in combination with constants for every possible trigger source.

This selection for the channel mask (A/D cards only) is realized with the SPC_TRIG_CH_ORMASK0 register in combination with constants for every possible channel trigger source.

In either case the sources are coded as a bitfield, so that they can be combined by one access to the driver with the help of a bitwise OR.



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 (analog window) trigger 0 for the OR mask. The card will trigger when the programmed condition for this input is valid. | |
| | SPC_TMASK_EXT1 | 4h | Enables the external (analog comparator) trigger 1 for the OR mask. The card will trigger when the programmed condition for this input is valid. | |

The following example shows, how to setup the OR mask, for the two external trigger inputs, ORing them together. When using just a single trigger, only this particular trigger must be used in the OR mask register, respectively. As an example a simple edge detection has been chosen for Ext1 input and a window edge detection has been chosen for Ext0 input. The explanation and a detailed description of the different trigger modes for the external trigger inputs will be shown in the dedicated passage within this chapter.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_LEVEL0, 1800);         // lower Window Trigger level set to 1.8 V
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_LEVEL1, 2000);         // upper Window Trigger level set to 2.0 V
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_WINENTER);// Setting up main window trigger for entering

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT1_LEVEL0, 2500);      // Trigger level set to 2.5 V
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT1_MODE, SPC_TM_POS);  // Setting up secondary trigger for rising edges

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

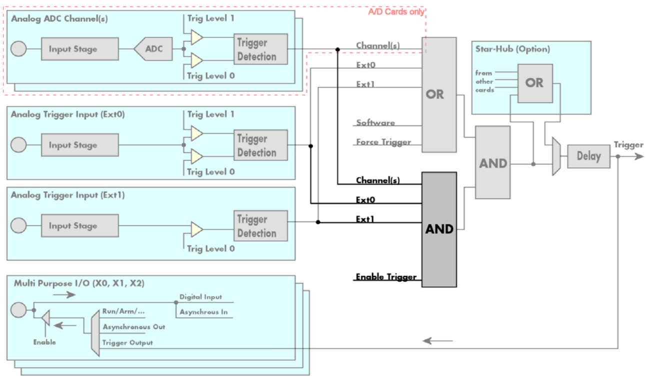
The table below is showing the registers for the channel OR mask (A/D cards only) and the possible constants that can be written to it.

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_TRIG_CH_AVAILORMASK0 | | 40450 | read | Bitmask, in which all bits of the below mentioned sources/channels (0…31) for the channel OR mask are set, if available. |
| SPC_TRIG_CH_ORMASK0 | | 40460 | read/write | Includes the analog or digital channels (0…31) within the channel trigger OR mask of the card. |
| | SPC_TMASK0_CH0 | 00000001h | Enables channel0 for recognition within the channel OR mask. | |
| | SPC_TMASK0_CH1 | 00000002h | Enables channel1 for recognition within the channel OR mask. | |
| | SPC_TMASK0_CH2 | 00000004h | Enables channel2 for recognition within the channel OR mask. | |
| | SPC_TMASK0_CH3 | 00000008h | Enables channel3 for recognition within the channel OR mask. | |
| | … | … | … | |
| | SPC_TMASK0_CH28 | 10000000h | Enables channel28 for recognition within the channel OR mask. | |
| | SPC_TMASK0_CH29 | 20000000h | Enables channel29 for recognition within the channel OR mask. | |
| | SPC_TMASK0_CH30 | 40000000h | Enables channel30 for recognition within the channel OR mask. | |
| | SPC_TMASK0_CH31 | 80000000h | Enables channel31 for recognition within the channel OR mask. | |

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

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ORMASK0, SPC_TMASK_CH0); // Enable channel0 trigger within the OR mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_LEVEL0, 0);          // Trigger level is zero crossing
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE, SPC_TM_POS);   // Setting up channel trigger for rising edges
```

## Trigger AND mask




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

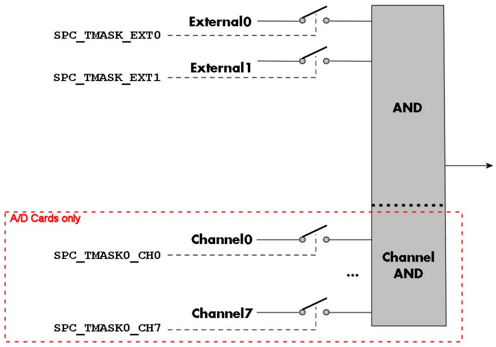
The AND mask shown in the overview before as one object, is separated into two parts: a general AND mask for external trigger and software trigger and a channel AND mask.

Every trigger source of the M4i series cards except the software trigger is wired to one of the above mentioned AND masks. The user then can program which trigger source will be recognized, and which one won't.

This selection for the general mask is realized with the SPC_TRIG_ANDMASK register in combination with constants for every possible trigger source.

This selection for the channel mask (A/D cards only) is realized with the SPC_TRIG_CH_ANDMASK0 register in combination with constants for every possible channel trigger source. In either case the sources are coded as a bitfield, so that they can be combined by one access to the driver with the help of a bitwise OR.

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_NONE | 0 | | No trigger source selected |
| SPC_TMASK_EXT0 | 2h | | Enables the external (analog window) trigger 0 for the AND mask. The card will trigger when the programmed condition for this input is valid. |
| SPC_TMASK_EXT1 | 4h | | Enables the external (analog comparator) trigger 1 for the AND mask. The card will trigger when the programmed condition for this input is valid. |

The following example shows, how to setup the AND mask, for an external trigger. As an example a simple high level detection has been chosen. When multiple external triggers shall be combined by AND, both of the external sources must be included in the AND mask register, similar to the OR mask example shown before. The explanation and a detailed description of the different trigger modes for the external trigger inputs will be shown in the dedicated passage within this chapter.

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ANDMASK, SPC_TMASK_EXT0); // Enable external trigger within the AND mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_LEVEL0, 2000);     // Trigger level is 2.0 V (2000 mV)
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_HIGH);// Setting up external trigger for HIGH level
```

The table below is showing the constants for the channel AND mask (A/D cards only) and all the constants for the different channels.

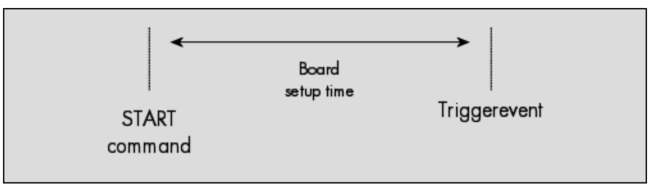
| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_CH_AVAILANDASK0 | 40470 | read | Bitmask, in which all bits of the below mentioned sources/channels (0…31) for the channel AND mask are set, if available. |
| SPC_TRIG_CH_ANDMASK0 | 40480 | read/write | Includes the analog or digital channels (0…31) within the channel trigger AND mask of the card. |
| SPC_TMASK0_CH0 | 00000001h | | Enables channel0 for recognition within the channel OR mask. |
| SPC_TMASK0_CH1 | 00000002h | | Enables channel1 for recognition within the channel OR mask. |
| SPC_TMASK0_CH2 | 00000004h | | Enables channel2 for recognition within the channel OR mask. |
| SPC_TMASK0_CH3 | 00000008h | | Enables channel3 for recognition within the channel OR mask. |
| … | … | | … |
| SPC_TMASK0_CH28 | 10000000h | | Enables channel28 for recognition within the channel OR mask. |
| SPC_TMASK0_CH29 | 20000000h | | Enables channel29 for recognition within the channel OR mask. |
| SPC_TMASK0_CH30 | 40000000h | | Enables channel30 for recognition within the channel OR mask. |
| SPC_TMASK0_CH31 | 80000000h | | Enables channel31 for recognition within the channel OR mask. |

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

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ANDMASK0, SPC_TMASK_CH0);// Enable channel0 trigger within AND mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_LEVEL0, 0);             // channel level to detect is zero level
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_MODE, SPC_TM_HIGH);     // Setting up ch0 trigger for HIGH levels
```

## Software trigger

The software trigger is the easiest way of triggering any Spectrum board. The acquisition or replay of data will start immediately after 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. | |

Example for setting up the software trigger:

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_SOFTWARE); // Internal software trigger mode is used
```

# Force- and Enable trigger

In addition to the 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.

Issuing the forcetrigger command will every time only generate one trigger event. If for example using Multiple Recording that will result in only one segment being acquired by force trigger. After execution of the forcetrigger command the trigger engine will fall back to the trigger mode that was originally programmed and will again wait for a trigger event.

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_M2CMD | | 100 | write | Command register of the M2i/M3i/M4i series cards. |
| | M2CMD_CARD_FORCETRIGGER | 10h | Forces a trigger event if the hardware is still waiting for a trigger event. | |

The example shows, how to use the forcetrigger command:

```
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_FORCETRIGGER); // Forcetrigger is used.
```

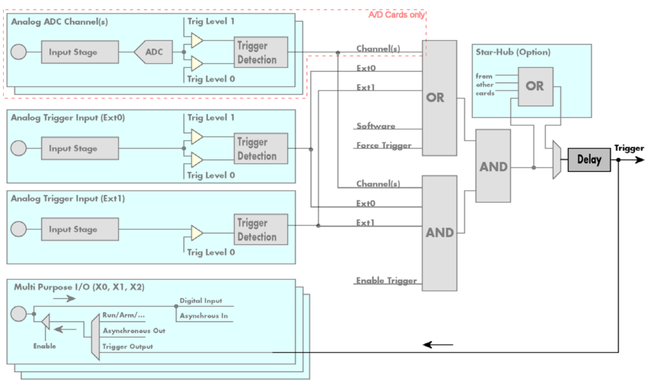
It is also possible to enable (arm) or disable (disarm) the card's whole triggerengine by software. By default the trigger engine is disabled.

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_M2CMD | | 100 | write | Command register of the M2i/M3i/M4i 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_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_ENABLETRIGGER); // Trigger engine is armed.
...
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_DISABLETRIGGER); // Trigger engine is disarmed.
```

# Trigger delay




All of the Spectrum M4i series cards allow the user to program an additional trigger delay. As shown in the trigger overview section, this delay is the last element in the trigger chain. Therefore the user does not have to care for the sources when programming the trigger delay.

As shown in the overview the trigger delay is located after the star-hub connection meaning that every M4i card being synchronized can still have its own trigger delay programmed. The Star-Hub will combine the original trigger events before the result is being delayed.

The delay is programmed in samples. The resulting time delay will therefore be [Programmed Delay] / [Sampling Rate].

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

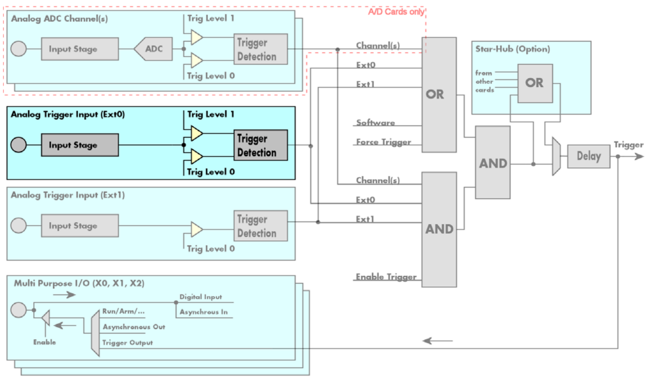
| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_AVAILDELAY | 40800 | read | Contains the maximum available delay as a decimal integer value. |
| SPC_TRIG_DELAY | 40810 | read/write | Defines the delay for the detected trigger events. |
| 0 | | | No additional delay will be added. The resulting internal delay is mentioned in the technical data section. |
| 16…[8G -8] in steps of 16 (12, 14, 16 bit cards) | | | Defines the additional trigger delay in number of sample clocks. The trigger delay can be programmed up to (8GSamples - 16) = 8589934576. Stepsize is 16 samples for 12, 14, 16 bit cards. |
| 32…[8G -32] in steps of 32 (8 bit cards) | | | Defines the additional trigger delay in number of sample clocks. The trigger delay can be programmed up to (8GSamples - 32) = 8589934560. Stepsize is 32 samples for 8 bit cards. |

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

