

M4i.66xx-x8

M4x.66xx-x4

**high-speed 16 bit AWG,
Arbitrary Waveform Generator
for PCI Express bus and PXI Express bus**

Valid for all versions

**Hardware Manual
Software Driver Manual**

Manual Printed: 28. February 2024

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

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

Table of Contents

Table of Contents	3
Safety Instructions	9
Symbols and Safety Labels	9
General safety information	9
Requirements for users and duties for operators	9
General safety at work	9
Bringing the product into service	10
Intended use	10
Application area of the product	10
Requirements for the technical state of the product	10
Requirements for operation	10
Electrical safety and power supply	10
Requirements for the location	10
Requirements on the ventilation	10
Maintenance	10
Repair/Service	11
Cleaning the module housing (NETBOX devices, cables, amplifiers, systems only)	11
Opening the module (NETBOX devices, amplifiers only)	11
Dismounting parts of the card (instrument card only)	11
Markings and Labelling	11
Packing list	12
PCIe (PCI Express) cards M4i.xxxx	12
PXIe (PXI Express) cards M4x.xxxx	12
Introduction	13
Preface	13
Overview	13
M4i cards for PCIe Express (PCIe)	13
M4x cards for PXI Express (PXIe)	13
General Information	14
Different models of the M4i.66xx series	15
Different models of the M4x.66xx series	16
Additional options	17
Star-Hub (M4i only)	17
Digital I/O with Dig-SMA (M4i.44xx only)	17
Hardware information	18
Block Diagrams	18
Technical Data	19
Bandwidth and Slewrate	21
Dynamic Parameters	21
SFDR and THD versus signal frequency	22
M4i Specific Technical Data	22
M4x Specific Technical Data	23
Order Information	24
M4i Order Information	24
M4x Order Information	25
Hardware Installation	27
ESD Precautions	27
Sources of noise	27
Cooling Precautions	27
Connector Handling Precautions	27
M4i PCIe Cards	28
System Requirements	28
Installing the M4i board in the system	28
Additional notes on PCIe x16 slot retention	29
Providing additional power to M4i.xxxx-x8 cards	29
Installing a M4i.44xx board with digital inputs/outputs mounted on an extra bracket	29
Installing multiple boards synchronized by star-hub option	30
Shipment of systems with Spectrum cards installed	31
M4x PXIe Cards	31
System Requirements	31
Installing the M4x board in the system	31

Software Driver Installation and Driver Update.....	32
Windows	32
Before initial installation	32
Running the driver Installer/Update.....	32
After installation	33
Linux.....	34
Overview	34
Driver Installation with Installation Script.....	34
Standard Driver Update	35
Compilation of kernel driver sources (optional and local cards only)	35
Update of a self compiled kernel driver	35
Installing the library only without a kernel (for remote devices)	36
Installation from Spectrum Repository	36
Control Center	37
Software	39
Software Overview.....	39
Card Control Center	39
Discovery of Remote Cards, digitizerNETBOX/generatorNETBOX/hybridNETBOX products	40
Wake On LAN of digitizerNETBOX/generatorNETBOX/hybridNETBOX.....	40
Netbox Monitor	41
Device identification	41
Hardware information.....	42
Firmware information	42
Software License information.....	43
Driver information.....	43
Installing and removing Demo cards	44
Feature upgrade.....	44
Software License upgrade.....	44
Performing card calibration (A/D only)	44
Performing memory test	45
Transfer speed test.....	45
Debug logging for support cases	45
Device mapping	46
Firmware upgrade	46
Accessing the hardware with SBench 6.....	47
C/C++ Driver Interface.....	47
Header files	47
General Information on Windows 64 bit drivers.....	48
Microsoft Visual C++ 6.0, 2005 and newer 32 Bit.....	48
Microsoft Visual C++ 2005 and newer 64 Bit.....	48
Linux Gnu C/C++ 32/64 Bit	48
C++ for .NET	49
Other Windows C/C++ compilers 32 Bit.....	49
Other Windows C/C++ compilers 64 Bit.....	49
Driver functions	49
Delphi (Pascal) Programming Interface	54
Driver interface	54
Examples.....	56
.NET programming languages	57
Library	57
Declaration.....	57
Using C#.....	57
Using Managed C++/CLI.....	59
Using VB.NET	59
Using J#	59
Python Programming Interface and Examples	60
Driver interface	60
Examples.....	60
Java Programming Interface and Examples	61
Driver interface	61
Examples.....	61
Julia Programming Interface and Examples	62
Driver interface	62
Examples.....	62
LabVIEW driver and examples	63
MATLAB driver and examples	63
SCAPP – CUDA GPU based data processing	64

Programming the Board	65
Overview	65
Register tables	65
Programming examples.....	65
Initialization.....	66
Initialization of Remote Products	66
Error handling.....	66
Gathering information from the card.....	67
Card type	68
Hardware and PCB version	68
Reading currently used PXI slot No. (M4x only)	69
Firmware versions.....	69
Production date	69
Last calibration date (analog cards only)	70
Serial number	70
Maximum possible sampling rate	70
Installed memory	70
Installed features and options	70
Miscellaneous Card Information	71
Function type of the card	72
Used type of driver	72
Custom modifications	73
Reset.....	73
Analog Outputs	74
Channel Selection	74
Important note on channel selection	75
Setting up the outputs.....	75
Output Enable.....	75
Output Amplitude Setting and Hysteresis	76
Output Filters	76
Differential Output	77
Double Out Mode	77
Programming the behavior in pauses and after replay.....	78
Read out of output features	78
Generation modes	79
Overview	79
Setup of the mode	79
Commands.....	79
Card Status.....	80
Acquisition cards status overview	81
Generation card status overview	81
Data Transfer	81
Standard Single Replay modes	84
Card mode	84
Memory setup	84
Continuous marker output	85
Example	86
FIFO Single replay mode.....	86
Card mode	86
Length of FIFO mode.....	86
Difference to standard single mode.....	86
Example (FIFO replay).....	87
Limits of segment size, memory size.....	88
Buffer handling	89
Output latency	92
Data organization	94
Sample format	94
Hardware data conversion	94
Clock generation	96
Overview	96
Clock Mode Register.....	96
The different clock modes	96
Details on the different clock modes.....	97
Standard internal sampling clock (PLL).....	97
Using Quartz2 with PLL (optional, M4i cards only).....	97
External clock (reference clock)	98
PXI Reference Clock (M4x cards only)	99

Trigger modes and appendant registers	100
General Description.....	100
Trigger Engine Overview.....	100
Trigger masks	101
Trigger OR mask	101
Trigger AND mask.....	102
Software trigger	104
Force- and Enable trigger	104
Trigger delay	105
Trigger Counter.....	105
Main external window trigger (Ext0/Trg0)	106
Trigger Mode.....	106
Trigger Input Termination.....	106
Trigger Input Coupling	107
Secondary external level trigger (Ext1/Trg1).....	107
Trigger Mode.....	107
Trigger level.....	107
Detailed description of the external analog trigger modes	108
PXI Trigger (M4x PXIe cards only).....	112
PXI Trigger Mode Registers	112
PXI Trigger Sources within the Trigger Masks	113
PXI Trigger Setup Example.....	114
Multi Purpose I/O Lines	115
On-board I/O lines (X0, X1, X2).....	115
Programming the behavior.....	115
Using asynchronous I/O	116
Special behavior of trigger output.....	116
Using synchronous digital outputs	117
Mode Multiple Replay.....	119
Trigger Modes	119
Programming examples.....	119
Replay modes	120
Standard Mode.....	120
FIFO Mode	120
Limits of segment size, memory size.....	121
Programming the behavior in pauses and after replay.....	121
Mode Gated Replay.....	123
Generation Modes	123
Standard Mode.....	123
Examples of Standard Standard Gated Replay with the use of SPC_LOOPS parameter	123
FIFO Mode	123
Examples of Fifo Gated Replay with the use of SPC_LOOPS parameter	124
Limits of segment size, memory size.....	124
Trigger.....	125
Detailed description of the external analog trigger modes	125
Programming examples	128
Programming the behavior in pauses and after replay.....	129
Sequence Replay Mode	130
Theory of operation	130
Define segments in data memory	130
Define steps in sequence memory	130
Programming	131
Gathering information	131
Setting up the registers	131
Changing sequences or step parameters during runtime	133
Changing data patterns during runtime	133
Synchronization	133
Programming example	134

Mode DDS (Firmware Option)	135
General Information	135
Feature Overview	135
Controlling the DDS functionality	135
Command FIFO	136
DDS Command buffer fill size, overrun and underrun	136
DDS Command	137
Internal trigger sources	137
DDS status register	137
General DDS information register	138
Trigger timer	138
DDS core connections	139
Programming the DDS cores	140
Frequency	141
Amplitude	141
Phase	141
Simple example for fixed frequency output	142
Slope update rate	142
Frequency Slope	142
Example for Frequency Slope	142
Amplitude Slope	143
Example for Amplitude Slope	143
Modifying stepsize of slope	143
Defining the Phase Behaviour	143
Mode Phase Jump	144
Mode Phase Shift	144
Additional functions in DDS core	145
Example for multi-purpose DDS output	145
Benefits of using the parameter array function	146
Transfer Mode Specification	146
Execute Now, Timer and Trigger timing behaviour	147
Pulse Generator (Firmware Option)	148
General Information	148
Principle of Operation	149
Setting up the Pulse Generator	150
Enabling, disabling and resetting a pulse generator	150
Defining the basic pulse parameters	150
Delaying (phase shifting) the Outputs	151
Defining the trigger behavior	151
Configuring the pulse generator's trigger source	152
Configuring Multi Purpose lines to output generated pulses	154
Programming Example	155
Option Star-Hub (M3i and M4i only)	156
Star-Hub introduction	156
Star-Hub trigger engine	156
Star-Hub clock engine	156
Software Interface	156
Star-Hub Initialization	156
Setup of Synchronization	158
Setup of Trigger	159
Run the synchronized cards	159
SH-Direct: using the Star-Hub clock directly without synchronization	160
Error Handling	161
Option Remote Server	162
Introduction	162
Installing and starting the Remote Server	162
Windows	162
Linux	162
Detecting the digitizerNETBOX/generatorNETBOX/hybridNETBOX	162
Discovery Function	162
Finding the digitizerNETBOX/generatorNETBOX/hybridNETBOX in the network	163
Troubleshooting	164
Accessing remote cards	164

Appendix	165
Error Codes	165
Spectrum Knowledge Base	166
Temperature sensors	167
Temperature read-out registers	167
Temperature hints	167
66xx temperatures and limits	167
Details on M4i/M4x cards I/O lines	168
Multi-Purpose I/O Lines	168
Interfacing with clock input	168
Interfacing with clock output.....	168
Details on M4i/M4x cards status LED	169
Turning on card identification LED	169
Continuous memory for increased data transfer rate	170
Background	170
Setup on Linux systems	171
Setup on Windows systems.....	172
Usage of the buffer	172
Abbreviations.....	174
List of Figures	175
List of Tables	177









Safety Instructions

This chapter contains information about the following topics:

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

Symbols and Safety Labels

Table 1: Symbols and Safety Labels

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

General safety information

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

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

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

Requirements for users and duties for operators

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

General safety at work

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

Bringing the product into service

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

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

Intended use

Application area of the product

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

Requirements for the technical state of the product

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

Requirements for operation

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

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

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

Electrical safety and power supply

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

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

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

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

Requirements for the location

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

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

Requirements on the ventilation

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

Maintenance

The product is maintenance-free.

Repair/Service

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

Please ensure suitable packaging to avoid damage during transport.

World-wide service address is:

Spectrum Instrumentation GmbH
Ahrensfelder Weg 13-17
22927 Grosshansdorf
Germany

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

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

Opening the module (NETBOX devices, amplifiers only)

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

Dismounting parts of the card (instrument card only)

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

Markings and Labelling

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

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

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

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

Packing list

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

PCIe (PCI Express) cards M4i.xxxx

Table 2: Packing List M4i card (PCI Express)

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

PXIe (PXI Express) cards M4x.xxxx

Table 3: Packing List M4x card (PXI Express)

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

Introduction

Preface

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

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

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

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

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

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



Overview

M4i cards for PCIe Express (PCIe)



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

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



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

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

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



M4x cards for PXI Express (PXIe)



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

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

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



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



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

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

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



General Information

The M4i.66xx and M4x.66xx series offer a wide range of ultra fast 16 bit D/A converter boards for PCI Express (PCIe) and PXI Express (PXIe) 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.

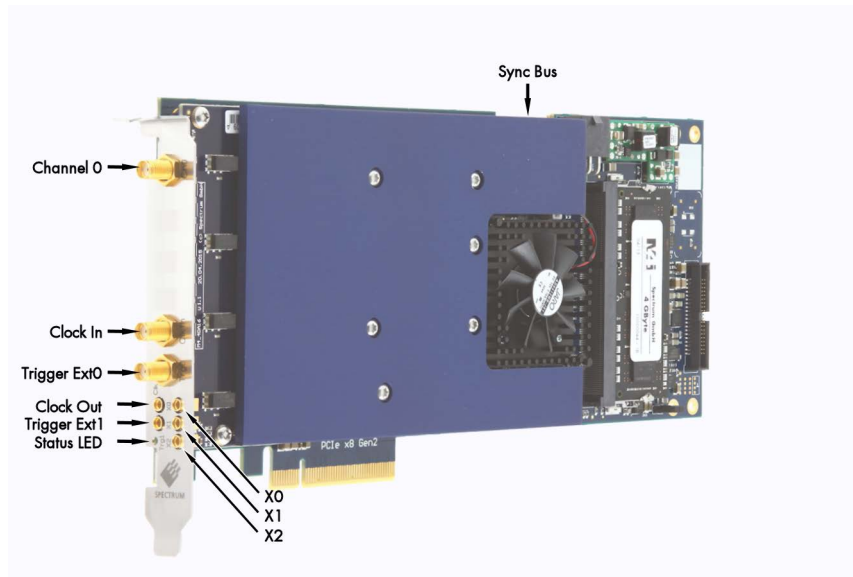
Several boards of the M4i.xxxx series may be connected together by the internal standard synchronization bus in combination with one of the star-hub options to work with the same time base. That allows to build system with multiple D/A channels or systems with combined A/D and D/A channels.

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

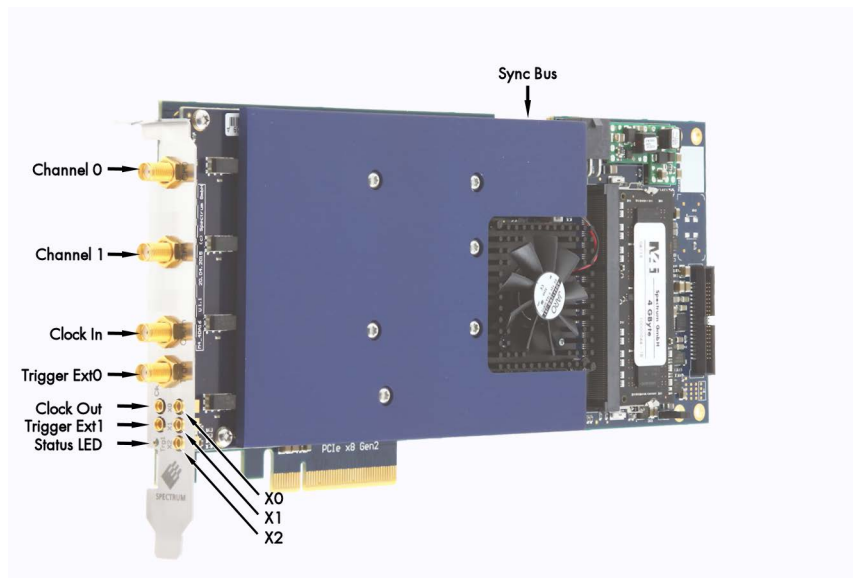
Different models of the M4i.66xx series

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 dependent location of the input connectors.

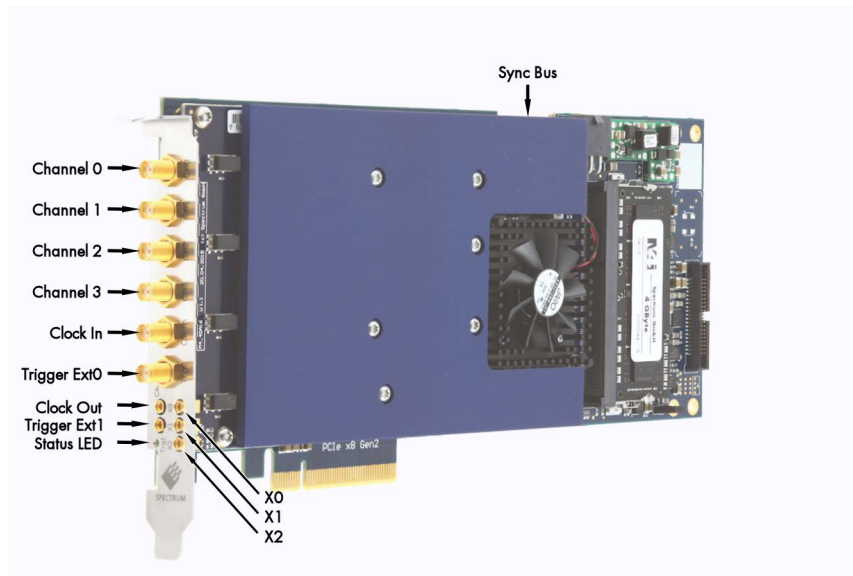
- **M4i.6620-x8**
- **M4i.6630-x8**



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



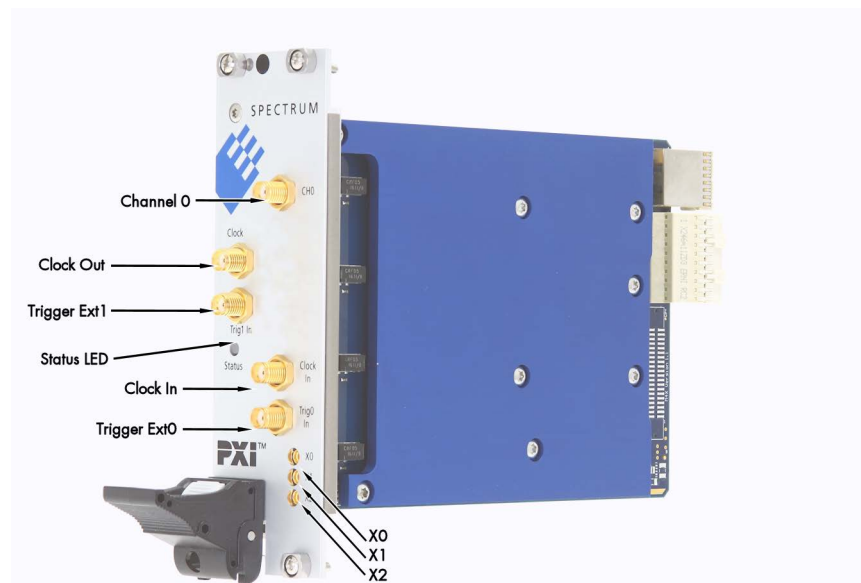
- **M4i.6622-x8**



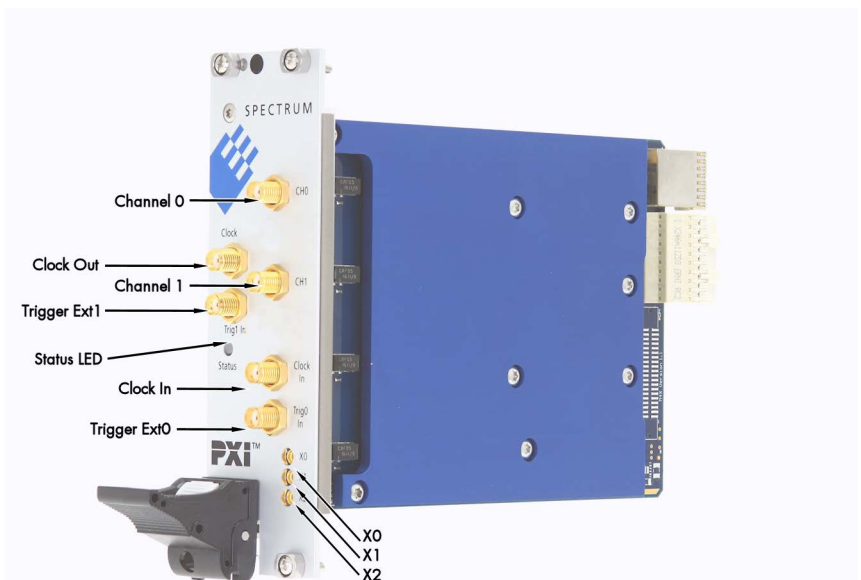
Different models of the M4x.66xx series

The following overview shows the different available models of the M4x.22xx series. They differ in the number of available channels. You can also see the model dependent location of the input connectors.