```
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_DELAY, 2000); // A detected trigger event will be
                                                  // delayed for 2000 sample clocks.
```

Using the delay trigger does not affect the ratio between pre trigger and post trigger recorded number of samples, but only shifts the trigger event itself. For changing these values, please take a look in the relating chapter about „Acquisition Modes".

# Main external window trigger (Ext0)




The M4i series has one main external trigger input consisting of an input stage with programmable termination and programmable AC/DC coupling and two comparators that can be programmed in the range of +/- 10000 mV. Using two comparators offers a wide range of different trigger modes that are support like edge, level, re-arm and window trigger.

The main external analog trigger can be easily combined with channel trigger or with the secondary external trigger being programmed as an additional external trigger input. The programming of the masks is shown in the chapters above.

## Trigger Mode

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

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_EXT0_AVAILMODES | 40500 | read | Bitmask showing all available trigger modes for external 0 (Ext0) = main analog trigger input |
| SPC_TRIG_EXT0_MODE | 40510 | read/write | Defines the external trigger mode for the external SMA connector trigger input. The trigger need to be added to either OR or AND mask input to be activated. |
| SPC_TM_NONE | 00000000h | | Channel is not used for trigger detection. This is as with the trigger masks another possibility for disabling channels. |
| SPC_TM_POS | 00000001h | | Trigger detection for positive edges (crossing level 0 from below to above) |
| SPC_TM_NEG | 00000002h | | Trigger detection for negative edges (crossing level 0 from above to below) |
| SPC_TM_POS \| SPC_TM_REARM | 01000001h | | Trigger detection for positive edges on level 0. Trigger is armed when crossing level 1 to avoid false trigger on noise |
| SPC_TM_NEG \| SPC_TM_REARM | 01000002h | | Trigger detection for negative edges on level 1. Trigger is armed when crossing level 0 to avoid false trigger on noise |
| SPC_TM_BOTH | 00000004h | | Trigger detection for positive and negative edges (any crossing of level 0) |
| SPC_TM_HIGH | 00000008h | | Trigger detection for HIGH levels (signal above level 0) |
| SPC_TM_LOW | 00000010h | | Trigger detection for LOW levels (signal below level 0) |
| SPC_TM_WINENTER | 00000020h | | Window trigger for entering area between level 0 and level 1 |
| SPC_TM_WINLEAVE | 00000040h | | Window trigger for leaving area between level 0 and level 1 |
| SPC_TM_INWIN | 00000080h | | Window trigger for signal inside window between level 0 and level 1 |
| SPC_TM_OUTSIDEWIN | 00000100h | | Window trigger for signal outside window between level 0 and level 1 |

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 main external trigger input for the OR mask |

Trigger Input Termination

The external trigger input is a high impedance input with 1 MOhm termination aginst GND. It is possible to program a 50 Ohm termination by software to terminate fast trigger signals correctly. If you enable the termination, please make sure, that your trigger source is capable to deliver the needed current. Please check carefully whether the source is able to fulfil the trigger input specification given in the technical data section.

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

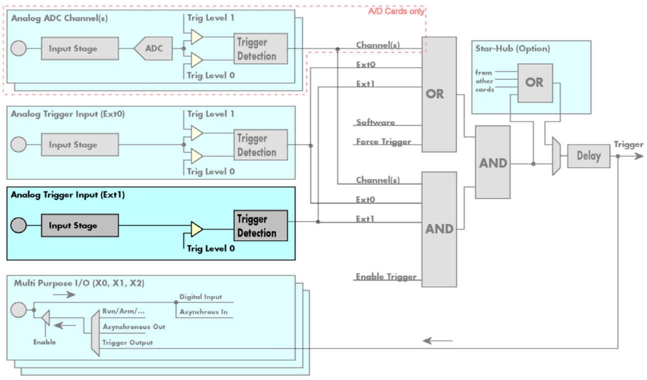
Please note that the signal levels will drop by 50% if using the 50 ohm termination and your source also has 50 ohm output impedance (both terminations will then work as a 1:2 divider). In that case it will be necessary to reprogram the trigger levels to match the new signal levels. In case of problems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

## Trigger Input Coupling

The external trigger input can be switched by software between AC and DC coupling. Please see the technical data section for details on the AC bandwidth.

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_EXT0_ACDC | 40120 | read/write | A „1“ sets the AC coupling for the external trigger input. A „0“ sets the DC coupling (default) |

# Secondary external level trigger (Ext1)




The M4i series has one secondary external trigger input consisting of an input stage with fixed 10 kOhm termination and one comparator that can be programmed in the range of +/- 10000 mV. Using one comparators offers a wide range of different logic levels for the available trigger modes that are support like edge, level.

The secondary external analog trigger can be easily combined with channel trigger or with the main external trigger being programmed as an additional external trigger input. The programming of the masks is shown in the chapters above.

## Trigger Mode

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

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_EXT1_AVAILMODES | 40501 | read | Bitmask showing all available trigger modes for external 0 (Ext0) = main analog trigger input |
| SPC_TRIG_EXT1_MODE | 40511 | read/write | Defines the external trigger mode for the external MMCX connector trigger input. The trigger need to be added to either OR or AND mask input to be activated. |
| SPC_TM_NONE | 00000000h | Channel is not used for trigger detection. This is as with the trigger masks another possibility for disabling channels. | |
| SPC_TM_POS | 00000001h | Trigger detection for positive edges (crossing level 0 from below to above) | |
| SPC_TM_NEG | 00000002h | Trigger detection for negative edges (crossing level 0 from above to below) | |
| SPC_TM_BOTH | 00000004h | Trigger detection for positive and negative edges (any crossing of level 0) | |
| SPC_TM_HIGH | 00000008h | Trigger detection for HIGH levels (signal above level 0) | |
| SPC_TM_LOW | 00000010h | Trigger detection for LOW levels (signal below level 0) | |

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

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_ORMASK | 40410 | read/write | Defines the OR mask for the different trigger sources. |
| SPC_TMASK_EXT1 | 4h | Enable secondary external trigger input for the OR mask | |

## Trigger level

All of the external (analog) trigger modes listed above require at least one trigger level to be set (except SPC_TM_NONE of course). Some like the window or the re-arm triggers require even two levels (upper and lower level) to be set. The meaning of the trigger levels is depending on the selected mode and can be found in the detailled trigger mode description that follows.

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

| Register | Value | Direction | Description | Range |
|---|---|---|---|---|
| SPC_TRIG_EXT_AVAIL0_MIN | 42340 | read | returns the minimum trigger level to be programmed in mV | |
| SPC_TRIG_EXT_AVAIL0_MAX | 42341 | read | returns the maximum trigger level to be programmed in mV | |
| SPC_TRIG_EXT_AVAIL0_STEP | 42342 | read | returns the step size of trigger level to be programmed in mV | |
| SPC_TRIG_EXT_AVAIL1_MIN | 42345 | read | returns the minimum trigger level to be programmed in mV | |
| SPC_TRIG_EXT_AVAIL1_MAX | 42346 | read | returns the maximum trigger level to be programmed in mV | |
| SPC_TRIG_EXT_AVAIL1_STEP | 42347 | read | returns the step size of trigger level to be programmed in mV | |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Trigger level 0 for external trigger Ext0 | -10000 mV to +10000 mV |
| SPC_TRIG_EXT0_LEVEL1 | 42330 | read/write | Trigger level 1 for external trigger Ext0 | -10000 mV to +10000 mV |
| SPC_TRIG_EXT1_LEVEL0 | 42321 | read/write | Trigger level 0 for external trigger Ext1 | -10000 mV to +10000 mV |

## Detailed description of the external analog trigger modes

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

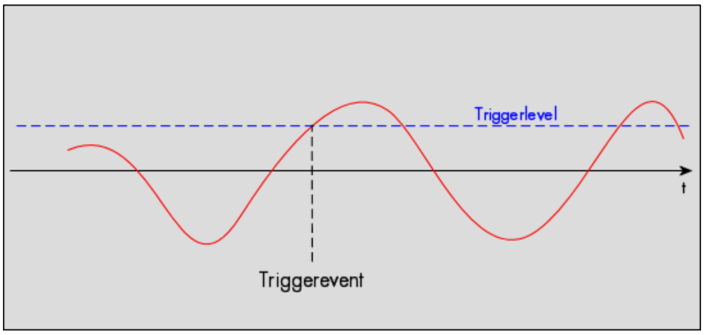
| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_ORMASK | 40410 | read/write | Defines the events included within the trigger OR mask of the card. |
| SPC_TRIG_ANDMASK | 40430 | read/write | Defines the events included within the trigger AND mask of the card. |
| SPC_TMASK_EXT0 | 2h | | Enables the main external (analog) trigger 0 for the mask. |
| SPC_TMASK_EXT1 | 4h | | Enables the secondary external (analog) trigger 0 for the mask. |

The following pages explain the available modes in detail. All modes that only require one single trigger level are available for both external trigger inputs. All modes that require two trigger levels are only available for the main external trigger input (Ext0).

### Trigger on positive edge

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

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

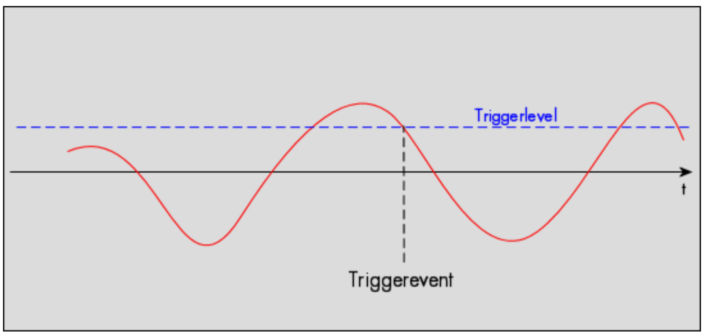



| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_POS | 1h |
| SPC_TRIG_EXT1_MODE | 40511 | read/write | SPC_TM_POS | 1h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the desired trigger level in mV | mV |

### Trigger on negative edge

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

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

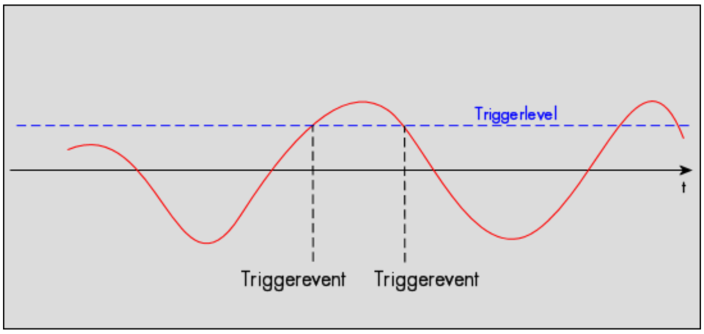



| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_NEG | 2h |
| SPC_TRIG_EXT1_MODE | 40511 | read/write | SPC_TM_NEG | 2h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the desired trigger level in mV | mV |

## Trigger on positive and negative edge

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

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

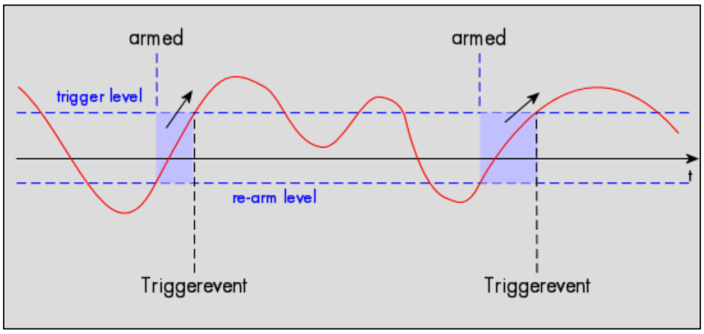



| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_BOTH | 4h |
| SPC_TRIG_EXT1_MODE | 40511 | read/write | SPC_TM_BOTH | 4h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the desired trigger level in mV | mV |

## Re-arm trigger on positive edge

The trigger 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 trigger signal from lower values to higher values (rising edge) then the trigger event 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.

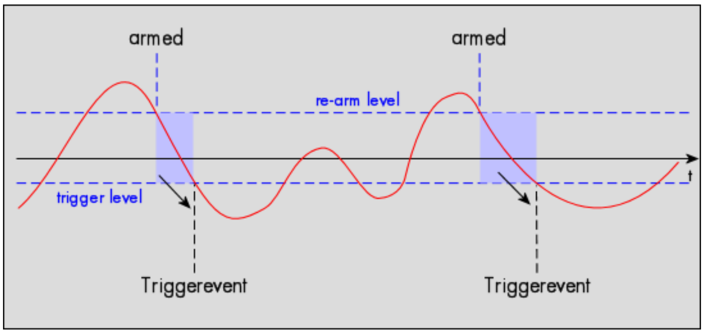



| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_POS \| SPC_TM_REARM | 01000001h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the desired trigger level in mV | mV |
| SPC_TRIG_EXT0_LEVEL1 | 42330 | read/write | Defines the re-arm level in mV | mV |

## Re-arm trigger on negative edge

The trigger 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 trigger signal from higher values to lower values (falling edge) then the trigger event will be detected and the trigger engine will be disarmed. A new trigger event is only detected, if the trigger engine is armed again.

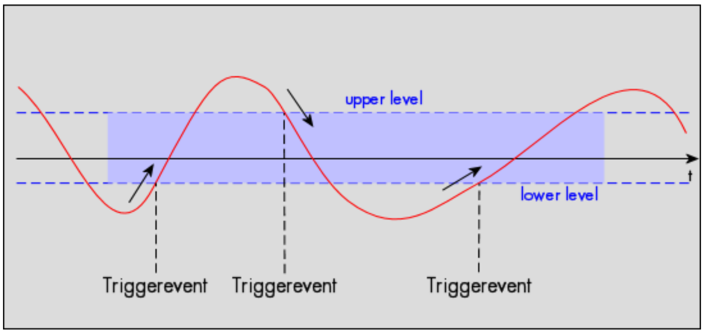
The re-arm trigger modes can be used to prevent the board from triggering on wrong edges in noisy signals.




| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_NEG \| SPC_TM_REARM | 01000002h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Defines the re-arm level in mV | mV |
| SPC_TRIG_EXT0_LEVEL1 | 42330 | read/write | Set it to the desired trigger level in mV | mV |

## Window trigger for entering signals

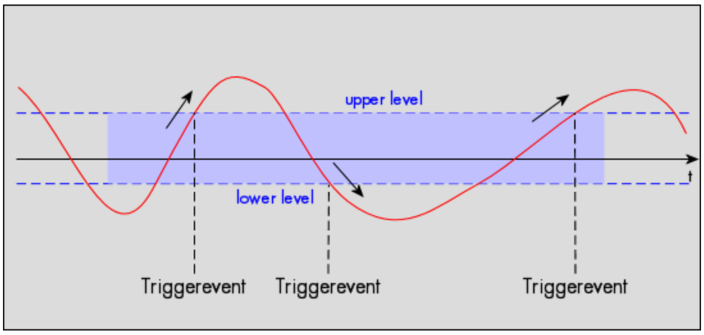
The trigger input is continuously sampled with the selected sample rate. The upper and the lower level define a window. Every time the signal enters the window from the outside, a trigger event will be detected.




| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_WINENTER | 00000020h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the upper trigger level in mV | mV |
| SPC_TRIG_EXT0_LEVEL1 | 42330 | read/write | Set it to the lower trigger level in mV | mV |

## Window trigger for leaving signals

The trigger 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 trigger event will be detected.

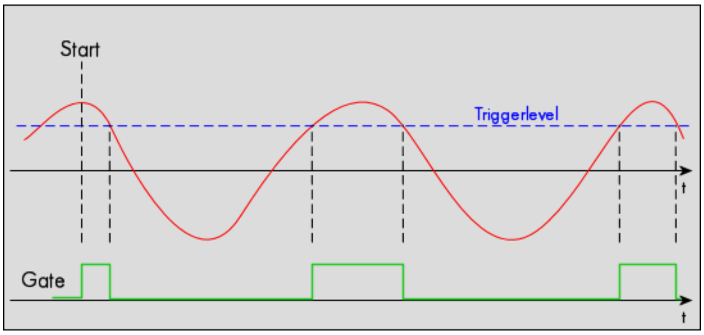



| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_WINLEAVE | 00000040h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the upper trigger level in mV | mV |
| SPC_TRIG_EXT0_LEVEL1 | 42330 | read/write | Set it to the lower trigger level in mV | mV |

## High level trigger

This trigger mode will generate an internal gate signal that can be useful in conjunction with a second trigger mode to gate that second trigger. If using this mode as a single trigger source the card will detect a trigger event at the time when entering the high level (acting like positive edge trigger) or if the trigger signal is already above the programmed level at the start it will immediately detect a trigger event.

The trigger input is continuously sampled with the selected sample rate. The trigger event will be detected if the trigger input is above the programmed trigger level.

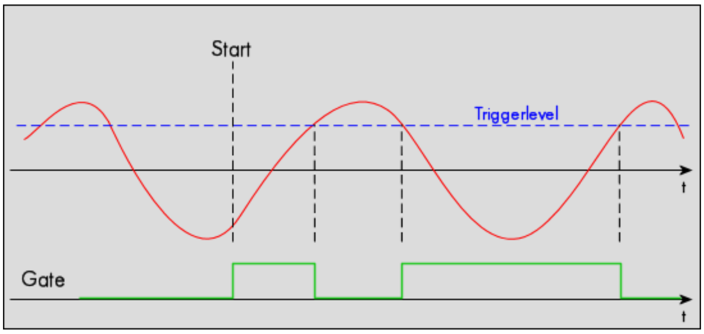



| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_HIGH | 00000008h |
| SPC_TRIG_EXT1_MODE | 40511 | read/write | SPC_TM_HIGH | 00000008h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the upper trigger level in mV | mV |

## Low level trigger

This trigger mode will generate an internal gate signal that can be useful in conjunction with a second trigger mode to gate that second trigger. If using this mode as a single trigger source the card will detect a trigger event at the time when entering the low level (acting like negative edge trigger) or if the trigger signal is already above the programmed level at the start it will immediately detect a trigger event.

The trigger input is continuously sampled with the selected sample rate. The trigger event will be detected if the trigger input is below the programmed trigger level.

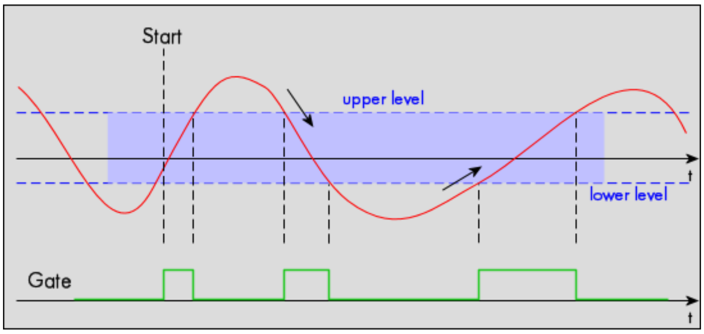



| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_LOW | 00000010h |
| SPC_TRIG_EXT1_MODE | 40511 | read/write | SPC_TM_LOW | 00000010h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the upper trigger level in mV | mV |

## In window trigger

This trigger mode will generate an internal gate signal that can be useful in conjunction with a second trigger mode to gate that second trigger. If using this mode as a single trigger source the card will detect a trigger event at the time when entering the window defined by the two trigger levels (acting like window enter trigger) or if the trigger signal is already inside the programmed window at the start it will immediately detect a trigger event.

The trigger input is continuously sampled with the selected sample rate. The trigger event will be detected if the trigger input is inside the programmed trigger window.

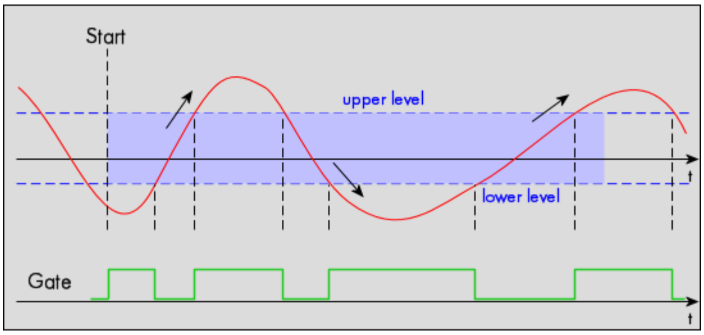



| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_INWIN | 00000080h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the upper trigger level in mV | mV |
| SPC_TRIG_EXT0_LEVEL1 | 42330 | read/write | Set it to the lower trigger level in mV | mV |

## Outside window trigger

This trigger mode will generate an internal gate signal that can be useful in conjunction with a second trigger mode to gate that second trigger. If using this mode as a single trigger source the card will detect a trigger event at the time when leaving the window defined by the two trigger levels (acting like leaving window trigger) or if the trigger signal is already outside the programmed windo at the start it will immediately detect a trigger event.

The trigger input is continuously sampled with the selected sample rate. The trigger event will be detected if the trigger input is outside the programmed trigger window.



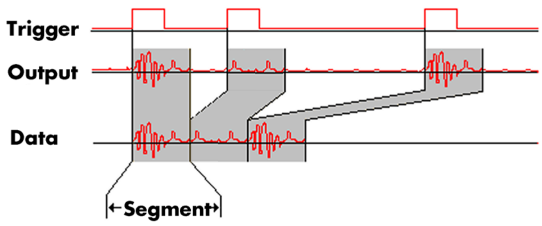

| Register | Value | Direction | set to | Value |
|---|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | SPC_TM_OUTSIDEWIN | 00000100h |
| SPC_TRIG_EXT0_LEVEL0 | 42320 | read/write | Set it to the upper trigger level in mV | mV |
| SPC_TRIG_EXT0_LEVEL1 | 42330 | read/write | Set it to the lower trigger level in mV | mV |

# Mode Multiple Replay

The Multiple Replay mode allows the generation of data blocks with multiple trigger events without restarting the hardware.

The on-board memory will be divided into several segments of the same size. On each trigger event one segment of data will be replayed.

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 replayed with each trigger event.

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_SEGMENTSIZE | 10010 | read/write | Size of one Multiple Replay segment: the total number of samples to be replayed after detection of one trigger event including the time recorded before the trigger (pre trigger). |

# 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.

# Programming examples

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

```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REP_STD_MULTI);  // Enables Standard Multiple Replay

spcm_dwSetParam_i32 (hDrv, SPC_SEGMENTSIZE, 1024);         // Set the segment size to 1024 samples
spcm_dwSetParam_i32 (hDrv, SPC_MEMSIZE,     4096);        // Set the total memsize for recording to 4096 samples
                                                           // so that actually four segments will be replayed

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 Replay in FIFO mode.

```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REP_FIFO_MULTI); // Enables FIFO Multiple Replay

spcm_dwSetParam_i32 (hDrv, SPC_SEGMENTSIZE,    2048);          // Set the segment size to 2048 samples
pcm_dwSetParam_i32 (hDrv, SPC_LOOPS            256);          // 256 segments will be replayed

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
```

# Replay modes

## Standard Mode

With every detected trigger event one data block is replayed. The length of one multiple replay segment is set by the value of the segment size register SPC_SEGMENTSIZE. The total amount of samples to be replayed 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 replay mode please refer to the according chapter earlier in this manual.

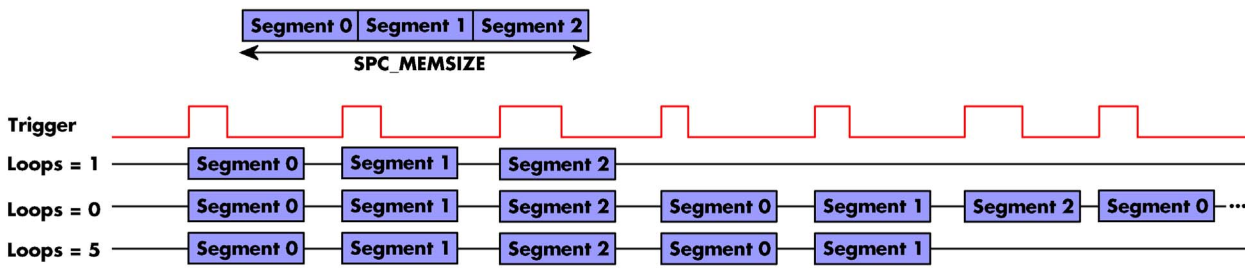
| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_CARDMODE | | 9500 | read/write | Defines the used operating mode |
| | SPC_REP_STD_MULTI | 200h | Enables Multiple Replay for standard replay. | |

The total number of samples to be replayed from the on-board memory in standard mode is defined by the SPC_MEMSIZE register. When using the SPC_LOOPS parameter one can further program whether all segments should be replayed once or continuously or whether a dedicated number of segments should be replayed

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_MEMSIZE | 10000 | read/write | Defines the total number of samples to be replayed. |
| SPC_LOOPS | 10020 | read/write | When writing a 1 the complete memory is replayed once, when writing a zero the replay continues from the beginning forever. When writing a number >1 this number of segments is replayed until the card stops automatically. |

| | |
|---|---|
| 0 | Replay will be infinite until the user stops it. When replay reaches the end of programmed memory it will start from the beginning again. |
| 1 | The complete memory is replayed once. |
| 2 … [4G - 1] | Defines the number of segments to be replayed. After replaying this number of segments the card will stop automatically. |

**Replay modes with the use of SPC_LOOPS**




## FIFO Mode

The Multiple Replay in FIFO mode is similar to the Multiple Replay in standard mode. In contrast to the standard mode it is not necessary to program the number of samples to be replayed. The replay is running until the user stops it. The data is written block by block by the driver as described under single FIFO mode example earlier in this manual. These blocks can be online calculated or loaded from hard disk. This mode significantly reduces the amount of data to be transfered on the PCI bus as gaps with no significant output did not have to be transferred. This enables you to use faster sample rates then you would be able to in FIFO mode without Multiple Recording.
The table below shows the dedicated register for enabling Multiple Replay. 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_REP_FIFO_MULTI | 1000h | | Enables Multiple Replay for FIFO mode. |