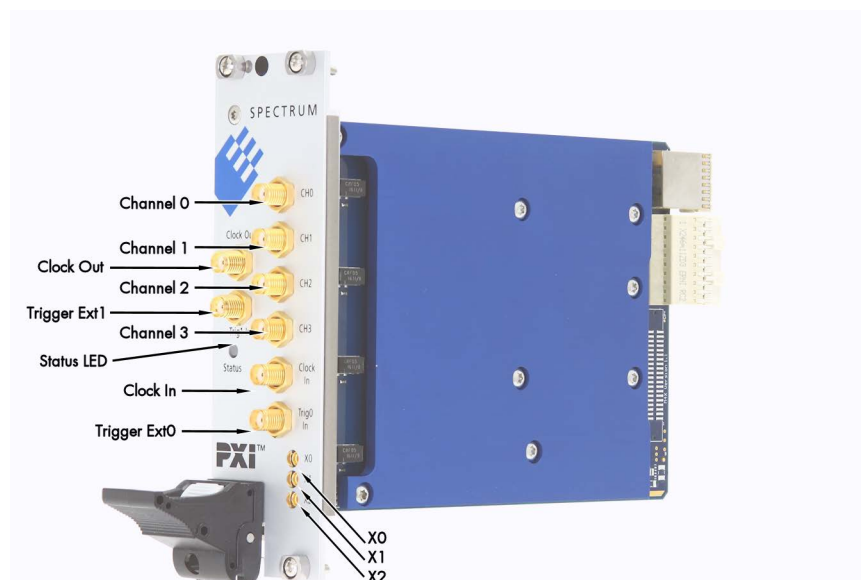
- **M4x.6620-x4**
- **M4x.6630-x4**



- **M4x.6621-x4**
- **M4x.6631-x4**



- **M4x.6622-x**



Additional options

Star-Hub (M4i only)

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

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

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

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

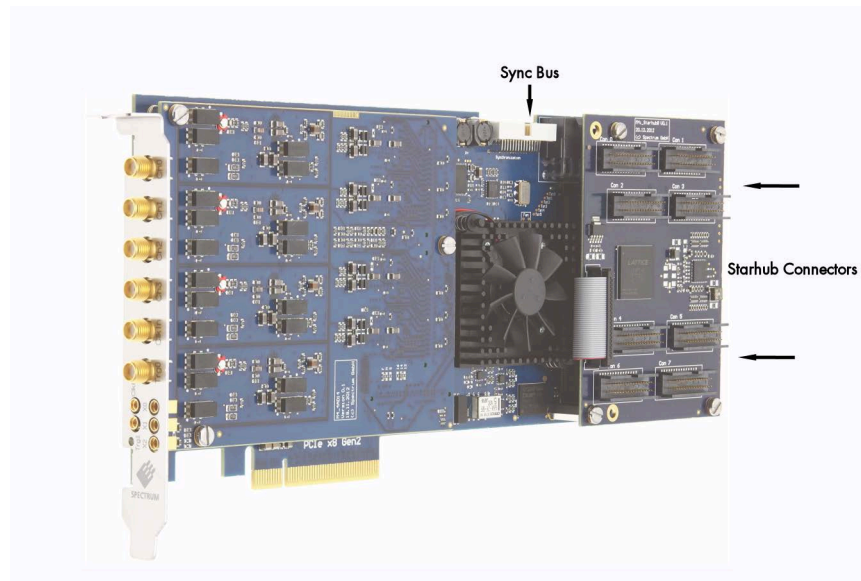


Image 1: M4i card showing mounted star-hub and sync bus connector

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

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

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

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

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

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

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

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

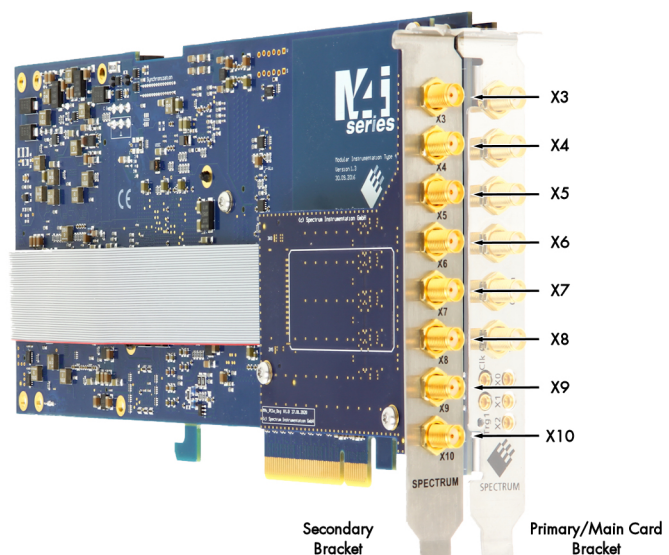


Image 2: M4i card showing mounted digital option

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



Hardware information

Block Diagrams

M4i.66xx Block Diagram

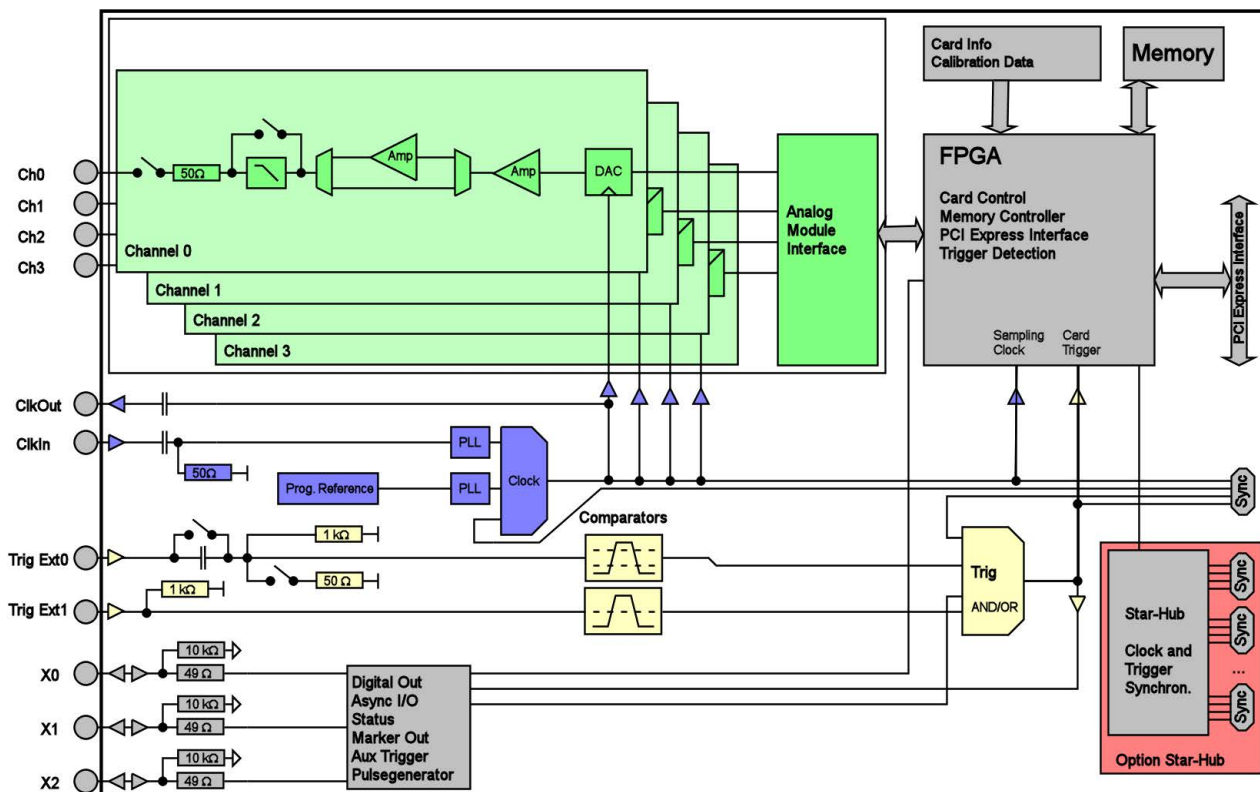


Image 3: M4i.66xx PCI Express series hardware block diagram

M4x.66xx Block Diagram

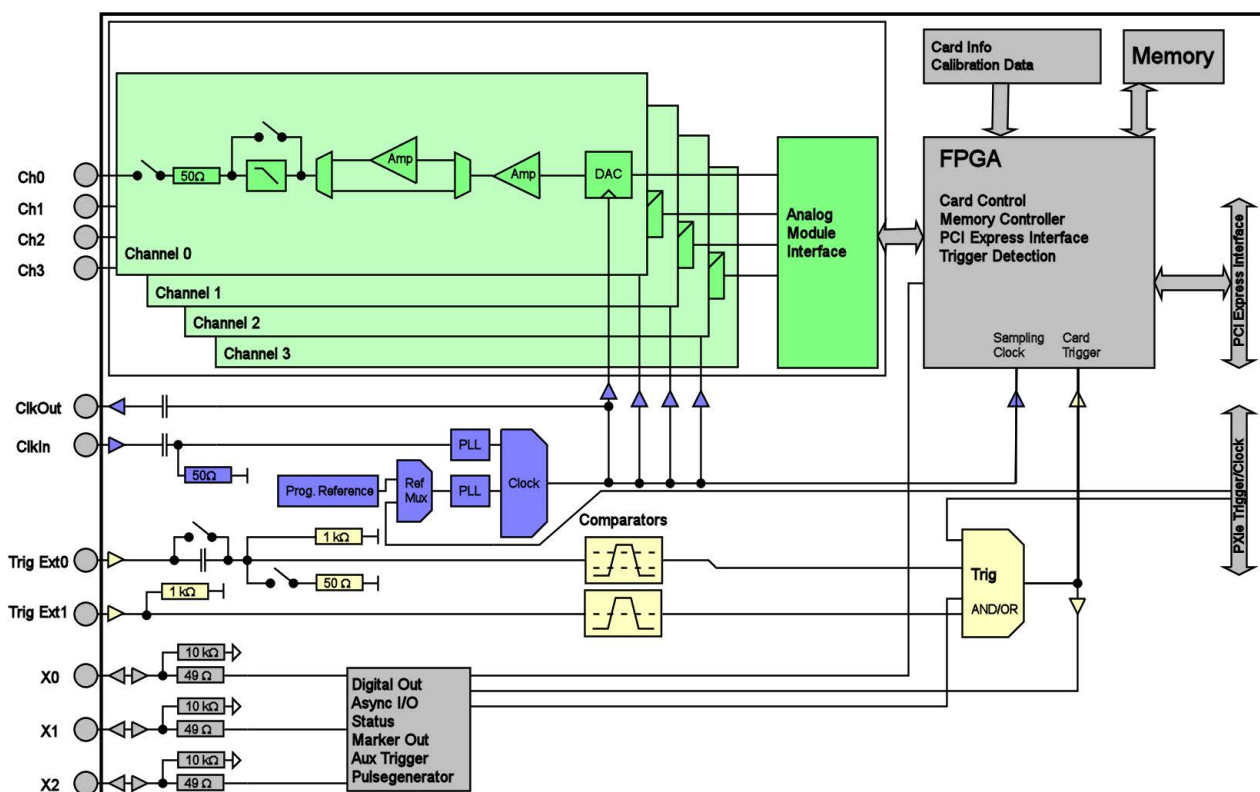


Image 4: M4x.66xx PXI Express series hardware block diagram

Technical Data



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

Analog Outputs

Resolution		16 bit	
D/A Interpolation		no interpolation	
		M4i.662x/M4x.662x DN2.662/DN6.662x DN2.82x-04	M4i.663x/M4x.663x DN2.663/DN6.663 DN2.82x-02
Output amplitude into 50 Ω termination	software programmable	±80 mV up to ±2.5 V	±80 mV up to ±2 V
Output amplitude into high impedance loads	software programmable	±160 mV up to ±5 V	±160 mV up to ±4 V
Stepsize of output amplitude (50 Ω termination)		1 mV	1 mV
Stepsize of output amplitude (high impedance)		2 mV	2 mV
10% to 90% rise/fall time of 0 V to 480 mV pulse		1.5 ns	1.1 ns
10% to 90% rise/fall time of 0 V to 2000 mV pulse		1.5 ns	1.1 ns
Output offset	fixed	0 V	
Output Amplifier Path Selection	automatically by driver	Low Power path: ±80 mV to ±480 mV (into 50 Ω) 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 switch 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	
Output coupling		DC	
Minimum output load		0 Ω (short circuit safe)	
Output accuracy	Low power path High power path	±0.5 mV ±0.1% of programmed output amplitude ±1.0 mV ±0.2% of programmed output amplitude	
Offset temperature drift	after warm-up and calibration	TBD	
Gain temperature drift	after warm-up and calibration	TBD	
Calibration	External	External calibration calibrates the on-board references. All calibration constants are stored in non-volatile memory. A yearly external calibration is recommended.	

Trigger

Available trigger modes	software programmable	External, Software, Window, Re-Arm, Or/And, Delay, PXI (M4x only)
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	sample rate ≤ 625 MS/s sample rate > 625 MS/s	238.5 sample clocks + 16 ns (valid for all modes except SPCSEQ_ENDLOOPONTRIG) 476.5 sample clocks + 16 ns (valid for all modes except SPCSEQ_ENDLOOPONTRIG)
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
Trigger accuracy (all sources)		1 sample
Minimum external trigger pulse width		≥ 2 samples
External trigger		Ext0
External trigger impedance	software programmable	50 Ω / 1 kΩ
External trigger coupling	software programmable	AC or DC
External trigger type		Window comparator
External input level		±10 V (1 kΩ), ±2.5 V (50 Ω),
External trigger sensitivity (minimum required signal swing)		2.5% of full scale range
External trigger level	software programmable	±10 V in steps of 10 mV
External trigger maximum voltage		±30V
External trigger bandwidth DC	50 Ω 1 kΩ	DC to 200 MHz DC to 150 MHz
External trigger bandwidth AC	50 Ω	20 kHz to 200 MHz
Minimum external trigger pulse width		≥ 2 samples

Ext1

1 kΩ
fixed DC
Single level comparator
±10 V
2.5% of full scale range = 0.5 V

Clock

Clock Modes	software programmable	internal PLL, external reference clock, Star-Hub sync (generatorNETBOX and M4i only), PXI Reference Clock (M4x only)
Internal clock accuracy		$\leq \pm 20$ 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 sampling clock possible in these gaps)
External reference clock range	software programmable	≥ 10 MHz and ≤ 1.25 GHz
External reference clock input impedance		50 Ω fixed
External reference clock input coupling		AC coupling
External reference clock input edge		Rising edge
External reference clock input type		Single-ended, sine wave or square wave
External reference clock input swing	square wave	0.3 V peak-peak up to 3.0 V peak-peak
External reference clock input swing	sine wave	1.0 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)
Minimum segment size	software programmable	384 samples (1 active channel), 192 samples (2 active channels), 96 samples (4 active channels), in steps of 32 samples.
Maximum segment size	software programmable	2 GS / active channels / number of sequence segments (round up to the next power of two)
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
Limitations for synchronized products		Software commands changing the sequence as well as „Loop until trigger“ are not synchronized between cards. This also applies to multiple AWG modules in a generatorNETBOX.

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 k Ω to 3.3 V
Input: maximum voltage level		-0.5 V to +4.0 V
Input: signal levels		3.3 V LVTTTL
Output: available signal types	software programmable	Asynchronous Digital-Out, Synchronous Digital-Out, Trigger Output, Run, Arm, Marker Output, System Clock
Output: impedance		50 Ω
Output: signal levels		3.3 V LVTTTL
Output: type		3.3V LVTTTL, TTL compatible for high impedance loads
Output: drive strength		Capable of driving 50 Ω loads, maximum drive strength ± 48 mA
Output: update rate		sampling clock

Option M4i.xxxx-DDS (multi-tone DDS firmware)

Number of available DDS cores per AWG card		23
DDS core routing options	software programmable	Routed cores can individually be activated for output Ch0: 8, 12, 16 or 20 cores; Ch1: 1 or 5 cores Ch2: 1 or 5 cores Ch3: 1 or 5 cores
DDS commands	individual for each core	Set Frequency,, Set Amplitude, Set Phase, Frequency Slope, Amplitude Slope
DDS commands	for all cores	Reset, Execute Now, Execute at Trigger/Timer
DDS command transfer mode		single or DMA
DDS time resolution		1.25 GS/s (800 ps)
DDS timer resolution	software programmable	83.2 ns up to 27.48 s with a resolution of 6.4 ns
DDS frequency range	per core programmable	0 Hz up to 1.25 GHz, frequencies above 625 MHz (Nyquist-Shannon) are mirrored
DDS amplitude range	per core programmable	-1.0 up to +1.0 with a resolution of $1/(2^{32})$ programmed in relation to output level: +1.0 = 100% output, -1.0 = 100% inverted output
DDS phase range	per core programmable	-360° to +360° with a resolution of $360/4096 = 0.088^\circ$
DDS command buffer	single mode	4k commands
	DMA mode	2G commands
Min user software to analog output latency	single mode	10 μ s
	DMA mode	20 μ s
Max continuous DDS command rate	single mode	400 kHz
	DMA mode	10 MHz
External trigger to DDS output change		ca. 554 ns (692 samples at 800 ps per sample)

Option M4i.xxxx-PulseGen

Number of internal pulse generators	4
Number of pulse generator output lines	3 (Existing multi-purpose outputs X0 to X2)
Time resolution of pulse generator	Pulse generator's sampling rate is derived from instrument's sampling rate and value can be read out. Maximum possible pulse generator update rate is 22xx: 156.25 MS/s (6.4 ns) 23xx: 156.25 MS/s (6.4 ns) 44xx: 125.00 MS/s (8.0 ns) 66xx: 156.25 MS/s (6.4 ns)
Programmable output modes	Single-shot, multiple repetitions on trigger, gated
Programmable trigger sources	Software, Card Trigger, Other Pulse Generator, XIO lines.
Programmable trigger gate	None, ARM state, RUN state
Programmable length (frequency)	2 to 4G samples in steps of 1 (32 bit)
Programmable width (duty cycle)	1 to 4G samples in steps of 1 (32 bit)
Programmable delay	0 to 4G samples in steps of 1 (32 bit)
Programmable loops	0 to 4G samples in steps of 1 (32 bit) - 0 = infinite
Output level of digital pulse generators	Please see section of multi-purpose I/O lines

Bandwidth and Slewrate

	Filter	Output Amplitude	M4i.663x-x8 M4x.663x-x8 DN2.663-xx DN6.663-xx DN2.82x-02	M4i.662x-x8 M4x.662x-x8 DN2.662-xx DN6.662-xx DN2.82x-04
Maximum Output Rate			1.25 GS/s	625 MS/s
-3dB Bandwidth	no Filter	±480 mV	400 MHz	200 MHz
-3dB Bandwidth	no Filter	±1000 mV	320 MHz	200 MHz
-3dB Bandwidth	no Filter	±2000 mV	320 MHz	200 MHz
-3dB Bandwidth	Filter	all	65 MHz	65 MHz
Slewrate	no Filter	±480 mV	4.5 V/ns	2.25 V/ns

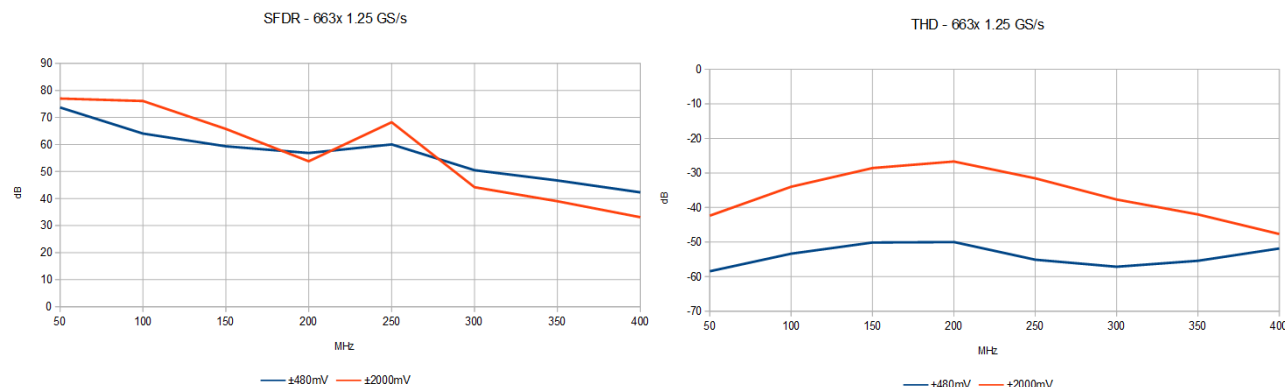
Dynamic Parameters

	M4i.662x-x8 M4x.662x-x8 DN2.662-xx DN6.662-xx DN2.82x-04						
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.663x-x8 M4x.663x-x8 DN2.663-xx DN6.663-xx DN2.82x-02						
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.