The number of segments to be replayed 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 replayed |
| 0 | | | Replay will be infinite until the user stops it. |
| 1 … [4G - 1] | | | Defines the total segments to be replayed. |

# Limits of segment size, 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 2 bytes of memory to be stored.

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 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 | | | Segment size SPC_SEGMENTSIZE | | | Loops SPC_LOOPS | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Min | Max | Step | Min | Max | Step | Min | Max | Step |
| 1 channel | Standard Single | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Single Restart | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Standard Multi | 32 | Mem | 32 | 16 | Mem/2 | 16 | | not used | |
| | Standard Gate | 32 | Mem | 32 | | not used | | | not used | |
| | FIFO Single | | not used | | 16 | 8G - 16 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Multi | | not used | | 16 | Mem/2 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Gate | | not used | | | not used | | 0 (∞) | 4G - 1 | 1 |
| 2 channels | Standard Single | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Single Restart | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Standard Multi | 32 | Mem | 32 | 16 | Mem/4 | 16 | | not used | |
| | Standard Gate | 32 | Mem | 32 | | not used | | | not used | |
| | FIFO Single | | not used | | 16 | 8G - 16 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Multi | | not used | | 16 | Mem/4 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Gate | | not used | | | not used | | 0 (∞) | 4G - 1 | 1 |
| 4 channels | Standard Single | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Single Restart | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Standard Multi | 32 | Mem | 32 | 16 | Mem/8 | 16 | | not used | |

| Activated Channels | Used Mode | Memory size SPC_MEMSIZE | | | Segment size SPC_SEGMENTSIZE | | | Loops SPC_LOOPS | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Min | Max | Step | Min | Max | Step | Min | Max | Step |
| | Standard Gate | 32 | Mem | 32 | not used | | | not used | | |
| | FIFO Single | not used | | | 16 | 8G - 16 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Multi | not used | | | 16 | Mem/8 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Gate | not used | | | 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 2 GSample** |
|---|---|
| Mem | 2 GSample |
| Mem / 2 | 1 GSample |
| Mem / 4 | 512 MSample |
| Mem / 8 | 256 MSample |

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

## Programming the behaviour in pauses and after replay

Usually the used outputs of the analog generation boards are set to zero level after replay. This is in most cases adequate. In some cases it can be necessary to hold the last sample, to output the maximum positive level or maximum negative level after replay. The stoplevel will stay on the defined level until the next output has been made. With the following registers you can define the behaviour after replay:

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_CH0_STOPLEVEL | | 206020 | read/write | Defines the behavior after replay for channel 0 |
| SPC_CH1_STOPLEVEL | | 206021 | read/write | Defines the behavior after replay for channel 1 |
| SPC_CH2_STOPLEVEL | | 206022 | read/write | Defines the behavior after replay for channel 2 |
| SPC_CH3_STOPLEVEL | | 206023 | read/write | Defines the behavior after replay for channel 3 |
| | SPCM_STOPLVL_ZERO | 16 | Defines the analog output to enter zero level (D/A converter is fed with digital zero value) | |
| | SPCM_STOPLVL_LOW | 2 | Defines the analog output to enter maximum negative level (D/A converter is fed with most negative level) | |
| | SPCM_STOPLVL_HIGH | 4 | Defines the analog output to enter maximum positive level (D/A converter is fed with most positive level) | |
| | SPCM_STOPLVL_HOLDLAST | 8 | Holds the last replayed sample on the analog output | |

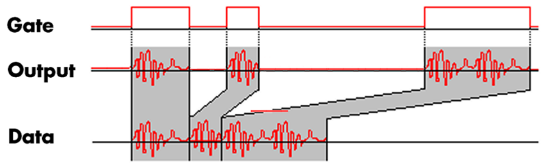
All outputs that are not activated for replay, will keep the programmed stoplevel also while the replay is in progress.

# Mode Gated Replay

The Gated Replay mode allows the data generation controlled by an external or an internal gate signal. Data will only be replayed if the programmed gate condition is true.

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

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




# Generation Modes

## Standard Mode

Data will be replayed as long as the gate signal fulfils the programmed gate condition. At the end of the gate interval the replay will be stopped and the card will pause until another gates signal appears. If loops (SPC_LOOPS) is set to 1 the card stops immediately as soon as the total amount of data (SPC_MEMSIZE) has been replayed. In that case the last gate segment is ended by the expiring memory size counter and not by the gate end signal. If loops is set to zero the Gated Replay mode will run in a continuous loop until explicitly stopped by user. If the replay reaches the end of the programmed memory it will start again at the beginning with no gap in between. If setting loops to a number larger than 1 this number of complete gates will be replayed and the card stopped afterwards automatically.

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.

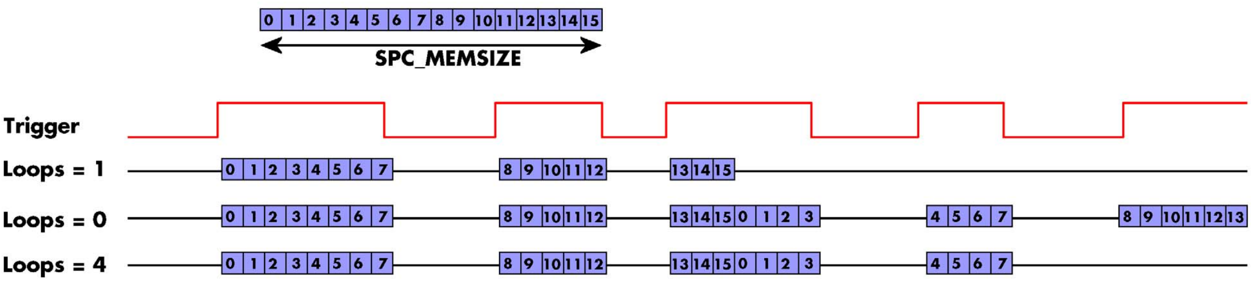
| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_CARDMODE | 9500 | read/write | Defines the used operating mode |
| SPC_REP_STD_GATE | 400h | | Enables Gated Sampling for standard acquisition. |

The total number of samples to be replayed from 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 replayed. |
| SPC_LOOPS | 10020 | read/write | Defines the number of gates to be replayed |
| 0 | | | Replay will be infinite until the user stops it. When replay reaches the end of programmed memory it will start from the beginning with no gap. |
| 1 | | | The complete memory is replayed once. The last gate segement is cut off when end of memory is reached. |
| 2 … [4G - 1] | | | Defines the number of gate segments to be replayed. |

## Examples of Standard Gated Replay with the use of SPC_LOOPS parameter

To keep the diagram easy to read there’s no delay shown in here and there’s also only a very small number of samples shown. Any further restrictions are described later in this chapter.




## FIFO Mode

The Gated Replay in FIFO mode is similar to the Gated Replay in standard mode. The replay can either run until the user stops it by software (infinite replay, loops = 0) or until a programmed number of gates has been played (loops = 1). The data is written continuously by the driver and can be either online calculated or loaded from 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_REP_FIFO_GATE | 2000h | | Enables Gated Replay with FIFO mode |

The number of gates to be replayed 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 replayed |
| 0 | | | Replay will be infinite until the user stops it or an underrun occurs |
| 1 … [4G - 1] | | | Defines the total gates to be replayed. |

# Limits of segment size, 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 2 bytes of memory to be stored.

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 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 | | | Segment size SPC_SEGMENTSIZE | | | Loops SPC_LOOPS | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Min | Max | Step | Min | Max | Step | Min | Max | Step |
| 1 channel | Standard Single | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Single Restart | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Standard Multi | 32 | Mem | 32 | 16 | Mem/2 | 16 | | not used | |
| | Standard Gate | 32 | Mem | 32 | | not used | | | not used | |
| | FIFO Single | | not used | | 16 | 8G - 16 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Multi | | not used | | 16 | Mem/2 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Gate | | not used | | | not used | | 0 (∞) | 4G - 1 | 1 |
| 2 channels | Standard Single | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Single Restart | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Standard Multi | 32 | Mem | 32 | 16 | Mem/4 | 16 | | not used | |
| | Standard Gate | 32 | Mem | 32 | | not used | | | not used | |
| | FIFO Single | | not used | | 16 | 8G - 16 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Multi | | not used | | 16 | Mem/4 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Gate | | not used | | | not used | | 0 (∞) | 4G - 1 | 1 |
| 4 channels | Standard Single | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Single Restart | 32 | Mem | 32 | | not used | | 0 (∞) | 4G - 1 | 1 |
| | Standard Multi | 32 | Mem | 32 | 16 | Mem/8 | 16 | | not used | |
| | Standard Gate | 32 | Mem | 32 | | not used | | | not used | |
| | FIFO Single | | not used | | 16 | 8G - 16 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Multi | | not used | | 16 | Mem/8 | 16 | 0 (∞) | 4G - 1 | 1 |
| | FIFO Gate | | not used | | | 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 2 GSample** |
|---|---|
| Mem | 2 GSample |
| Mem / 2 | 1 GSample |
| Mem / 4 | 512 MSample |
| Mem / 8 | 256 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.

# Allowed trigger modes

## 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:

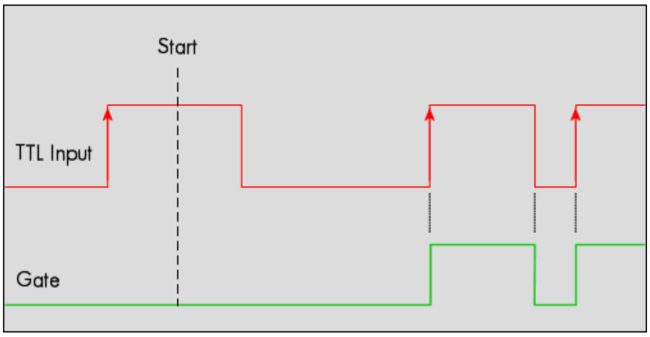
| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_ORMASK | 40410 | read/write | Defines the OR mask for the different trigger sources. |
| SPC_TMASK_EXT0 | 2h | | Enable external trigger input for the OR mask |
| SPC_TMASK_XIO0 | 100h | | Enable extra TTL input 0 for the OR mask. On plain cards this input is only available if the option BaseXIO is installed. As part of the digitizerNETBOX this input is available as connector Trigger B. |
| SPC_TMASK_XIO1 | 200h | | Enable extra TTL input 1 for the OR mask. These trigger inputs are only available, when option BaseXIO is installed. |

## 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.



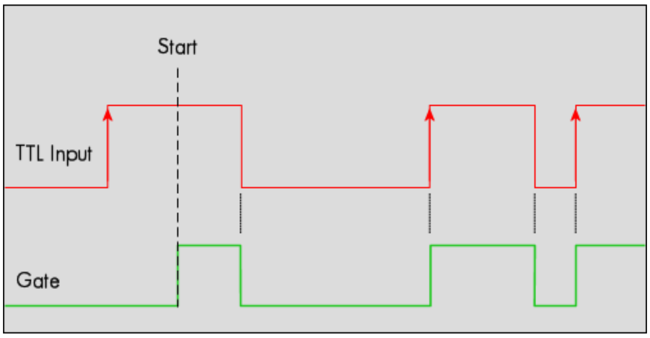

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | Sets the external trigger mode for the board |
| SPC_TM_POS | 1h | | Sets the trigger mode for external TTL trigger to detect positive edges |

## 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.



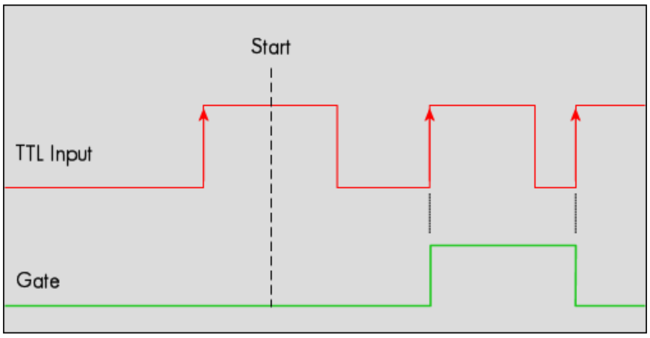

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | Sets the external trigger mode for the board |
| SPC_TM_HIGH | 8h | | Sets the trigger mode for external TTL trigger to detect high levels. |

## 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.



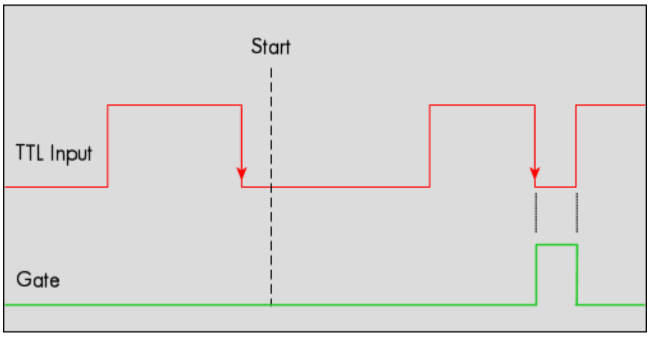

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | Sets the external trigger mode for the board |
| SPC_TM_POS \| SPC_TM_DOUBLEEDGE | 08000001h | | Sets the gate mode for external TTL trigger to start and stop on positive edges. |

## 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.



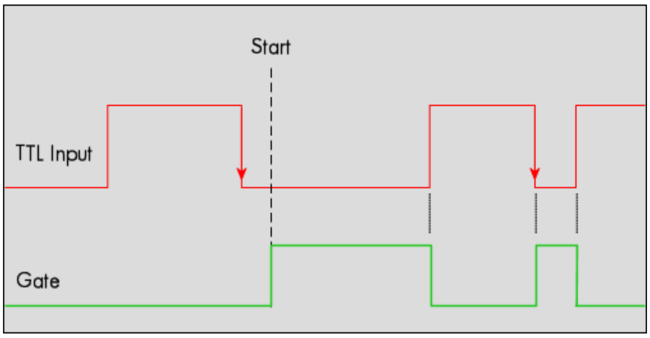

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | Sets the external trigger mode for the board |
| SPC_TM_NEG | 2h | Sets the trigger mode for external TTL trigger to detect negative edges. | |

## 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.



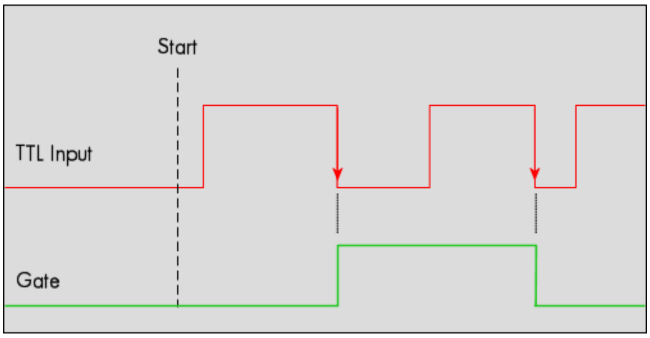

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_EXT0_MODE | 40510 | read/write | Sets the external trigger mode for the board |
| SPC_TM_LOW | 10h | Sets the trigger mode for external TTL trigger to detect low levels. | |

## 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:

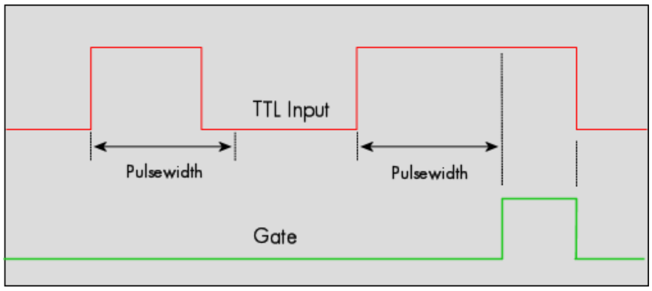
| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_ORMASK | 40410 | read/write | Defines the OR mask for the different trigger sources. |
| SPC_TMASK_EXT0 | 2h | | Enable external trigger input for the OR mask |
| SPC_TMASK_XIO0 | 100h | | Enable extra TTL input 0 for the OR mask. On plain cards this input is only available if the option BaseXIO is installed. As part of the digitizerNETBOX this input is available as connector Trigger B. |
| SPC_TMASK_XIO1 | 200h | | Enable extra TTL input 1 for the OR mask. These trigger inputs are only available, when option BaseXIO is installed. |

### 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 pulsewidth, 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.



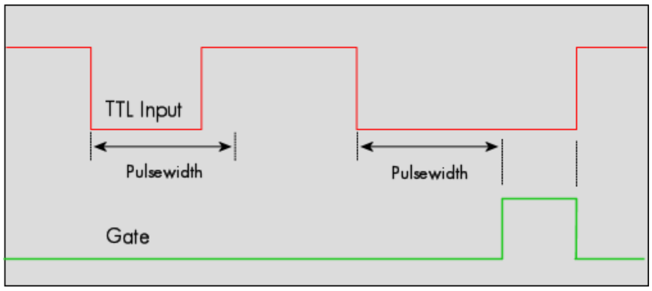

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_TRIG_EXT0_PULSEWIDTH | 44210 | read/write | Sets the pulsewidth in samples. Values from 2 to 65535 are allowed. |
| SPC_TRIG_EXT0_MODE | 40510 | read/write | Sets the trigger mode for the board. |
| (SPC_TM_POS \| SPC_TM_PW_GREATER) | 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 pulsewidth, 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
```

## Programming examples

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

```
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REP_STD_GATE);  // Enables Standard Gated Replay

spcm_dwSetParam_i32 (hDrv, SPC_MEMSIZE, 8192);              // Set the total memsize for replay 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_REP_FIFO_GATE); // Enables FIFO Gated Replay

pcm_dwSetParam_i32 (hDrv, SPC_LOOP,    1024);              // 1024 gates will be replayed

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
```

## Programming the behaviour in pauses and after replay

Usually the used outputs of the analog generation boards are set to zero level after replay. This is in most cases adequate. In some cases it can be necessary to hold the last sample, to output the maximum positive level or maximum negative level after replay. The stoplevel will stay on the defined level until the next output has been made. With the following registers you can define the behaviour after replay:

| Register | | Value | Direction | Description |
|---|---|---|---|---|
| SPC_CH0_STOPLEVEL | | 206020 | read/write | Defines the behavior after replay for channel 0 |
| SPC_CH1_STOPLEVEL | | 206021 | read/write | Defines the behavior after replay for channel 1 |
| SPC_CH2_STOPLEVEL | | 206022 | read/write | Defines the behavior after replay for channel 2 |
| SPC_CH3_STOPLEVEL | | 206023 | read/write | Defines the behavior after replay for channel 3 |
| | SPCM_STOPLVL_ZERO | 16 | Defines the analog output to enter zero level (D/A converter is fed with digital zero value) | |
| | SPCM_STOPLVL_LOW | 2 | Defines the analog output to enter maximum negative level (D/A converter is fed with most negative level) | |
| | SPCM_STOPLVL_HIGH | 4 | Defines the analog output to enter maximum positive level (D/A converter is fed with most positive level) | |
| | SPCM_STOPLVL_HOLDLAST | 8 | Holds the last replayed sample on the analog output | |

All outputs that are not activated for replay, will keep the programmed stoplevel also while the replay is in progress.

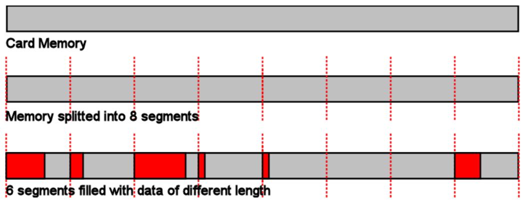
# Sequence Replay Mode

The sequence replay mode is a special firmware mode that allows to program an output sequence by defining one or more sequences each associated with a certain memory pattern. Therefore the user is provided with two different memories, one for the sequence steps and one for the data patterns. The separated sequence memory can hold different sequence steps (the actual number depends on the hardware and can be found in the technical data section). Each step itself contains information about how often it should be repeated in a loop, which step will be next and on what condition the change will happen. To define the pattern for the steps, the on-board memory is split up into several segments of different length. The switch over from one segment to the other is seamless, without any missing samples. The powerful sequence mode option adds a huge variety of different application areas to Spectrum's generator cards.

## Theory of operation

### Define segments in data memory

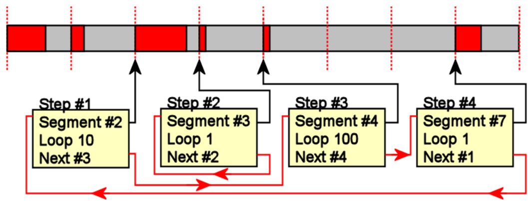
The complete installed on-board memory of the card is divided into a user definable number of segments. Each segment space has the same length limiting the maximum length of one data segment to [Installed Memory] / [Number of Segments]. Each data segment can be filled by the user with patterns of different lengths or can even be left completely empty if unused:




In our example we see the complete installed card memory is being split into 8 segments and 6 of these segments are actually filled with data sequences of different length afterwards (indicated in red). Two of these segments are not needed for the assumed sequence and therefore left empty as an example. Due to the fact that each sequence step can be associated with any of the data segments, it is also possible to use one data segment in multiple steps or to just once upload the data for multiple sequences, and just change the order of the sequence.

### Define steps in sequence memory

The sequence memory defines a number of data loop steps that are executed step by step either linear or interrupted by waiting for trigger event. The first step that is entered after a card start is separately defined by software. When being entered, each step first repeats the associated data segment the number times defined by its loop parameter. Afterwards the sequencer will either automatically proceed either unconditionally or check for a trigger event as a condition to change over to the next step, which is defined by the steps next parameter. This next segment can be the same segment again performing an endless loop or the beginning of the sequence to repeat the sequence until being stopped by the user. Additionally a step can also be defined to be the last step in a sequence such that the card is stopped afterwards.




In our example 4 steps have been defined. Three of them (Step #1, Step #3, Step #4) perform an endless loop that will be repeated continuously. The output of the card will then be 10 times data segment #2, 100 times data segment #4, 1 time data segment #7 and then starting over with 10 times data segment #2 and so on...
In this first simple example the sequence consisting of the three steps is once defined prior to the card start and not changed during runtime, therefore the shown Step #2 is not used here. There will be an extra passage later, that shows how the sequence memory can be updated or modified even during runtime, whilst the replay is in progress.

## Programming

Programming of the sequence mode is done using the known driver interface with the addition of a few new registers.

### Gathering information

If the sequence mode is installed on the card, the different details and limits of the sequence programming can be read out:

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_PCIFEATURES | 2120 | read only | PCI feature register. Holds the installed features and options as a bit field. The return value must be masked out with one of the masks below to get information about one certain feature. |
| SPCM_FEAT_SEQUENCE | 1000h | Replay sequence mode available (only available for arbitrary generator and digital I/O cards). | |

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_SEQMODE_AVAILMAXSEGMENT | 349900 | read only | Returns the maximum number of segments the memory can be divided into. Please note that only dividers with a power of 2 are possible return values. |
| SPC_SEQMODE_AVAILMAXSTEPS | 349901 | read only | Returns the maximum number of sequence steps that can be used on this card. |
| SPC_SEQMODE_AVAILMAXLOOP | 349902 | read only | Returns the maximum number of loops that can be programmed for a step. |
| SPC_SEQMODE_AVAILFEATURES | 349903 | read only | Returns the available features for each sequence step as shown below: |
| SPCSEQ_ENDLOOPONTRIG | 40000000h | The step runs endless until a trigger is received. If no trigger has been detected, the step will enter itself again, counting down its own loops and check for a trigger again. For a minimum reaction time on an external trigger event it is good practice to set the loop parameter to 1 in the step checking for the trigger. | |

| | | |
|---|---|---|
| SPCSEQ_END | 80000000h | This sequence step is the end of the sequence. The card is stopped at the end of this segment after the loop counter has reached his end. |

# Setting up the registers

## Define the card mode

To enable the sequencer the card mode needs to be set appropriately first:

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_CARDMODE | 9500 | read/write | Defines the used operating mode. |
| SPC_REP_STD_SEQUENCE | 40000h | | Data generation from on-board memory, by splitting the memory into several segments and replaying the data using a programmable order coming from a special sequence memory. |

## Prepare the data memory

Setting up the segmentation of the on-board data memory is done by using the following registers:

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_SEQMODE_MAXSEGMENTS | 349910 | read/write | Programs the number of segments the on-board memory should be divided into. If changing the number of segments all information that has been stored before is lost and all sequence data and all sequence setup has to be written again. Only a power of two is allowed, but not all of the segments must be actually used in the sequence.<br>If reading this register the number of segments the memory is currently divided into is returned. |
| SPC_SEQMODE_WRITESEGMENT | 349920 | read/write | Defines the current segment to be addressed by the user. Must be programmed prior to changing any segment parameters. |
| SPC_SEQMODE_SEGMENTSIZE | 349940 | read/write | Defines the number of valid/to be replayed samples for the current selected memory segment. |

Due to the internal organization of the card memory there is a certain minimum, maximum and stepsize when setting the segmentsize for the sequence memory. The following table gives you an overview of all limits. 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:

## For analog waveform generator (D/A) cards

| Activated Channels | For cards with 16 bit converter resolution Pattern size for register SPC_SEQMODE_SEGMENTSIZE | | |
|---|---|---|---|
| | Min | Max | Step |
| 1 channel | 384 | (Mem/1) / SPC_SEQMODE_MAXSEGMENTS) | 32 |
| 2 channels | 192 | (Mem/2) / SPC_SEQMODE_MAXSEGMENTS) | 32 |
| 4 channels | 96 | (Mem/4) / SPC_SEQMODE_MAXSEGMENTS) | 32 |

## Definition of the transfer buffer

The data transfer itself is done using the standard data transfer commands, with the exception that the buffer type and the direction is fixed in combination with the sequence mode. 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,              // fixed SPCM_BUF_DATA (segment memory is always in on-board memory)
    uint32      dwDirection,            // fixed SPCM_DIR_PCTOCARD (only available for replay cards)
    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 relation to the currently selected segment
    uint64      qwTransferLen);         // buffer length for the currently selected segment
```

The programming examples further below will show the setup and also some examples of data transfer.