SFDR and THD versus signal frequency



- Measurements done with a spectrum analyzer bandwidth of 1.5 GHz
- Please note that the bandwidth of the high range output is limited to 320 MHz
- Please note that the output bandwidth limit also affects the THD as harmonics higher than the bandwidth are filtered

M4i Specific Technical Data

Connectors

Analog Inputs/Analog Outputs
 Trigger 0 Input
 Clock Input
 Trigger 1 Input
 Clock Output
 Multi Purpose I/O

SMA female (one for each single-ended input)
 SMA female
 SMA female
 MMCX female
 MMCX female
 MMCX female (3 lines)

Cable-Type: Cab-3mA-xx-xx
 Cable-Type: Cab-3mA-xx-xx
 Cable-Type: Cab-3mA-xx-xx
 Cable-Type: Cab-1m-xx-xx
 Cable-Type: Cab-1m-xx-xx
 Cable-Type: Cab-1m-xx-xx

Connection Cycles

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

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

Environmental and Physical Details

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

PCI Express specific details

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

Certification, Compliance, Warranty

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

Power Consumption

		PCI EXPRESS		
		3.3V	12 V	Total
M4i.6620-x8	Typical values: All channels activated, Sample rate: 625 MSps	0.2 A	2.5 A	31 W
M4i.6621-x8	Output signal: 31.25 MHz sine wave, Output level: +/- 1 V into 50 Ω load	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	0.2 A	2.6 A	32 W
M4i.6621-x8	Output signal: 31.25 MHz sine wave, Output level: +/- 2.5 V into 50 Ω load	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	0.2 A	2.7 A	33 W
M4i.6631-x8	Output signal: 31.25 MHz sine wave, Output level: +/- 1 V into 50 Ω load	0.2 A	3.0 A	36 W
M4i.6630-x8	Typical values: All channels activated, Sample rate: 1.25 GSps	0.2 A	2.9 A	35 W
M4i.6631-x8	Output signal: 31.25 MHz sine wave, Output level: +/- 2.0 V into 50 Ω load	0.2 A	3.3 A	40 W

MTBF

MTBF	400.000
------	---------

M4x Specific Technical Data**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	SMA female	Cable-Type: Cab-3mA-xx-xx
Clock Output	SMA female	Cable-Type: Cab-3mA-xx-xx
Multi Purpose I/O	MMCX female (3 lines)	Cable-Type: Cab-1m-xx-xx

Connection Cycles

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

SMA connector	500 connection cycles
MMCX connector	500 connection cycles
PXIe connector	250 connection cycles

Environmental and Physical Details

Dimension (Single Card)	(PCB only)	160 mm x 100 mm (Standard 3U)
Width		2 slots
Weight (M4x.44xx series)	maximum	340 g
Weight (M4x.22xx, M4x.66xx series)	maximum	450 g
Warm up time		10 minutes
Operating temperature		0°C to 50°C
Storage temperature		-10°C to 70°C
Humidity		10% to 90%
Dimension of packing	1 or 2 cards	470 mm x 250 mm x 130 cm
Volume weight of packing	1 or 2 cards	4 kg

PXI Express specific details

PXIe slot type	4 Lanes, PCIe Gen 2 (x4 Gen2)
PXIe hybrid slot compatibility	Fully compatible
Sustained streaming mode (Card-to-System: M4x.22xx, M4x.44xx)	> 1.7 GB/s (measured with a chipset supporting a TLP size of 256 bytes, using PXIe x4 Gen2)
Sustained streaming mode (System-to-Card: M4x.66xx)	> 1.4 GB/s (measured with a chipset supporting a TLP size of 256 bytes, using PXIe x4 Gen2)

Certification, Compliance, Warranty

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

Power Consumption

		PCI EXPRESS		
		3.3V	12 V	Total
M4x.6620-x4	Typical values: All channels activated, Sample rate: 625 MSps	0.25 A	2.5 A	31 W
M4x.6621-x4	Output signal: 31.25 MHz sine wave, Output level: +/- 1 V into 50 Ω load	0.25 A	2.7 A	33 W
M4x.6622-x4		0.25 A	3.0 A	36 W
M4x.6620-x4	Typical values: All channels activated, Sample rate: 625 MSps	0.25 A	2.6 A	32 W
M4x.6621-x4	Output signal: 31.25 MHz sine wave, Output level: +/- 2.5 V into 50 Ω load	0.25 A	2.9 A	35 W
M4x.6622-x4		0.25 A	3.3 A	40 W
M4x.6630-x4	Typical values: All channels activated, Sample rate: 1.25 GSps	0.25 A	2.7 A	33 W
M4x.6631-x4	Output signal: 31.25 MHz sine wave, Output level: +/- 1 V into 50 Ω load	0.25 A	3.0 A	36 W
M4x.6630-x4	Typical values: All channels activated, Sample rate: 1.25 GSps	0.25 A	2.9 A	35 W
M4x.6631-x4	Output signal: 31.25 MHz sine wave, Output level: +/- 2.0 V into 50 Ω load	0.25 A	3.3 A	40 W

MTBF

MTBF	400.000 hours
------	---------------

Order Information**M4i 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), IVI, .NET, Delphi, Java, Python, Julia and a Base license of the measurement 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

Options

Order no.	Option
M4i.663x-hbw	High bandwidth option 600 MHz. Available for 663x products with 1.25 GS/s only. Output level limited to ±480 mV into 50 Ω. Needs external reconstruction filter. One option needed per AWG card.

Firmware Options

Order no.	Option
M4i.66xx-DDS	Firmware Option multi-carrier DDS mode: adds 23 programmable DDS cores to the AWG. Each core can be programmed with single commands for frequency, amplitude, phase, frequency slope, amplitude slope.
M4i.xxxx-PulseGen	Firmware Option: adds 4 freely programmable digital pulse generators that use the XIO lines for output (later installation by firmware-upgrade available)

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	Cab-3mA-3mA-80		Cab-3f-3mA-80	
Analog/Clock-In/Trig-In	200 cm	Cab-3mA-9m-200	Cab-3mA-9f-200	Cab-3mA-3mA-200		Cab-3f-3mA-200	
Probes (short)	5 cm		Cab-3mA-9f-5				

Standard Cables

for Connections	Length	Order no.					
		to BNC male	to BNC female	to SMA male	to SMA female	to SMB female	
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-3f200	
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						

Services

Order no.	
Recal	Recalibration at Spectrum incl. calibration protocol

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/M2p/M5i cards
SPc-SCAPP	Spectrum's CUDA Access for Parallel Processing - SDK for direct data transfer between Spectrum card and CUDA GPU. Includes RDMA activation and examples.

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

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

M4x 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), IVI, .NET, Delphi, Java, Python, Julia and a Base license of the measurement software SBench 6 are included.

Adapter cables are not included. Please order separately!

PXI Express x4

Order no.	Bandwidth	Standard mem	1 channel	2 channels	4 channels
M4x.6620-x4	200 MHz	2 GSample	625 MS/s		
M4x.6621-x4	200 MHz	2 GSample	625 MS/s	625 MS/s	
M4x.6622-x4	200 MHz	2 GSample	625 MS/s	625 MS/s	625 MS/s
M4x.6630-x4	400 MHz	2 GSample	1.25 GS/s		
M4x.6631-x4	400 MHz	2 GSample	1.25 GS/s	1.25 GS/s	

Options

Order no.	Option
M4i.663x-hbw	High bandwidth option 600 MHz. Available for 663x products with 1.25 GS/s only. Output level limited to ± 480 mV into 50 Ω . Needs external reconstruction filter. One option needed per AWG card.

Firmware Options

Order no.	Option
M4i.66xx-DDS	Firmware Option multi-carrier DDS mode: adds 23 programmable DDS cores to the AWG. Each core can be programmed with single commands for frequency, amplitude, phase, frequency slope, amplitude slope.
M4i.xxxx-PulseGen	Firmware Option: adds 4 freely programmable digital pulse generators that use the XIO lines for output (later installation by firmware-upgrade available)

Services

Order no.	
Recal	Recalibration at Spectrum incl. calibration protocol

Standard Cables

for Connections	Length	Order no.					
		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	Cab-3mA-3mA-80		Cab-3f3mA-80	
Analog/Clock-In/Trig-In	200 cm	Cab-3mA-9m-200	Cab-3mA-9f-200	Cab-3mA-3mA-200		Cab-3f3mA-200	
Probes (short)	5 cm		Cab-3mA-9f-5				
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-3f200	
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/M2p/M5i cards
SPc-SCAPP	Spectrum's CUDA Access for Parallel Processing - SDK for direct data transfer between Spectrum card and CUDA GPU. Includes RDMA activation and examples.

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

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

Hardware Installation

ESD Precautions

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

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



Sources of noise

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

Cooling Precautions

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

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



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

Connector Handling Precautions

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

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



M4i PCIe Cards

System Requirements

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

Installing the M4i board in the system

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

Installing a single board without any options

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

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



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



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

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

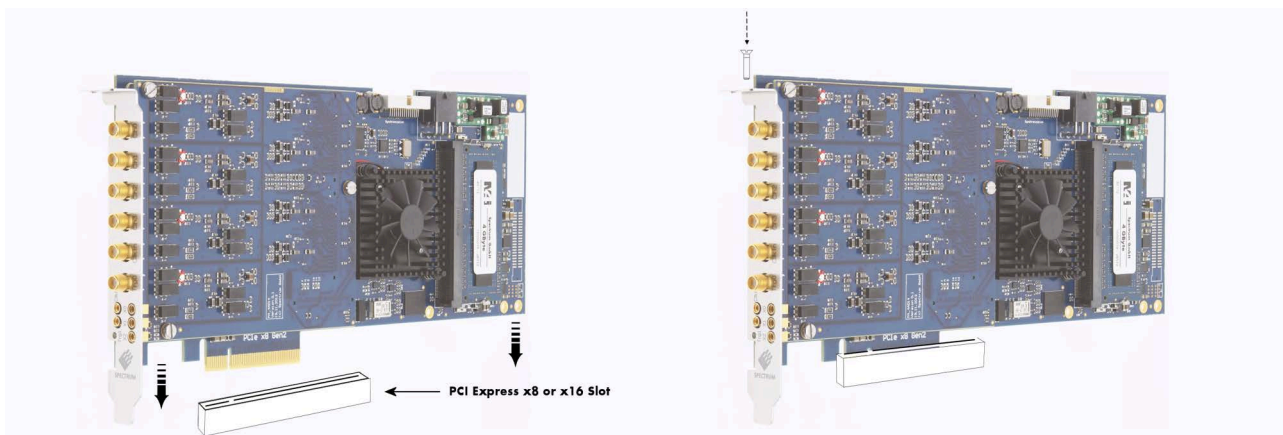


Image 5: Mounting M4i PCIe card into connector

Additional notes on PCIe x16 slot retention

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

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

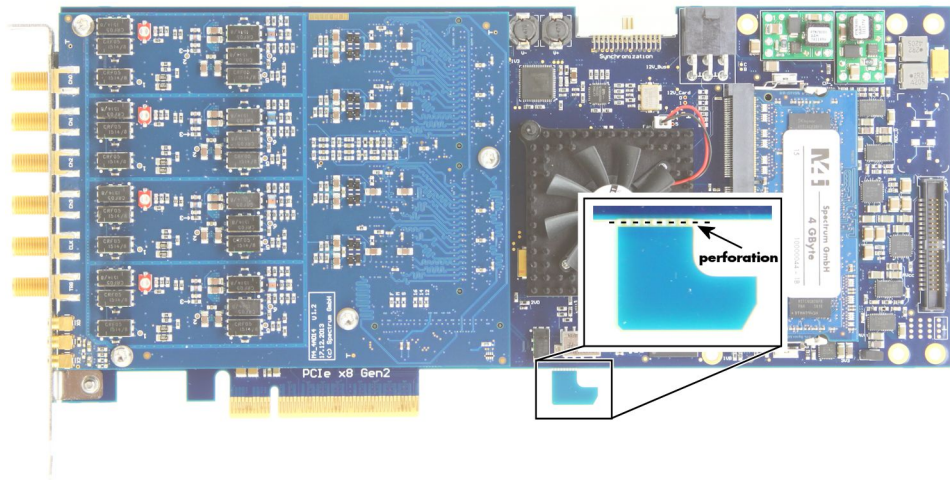


Image 6: M4i card slot retention with perforation

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



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



Providing additional power to M4i.xxxx-x8 cards

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

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

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

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

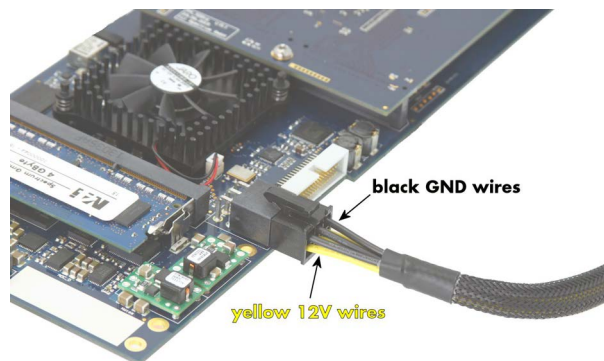


Image 7: M4i card additional power connection usage

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



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

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

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



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



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



Image 8: M4i card with digital option mechanical installation and position of screws

Installing multiple boards synchronized by star-hub option

Precautions M4i cards with Star-Hub option (SH8ex)

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

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

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

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



Hooking up the boards

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

Only use the included flat ribbon cables.

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

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

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

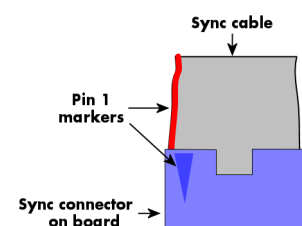
Mounting the wired boards

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

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



Image 9: M4i cards with star-hub ex installed and connecting cables



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

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



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



Shipment of systems with Spectrum cards installed

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

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

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

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

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

M4x PXIe Cards

System Requirements

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

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



Installing the M4x board in the system

Installing a single board without any options

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

While inserting the board take care not to tilt it.



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

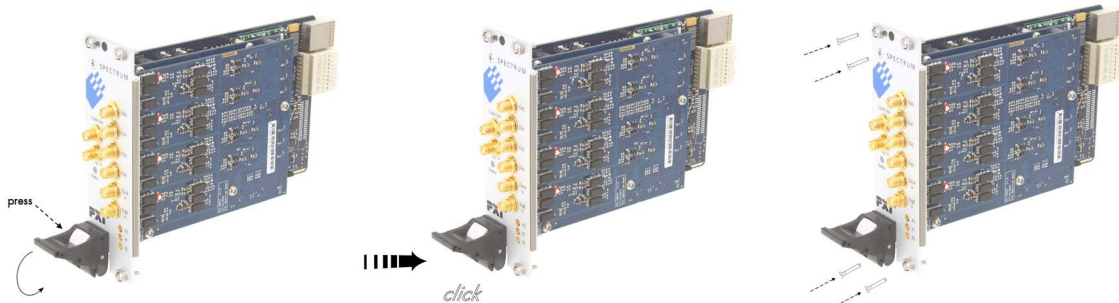


Image 10: Installation of PXIe cards with connector handling and mounting screws

Software Driver Installation and Driver Update

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

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



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

Windows

Before initial installation

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

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

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

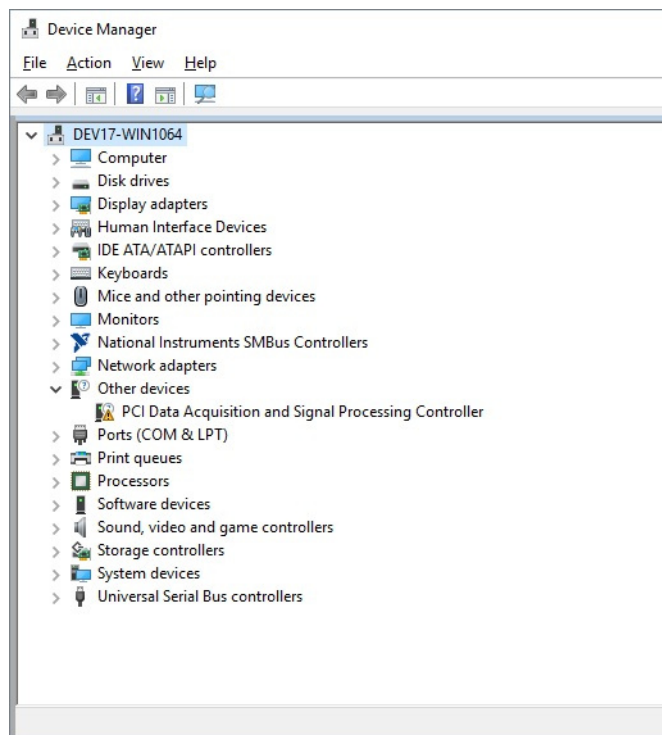


Image 11: Windows Device Manager showing a new Spectrum card

Running the driver Installer/Update

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

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

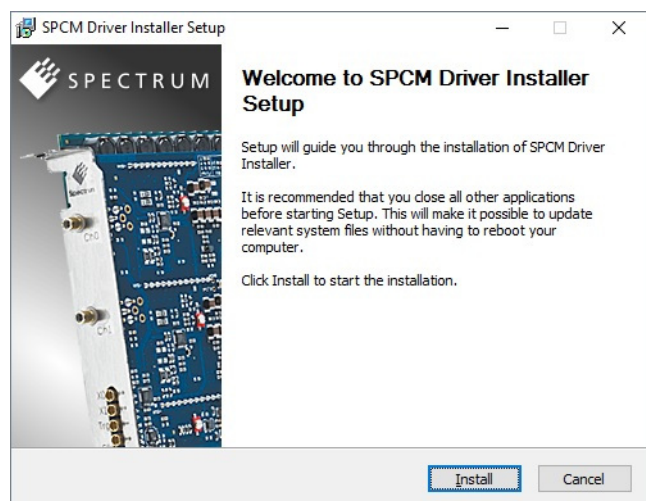


Image 12: Spectrum Driver Installer Welcome Screen

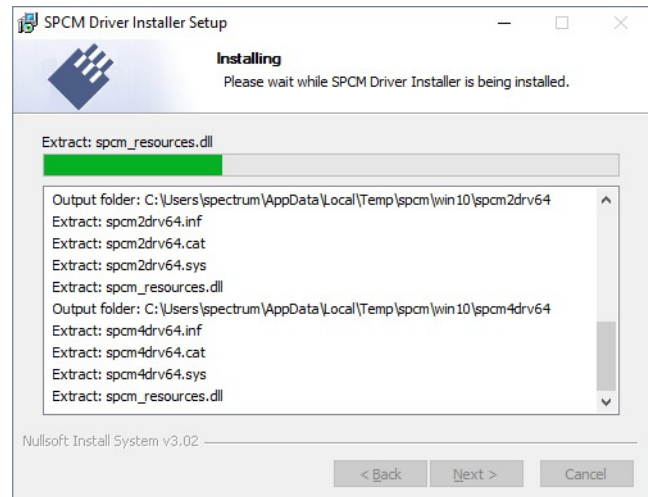


Image 13: Spectrum Driver Installer - Progress

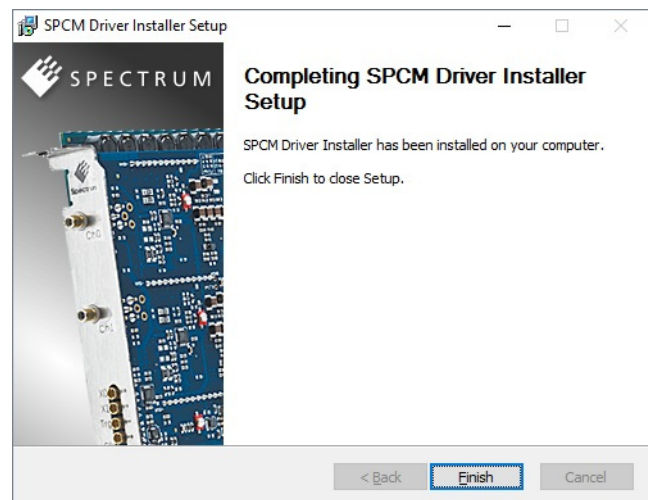


Image 14: Spectrum Driver Installer - finished

After installation

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

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

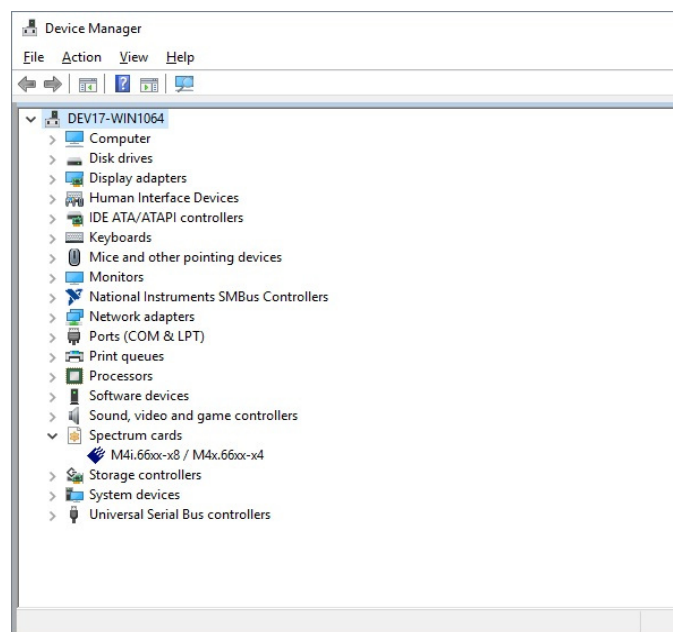


Image 15: Windows Device Manager showing properly installed Spectrum card

Linux

Overview

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

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

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

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



Driver Installation with Installation Script

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

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



Login as root

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

Call the `install.sh <install_path> script`

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

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

Udev support

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

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

Start the driver

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

For M2i and M3i cards:

```
modprobe spcm
```

For M5i, M4i, M4x and M2p cards:

```
modprobe spcm4
```

Get first driver info

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

For M2i and M3i cards:

```
cat /proc/spcm_cards
```

For M5i, M4i, M4x and M2p cards:

```
cat /proc/spcm4_cards
```

Stop the driver

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

For M2i and M3i cards:

```
modprobe -r spcm
```

For M5i, M4i, M4x and M2p cards:

```
modprobe -r spcm4
```

Standard Driver Update

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

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

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

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

Login as root

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

Call the compile script

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

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

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

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

Update of a self compiled kernel driver

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

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

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

```
sh install_libonly.sh
```

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

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

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

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

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

```
sh install_libonly.sh
```

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

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

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

Installation from Spectrum Repository

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

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

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

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

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

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

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

Then run

```
sudo apt update
```

to update the repository information.