## Set up the sequence memory

Sequence steps are programmed using a dedicated register for each step. Please note that the register has to be written with 64 bit of data to cover all settings. It is possible to either use raw 64 bit access or multiplexed 64 bit access (2 times 32 bit data). The masks mentioned in the table below are 32 bit masks only, so that they can be used for 64 bit and 32 bit accesses.

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_SEQMODE_STEPMEM0 | 340000 | read/write | First address (sequence step 0) of the 64 bit organized sequence memory. |
| ... | ... | ... | ... |
| SPC_SEQMODE_STEPMEM0 + 4095 | 344095 | read/write | Writes the sequence step 4095, as an example. The maximum number of steps should be read out by using the SPC_SEQMODE_AVAILMAXSTEPS register as described above. |
| **Lower 32 bit:** | | | |
| SPCSEQ_SEGMENTMASK | 0000FFFFh | | Associates the current sequence step with one of the memory segments. |
| SPCSEQ_NEXTSTEPMASK | FFFF0000h | | Defines the next step in the sequence. |
| **Upper 32 bit:** | | | |
| SPCSEQ_LOOPMASK | 000FFFFFh | | Defines how often the memory segment associated with the current step will be repeated before the next step condition will be evaluated. |

| | | |
|---|---|---|
| SPCSEQ_ENDLOOPALWAYS | 0h | Unconditionally change to the next step, if defined loops for the current segment have been replayed. |
| SPCSEQ_ENDLOOPONTRIG | 40000000h | Feature flag that marks the step to conditionally change to the next step on a trigger condition. The occurrence of a trigger event is repeatedly checked each time the defined loops for the current segment have been replayed. A temporary valid trigger condition will be stored until evaluation at the end of the step. |
| SPCSEQ_END | 80000000h | Feature flag that marks the current step to be the last in the sequence. The card is stopped at the end of this segment after the loop counter has reached his end. |

The start step register allows to define which of the set up steps is used first after card start. Therefore is possible to upload multiple sequences prior to the start and switch between these sequences by using a simple command, setting a different starting point:

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_SEQMODE_STARTSTEP | 349930 | read/write | Defines which of all defined steps in the sequence memory will be used first directly after the card start. |

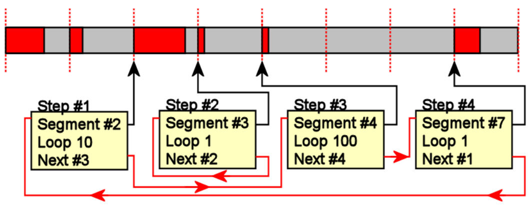
### Read out the currently replayed sequence step

In case one wants to change the sequence on the fly or one needs to know which part of the sequence is currently replayed it is possible to read out the sequence step number that is currently at the output connector of the card. This could be extremely useful if external equipm,ent has to be changed after a dedicated sequence has been replayed or if the AWG is changing between different patterns in automatic test environment.

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_SEQMODE_STATS | 349950 | read | Number of the sequence step that is currently replayed. |

## Changing sequences or step parameters during runtime

Due to the strict separation of the two memory areas it is also possible to change the sequence memory during runtime. If we look again on the example sequence below, we can see that there is an unused step #2:




In our example 3 steps have been defined, prior to the card start, and these at first are not changed. Additionally Step#2 is set up to repeat itself, but due to the defined start step it is normally not used. Due to the nature of the sequence memory (read-before-write) it is possible to write to any step register in the sequence memory during runtime without corrupting the sequence memory. By addressing a certain step and changing for example its next parameter, it is possible switch between two sequences by software. Because the user does not know what sequence is currently replayed, one cannot leave the „current" step but instead has to address one certain step and therefore defines an exit/change state.

Assuming in the example above, that we change the next parameter of Step#4 from Next=1 to Next=2, the infinitely executed 3-step sequence that is used as default after card start will be left the next time that the replay finishes the last sample of the pattern associated with Step#4 (which in this case is Segment#7), will then jump to step #2 and seamlessly continue replaying with the first sample off the associated segment #3. As step #2 links back to itself it will generate data segment #3 in an endless loop until being stopped by a software command.

Any of the three step parameters „Next", „Segment" and „Loop" of any step in the sequence memory can be changed during runtime, without corruption the sequence memory. However once a step is entered, it will first execute the current parameters such as replay the associated pattern and repeating it the programmed number of times.

## Changing data patterns during runtime

In addition to the possible runtime changes within the sequence memory as described above, it is also possible to change the parts of the pattern memory.

**However since the data memory's nature is not „read-before-write", the user must take care not to change the content of the memory segments, which are used within the currently active sequence.**

Changing the data pattern can be usefull in applications, where the data for the next test needs to be updated based on results from the currently running test. Remember to update the sequence step entries if the segment length has changed.

# Synchronization

**Please note that the sequence mode is NOT synchronized using the star-hub. Using sequence mode together with star-hub, it is still possible to synchronize the clock and the start of the cards. However it is neither possible to synchronize any changes inside the step memory nor to synchronize software commands that change the step memory order.**

# Programming example

The following example shows a very simple sequence as an example. Only two segments are used, the first is replayed 10 times and then unconditionally left and replay switches over to the second segment. This segment is repeated until a trigger event is detected by the card. After the trigger has been detected the sequence starts over again... until the card is stopped.

```
// Setup of channel enable, output conditioning as well as trigger setup not shown for simplicity

#define MAX_SEGMENTS         2 // only 2 segments used here for simplicity
int32 lBytesPerSample;

// Read out used bytes per sample
spcm_dwGetParam_i32 (hDrv, SPC_MIINST_BYTESPERSAMPLE, &lBytesPerSample);

// Setting up the card mode
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REP_STD_SEQUENCE); // enable sequence mode
spcm_dwSetParam_i32 (hDrv, SPC_SEQMODE_MAXSEGMENTS,         2); // Divide on-board mem in two parts
spcm_dwSetParam_i32 (hDrv, SPC_SEQMODE_STARTSTEP,           0); // Step#0 is the first step after card start

// Setting up the data memory and transfer data
spcm_dwSetParam_i32 (hDrv, SPC_SEQMODE_WRITESEGMENT,   0); // set current configuration switch to segment 0
spcm_dwSetParam_i32 (hDrv, SPC_SEQMODE_SEGMENTSIZE, 1024); // define size of current segment 0

// it is assumed, that the Buffer memory has been allocated and is already filled with valid data
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_PCTOCARD, 0, pData, 0, 1024 * lBytesPerSample);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

// Setting up the data memory and transfer data
spcm_dwSetParam_i32 (hDrv, SPC_SEQMODE_WRITESEGMENT,   1); // set current configuration switch to segment 1
spcm_dwSetParam_i32 (hDrv, SPC_SEQMODE_SEGMENTSIZE,  512); // define size of current segment 1

// it is assumed, that the Buffer memory has been allocated and is already filled with valid data
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_PCTOCARD, 0, pData, 0, 512 * lBytesPerSample);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

// Setting up the sequence memory (Only two steps used here as an example)
lStep =       0;                    // current step is Step#0
llSegment =   0;                    // associated with memory
llLoop =     10;                    // Pattern will be repeated 10 times
llNext =      1;                    // Next step is Step#1
llCondition = SPCSEQ_ENDLOOPALWAYS; // Unconditionally leave current step

// combine all the parameters to one int64 bit value
llValue =   (llCondition << 32) | (llLoop << 32) | (llNext << 16) | (llSegment);
spcm_dwSetParam_i64 (hDrv, SPC_SEQMODE_STEPMEM0 + lStep, llValue);

lStep =       1;                    // current step is Step#1
llSegment =   1;                    // associated with memory segment 1
llLoop =      1;                    // Pattern will be repeated once before condition is checked
llNext =      0;                    // Next step is Step#0
llCondition = SPCSEQ_ENDLOOPNTRIG;  // Repeat current step until a trigger has occurred

llValue =   (llCondition << 32) | (llLoop << 32) | (llNext << 16) | (llSegment);
spcm_dwSetParam_i64 (hDrv, SPC_SEQMODE_STEPMEM0 + lStep, llValue);

// Start the card
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);

// ... wait here or do something else ...

// Stop the card
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_STOP);
```

# Option Star-Hub

## Star-Hub introduction

The purpose of the Star-Hub is to extend the number of channels available for acquisition or generation by interconnecting multiple cards and running them simultaneously.

The Star-Hub option allows to synchronize several cards of the same M3i/M4i series that are mounted within one host system (PC):

- For the M3i series there are the two different versions available: a small version with 4 connectors (option SH4) for synchronizing up to four cards and a big version with 8 connectors (option SH8) for synchronizing up to eight cards.

- For the M4i series there are the two different mechanical versions available, with 8 connectors for synchronizing up to eight cards.

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

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

Either which of the two available Star-Hub options is used, there will be no phase delay between the sampling clocks of the synchronized cards and either no delay between the trigger events. The card holding the Star-Hub is automatically also the clock master. Any one of the synchronized cards can be part of the trigger generation.

### Star-Hub trigger engine

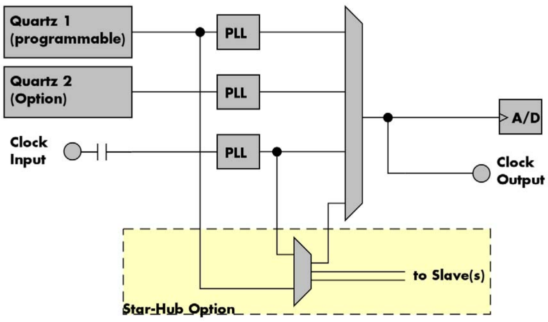
The trigger bus between an M3i/M4i card and the Star-Hub option consists of several lines. Some of them send the trigger information from the card's trigger engine to the Star-Hub and some receives the resulting trigger from the Star-Hub. All trigger events from the different cards connected are combined with OR on the star-hub.

While the returned trigger is identical for all synchronized cards, the sent out trigger of every single card depends on their trigger settings.

### Star-Hub clock engine

The card holding the Star-Hub is the clock master for the complete system. If you need to feed in an external clock to a synchronized system the clock has to be connected to the master card. Slave cards cannot generate a star-hub system clock. As shown in the drawing on the right the clock master can use either the programmable quartz 1 or the external clock input to be 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.




## Software Interface

The software interface is similar to the card software interface that is explained earlier in this manual. The same functions and some of the registers are used with the Star-Hub. The Star-Hub is accessed using its own handle which has some extra commands for synchronization setup. All card functions are programmed directly on card as before. There are only a few commands that need to be programmed directly to the Star-Hub for synchronization.

The software interface as well as the hardware supports multiple Star-Hubs in one system. Each set of cards connected by a Star-Hub then runs totally independent. It is also possible to mix cards that are connected with the Star-Hub with other cards that run independent in one system.

### Star-Hub Initialization

The interconnection between the Star-Hubs is probed at driver load time and does not need to be programmed separately. Instead the cards can be accessed using a logical index. This card index is only based on the ordering of the cards in the system and is not influenced by the current cabling. It is even possible to change the cable connections between two system starts without changing the logical card order that is used for Star-Hub programming.

**The Star-Hub initialization must be done AFTER initialization of all cards in the system. Otherwise the interconnection won't be received properly.**

The Star-Hubs are accessed using a special device name „sync" followed by the index of the star-hub to access. The Star-Hub is handled completely like a physical card allowing all functions based on the handle like the card itself.