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

```
sudo apt install sbench6
```

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

RPM based distributions

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

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

The repository information will be updated automatically.

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

```
sudo zypper install SBench6
```

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

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

The repository information will be updated automatically.

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

```
sudo dnf install SBench6
```

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

Control Center

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

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

Installation

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

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

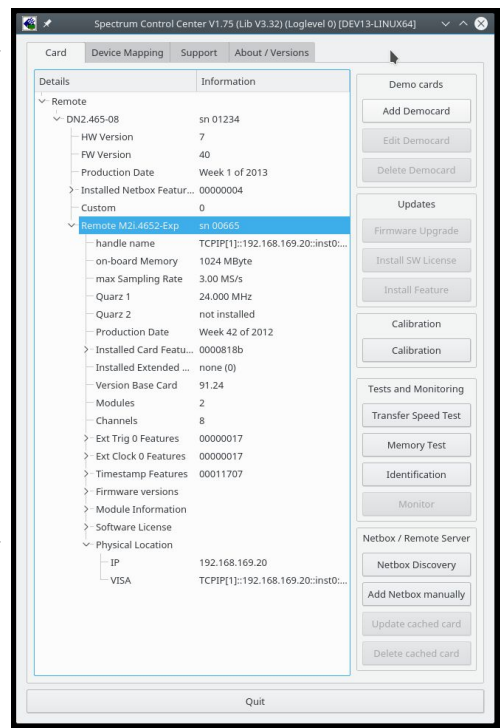


Image 16: Device Manager showing a new Spectrum card

Manual Installation

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

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

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

Troubleshooting

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

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

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

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

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

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

Software

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

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



Software Overview

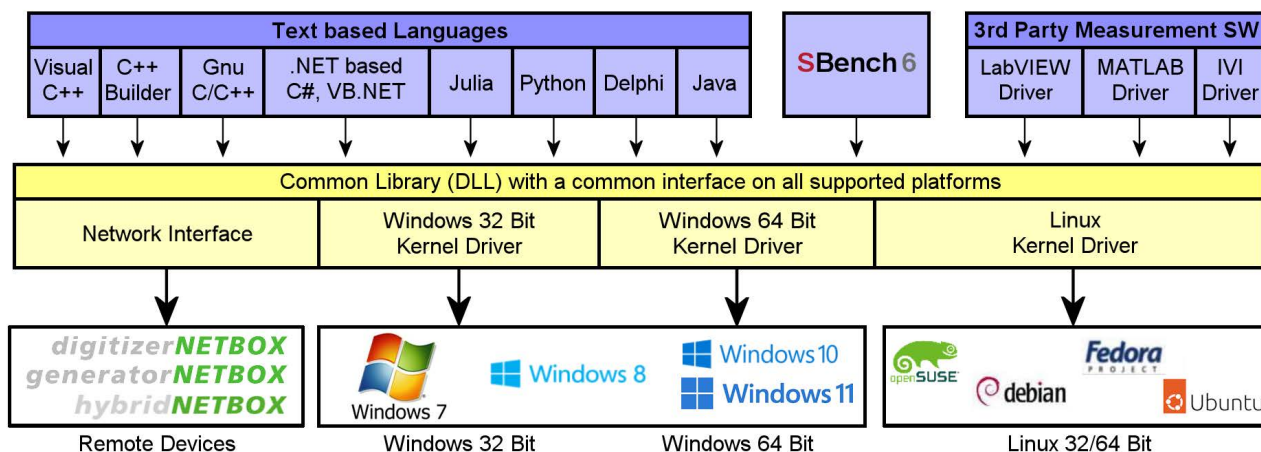


Image 17: Spectrum Kernel Driver, API Library and Software structure

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

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

Card Control Center

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

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

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

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

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

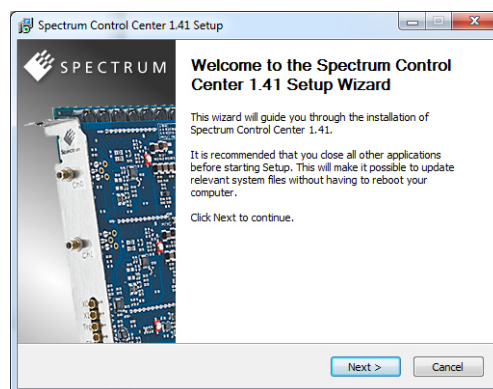


Image 18: Spectrum Control Center Installer



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

Discovery of Remote Cards, digitizerNETBOX/generatorNETBOX/hybridNETBOX products

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

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

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

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

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

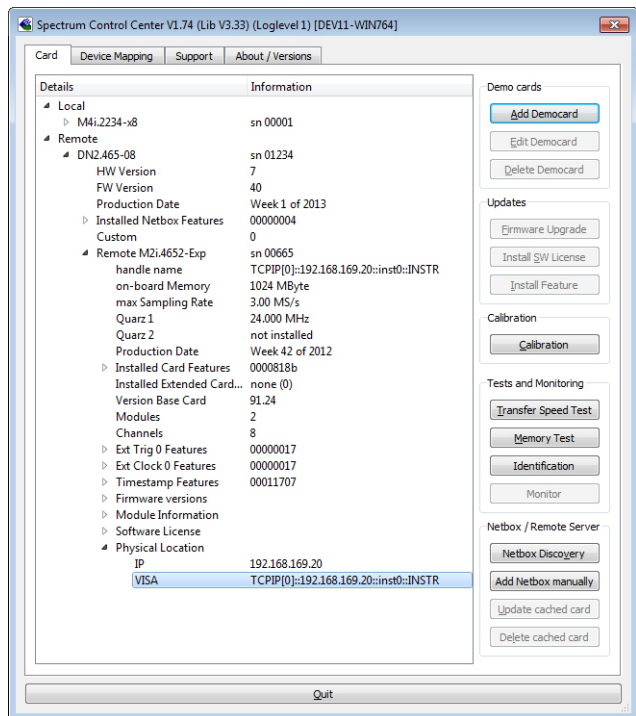


Image 19: Spectrum Control Center showing detail card information

Enter IP Address of digitizerNETBOX/generatorNETBOX/hybridNETBOX manually

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

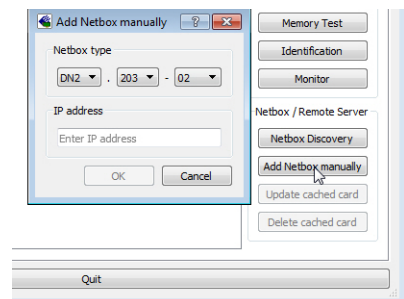


Image 20: Spectrum Control Center - entering an IP address for a NETBOX

Wake On LAN of digitizerNETBOX/generatorNETBOX/hybridNETBOX

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

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

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

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

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

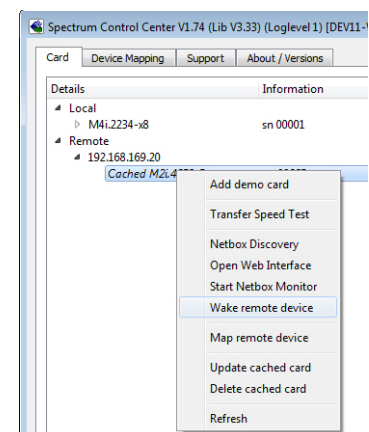


Image 21: Spectrum Control Center: wake on LAN for a cached card

Netbox Monitor

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

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

```
NetboxMonitor 192.168.169.22
```

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



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

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

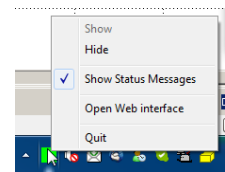


Image 22: Netbox Monitor activation

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

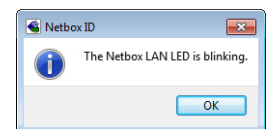
The status colors:

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

Device identification

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

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



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

Hardware information

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

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

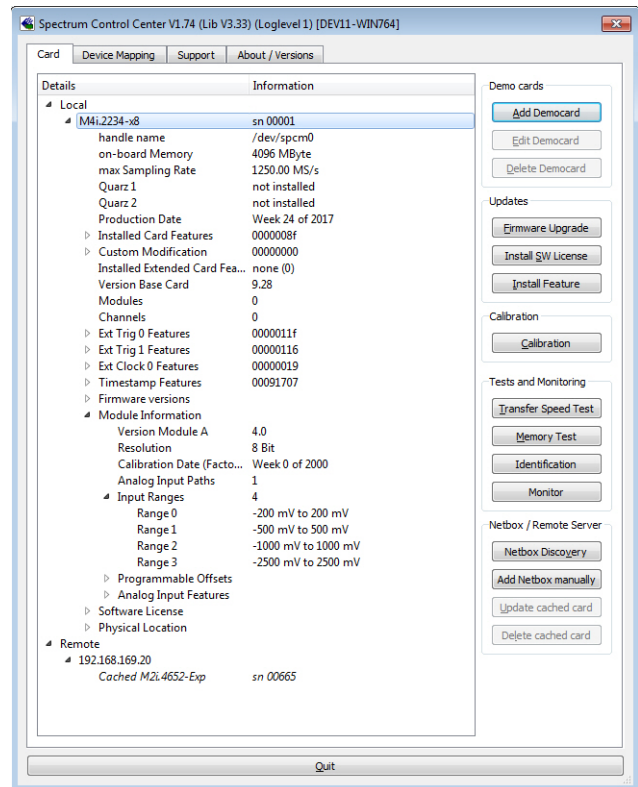


Image 23: Spectrum Control Center: detailed hardware information on installed card

Firmware information

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

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

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

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

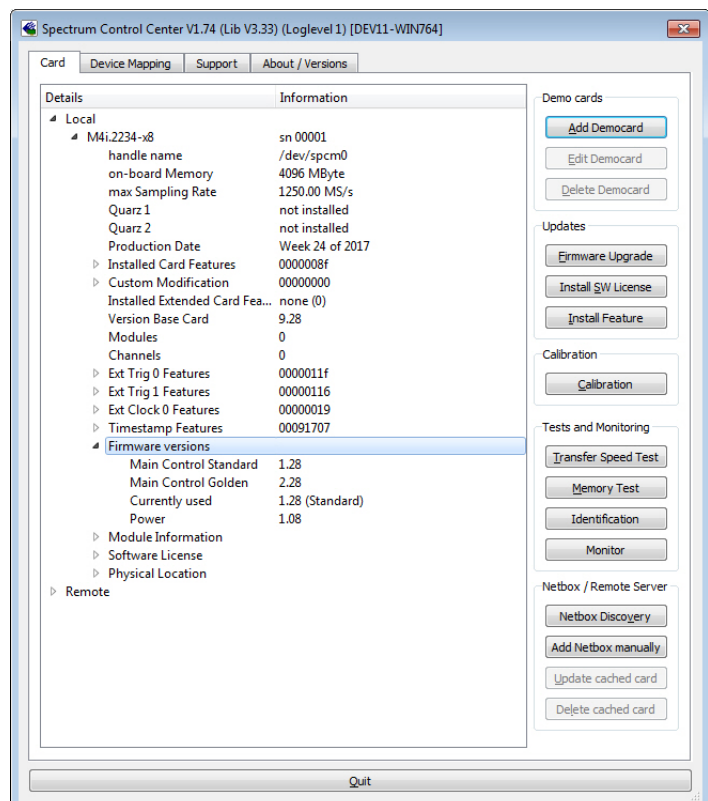


Image 24: Spectrum Control Center - showing firmware information of an installed card

Software License information

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

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

The number of demo starts left can be seen here.

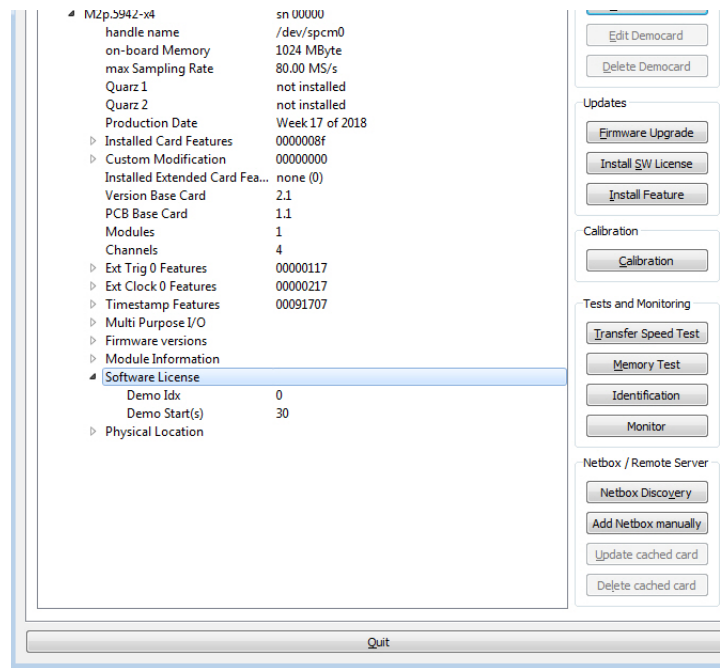


Image 25: Spectrum Control Center - showing firmware information of an installed card

Driver information

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

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

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

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

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

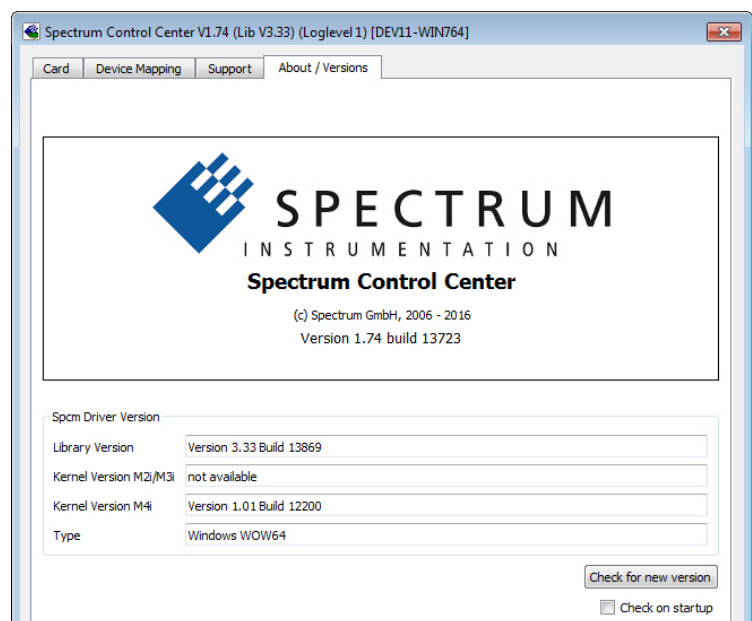


Image 26: Spectrum Control Center - showing driver information details

Installing and removing Demo cards

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

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

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

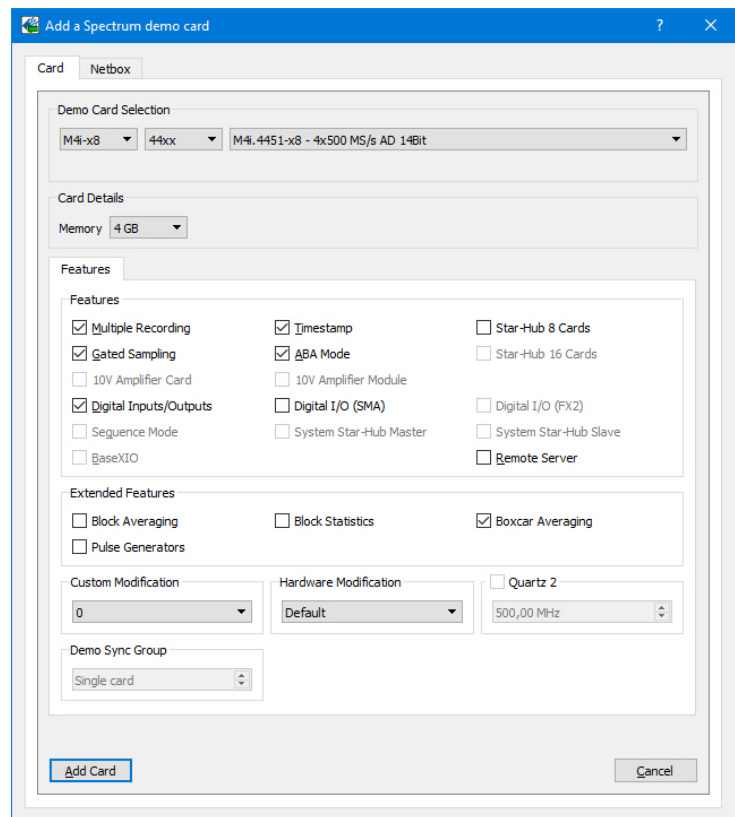


Image 27: Spectrum Control Center - adding a demo card to the system

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



Feature upgrade

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

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

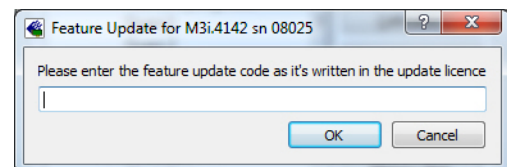


Image 28: Spectrum Control Center - feature update, code entry

Software License upgrade

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

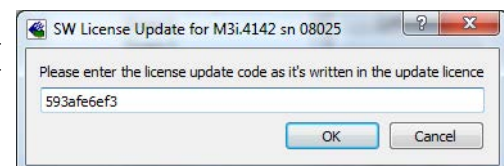


Image 29: Spectrum Control Center - software license install

Performing card calibration (A/D only)

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

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

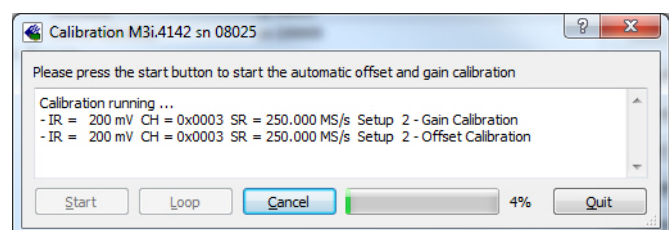


Image 30: Spectrum Control Center - running an on-board calibration

Performing memory test

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

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

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

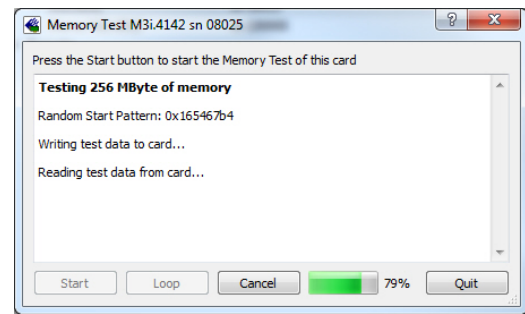


Image 31: Spectrum Control Center - performing memory test

Transfer speed test

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

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

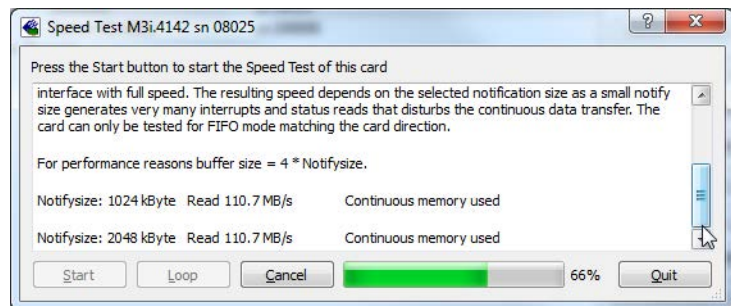


Image 32: Spectrum Control Center - running a transfer speed test of one card

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

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

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

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

Debug logging for support cases

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

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

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

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

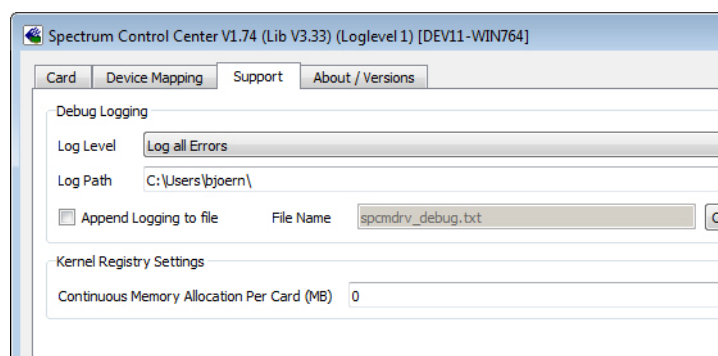


Image 33: Spectrum Control Center - activate debug logging for support cases



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

Device mapping

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

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

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

The remote digitizerNETBOX device is mapped to spcm2.

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

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

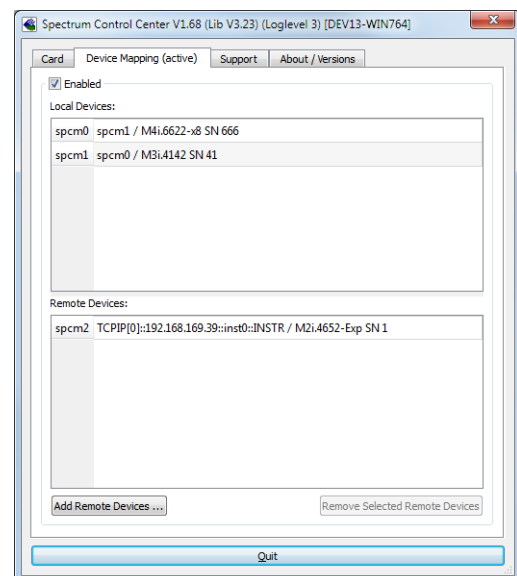


Image 34: Spectrum Control Center - using device mapping

Firmware upgrade

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

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

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

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

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

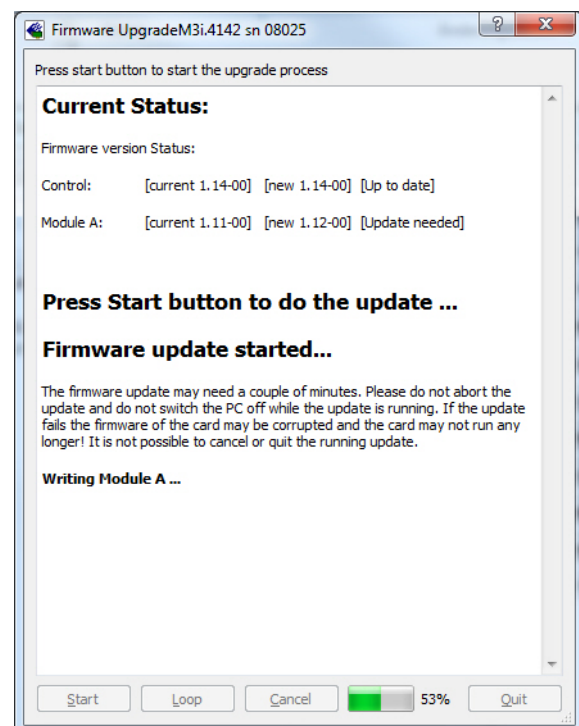


Image 35: Spectrum Control Center - doing a firmware update for one device



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



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

Accessing the hardware with SBench 6

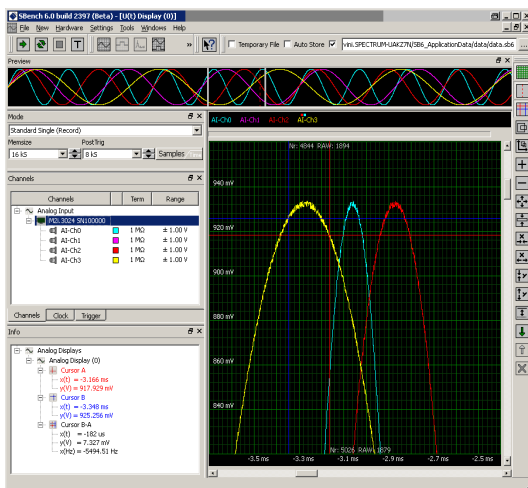


Image 36: SBench 6 overview of main functionality with demo data

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

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

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

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

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

C/C++ Driver Interface

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

Header files

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

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

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

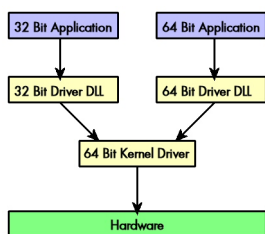
Example for including the header files:

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



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

General Information on Windows 64 bit drivers



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

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

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

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

Include Driver

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

Examples

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

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

Microsoft Visual C++ 2005 and newer 64 Bit

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

Include Driver

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

Linux Gnu C/C++ 32/64 Bit

Include Driver

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

```

COMPILER = gcc
EXECUTABLE = test_prg
LIBS = -lspcm_linux

OBJECTS = test.o\
          test2.o

all: $(EXECUTABLE)

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

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

Examples

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

C++ for .NET

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

Other Windows C/C++ compilers 32 Bit

Include Driver

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

Example of function loading:

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

Other Windows C/C++ compilers 64 Bit

Include Driver

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

Example of function loading:

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

Driver functions

The driver contains seven main functions to access the hardware.

Own types used by our drivers

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

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

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

Notation of variables and functions

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

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

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

Function `spcm_hOpen`

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

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

Function `spcm_hOpen (const char* szDeviceName):`

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

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

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

Example for local installed cards

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

Example for digitizerNETBOX/generatorNETBOX and remote installed cards

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

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

Function `spcm_vClose`

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

Function `spcm_vClose:`

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

Example:

```
spcm_vClose (hDrv);
```

Function `spcm_dwSetParam`

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

Function `spcm_dwSetParam`

```

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

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

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

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

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

```

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

Example:

```

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

```

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

Function `spcm_dwGetParam`

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

Function `spcm_dwGetParam`

```

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

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

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

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

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

```

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

Example:

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

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

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

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

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

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

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

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

Function spcm_dwGetContBuf

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

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

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



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



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

Function `spcm_dwDefTransfer`

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

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

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

Function `spcm_dwDefTransfer`

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

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

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

Example:

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

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

Function `spcm_dwInvalidateBuf`

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

Function `spcm_dwInvalidateBuf`

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

Function `spcm_dwGetErrorInfo`

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

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



Function `spcm_dwGetErrorInfo`

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

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

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

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

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

Example:

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

Delphi (Pascal) Programming Interface

Driver interface

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

file `spcm_win32.pas`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

file spcm_win64.pas

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

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

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

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

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

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

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

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

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

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

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

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

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

file SpcRegs.pas

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

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

file SpcErr.pas

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

Including the driver files

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

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

  SpcRegs, SpcErr, spcm_win32;
```

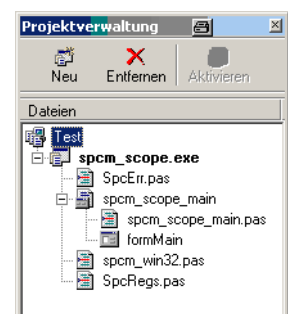


Image 37: Structure of the Delphi examples

Examples

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

spcm_scope

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

The program shows the following steps:

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

.NET programming languages

Library

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

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

Declaration

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



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

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

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

Using C#

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

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

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

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

Example for digitizerNETBOX/generatorNETBOX and remotely installed cards:

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

Using Managed C++/CLI

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

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

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

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

Example for digitizerNETBOX/generatorNETBOX and remotely installed cards:

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

Using VB.NET

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

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

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

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

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

Example for digitizerNETBOX/generatorNETBOX and remotely installed cards:

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

Using J#

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

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

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

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

Example for digitizerNETBOX/generatorNETBOX and remotely installed cards:

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

Python Programming Interface and Examples

Driver interface

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

file pyspcm.py

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

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

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

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

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

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

...
```

file regs.py

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

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

file spcerr.py

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

Examples

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

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



Java Programming Interface and Examples

Driver interface

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

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

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

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

SpcmDrv32.java / SpcmDrv64.java

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

```
...

public interface SpcmWin64 extends StdCallLibrary {

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

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

    int spcm_dwInvalidateBuf (long hDevice, int lBufType);

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

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

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

SpcmRegs.java

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

```
...

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

SpcmErrors.java

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

Examples

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

Julia Programming Interface and Examples

Driver interface

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

file `spcm_drv.jl`

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

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

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

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

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

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

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

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

file `regs.jl`

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

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

file `spcerr.jl`

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

Examples

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

LabVIEW driver and examples

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

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

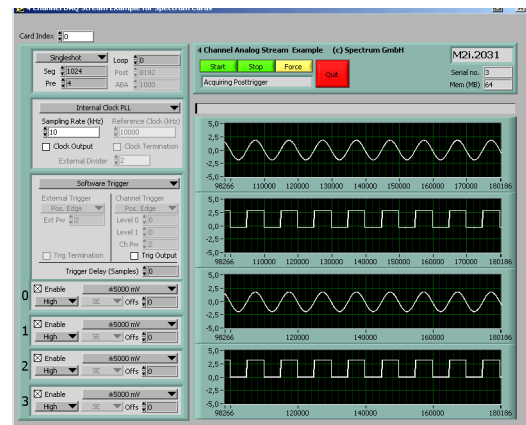


Image 38: LabVIEW driver oscilloscope example

MATLAB driver and examples

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

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

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

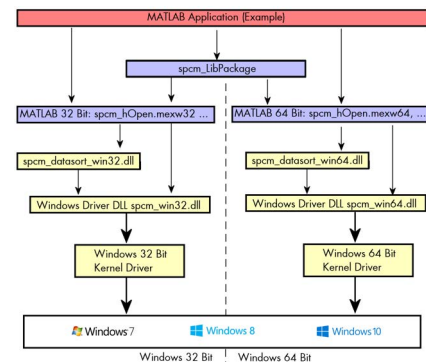


Image 39: Spectrum MATLAB driver structure

SCAPP – CUDA GPU based data processing

Spectrum's CUDA Access for Parallel Processing

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

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

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

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

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

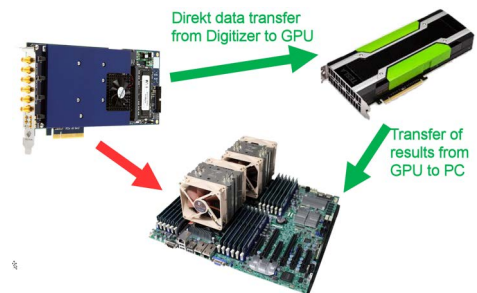


Image 40: GPU usage with SCAPP SDK: data transfer options

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

Programming the Board

Overview

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

Register tables

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

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

Table 8: Spectrum API: Command register and basic commands

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

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

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



Programming examples

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

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

Complete C/C++ Example

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

#include <stdio.h>

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

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

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

    return 0;
}
```

Initialization

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

Open/Close Example

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

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

... do any work with the driver

spcm_vClose (hDrv);
return 0;
```

Initialization of Remote Products

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

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

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

...
```

Multiple cards are opened by indexing the remote card number:

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

Error handling

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

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



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

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

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

```
char szErrorText[ERRORTXTLEN];

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

This short program then would generate a printout as:

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

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



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

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

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

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

Gathering information from the card

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

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

Card type

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

Table 9: Spectrum API: Card Type Register

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

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

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

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

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

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

..

Table 10: Spectrum API: list of card type codes for M4i.66xx series

Card type	Card type as defined in regs.h	Value hexadecimal	Value decimal	Card type	Card type as defined in regs.h	Value hexadecimal	Value decimal
M4i.6620-x8	TYP_M4i6620_X8	76620h	484896	M4i.6630-x8	TYP_M4i6630_X8	76630h	484912
M4i.6621-x8	TYP_M4i6621_X8	76621h	484897	M4i.6631-x8	TYP_M4i6631_X8	76631h	484913
M4i.6622-x8	TYP_M4i6622_X8	76622h	484898				

Table 11: Spectrum API: list of card type codes for M4x.66xx series

Card type	Card type as defined in regs.h	Value hexadecimal	Value decimal	Card type	Card type as defined in regs.h	Value hexadecimal	Value decimal
M4x.6620-x4	TYP_M4X6620_X4	86620h	550432	M4x.6630-x4	TYP_M4X6630_X4	86630h	550448
M4x.6621-x4	TYP_M4X6621_X4	86621h	550433	M4x.6631-x4	TYP_M4X6631_X4	86631h	550449
M4x.6622-x4	TYP_M4X6622_X4	86622h	550434				

Hardware and PCB version

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

Table 12: Spectrum API: hardware and PCB version register overview

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

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

Table 13: Spectrum API: extension module hardware and PCB version register

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

Reading currently used PXI slot No. (M4x only)

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

Table 14: Spectrum API: register for reading back the PXIe card slot number

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

Firmware versions

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

Table 15: Spectrum API: Register overview of firmware versions

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

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

Table 16: Spectrum API: Register overview of reading current firmware

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

Production date

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

Table 17: Spectrum API: production date register

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

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

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

Last calibration date (analog cards only)

This register informs you about the date of the last factory calibration. When receiving a new card this date is similar to the delivery date when the production calibration is done. When returning the card to calibration this information is updated. This date is not updated when the user does an on-board calibration. The date is returned as one 32 bit long word. The lower word is holding the information about the year, while the upper word informs about the week of the year.

Table 18: Spectrum API: calibration date register

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

Serial number

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

Table 19: Spectrum API: hardware serial number register

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

Maximum possible sampling rate

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

Table 20: Spectrum API: maximum sampling rate register

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

Installed memory

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

Table 21: Spectrum API: installed memory registers. 32 bit read is limited to a maximum of 1 GByte

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

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

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

Installed features and options

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

Table 22: Spectrum API: Feature Register and available feature flags

Register	Value	Direction	Description
SPC_PCIFEATURES	2120	read	PCI feature register. Holds the installed features and options as a bitfield. The read value must be masked out with one of the masks below to get information about one certain feature.
SPCM_FEAT_MULTI	1h		Is set if the feature Multiple Recording / Multiple Replay is available.
SPCM_FEAT_GATE	2h		Is set if the feature Gated Sampling / Gated Replay is available.
SPCM_FEAT_DIGITAL	4h		Is set if the feature Digital Inputs / Digital Outputs is available.

SPCM_FEAT_TIMESTAMP	8h	Is set if the feature Timestamp is available.
SPCM_FEAT_STARHUB6_EXTM	20h	Is set on the card, that carries the star-hub extension or piggy-back module for synchronizing up to 6 cards (M2p).
SPCM_FEAT_STARHUB8_EXTM	20h	Is set on the card, that carries the star-hub extension or piggy-back module for synchronizing up to 8 cards (M4i).
SPCM_FEAT_STARHUB4	20h	Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 4 cards (M3i).
SPCM_FEAT_STARHUB5	20h	Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 5 cards (M2i).
SPCM_FEAT_STARHUB16_EXTM	40h	Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 16 cards (M2p).
SPCM_FEAT_STARHUB8	40h	Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 8 cards (M3i and M5i).
SPCM_FEAT_STARHUB16	40h	Is set on the card, that carries the star-hub piggy-back module for synchronizing up to 16 cards (M2i).
SPCM_FEAT_ABA	80h	Is set if the feature ABA mode is available.
SPCM_FEAT_BASEXIO	100h	Is set if the extra BaseXIO option is installed. The lines can be used for asynchronous digital I/O, extra trigger or timestamp reference signal input.
SPCM_FEAT_AMPLIFIER_10V	200h	Arbitrary Waveform Generators only: card has additional set of calibration values for amplifier card.
SPCM_FEAT_STARHUBSYMASTER	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_STARHUBSYSLAVE	4000h	Is set in the card that carries a System Star-Hub Slave module to connect with System Star-Hub master systems (M2i).
SPCM_FEAT_NETBOX	8000h	The card is physically mounted within a digitizerNETBOX, generatorNETBOX or hybridNETBOX.
SPCM_FEAT_REMOTESERVER	10000h	Support for the Spectrum Remote Server option is installed on this card.
SPCM_FEAT_SCAPP	20000h	Support for the SCAPP option allowing CUDA RDMA access to supported graphics cards for GPU calculations (M5i, M4i and M2p).
SPCM_FEAT_DIG16_SMB	40000h	M2p: Set if option M2p.xxxx-DigSMB is installed, adding 16 additional digital I/Os via SMB connectors.
SPCM_FEAT_DIG16_FX2	80000h	M2p: Set if option M2p.xxxx-DigFX2 is installed, adding 16 additional digital I/Os via FX2 multipin connectors.
SPCM_FEAT_DIGITALBWFILTER	100000h	A digital (boxcar) bandwidth filter is available that can be globally enabled/disabled for all channels.
SPCM_FEAT_CUSTOMMOD_MASK	F0000000h	The upper 4 bit of the feature register is used to mark special custom modifications. This is only used if the card has been specially customized. Please refer to the extra documentation for the meaning of the custom modifications. (M2i/M3i). For M5i, M4i, M4x and M2p cards see „Custom modifications“ chapter instead.

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

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

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

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

Installed extended Options and Features

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

Table 23: Spectrum API: Extended feature register and available extended feature flags

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

Miscellaneous Card Information

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

Table 24: Spectrum API: register overview of miscellaneous cards information

Register	Value	Direction	Description
SPC_MIINST_MODULES	1100	read	Number of the installed front-end modules on the card.
SPC_MIINST_CHPERMODULE	1110	read	Number of channels installed on one front-end module.
SPC_MIINST_BYTESPERSAMPLE	1120	read	Number of bytes used in memory by one sample.
SPC_MIINST_BITSPERSAMPLE	1125	read	Resolution of the samples in bits.
SPC_MIINST_MAXADCVALUE	1126	read	Decimal code of the full scale value.
SPC_MIINST_MINEXTCLOCK	1145	read	Minimum external clock that can be fed in for direct external clock (if available for card model).

Table 24: Spectrum API: register overview of miscellaneous cards information

Register	Value	Direction	Description
SPC_MIINST_MAXEXTCLOCK	1146	read	Maximum external clock that can be fed in for direct external clock (if available for card model).
SPC_MIINST_MINEXTREFCLOCK	1148	read	Minimum external clock that can be fed in as a reference clock.
SPC_MIINST_MAXEXTREFCLOCK	1149	read	Maximum external clock that can be fed in as a reference clock.
SPC_MIINST_ISDEMOCARD	1175	read	Returns a value other than zero, if the card is a demo card.

Function type of the card

This register returns the basic type of the card:

Table 25: Spectrum API: register card function type and possible types

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

Used type of driver

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

Table 26: Spectrum API: register driver type information and possible driver types

Register	Value	Direction	Description
SPC_GETDRVTYPE	1220	read	Gives information about what type of driver is actually used
DRVTYPE_LINUX32	1		Linux 32bit driver is used
DRVTYPE_WDM32	4		Windows WDM 32bit driver is used (XP/Vista/Windows 7/8/10/11).
DRVTYPE_WDM64	5		Windows WDM 64bit driver is used by 64bit application (XP64/Vista/Windows 7/8/10/11).
DRVTYPE_WOW64	6		Windows WDM 64bit driver is used by 32bit application (XP64/Vista/Windows 7/8/10/11).
DRVTYPE_LINUX64	7		Linux 64bit driver is used

Driver version

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

Table 27: Spectrum API: driver version read register

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

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

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

Kernel Driver version

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

Table 28: Spectrum API: kernel driver version read register

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

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

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

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