Example with 4 cards and one Star-Hub (no error checking to keep example simple)

```
drv_handle  hSync;
drv_handle  hCard[4];

for (i = 0; i < 4; i++)
    {
    sprintf (s, "/dev/spcm%d", i);
    hCard[i] = spcm_hOpen (s);
    }
hSync = spcm_hOpen ("sync0");

...

spcm_vClose (hSync);
for (i = 0; i < 4; i++)
    spcm_vClose (hCard[i]);
```

Example for a digitizerNETBOX with two internal digitizer modules, This example is also suitable for accessing a remote server with two cards installed:

```
drv_handle  hSync;
drv_handle  hCard[2];

for (i = 0; i < 2; i++)
    {
    sprintf (s, "TCPIP:192.168.169.14::INST%d::INSTR", i);
    hCard[i] = spcm_hOpen (s);
    }
hSync = spcm_hOpen ("sync0");

...

spcm_vClose (hSync);
for (i = 0; i < 2; i++)
    spcm_vClose (hCard[i]);
```

When opening the Star-Hub the cable interconnection is checked. The Star-Hub may return an error if it sees internal cabling problems or if the connection between Star-Hub and the card that holds the Star-Hub is broken. It can't identify broken connections between Star-Hub and other cards as it doesn't know that there has to be a connection.

The synchronization setup is done using bit masks where one bit stands for one recognized card. All cards that are connected with a Star-Hub are internally numbered beginning with 0. The number of connected cards as well as the connections of the star-hub can be read out after initialization. For each card that is connected to the star-hub one can read the index of that card:

| 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_CARDIDX7 | 49007 | read | Index of card that is connected to star-hub logical index 7 (mask 0x0080) |
| SPC_SYNC_READ_CARDIDX8 | 49008 | read | M2i only: Index of card that is connected to star-hub logical index 8 (mask 0x0100) |
| ... | | read | ... |
| SPC_SYNC_READ_CARDIDX15 | 49015 | read | M2i only: Index of card that is connected to star-hub logical index 15 (mask 0x8000) |
| SPC_SYNC_READ_CABLECON0 | | read | Returns the index of the cable connection that is used for the logical connection 0. The cable connections can be seen printed on the PCB of the star-hub. Use these cable connection information in case that there are hardware failures with the star-hub cabeling. |
| ... | 49100 | read | ... |
| SPC_SYNC_READ_CABLECON15 | 49115 | read | Returns the index of the cable connection that is used for the logical connection 15. |

In standard systems where all cards are connected to one star-hub reading the star-hub logical index will simply return the index of the card again. This results in bit 0 of star-hub mask being 1 when doing the setup for card 0, bit 1 in star-hub mask being 1 when setting up card 1 and so on. On such systems it is sufficient to read out the SPC_SYNC_READ_SYNCCOUNT register to check whether the star-hub has found the expected number of cards to be connected.

```
spcm_dwGetParam_i32 (hSync, SPC_SYNC_READ_SYNCCOUNT, &lSyncCount);
for (i = 0; i < lSyncCount; i++)
    {
    spcm_dwGetParam_i32 (hSync, SPC_SYNC_READ_CARDIDX0 + i, &lCardIdx);
    printf ("star-hub logical index %d is connected with card %d\n", i, lCardIdx);
    }
```

In case of 4 cards in one system and all are connected with the star-hub this program 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

The synchronization setup only requires one additional registers to enable the cards that are synchronized in 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.

In our example we synchronize all four cards. The star-hub is located on card #2 and is therefor the clock master

```
spcm_dwSetParam_i32 (hSync, SPC_SYNC_ENABLEMASK, 0x000F); // all 4 cards are masked

// set the clock master to 100 MS/s internal clock
spcm_dwSetParam_i32 (hCard[2], SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hCard[2], SPC_SAMPLEATE, MEGA(100));

// set all the slaves to run synchronously with 100 MS/s
spcm_dwSetParam_i32 (hCard[0], SPC_SAMPLEATE, MEGA(100));
spcm_dwSetParam_i32 (hCard[1], SPC_SAMPLEATE, MEGA(100));
spcm_dwSetParam_i32 (hCard[3], SPC_SAMPLEATE, MEGA(100));
```

## Setup of Trigger

Setting up the trigger does not need any further steps of synchronization setup. Simply all trigger settings of all cards that have been enabled for synchronization are connected together. All trigger sources and all trigger modes can be used on synchronization as well.

Having positive edge of external trigger on card 0 to be the trigger source for the complete system needs the following setup:

```
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_ORMASK, SPC_TMASK_EXT0);
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_EXT0_MODE, SPC_TM_POS);

spcm_dwSetParam_i32 (hCard[1], SPC_TRIG_ORMASK, SPC_TM_NONE);
spcm_dwSetParam_i32 (hCard[2], SPC_TRIG_ORMASK, SPC_TM_NONE);
spcm_dwSetParam_i32 (hCard[3], SPC_TRIG_ORMASK, SPC_TM_NONE);
```

Assuming that the 4 cards are analog data acquisition cards with 4 channels each we can simply setup a synchronous system with all channels of all cards being trigger source. The following setup will show how to set up all trigger events of all channels to be OR connected. If any of the channels will now have a signal above the programmed trigger level the complete system will do an acquisition:

```
for (i = 0; i < lSyncCount; i++)
    {
    int32 lAllChannels = (SPC_TMASK0_CH0 | SPC_TMASK0_CH1 | SPC_TMASK_CH2 | SPC_TMASK_CH3);
    spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_CH_ORMASK0, lAllChannels);
    for (j = 0; j < 2; j++)
        {

        // set all channels to trigger on positive edge crossing trigger level 100
        spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_CH0_MODE + j, SPC_TM_POS);
        spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_CH0_LEVEL0 + j, 100);
        }
    }
```

## Run the synchronized cards

Running of the cards is very simple. The star-hub acts as one big card containing all synchronized cards. All card commands have to be omitted directly to the star-hub which will check the setup, do the synchronization and distribute the commands in the correct order to all synchronized cards. The same card commands can be used that are also possible for single cards:

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_M2CMD | 100 | write only | Executes a command for the card or data transfer |
| M2CMD_CARD_RESET | 1h | | Performs a hard and software reset of the card as explained further above |
| M2CMD_CARD_WRITESETUP | 2h | | Writes the current setup to the card without starting the hardware. This command may be useful if changing some internal settings like clock frequency and enabling outputs. |
| M2CMD_CARD_START | 4h | | Starts the card with all selected settings. This command automatically writes all settings to the card if any of the settings has been changed since the last one was written. After card has been started none of the settings can be changed while the card is running. |
| M2CMD_CARD_ENABLETRIGGER | 8h | | The trigger detection is enabled. This command can be either send together with the start command to enable trigger immediately or in a second call after some external hardware has been started. |
| M2CMD_CARD_FORCETRIGGER | 10h | | This command forces a trigger even if none has been detected so far. Sending this command together with the start command is similar to using the software trigger. |
| M2CMD_CARD_DISABLETRIGGER | 20h | | The trigger detection is disabled. All further trigger events are ignored until the trigger detection is again enabled. When starting the card the trigger detection is started disabled. |
| M2CMD_CARD_STOP | 40h | | Stops the current run of the card. If the card is not running this command has no effect. |

All other commands and settings need to be send directly to the card that it refers to.

This example shows the complete setup and synchronization start for our four cards:

```
spcm_dwSetParam_i32 (hSync, SPC_SYNC_ENABLEMASK, 0x000F); // all 4 cards are masked

// to keep it easy we set all card to the same clock and disable trigger
for (i = 0; i < 4; i++)
    {
    spcm_dwSetParam_i32 (hCard[i], SPC_CLOCKMODE, SPC_CM_INTPLL);
    spcm_dwSetParam_i32 (hCard[i], SPC_SAMPLERATE, MEGA(100));
    spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_ORMASK, SPC_TM_NONE);
    }

// card 0 is trigger master and waits for external positive edge
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_ORMASK, SPC_TMASK_EXT0);
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_EXT0_MODE, SPC_TM_POS);

// start the cards and wait for them a maximum of 1 second to be ready
spcm_dwSetParam_i32 (hSync, SPC_TIMEOUT, 1000);
spcm_dwSetParam_i32 (hSync, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);
if (spcm_dwSetParam_i32 (hSync, SPC_M2CMD, M2CMD_CARD_WAITREADY) == ERR_TIMEOUT)
    printf ("Timeout occured - no trigger received within time\n")
```

**Using one of the wait commands for the star-hub will return as soon as the card holding the star-hub has reached this state. However when synchronizing cards with different memory sizes there may be other cards that still haven't reached this level.**

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

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

For all M2i cards is is also possible to run the „slave" cards with a divided clock. Therefore please program a desired divided sampling rate in the SPC_SAMPLERATE register (example: running the star-hub card with 10 MS/s and the independent cards with 1 MS/s). The sampling rate is automatically adjusted by the driver to the next matching value.

### What is necessary?

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

### Setup

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

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

**When using SH_Direct mode, the register call to SPC_CLOCKMODE enabling this mode must be written before initiating a card start command to any of the connected cards. Also it is not allowed to be modified later in the programming sequence to prevent the driver from calculating wrong sample rates.**

### Example

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

```
// setup of the generator card
spcm_dwSetParam_i32 (hCard[0], SPC_CARDMODE, SPC_REP_STD_SINGLE);
spcm_dwSetParam_i32 (hCard[0], SPC_LOOPS, 0);                     // infinite data replay
spcm_dwSetParam_i32 (hCard[0], SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hCard[0], SPC_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);
```

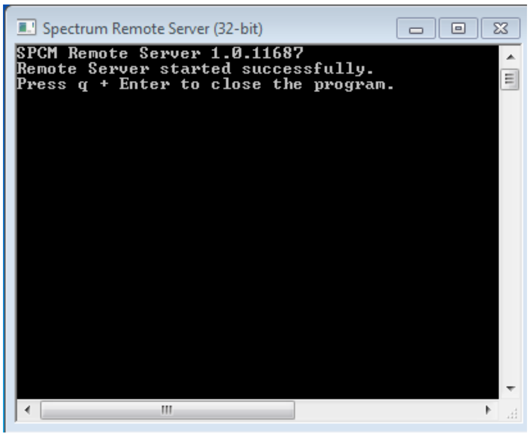
## Error Handling

The Star-Hub error handling is similar to the card error handling and uses the function spcm_dwGetErrorInfo_i32. Please see the example in the card error handling chapter to see how the error handling is done.

# Option Remote Server

## Introduction

Using the Spectrum Remote Server (order code -SPc-RServer) it is possible to access the M2i/M3i/M4i card(s) installed in one PC (server) from another PC (client) via local area network (LAN), similar to using a digitizerNETBOX.




It is possible to use different operating systems on both server and client. For example the Remote Server is running on a Linux system and the client is accessing them from a Windows system.

The Remote Server software requires, that the option „-SPc-RServer“ is installed on at least one card installed within the server side PC.

You can either check this with the Control Center in the "Installed Card features" node or by reading out the feature register, as described in the „Installed features and options“ passage, earlier in this manual.

**To run the the Remote Server software, it is required to have least version 3.18 of the Spectrum SPCM driver installed. Additionally at least on one card in the server PC the feature flag SPCM_FEAT_REMOTESERVER must be set.**

## Installing and starting the Remote Server

### Windows

Windows users find the Control Center installer on the CD under „Install\win\spcm_remote_install.exe“.

After the installation has finished there will be a new start menu entry in the Folder "Spectrum GmbH" to start the Remote Server. To start the Remote Server automatically after login, just copy this shortcut to the Autostart directory.

### Linux

Linux users find the versions of the installer for the different StdC libraries under under /Install/linux/spcm_control_center/ as RPM packages.

To start the Remote Server type "spcm_remote_server" (without quotation marks). To start the Remote Server automatically after login, add the following line to the .bashrc or .profile file (depending on the used Linux distribution) in the user's home directory:

```
spcm_remote_server&
```

## Accessing remote cards

To detect remote card(s) from the client PC, start the Spectrum Control Center on the client and click "Netbox Discovery". All discovered cards will be listed under the "Remote" node.

Using remote cards instead of using local ones is as easy as using a digitizerNETBOX and only requires a few lines of code to be changed compared to using local cards.

Instead of opening two locally installed cards like this:

```
hDrv0 = spcm_hOpen ("/dev/spcm0"); // open local card spcm0
hDrv1 = spcm_hOpen ("/dev/spcm1"); // open local card spcm1
```

one would call spcm_hOpen() with a VISA string as a parameter instead:

```
hDrv0 = spcm_hOpen ("TCPIP::192.168.1.2::inst0::INSTR"); // open card spcm0 on a Remote Server PC
hDrv1 = spcm_hOpen ("TCPIP::192.168.1.2::inst1::INSTR"); // open card spcm1 on a Remote Server PC
```

to open cards on the Remote Server PC with the IP address 192.168.1.2. The driver will take care of all the network communication.