```

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

```

This small program will generate an output like this:

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

Custom modifications

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

Table 29: Spectrum API: custom modification register and different bitmasks to split the register in various hardware parts

Register	Value	Direction	Description
SPCM_CUSTOMMOD	3130	read	Dedicated feature register used to mark special custom modifications of the base card and/or the front-end module and/or the Star-Hub module. This is only used if the card has been specially customized. Please refer to the extra documentation for the meaning of the custom modifications. This register is supported for all M5i, M4i, M4x, M2p cards and all digitizerNETBOX, generatorNETBOX or hybridNETBOX based upon these series of cards.
SPCM_CUSTOMMOD_BASE_MASK	000000Fh		Mask for the custom modification of the base card.
SPCM_CUSTOMMOD_MODULE_MASK	0000FF00h		Mask for the custom modification of the front-end module(s).
SPCM_CUSTOMMOD_STARHUB_MASK	00FF0000h		Mask out custom modification of the Star-Hub module.

Reset

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

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

Table 30: Spectrum API: command register and reset command

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

Analog Outputs

Channel Selection

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

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

Table 31: Spectrum API: channel enable register and register settings

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_i64 (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 channel.

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.

Table 32: Spectrum API: channel count register

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

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

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

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

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

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

Important note on channel selection

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



Setting up the 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.

Table 33: Spectrum API: output enable register and register settings

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

This arbitrary waveform generator board uses separate output amplifiers for each channel. This gives you the possibility to separately 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.

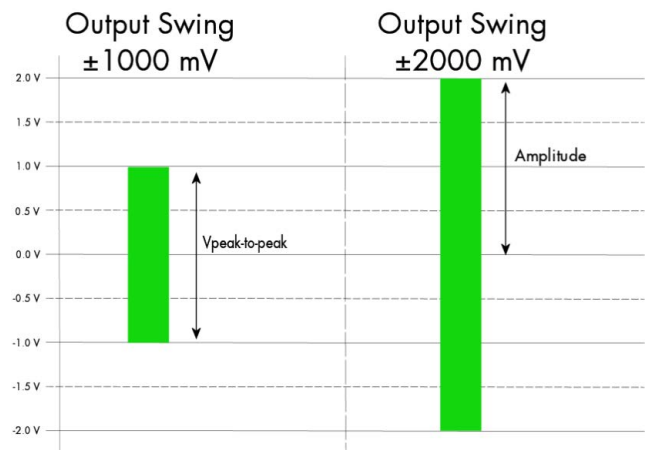


Image 41: Scaling the output swing using the output amplitude registers

Table 34: Spectrum API: output amplitude registers and register settings depending on board type

Register	Value	Direction	Description	Amplitude range	Amplitude range
				M4i.6620 M4i.6621 M4i.6622	M4i.6630 M4i.6631
SPC_AMP0	30010	read/write	Defines the amplitude of channel0 into 50 Ohm load in mV.	80 up to 2500 (in mV)	80 up to 2000 (in mV)
SPC_AMP1	30110	read/write	Defines the amplitude of channel1 into 50 Ohm load in mV.	80 up to 2500 (in mV)	80 up to 2000 (in mV)
SPC_AMP2	30210	read/write	Defines the amplitude of channel2 into 50 Ohm load in mV.	80 up to 2500 (in mV)	80 up to 2000 (in mV)
SPC_AMP3	30310	read/write	Defines the amplitude of channel3 into 50 Ohm load in mV.	80 up to 2500 (in mV)	80 up to 2000 (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 2000 mV (4000 mV peak-to-peak voltage) will result into an amplitude of 4000 mV (80000 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 paths (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 relay. When switching from one path to the other the driver will automatically disable the output (zero volt level) for the „path switching time“ to avoid a 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 2500mV (M4i.662x) or 2000 mV (M4i.663x) 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 different 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.

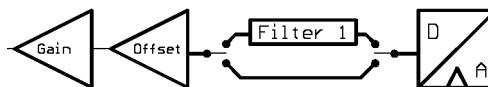


Image 42: output stage showing amplifier and filters

Table 35: Spectrum API: output filter registers and register settings

Register			
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.	

Table 36: output filter specifications depending on card version

Filter	Specifications	M4i.6620-x8, M4x.6620-x4 M4i.6621-x8, M4x.6621-x4 M4i.6622-x8, M4x.6622-x4	M4i.6630-x8, M4x.6630-x4 M4i.6631-x8, M4x.6631-x4
filter 0		No filter will be used.	
filter 1	-3 dB bandwidth	65 MHz	65 MHz

Differential Output

The differential 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.

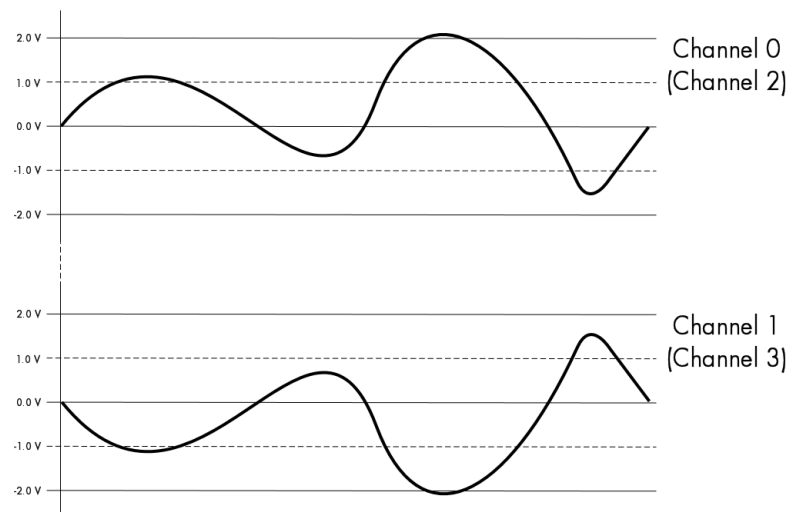


Table 37: Spectrum API: differential output register and register settings

Register			
SPC_DIFF0	30040	read/write	Sets channel 0/1 to differential mode.
SPC_DIFF2	30240	read/write	Sets channel 2/3 to differential mode.

Table 38: availability of differential output mode depending on AWG model

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.

When switching to differential/double out mode the following settings need to match:

- The channel enable mask must contain the primary channel only. That means, for example, when switching to this mode on a four channel card with only channel 0 and channel 2 must be enables. Channel 1 and channels 3 must be disabled
- All output setup like offset, gain or filter must be programmed for each channel of the pair
- Each channel pair only receives one sample of data for output. The second channel will be automatically generated according to the selected mode

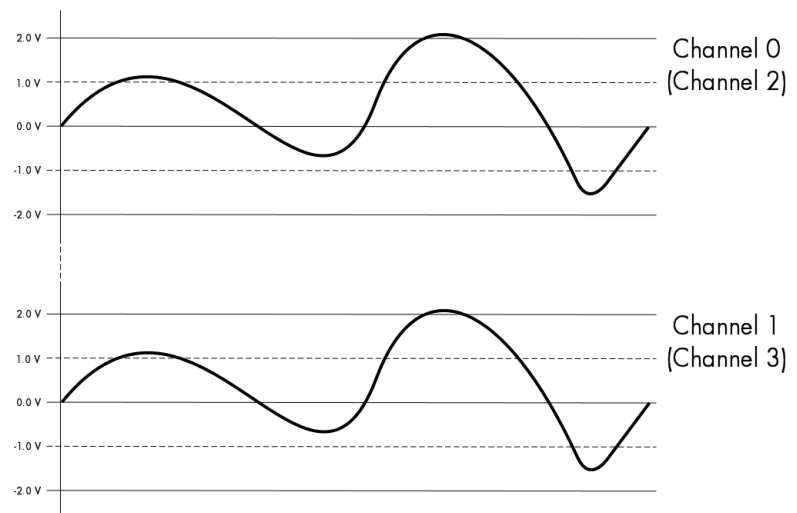


Image 43: schematics of double output mode

Table 39: Spectrum API: double output mode registers

Register			
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.

Table 40: availability of double output mode depending on AWG model

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

Programming the behavior 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 behavior after replay:

Table 41: Spectrum API: stop level register and register settings

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). When synchronous digital bits are replayed, these will be set to LOW state during pause.
SPCM_STOPLVL_LOW	2		Defines the analog output to enter maximum negative level (D/A converter is fed with most negative level). When synchronous digital bits are replayed, these will be set to LOW state during pause.
SPCM_STOPLVL_HIGH	4		Defines the analog output to enter maximum positive level (D/A converter is fed with most positive level). When synchronous digital bits are replayed, these will be set to HIGH state during pause.
SPCM_STOPLVL_HOLDLAST	8		Holds the last replayed sample on the analog output. When synchronous digital bits are replayed, their last state will also be hold.
SPCM_STOPLVL_CUSTOM	32		Allows to define a 16bit wide custom level per channel for the analog output to enter in pauses. The sample format is exactly the same as during replay, as described in the „sample format“ section. When synchronous digital bits are replayed along, the custom level must include these as well and therefore allows to set a custom level for each multi-purpose line separately.

When using SPCM_STOPLVL_CUSTOM, the sample value for the pauses must be defined via the following registers:

Table 42: Spectrum API: custom stop level registers

Register	Value	Direction	Description
SPC_CH0_CUSTOM_STOP	206050	read/write	Defines the custom stop level for channel 0 when using SPCM_STOPLVL_CUSTOM.
SPC_CH1_CUSTOM_STOP	206051	read/write	Defines the custom stop level for channel 1 when using SPCM_STOPLVL_CUSTOM.
SPC_CH2_CUSTOM_STOP	206052	read/write	Defines the custom stop level for channel 2 when using SPCM_STOPLVL_CUSTOM.
SPC_CH3_CUSTOM_STOP	206053	read/write	Defines the custom stop level for channel 3 when using SPCM_STOPLVL_CUSTOM.

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

Example showing how to set a custom stoplevel for channel 0:

```
// enable the use of custom stop level and use raw value 10487 as stop value
spcm_dwSetParam_i32 (stCard.hDrv, SPC_CH0_STOPLEVEL, SPCM_STOPLVL_CUSTOM);
spcm_dwSetParam_i32 (stCard.hDrv, SPC_CH0_CUSTOM_STOP, 10487);
```

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	0000002h		Output is single-ended. If available together with SPC_AO_DIFF: output type is software selectable
SPCM_AO_DIFF	0000004h		Output is differential. If available together with SPC_AO_SE: output type is software selectable
SPCM_AO_PROGFILTER	0000008h		Software selectable output filters are available.
SPCM_AO_PROGOFFSET	0000010h		Output offset is software programmable.
SPCM_AO_PROGGAIN	0000020h		Output gain is software programmable.
SPCM_AO_PROGSTOPELVL	0000040h		The output level between segments and after replay of generated data is programmable.
SPCM_AO_DOUBLEOUT	0000080h		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	0000100h		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 behavior and there are only a very few differences between them.

Table 43: Spectrum API: card mode and read out of available card mode software registers

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

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 mode.
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 mode.
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.
SPC_REP_STD_DDS	4000000h	DDS replay mode functionality available (firmware option required)

Commands

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

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

Table 44: Spectrum API: card command register and different commands with descriptions

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

Card execution commands

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

Card wait commands

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

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

Wait command timeout

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

Table 45: Spectrum API: timeout definition register

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

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

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

Example for card control:

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

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

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

Card Status

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

Table 46: Spectrum API: card status register and possible status values with descriptions of the status

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

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

Acquisition cards status overview

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

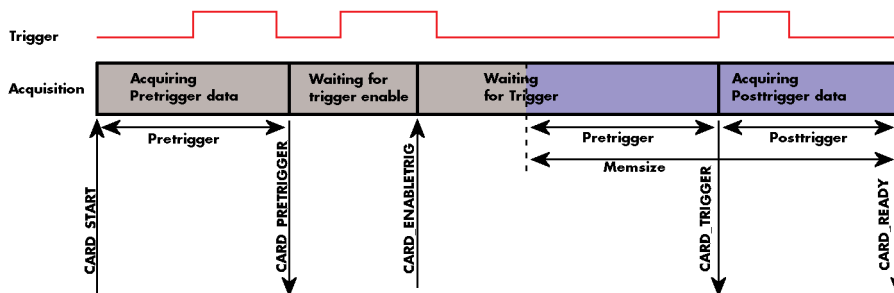


Image 44: Acquisition cards: graphical overview of acquisition status and card command interaction

Generation card status overview

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

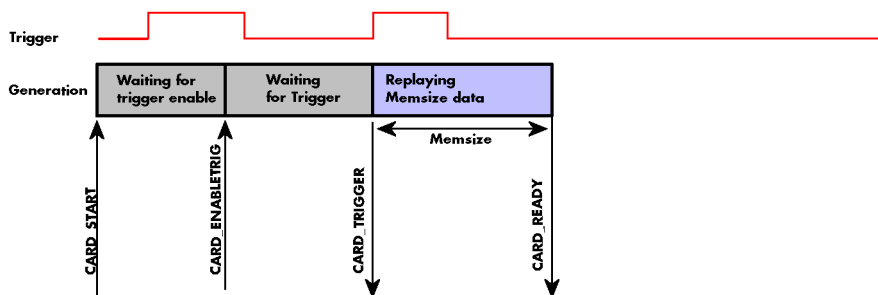


Image 45: Generation cards: graphical overview of generation status and card command interaction

Data Transfer

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

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

Definition of the transfer buffer

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

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

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

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

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

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

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



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

M2i, M3i, M4i, M4x and M2p cards:

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



M5i:

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



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

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



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

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

M5i cards only:

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



Memory test mode

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

Table 47: Spectrum API: memory test register

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

Invalidation of the transfer buffer

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

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

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

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

Commands and Status information for data transfer buffers.

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

Table 48: Spectrum API: Command register and commands for DMA transfers

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

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

Table 49: Spectrum API: status register and status codes for DMA data transfer

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

Example of data transfer

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

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

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

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

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

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



Standard Single 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.

Table 50: Spectrum API: card mode register and single mode settings

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.



Table 51: Spectrum API: memory and loop settings

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 stops 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 (SPC_LOOPS = 1):

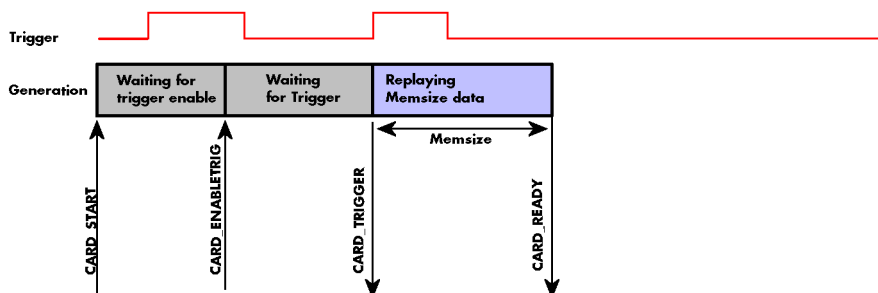


Image 46: timing diagram of single replay mode with commands and status changes

Replay for a defined number of times (2 in the example shown with $\text{SPC_LOOPS} = 2$):

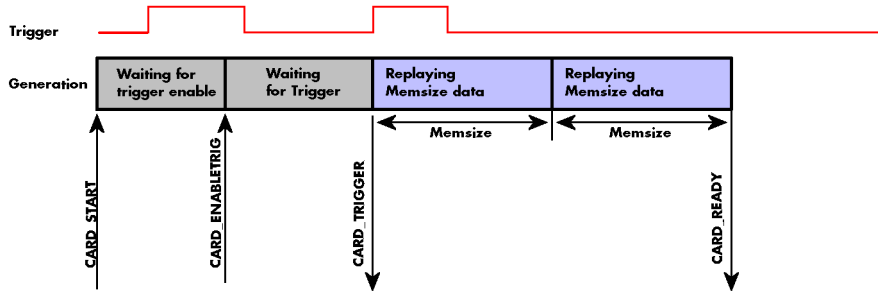


Image 47: timing diagram of single replay mode with two loops with commands and status changes

Replay continuously until the replay is stopped/aborted by the user ($\text{SPC_LOOPS} = 0$):

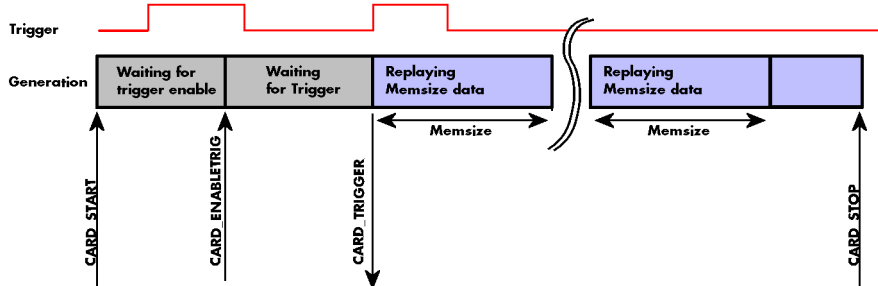


Image 48: timing diagram of continuous replay mode stopped by user with commands and status changes

SPC REP STD SINGLERESTART

This mode behaves like multiple shots of $\text{SPC_REP_STD_SINGLE}$ but with a very small re-arming time in between. 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.

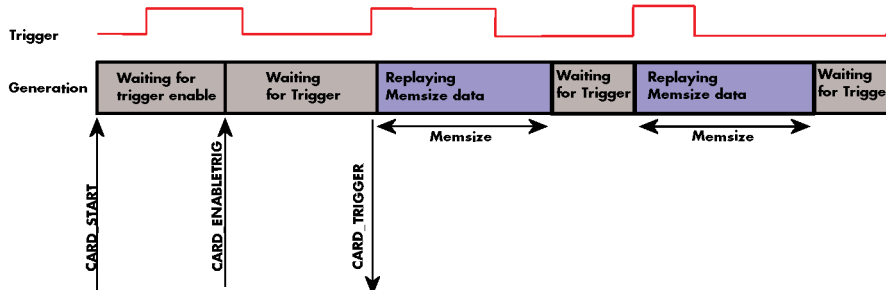


Image 49: timing diagram of single restart mode with commands and status changes

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

Table 52: Spectrum API: overview of mode settings in relation to loops settings and resulting behaviour

	$\text{SPC_LOOPS} = 0$	$\text{SPC_LOOPS} = 1$	$\text{SPC_LOOPS} = N$
$\text{SPC_REP_STD_SINGLE}$	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.
$\text{SPC_REP_STD_SINGLERESTART}$	The programmed memory is replayed once on every trigger event. This continues until stopped by the user.	n.a. (similar to $\text{SPC_REP_STD_SINGLE}$)	The programmed memory is replayed once on every trigger event. This continues until the memory is N-times replayed. Afterwards the card stops.

Continuous marker output

If using the continuous output with internal trigger one can activate a marker output on the multi-purpose I/O connectors marking the beginning of each loop.

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.

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_i64 (hDrv, SPC_MEMSIZE, lMemsize); // replay length
spcm_dwSetParam_i64 (hDrv, SPC_LOOPS, 1); // replay memsize once

void* pvData = pvAllocMemPageAligned (2 * lMemsize); // create a data buffer, 2 bytes per sample
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.

Table 53: Spectrum API: FIFO single replay mode register and settings

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

Table 54: Spectrum API: FIFO mode length settings registers

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

// in FIFO mode we need to define the buffer before starting the transfer
int16* pData = (int16*) pvAllocMemPageAligned (llBufsizeInSamples * 2); // assuming 2 byte per sample
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_PCTOCARD, 4096,
                       (void*) pData, 0, 2 * llBufsizeInSamples);

// before start we once have to fill some data in for the start of the output
vCalcOrLoadData (&pData[0], 2 * llBufsizeInSamples);
spcm_dwSetParam_i64 (hDrv, SPC_DATA_AVAIL_CARD_LEN, 2 * llBufsizeInSamples);
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

// now the first <notifiesize> 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 chunks
llTotalBytes = 2 * llBufsizeInSamples;
while (!dwError)
{
    // read out the available bytes that are free again
    spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_LEN, &llAvailBytes);
    spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_POS, &llUserPosInBytes);

    // be sure not to make a rollover and limit the data to be processed
    if ((llUserPosInBytes + llAvailBytes) > (2 * llBufsizeInSamples))
        llAvailBytes = (2 * llBufsizeInSamples) - llUserPosInBytes;
    llTotalBytes += llAvailBytes;

    // generate some new data
    vCalcOrLoadData (&pData[llUserPosInBytes / 2], llAvailBytes);
    printf ("Currently Available: %lld, total: %lld\n", llAvailBytes, llTotalBytes);

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