# Appendix

## Error Codes

The following error codes could occur when a driver function has been called. Please check carefully the allowed setup for the register and change the settings to run the program.

| error name | value (hex) | value (dec.) | error description |
|---|---|---|---|
| ERR_OK | 0h | 0 | Execution OK, no error. |
| ERR_INIT | 1h | 1 | An error occurred when initializing the given card. Either the card has already been opened by another process or an hardware error occurred. |
| ERR_TYP | 3h | 3 | 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_BOARDINUSE | Bh | 11 | A card with given name is already in use by another application. |
| ERR_EXPHW64BITADR | Ch | 12 | Express hardware version not able to handle 64 bit addressing -> update needed. |
| ERR_FWVERSION | Dh | 13 | Firmware versions of synchronized cards or for this driver do not match -> update needed. |
| ERR_LASTERR | 10h | 16 | Old error waiting to be read. Please read the full error information before proceeding. The driver is locked until the error information has been read. |
| ERR_BOARDINUSE | 11h | 17 | Board is already used by another application. It is not possible to use one hardware from two different programs at the same time. |
| ERR_ABORT | 20h | 32 | Abort of wait function. This return value just tells that the function has been aborted from another thread. The driver library is not locked if this error occurs. |
| ERR_BOARDLOCKED | 30h | 48 | The card is already in access and therefore locked by another process. It is not possible to access one card through multiple processes. Only one process can access a specific card at the time. |
| ERR_DEVICE_MAPPING | 32h | 50 | The device is mapped to an invalid device. The device mapping can be accessed via the Conrol Center. |
| ERR_NETWORKSETUP | 40h | 64 | The network setup of a digitizerNETBOX has failed. |
| ERR_NETWORKTRANSFER | 41h | 65 | The network data transfer from/to a digitizerNETBOX has failed. |
| ERR_FWPOWERCYCLE | 42h | 66 | Power cycle (PC off/on) is needed to update the card's firmware (a simple OS reboot is not sufficient !) |
| ERR_NETWORKTIMEOUT | 43h | 67 | A network timeout has occurred. |
| ERR_BUFFERSIZE | 44h | 68 | The buffer size is not sufficient (too small). |
| ERR_RESTRICTEDACCESS | 45h | 69 | The access to the card has been intentionally restricted. |
| ERR_INVALIDPARAM | 46h | 70 | An invalid parameter has been used for a certain function. |
| ERR_REG | 100h | 256 | The register is not valid for this type of board. |
| ERR_VALUE | 101h | 257 | The value for this register is not in a valid range. The allowed values and ranges are listed in the board specific documentation. |
| ERR_FEATURE | 102h | 258 | Feature (option) is not installed on this board. It's not possible to access this feature if it's not installed. |
| ERR_SEQUENCE | 103h | 259 | Command sequence is not allowed. Please check the manual carefully to see which command sequences are possible. |
| ERR_READABORT | 104h | 260 | Data read is not allowed after aborting the data acquisition. |
| ERR_NOACCESS | 105h | 261 | Access to this register is denied. This register is not accessible for users. |
| ERR_TIMEOUT | 107h | 263 | A timeout occurred while waiting for an interrupt. This error does not lock the driver. |
| ERR_CALLTYPE | 108h | 264 | The access to the register is only allowed with one 64 bit access but not with the multiplexed 32 bit (high and low double word) version. |
| ERR_EXCEEDSINT32 | 109h | 265 | The return value is int32 but the software register exceeds the 32 bit integer range. Use double int32 or int64 accesses instead, to get correct return values. |
| ERR_NOWRITEALLOWED | 10Ah | 266 | The register that should be written is a read-only register. No write accesses are allowed. |
| ERR_SETUP | 10Bh | 267 | The programmed setup for the card is not valid. The error register will show you which setting generates the error message. This error is returned if the card is started or the setup is written. |
| ERR_CLOCKNOTLOCKED | 10Ch | 268 | Synchronisation to external clock failed: no signal connected or signal not stable. Please check external clock or try to use a different sampling clock to make the PLL locking easier. |
| ERR_CHANNEL | 110h | 272 | The channel number may not be accessed on the board: Either it is not a valid channel number or the channel is not accessible due to the current setup (e.g. Only channel 0 is accessible in interlace mode) |
| 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. |

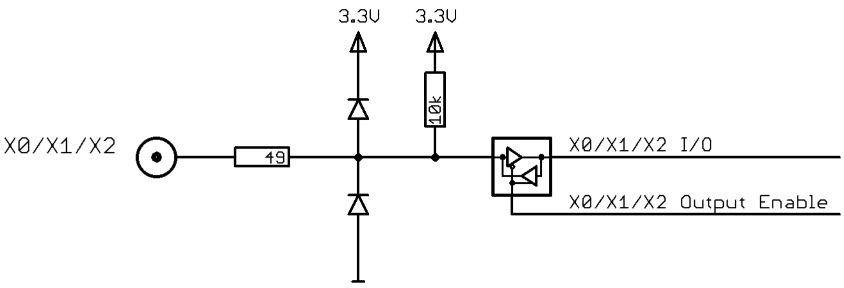
| error name | value (hex) | value (dec.) | error description |
|---|---|---|---|
| ERR_EDGEPERMOD | 149h | 329 | This card is only capable to have one programmed trigger edge for each module that is installed. It is not possible to mix different trigger edges on one module. |
| ERR_DOLEVELMINDIFF | 14Ah | 330 | The minimum difference between low output level and high output level is not reached. |
| ERR_STARHUBENABLE | 14Bh | 331 | The card holding the star-hub must be enabled when doing synchronization. |
| ERR_PATPWSMALLEDGE | 14Ch | 332 | Combination of pattern with pulsewidth smaller and edge is not allowed. |
| ERR_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_TIMESTAMP_SYNC | 310h | 784 | Synchronization to timestamp reference clock failed. Please check the connection and the signal levels of the reference clock input. |
| ERR_STARHUB | 320h | 800 | The auto routing function of the Star-Hub initialization has failed. Please check whether all cables are mounted correctly. |
| ERR_INTERNAL_ERROR | FFFFh | 65535 | Internal hardware error detected. Please check for driver and firmware update of the card. |

# Details on M4i cards I/O lines

## Multi Purpose I/O Lines

The MMCX Multi Purpose I/O connectors (X0, X1 and X2) of the M4i cards from Spectrum are protected against over voltage conditions.

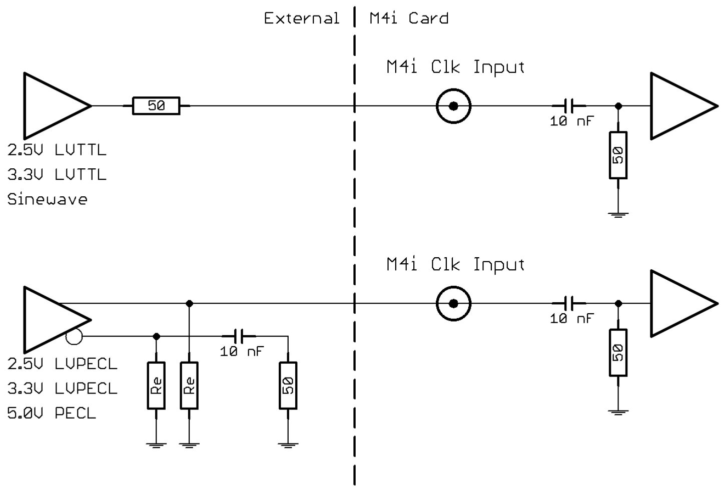
For this purpose clamping diodes of the types CD1005 are used in conjunction with a series resistor. All three I/O lines are internally clamped to signal ground and to 3.3V clamping voltage. So when connecting sources with a higher level than the clamping voltage plus the forward voltage of typically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




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

## Interfacing with clock input

The clock input of the M4i cards is AC-coupled, single-ended PECL type. Due to the internal biasing and a relatively high maximum input voltage swing, it can be directlly connected to various logic standards, without the need for external level converters.




#### Single-ended LVTTL sources

All LVTTL sources, be it 2.5V LVTTL or 3.3V LVTTL must be terminated with a 50 Ohm series resistor to avoid reflections and limit the maximum swing for the M4i card.

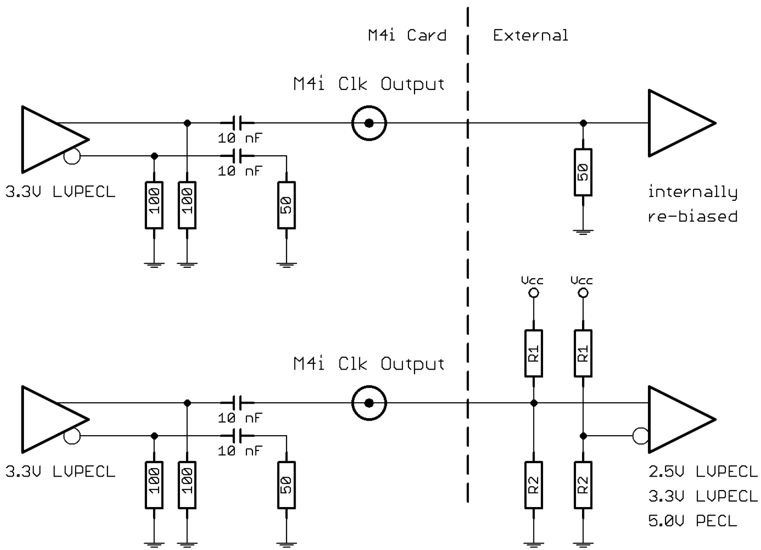
#### Differential (LV)PECL sources

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

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

## Interfacing with clock output

The clock output of the M4i cards is AC-coupled, single-ended PECL type. The output swing of the M3i clock output is approximately 800 $mV_{PP}$.




#### Internal biased single-ended receivers

Because of the AC coupling of the M4i clock output, the signal must be properly re-biased for the receiver. Receivers that provide an internal re-bias only require the signal to be terminated to ground by a 50 Ohm resistor.

#### Differential (LV)PECL receivers

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

# Details on M4i cards status LED

Every M4i card has a two-color status LED mounted within the multi-purpose I/O connector field on the card bracket. This chapter explains the different color codings and offers some possible solutions in case of an error condition.


| Condition | LED color | Status | Solution |
|---|---|---|---|
| Off | Off | Card not powered | Power on the PC. |
| Error | Static: red | Power supply error | Restart the PC. In case that the error persists, please contact Spectrum support for further assistance. |
| | Fast blinking (aprox. 4 Hz):<br>red - green - red - green ... | Power supply error | Restart the PC. In case that the error persists, please contact Spectrum support for further assistance. |
| | Blinking: red - off - red - off ... | Over temperature error | Power down the PC, let the card cool down and restart the system. Please make sure that you have a proper cooling fan installed to supply the M4i card in the PCIe slot with a constant air flow. |
| | Slow blinking (aprox. 1 Hz):<br>red - green - red - green ... | PCI Express link training has not finished | 1) Power down the PC, un-plug and re-plug the card to verify that there is a proper contact between the card and the slot.<br><br>2) Try another PCI Express slot, maybe the currently used one is not properly working.<br><br>3) In case that this error is occuring after a firmware update or of the above steps did not help, please contact Spectrum support for assistance on how to boot the card's golden recovery image. |
| O.K. | Static: green | Card is ready for operation<br>(at full PCIe speed) | A full speed PCIe link has been established (PCIe x8, Gen 2) and the card is ready for operation. |
| | Slow blinking (approx. 1 Hz):<br>green - off - green - off ... | Indicator mode on<br>(at full PCIe speed) | To ease the identification of a specific card in a multi-card system without un-installing the card it is possible to activate the card identification status by software. This mode changes the static „Ready for Operation" green into a blinking state. |
| | Static: green & red (yellow) | Card is ready for operation<br>(at reduced PCIe speed) | A reduced speed PCIe link has been established either with less than all of the possible 8 lanes and/or the card is installed in a PCIe Gen 1 slot. The card is ready for operation, but the data transfer throughput over the PCI Express bus is reduced.<br><br>For getting the highest PCIe performance please consult your PC's or motherboard's manual for details on the PCI Express slots of your system. |
| | Slow blinking (aprox. 1 Hz):<br>yellow - off - yellow - off ... | Indicator mode on<br>(at reduced PCIe speed) | To ease the identification of a specific card in a multi-card system without un-installing the card it is possible to activate the card identification status by software. This mode changes the static „Ready for Operation" yellow into a blinking state. |

## Turning on card identification LED

To enable/disable the cards LED indicator mode or to read out the current setting, please use the following register:

| Register | Value | Direction | Description |
|---|---|---|---|
| SPC_CARDIDENTIFICATION | 201500 | read/write | Writing a '1' turns on the LED card indicator mode, writing a '0' turns off the LED indicator mode. |

The default for the card identification register is the OFF state.