```

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 sample 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:

Table 55: Spectrum API: limits of segment size, memory size and loops registers depending on selected mode

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	0 (∞)	1	1
	Standard Gate	32	Mem	32	not used			0 (∞)	1	1
	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/2	32	not used			0 (∞)	4G - 1	1
	Single Restart	32	Mem/2	32	not used			0 (∞)	4G - 1	1
	Standard Multi	32	Mem/2	32	16	Mem/4	16	0 (∞)	1	1
	Standard Gate	32	Mem/2	32	not used			0 (∞)	1	1
	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/4	32	not used			0 (∞)	4G - 1	1
	Single Restart	32	Mem/4	32	not used			0 (∞)	4G - 1	1
	Standard Multi	32	Mem/4	32	16	Mem/8	16	0 (∞)	1	1
	Standard Gate	32	Mem/4	32	not used			0 (∞)	1	1
	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 M5i/M4i/M4x/M2p series cards, there is a very reliable two step buffer strategy set up. The on-board memory of the card can be completely used as a real FIFO buffer. In addition a part of the PC memory can be used as an additional software buffer. Transfer between hardware FIFO and software buffer is performed interrupt driven and automatically by the driver to get best performance. The following drawing will give you an overview of the structure of the data transfer handling:

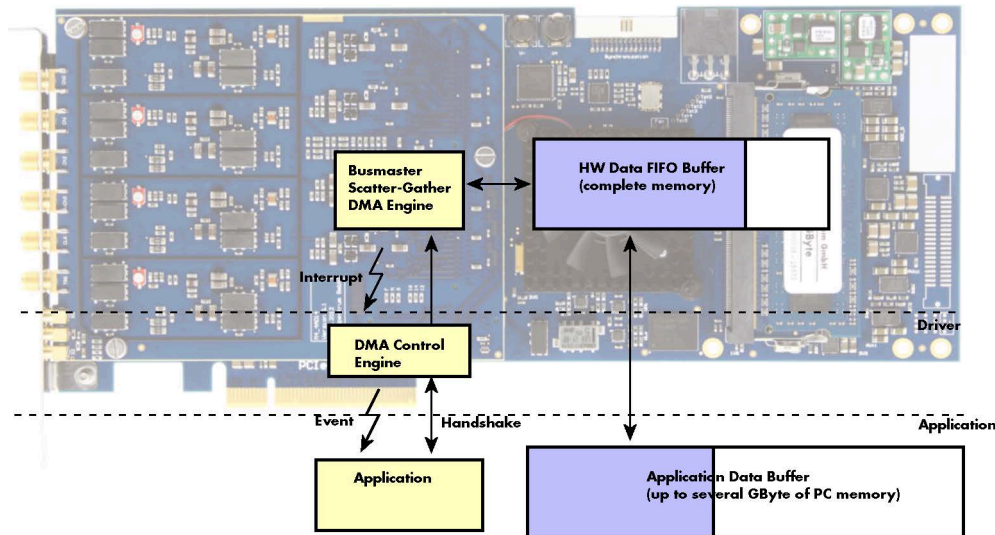


Image 50: Overview of buffer handling for DMA transfers showing and the interaction with the DMA engine

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

Table 56: Spectrum API: registers for DMA buffer handling

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

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

Table 57: Spectrum API: content of DMA buffer handling registers for different use cases

Transfer direction	Register	Direction	Description
Write to card	SPC_DATA_AVAIL_USER_LEN	read	This register contains the currently available number of bytes that are free to write new data to the card. The user can now fill this amount of bytes with new data to be transferred.
	SPC_DATA_AVAIL_CARD_LEN	write	After filling an amount of the buffer with new data to transfer to card, the user tells the driver with this register that the amount of data is now ready to transfer.
Read from card	SPC_DATA_AVAIL_USER_LEN	read	This register contains the currently available number of bytes that are filled with newly transferred data. The user can now use this data for own purposes, copy it, write it to disk or start calculations with this data.
	SPC_DATA_AVAIL_CARD_LEN	write	After finishing the job with the new available data the user needs to tell the driver that this amount of bytes is again free for new data to be transferred.
Any direction	SPC_DATA_AVAIL_USER_POS	read	The register holds the current byte index position where the available bytes start. The register is just intended to help you and to avoid own position calculation
Any direction	SPC_FILLSIZEPROMILLE	read	The register holds the current fill size of the on-board memory (FIFO buffer) in promille (1/1000) of the full on-board memory. Please note that the hardware reports the fill size only in 1/16 parts of the full memory. The reported fill size is therefore only shown in $1000/16 = 63$ promille steps.

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

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



Remarks

- The transfer between hardware FIFO buffer and application buffer is done with scatter-gather DMA using a busmaster DMA controller located on the card. Even if the PC is busy with other jobs data is still transferred until the application data buffer is completely used.
- Even if application data buffer is completely used there's still the hardware FIFO buffer that can hold data until the complete on-board

memory is used. Therefore a larger on-board memory will make the transfer more reliable against any PC dead times.

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

M2i, M3i, M4i, M4x and M2p cards:

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



M5i:

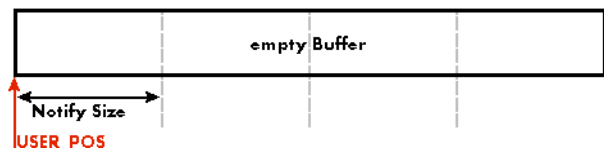
The Notify size sticks to the page size which is defined by the PC hardware and the operating system. Therefore the notify size must be a multiple of 4 kByte. For main data transfer it may also be a fraction of 4k in the range of 64, 128, 256, 512, 1k or 2k. No other values are allowed. For timestamp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



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

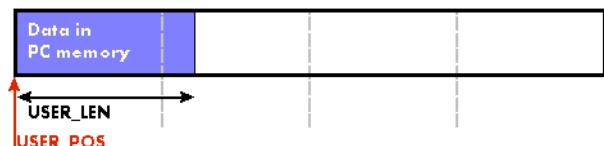
Step 1: Buffer definition

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



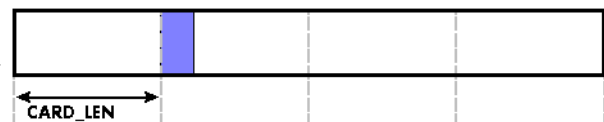
Step 2: Start and first data available

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



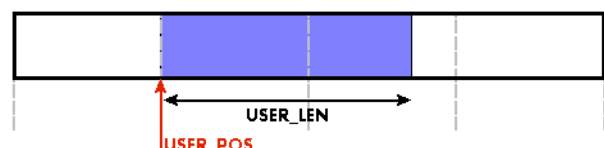
Step 3: set the first data available for card

Now the data can be processed. If transfer is going from card to PC that may be storing to hard disk or calculation of any figures. If transfer is going from PC to card that means we have to fill the available buffer again with data. After the amount of data that has been processed by the user application we set it available for the card and for the next step.



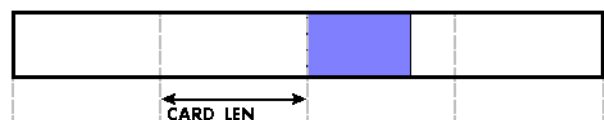
Step 4: next data available

After reaching the next border of the notify size we get the next part of the data buffer to be available. In our example at the time when reading the USER_LEN even some more data is already available. The user position will now be at the position of the previous set CARD_LEN.



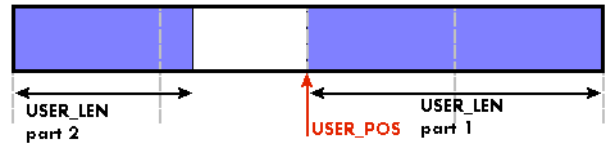
Step 5: set data available again

Again after processing the data we set it free for the card use. In our example we now make something else and don't react to the interrupt for a longer time. In the background the buffer is filled with more data.

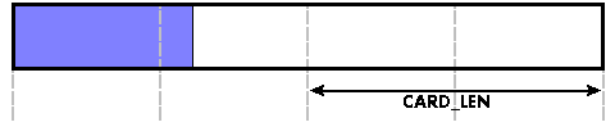


Step 6: roll over the end of buffer

Now nearly the complete buffer is filled. Please keep in mind that our current user position is still at the end of the data part that we processed and marked in step 4 and step 5. Therefore the data to process now is split in two parts. Part 1 is at the end of the buffer while part 2 is starting with address 0.

**Step 7: set the rest of the buffer available**

Feel free to process the complete data or just the part 1 until the end of the buffer as we do in this example. If you decide to process complete buffer please keep in mind the roll over at the end of the buffer.



This buffer handling can now continue endless as long as we manage to set the data available for the card fast enough. The USER_POS and USER_LEN for step 8 would now look exactly as the buffer shown in step 2.

Buffer handling example for transfer from card to PC (Data acquisition)

```
int8* pcData = (int8*) pvAllocMemPageAligned (llBufferSizeInBytes);

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

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

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

        // if there was no error we can proceed and read out the available bytes that are free again
        spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_LEN, &llAvailBytes);
        spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_POS, &llBytePos);

        printf ("We now have %lld new bytes available\n", llAvailBytes);
        printf ("The available data starts at position %lld\n", llBytePos);

        // we take care not to go across the end of the buffer, handling the wrap-around
        if ((llBytePos + llAvailBytes) >= llBufferSizeInBytes)
            llAvailBytes = llBufferSizeInBytes - llBytePos;

        // our do function gets a pointer to the start of the available data section and the length
        vDoSomething (&pcData[llBytePos], llAvailBytes);

        // the buffer section is now immediately set available for the card
        spcm_dwSetParam_i64 (hDrv, SPC_DATA_AVAIL_CARD_LEN, llAvailBytes);
    }
}
while (!dwError); // we loop forever if no error occurs
```

Buffer handling example for transfer from PC to card (Data generation)

```

int8* pcData = (int8*) pvAllocMemPageAligned (llBufferSizeInBytes);

// before starting transfer we first need to fill complete buffer memory with meaningful data
vDoGenerateData (&pcData[0], llBufferSizeInBytes);

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

// and transfer some data to the hardware buffer before the start of the card
spcm_dwSetParam_i32 (hDrv, SPC_DATA_AVAIL_CARD_LEN, llBufferSizeInBytes);
dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

do
{
    if (!dwError)
    {
        // if there was no error we can proceed and read out the current amount of available data
        spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_LEN, &llAvailBytes);
        spcm_dwGetParam_i64 (hDrv, SPC_DATA_AVAIL_USER_POS, &llBytePos);

        printf ("We now have %lld free bytes available\n", llAvailBytes);
        printf ("The available data starts at position %lld\n", llBytePos);

        // we take care not to go across the end of the buffer, handling the wrap-around
        if ((llBytePos + llAvailBytes) >= llBufferSizeInBytes)
            llAvailBytes = llBufferSizeInBytes - llBytePos;

        // our do function gets a pointer to the start of the available data section and the length
        vDoGenerateData (&pcData[llBytePos], llAvailBytes);

        // now we mark the number of bytes that we just generated for replay
        // and wait for the next free buffer
        spcm_dwSetParam_i64 (hDrv, SPC_DATA_AVAIL_CARD_LEN, llAvailBytes);
        dwError = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_WAITDMA);
    }
}
while (!dwError); // we loop forever if no error occurs

```

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



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.

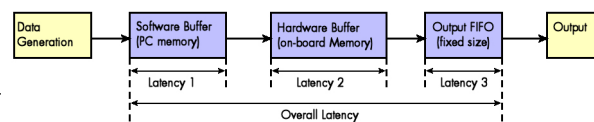


Image 51: output latency involved components

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:

Table 58: Spectrum API: output buffer size register and register settings

Register			
SPC_DATA_OUTBUFSIZE	209	read/write	Programs 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 1k. Resulting in allowed settings of 1k, 2k, 4k, 8k, 16k, ... up to the installed on-board memory size.

When the hardware buffer is adjusted, this must be followed by a M2CMD_CARD_WRITESETUP command and done after defining the card mode but before defining the transfer buffers via the spcm_dwDefTransfer function and , as shown in the example below.



The size of the output FIFO is fixed to 192 kByte (Latency 3) and cannot be changed. If setting a hardware buffer to 64 kByte (Latency 2) and using a software buffer of 64 kByte (Latency 1), the total size of buffered data is hence 320 kByte. Please see the following table for some example output latency calculations (1 sample = 2 bytes), taking buffers and the clock rate into account:

Table 59: output latency depending on channel settings, buffer settings and output FIFO

Configuration	Sampling rate	Software Buffer		Hardware Buffer		Output FIFO		Overall Latency
		Size	Latency	Size	Latency	Size (max)	Latency (max)	
1 x 16 Bit Channel	1.25 GS/s	8 MByte	3.36 ms	4 GByte	1717.98 ms	192 kByte	0.16 ms	1721.5 ms
...	...	8 MByte	3.36 ms	8 MByte	3.36 ms	192 kByte	0.16 ms	6.9 ms
...	...	1 MByte	0.42 ms	1 MByte	0.42 ms	192 kByte	0.16 ms	1.0 ms
...	...	64 kByte	0.026 ms	64 kByte	0.0026 ms	192 kByte	0.16 ms	0.2 ms
1 x 16 Bit Channel	625 MS/s	8 MByte	6.71 ms	8 MByte	6.71 ms	192 kByte	0.31 ms	13.7 ms
...	...	1 MByte	0.84 ms	1 MByte	0.84 ms	192 kByte	0.31 ms	2.0 ms
...	...	64 kByte	0.05 ms	64 kByte	0.05 ms	192 kByte	0.31 ms	0.4 ms
1 x 16 Bit Channel	100 MS/s	8 MByte	41.94 ms	8 MByte	41.94 ms	192 kByte	1.97 ms	85.9 ms
...	...	1 MByte	5.24 ms	1 MByte	5.24 ms	192 kByte	1.97 ms	12.5 ms
...	...	64 kByte	0.33 ms	64 kByte	0.33 ms	192 kByte	1.97 ms	2.6 ms
4 x 16 Bit Channel	100 MS/s	8 MByte	40.48 ms	8 MByte	40.48 ms	192 kByte	0.49 ms	81.5 ms
...	...	1 MByte	1.31 ms	1 MByte	1.31 ms	192 kByte	0.49 ms	3.1 ms
...	...	64 kByte	0.08 ms	64 kByte	0.08 ms	192 kByte	0.49 ms	0.7 ms

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.



The above mentioned latency calculations are only an example on how to calculate the time. They're not tested in real life to run continuously with that sampling speed.

```
void* pvBuffer = NULL;
int64 llHWBufSize = KILO_B(64);
int64 llSWBufSize = KILO_B(128); // must be an integer multiple of llNotifySize
uint32 dwNotifySize = KILO_B(8);
uint32 dwErr;

// define card mode first
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REP_STD_SINGLE);

// secondly define the hardware buffer and write it to the hardware
spcm_dwSetParam_i64 (hDrv, SPC_DATA_OUTBUFSIZE, llHWBufSize);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_WRITESETUP);

// and then allocate and setup the software fifo buffer
pvBuffer = pvAllocMemPageAligned ((uint32) llSWBufSize);
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_PCTOCARD, dwNotifySize, pvBuffer, 0, llSWBufSize);

// --> now fill the buffer with initial data (not shown here)

spcm_dwSetParam_i64 (hDrv, SPC_DATA_AVAIL_CARD_LEN, llSWBufSize);

// now that SW-buffer is filled, we start the data transfer (replay itself is not started yet)
// and wait for the data to be transferred.
spcm_dwSetParam_i32 (stCard.hDrv, SPC_TIMEOUT, 1000);
dwErr = spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

if (!dwErr)
{
    // please see FIFO replay examples for further details regarding the complete data transfer ...
}
```

Data organization

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

Table 60: M4i and M4x cards data organization

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 two's complement as a 16 bit signed data word. 16 bit resolution means that data is ranging from -32768...to...+32767. Data is stored in little-endian format, the upper 8 bit come first and the lower 8 bit second.

A channel's samples can contain also information for the synchronous digital output channels, with up to three digital channels combined with the analog sample within one data word. When extracting the digital channels from the data word, the analog data will automatically be shifted upwards, to not lose any gain information. The analog data is still in the same two's complement format.

Table 61: Spectrum API: data format and DAC resolution depending on selected mode and digital output modes

	Standard Mode No embedded digital Bit 16 bit DAC resolution	Digital outputs enabled 1 embedded digital Bit 15 bit DAC resolution	Digital outputs enabled 2 embedded digital Bits 14 bit DAC resolution	Digital outputs enabled 3 embedded digital Bits 13 bit DAC resolution
D15	DAx Bit 15 (MSB)	Digital „Bit15“ of channel x	Digital „Bit15“ of channel x	Digital „Bit15“ of channel x
D14	DAx Bit 14	DAx Bit 15 (MSB)	Digital „Bit14“ of channel x	Digital „Bit14“ of channel x
D13	DAx Bit 13	DAx Bit 14	DAx Bit 15 (MSB)	Digital „Bit13“ of channel x
D12	DAx Bit 12	DAx Bit 13	DAx Bit 14	DAx Bit 15 (MSB)
D11	DAx Bit 11	DAx Bit 12	DAx Bit 13	DAx Bit 14
D10	DAx Bit 10	DAx Bit 11	DAx Bit 12	DAx Bit 13
D9	DAx Bit 9	DAx Bit 10	DAx Bit 11	DAx Bit 12
D8	DAx Bit 8	DAx Bit 9	DAx Bit 10	DAx Bit 11
D7	DAx Bit 7	DAx Bit 8	DAx Bit 9	DAx Bit 10
D6	DAx Bit 6	DAx Bit 7	DAx Bit 8	DAx Bit 9
D5	DAx Bit 5	DAx Bit 6	DAx Bit 7	DAx Bit 8
D4	DAx Bit 4	DAx Bit 5	DAx Bit 6	DAx Bit 7
D3	DAx Bit 3	DAx Bit 4	DAx Bit 5	DAx Bit 6
D2	DAx Bit 2	DAx Bit 3	DAx Bit 4	DAx Bit 5
D1	DAx Bit 1	DAx Bit 2	DAx Bit 3	DAx Bit 4
D0	DAx Bit 0 (LSB)	DAx Bit 1 (LSB)	DAx Bit 2 (LSB)	DAx Bit 3 (LSB)

Hardware data conversion

The data conversion modes allow the conversion of input data in hardware. This is especially useful when replaying previously recorded data of acquisition cards with either 15 bit, 14 bit or 12 bit resolution. The conversion takes place in hardware and therefore avoids a possible time consuming shift in the user application software.

Table 62: Spectrum API: hardware data conversion registers and available register settings

Register	Value	Direction	Description
SPC_AVALDATACONVERSION	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 global hardware data conversion mode for all channels or reads out the currently selected one.
SPCM_DC_NONE	0h		16 bit input data is assumed and no hardware data conversion will be done.
SPCM_DC_12BIT_TO_16BIT	4h		12 bit input data is assumed and all samples of all currently active channels will be logically shifted upwards to use the available 16 bit DAC resolution.

SPCM_DC_14BIT_TO_16BIT	8h	14 bit input data is assumed and all samples of all currently active channels will be logically shifted upwards to use the available 16 bit DAC resolution.
SPCM_DC_15BIT_TO_16BIT	10h	15 bit input data is assumed and all samples of all currently active channels will be logically shifted upwards to use the available 16 bit DAC resolution.

The hardware data conversion shifts the 16bit data words no matter what their content is or what channel they belong to. In case that you would like to replay also some digital data from a previous recording included within the samples, the added width of the digital data would have to be taken into account.



For example when replaying a recording from an M4i.4420 card with one digital bit included, you can either use no data conversion and replay that digital bit through your generators X0, X1 or X2 line by selecting SPCM_DC_NONE for the data conversion and as such treating that sample as 16bit. Additionally you select the digital output of one bit accordingly as described in the „Multi Purpose I/O Lines“ section later in this manual, which will properly split the in this case 15 bit analog data and the 1 bit digital data.

Or in case, that you want to get rid of the recorded digital bits and output only the pure analog data, you would select a data conversion of SPCM_DC_15BIT_TO_16BIT and hence treat this sample as 15 bit.

Clock generation

Overview

The Spectrum M4i PCI Express (PCIe) and M4x PXI Express (PXIe) cards offer a wide variety of different clock modes to match all the customers' needs. 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 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.

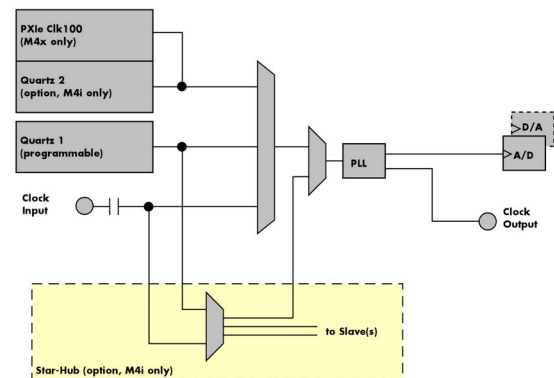


Image 52: M4i/M4x clock section overview

Clock Mode Register

The selection of the different clock modes has to be done by the SPC_CLOCKMODE register. All available modes, can be read out by the help of the SPC_AVAILCLOCKMODES register.

Table 63: Spectrum API: clock mode register and available clock modes

Register	Value	Direction	Description
SPC_AVAILCLOCKMODES	20201	read	Bitmask, in which all bits of the below mentioned clock modes are set, if available.
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode or reads out the actual selected one.
SPC_CM_INTPLL	1		Enables internal programmable high precision Quartz 1 for sample clock generation
SPC_CM_QUARTZ2	4		Enables optional Quartz 2 as reference for sample clock generation
SPC_CM_EXTREFCLOCK	32		Enables internal PLL with external reference for sample clock generation
SPC_CM_PXIREFCLOCK	64		M4x cards only: Enables internal PLL with PXIe backplane clock as reference for sample clock generation

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

The different clock modes

Standard internal sample rate (programmable reference quartz 1)

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

Quartz2 with PLL (option, M4i cards only)

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

External 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.

PXIe Reference Clock (M4x cards only)

The PXIe reference clock is a 100 MHz high-quality differential clock signal with an accuracy of ± 100 ppm or better. This reference clock is located on the PXIe backplane and is routed to every PXIe slot with the same trace length on the mainboard's PCB. PXIe cards from Spectrum are able to use the PXIe reference clock for sampling clock generation. One big advantage of using the reference clock is the fact that all cards that are synchronized to the reference clock are running with the same clock frequency.

Synchronization Clock (option Star-Hub, M4i cards only)

The star-hub option allows the synchronization of up to 8 cards of the M4i series from Spectrum with a minimal phase delay between the different cards. The clock is distributed from the master card to all connected cards. As a source it is possible to either use the programmable Quartz 1 clock or the external ExtIO reference clock input of the master card. For details on the synchronization option please take a look at the dedicated chapter later in this manual.

Details on the different clock modes

Standard internal sampling clock (PLL)

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

Table 64: Spectrum API: clock mode register and internal clock mode

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

The user does not have to care about how the desired sampling rate is generated by multiplying and dividing internally. You simply write the desired sample rate to the according register shown in the table below and the driver makes all the necessary calculations. If you want to make sure the sample rate has been set correctly you can also read out the register and the driver will give you back the sampling rate that is matching your desired one best.

Table 65: Spectrum API: samplerate register

Register	Value	Direction	Description
SPC_SAMPLERATE	20000	write	Defines the sample rate in Hz for internal sample rate generation.
		read	Read out the internal sample rate that is nearest matching to the desired one.

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

Table 66: Spectrum API: clock output and clock output frequency register

Register	Value	Direction	Description
SPC_CLOCKOUT	20110	read/write	Writing a „1“ enables clock output on external clock output connector. Writing a „0“ disables the clock output (tristate)
SPC_CLOCKOUTFREQUENCY	20111	read	Allows to read out the frequency of an internally synthesized clock present at the clock output.

Example on writing and reading internal sampling rate

```

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

```

Minimum internal sampling rate

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

Using Quartz2 with PLL (optional, M4i cards only)

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:

Table 67: Spectrum API: clock mode register and quartz 2 settings

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.

Table 68: Spectrum API: clock output and clock output frequency register

Register	Value	Direction	Description
SPC_CLOCKOUT	20110	read/write	Writing a „1“ enables clock output on external clock output connector. Writing a „0“ disables the clock output (tristate)
SPC_CLOCKOUTFREQUENCY	20111	read	Allows to read out the frequency of an internally synthesized clock present at the clock output.

External clock (reference clock)

The external clock input is fed through a PLL to the clock system. Therefore the input will act as a reference clock input thus allowing to either use a copy of the external clock or to generate any sampling clock within the allowed range from the reference clock. Please note the limited setup granularity in comparison to the internal sampling clock generation. Details are found in the technical data section.

Table 69: Spectrum API: clock mode register and external reference clock setup

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

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

Table 70: Spectrum API: reference clock register and available settings

Register	Value	Direction	Description
SPC_REFERENCECLOCK	20140	read/write	Programs the external reference clock in the range stated in the technical data section.
	External sampling rate in Hz as an integer value		You need to set up this register exactly to the frequency of the external fed in clock.

Example of reference clock:

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

It is recommended that the sampling clock is always a multiple of the reference clock. If the sampling clock is a division of the reference clock, the clock starting phase is undetermined and may change between resets or clock configuration changes.



PLL Locking Error

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

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

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

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

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

Table 71: Spectrum API: clock output and clock output frequency register

Register	Value	Direction	Description
SPC_CLOCKOUT	20110	read/write	Writing a „1“ enables clock output on external clock output connector. Writing a „0“ disables the clock output (tristate)
SPC_CLOCKOUTFREQUENCY	20111	read	Allows to read out the frequency of an internally synthesized clock present at the clock output.

PXI Reference Clock (M4x cards only)

Table 72: Spectrum API: clock mode register and PXI reference clock usage

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_PXIREFCLOCK	64		Enables internal PLL with PXI reference for sample clock generation

The 100 MHz PXIe system reference clock can be used as a reference clock for internal sample rate generation on all M4x PXIe cards from Spectrum. With the above mentioned software command the PXIe reference clock is routed to the internal PLL. Afterwards you only have to program the sample rate register to the desired sampling rate. The remaining internal calculations will be automatically done by the driver.

Example of PXI reference clock:

```
spcm_dwSetParam_i32 (hDrv, SPC_CLOCKMODE, SPC_CM_PXIREFCLOCK); // Set to PXI reference clock mode
spcm_dwSetParam_i64 (hDrv, SPC_SAMPLERATE, 65200000); // We want to have 62.5 MHz as sampling rate
```

PLL Locking Error

The PXI reference signal is routed to a PLL to generate any sampling clock from this external clock. The PLL needs a stable and defined external clock with no gaps and no variation in the frequency. Some backplanes might allow to turn off the reference clock. The PXI clock must be present when issuing the start command. It is not possible to start the card with external clock activated and no external clock available.

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

However if the PLL can not lock on the PXI clock because there is no clock available (if however disabled on the backplane), the driver will return with an error code ERR_CLOCKNOTLOCKED. In that case it is necessary to check the external clock connection. Please see the example below:

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

Trigger modes and appendant registers

General Description

The trigger modes of the Spectrum M4i/M4x series A/D and D/A cards are very extensive and give you the possibility to detect nearly any trigger event you can think of.

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

Trigger Engine Overview

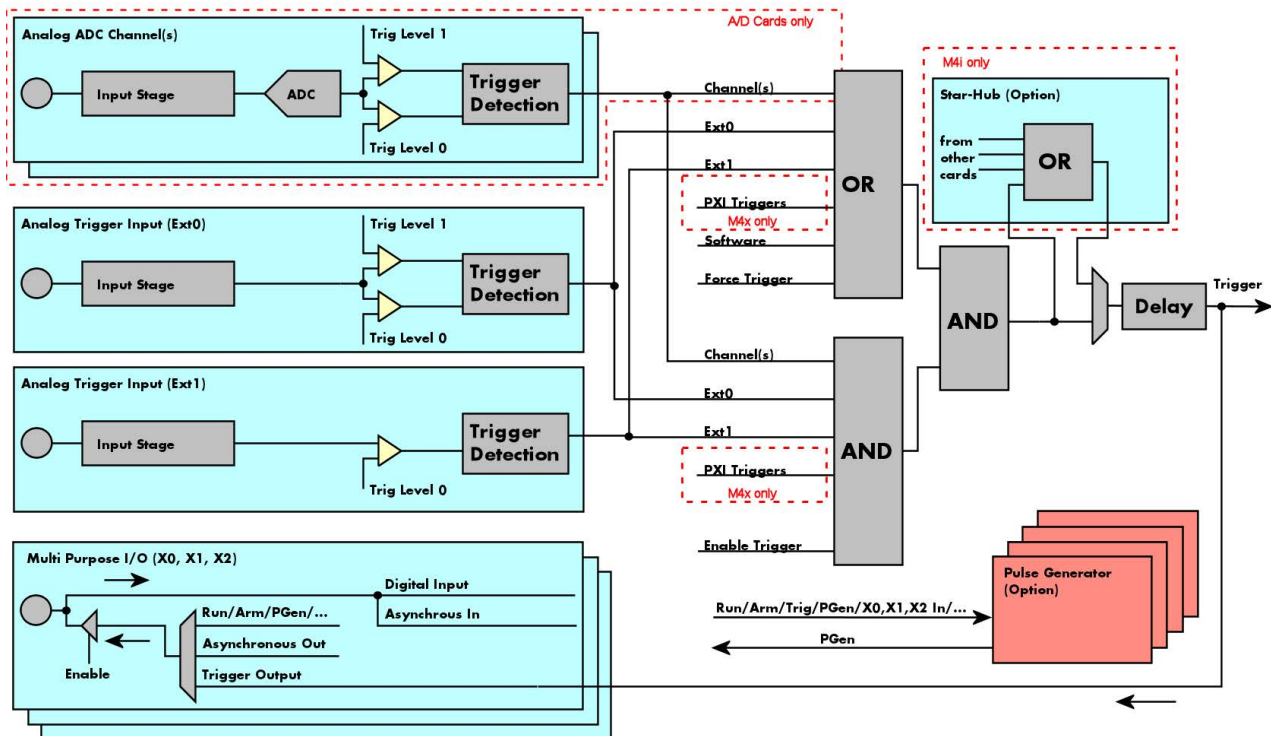


Image 53: Trigger Engine Overview. Red marked parts not available on all card types

The trigger engine of the M4iM4x 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 two different additional external trigger sources. One main trigger source (Ext0, labelled Trg0 on front panel) which also has two analog level comparators also allowing to use edge and windowed trigger detection and one secondary analog trigger (Ext1, labelled Trg1 on front panel) with one analog level comparator. Additionally three multi purpose in/outputs that can be software programmed to either inputs or outputs some extended status signals.

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

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

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

Trigger masks

Trigger OR mask

The purpose of this passage is to explain the trigger OR mask (see left figure) and all the 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, the various PXI triggers (available on M4x PXIe cards only) and software trigger and a channel OR mask.

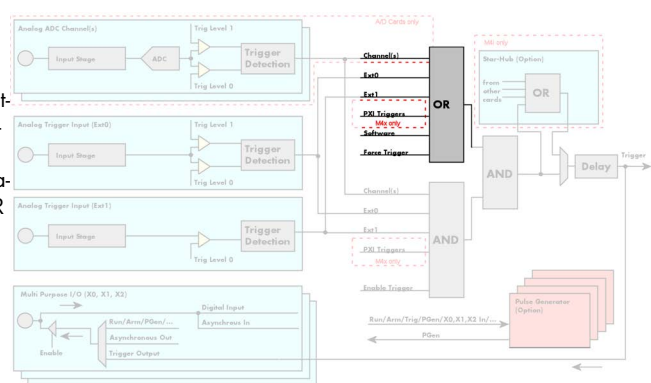


Image 54: trigger engine overview with trigger OR mask shown

Every trigger source of the M4i/M4x series cards is wired to one of the above mentioned OR masks. The user then can program which trigger source will be recognized, and which one won't.

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

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

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

If no input is enabled, the output will be a logic "true", to not block the following static AND mask.

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

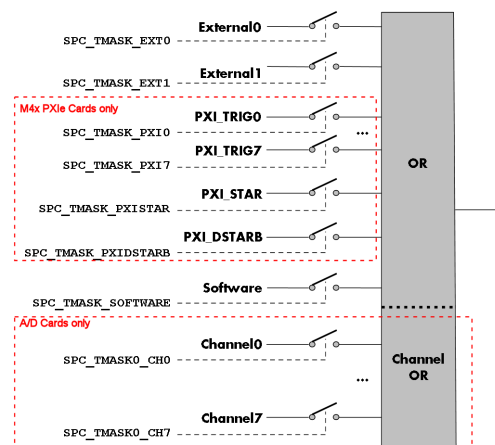


Image 55: trigger engine OR mask details

Table 73: Spectrum API: general trigger OR mask register and available settings

Register	Value	Direction	Description
SPC_TRIG_AVAILORMASK	40400	read	Bitmask, in which all bits of the below mentioned sources for the OR mask are set, if available.
SPC_TRIG_ORMASK	40410	read/write	Defines the events included within the trigger OR mask of the card.
SPC_TMASK_NONE	0		No trigger source selected
SPC_TMASK_SOFTWARE	1h		Enables the software trigger for the OR mask. The card will trigger immediately after start.
SPC_TMASK_EXT0	2h		Enables the external (analog window) trigger 0 (labelled Trg0 on front panel) 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 (labelled Trg1 on front panel) for the OR mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI0	100000h		Enables the PXI_TRIG0 for the OR mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI1	200000h		Enables the PXI_TRIG1 for the OR mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI2	400000h		Enables the PXI_TRIG2 for the OR mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI3	800000h		Enables the PXI_TRIG3 for the OR mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI4	1000000h		Enables the PXI_TRIG4 for the OR mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI5	2000000h		Enables the PXI_TRIG5 for the OR mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI6	4000000h		Enables the PXI_TRIG6 for the OR mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI7	8000000h		Enables the PXI_TRIG7 for the OR mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI_STAR	10000000h		Enables the PXI_STAR line for the OR mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI_STARB	20000000h		Enables the PXI_STARB for the OR mask. The card will trigger when the signal on this input is HIGH.



Please note that as default the SPC_TRIG_ORMASK is set to SPC_TMASK_SOFTWARE. When not using any trigger mode requiring values in the SPC_TRIG_ORMASK register, this mask should explicitly be cleared, as otherwise the software trigger will override other modes.

The following example shows, how to setup the OR mask, for the two external trigger inputs, ORing them together. When using just a single trigger, only this particular trigger must be used in the OR mask register, respectively. As an example a simple edge detection has been

chosen for Ext1 input and a window edge detection has been chosen for Ext0 input. The explanation and a detailed description of the different trigger modes for the external trigger inputs will be shown in the dedicated passage within this chapter.

```

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_LEVEL0, 1800); // 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.

Table 74: Spectrum API: channel trigger OR mask registers and available settings

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILORMASK0	40450	read	Bitmask, in which all bits of the below mentioned sources/channels (0...7) for the channel OR mask are set, if available.
SPC_TRIG_CH_ORMASK0	40460	read/write	Includes the analog channels (0...7) 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_CH4	00000010h		Enables channel4 for recognition within the channel OR mask.
SPC_TMASK0_CH5	00000020h		Enables channel5 for recognition within the channel OR mask.
SPC_TMASK0_CH6	00000040h		Enables channel6 for recognition within the channel OR mask.
SPC_TMASK0_CH7	00000080h		Enables channel7 for recognition within the channel OR mask.

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

```

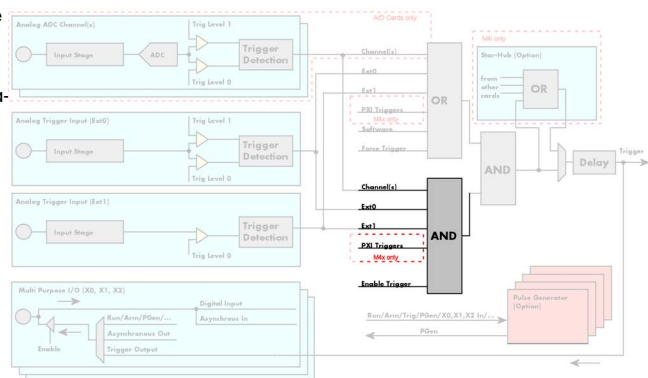
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_NONE); // disable default software trigger
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH_ORMASK0, SPC_TMASK_CH0); // Enable channel0 trigger within the OR mask
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_CH0_LEVEL0, 0); // Trigger level is zero crossing
spcm_dwSetParam_i32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

```

Trigger AND mask

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

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



Every trigger source of the M4i/M4x series cards except the software trigger is wired to one of the above mentioned AND masks. The user then can program which trigger source will be recognized, and which one won't.

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

This selection for the channel mask (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 bit-field, so that they can be combined by one access to the driver with the help of a bitwise OR.

If no input is enabled, the output will be a logic "true", to not block the following static AND mask.

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

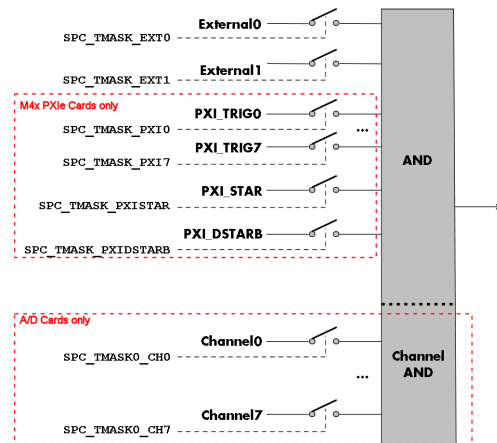


Image 57: trigger engine AND mask details

Table 75: Spectrum API: general trigger AND mask registers and available settings

Register	Value	Direction	Description
SPC_TRIG_AVAILANDMASK	40420	read	Bit mask, 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 (labelled Trg0 on front panel) 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 (labelled Trg1 on front panel) for the AND mask. The card will trigger when the programmed condition for this input is valid.
SPC_TMASK_PXI0	100000h		Enables the PXI_TRIG0 for the AND mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI1	200000h		Enables the PXI_TRIG1 for the AND mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI2	400000h		Enables the PXI_TRIG2 for the AND mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI3	800000h		Enables the PXI_TRIG3 for the AND mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI4	1000000h		Enables the PXI_TRIG4 for the AND mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI5	2000000h		Enables the PXI_TRIG5 for the AND mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI6	4000000h		Enables the PXI_TRIG6 for the AND mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI7	8000000h		Enables the PXI_TRIG7 for the AND mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXISTAR	10000000h		Enables the PXISTAR line for the AND mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXIDSTARB	20000000h		Enables the PXI_DSTARB for the AND mask. The card will trigger when the signal on this input is HIGH.

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

```

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

```

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

Table 76: Spectrum API: channel trigger AND mask registers and available settings

Register	Value	Direction	Description
SPC_TRIG_CH_AVAILANDMASK0	40470	read	Bitmask, in which all bits of the below mentioned sources/channels (0...7) for the channel AND mask are set, if available.
SPC_TRIG_CH_ANDMASK0	40480	read/write	Includes the analog or digital channels (0...7) 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_CH4	00000010h		Enables channel4 for recognition within the channel OR mask.
SPC_TMASK0_CH5	00000020h		Enables channel5 for recognition within the channel OR mask.
SPC_TMASK0_CH6	00000040h		Enables channel6 for recognition within the channel OR mask.
SPC_TMASK0_CH7	00000080h		Enables channel7 for recognition within the channel OR mask.

The following example shows, how to setup the AND mask for a channel trigger. As an example a simple level detection has been chosen.

The explanation and a detailed description of the different trigger modes for the channel trigger modes will be shown in the dedicated passage within this chapter.

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

Software trigger

The software trigger is the easiest way of triggering any Spectrum board. The acquisition or replay of data will start immediately after the card is started and the trigger engine is armed. The resulting delay upon start includes the time the board needs for its setup and the time for recording the pre-trigger area (for acquisition cards).

For enabling the software trigger one simply has to include the software event within the trigger OR mask, as the following table is showing:



Table 77: Spectrum API: software register and register setting for software trigger

Register	Value	Direction	Description
SPC_TRIG_ORMASK	40410	read/write	Defines the events included within the trigger OR mask of the card.
SPC_TMASK_SOFTWARE	1h		Sets the trigger mode to software, so that the recording/replay starts immediately.

Example for setting up the software trigger:

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

Force- and Enable trigger

In addition to the software trigger (free run) it is also possible to force a trigger event by software while the board is waiting for a real physical trigger event. The forcetrigger command will only have any effect, when the board is waiting for a trigger event. The command for forcing a trigger event is shown in the table below.

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

Table 78: Spectrum API: command register and force trigger command

Register	Value	Direction	Description
SPC_M2CMD	100	write	Command register of the M2i/M3i/M4i/M4x/M2p/M5i series cards.
M2CMD_CARD_FORCETRIGGER	10h		Forces a trigger event if the hardware is still waiting for a trigger event.

The example shows, how to use the forcetrigger command:

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

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

Table 79: Spectrum API: command register and trigger enable/disable command

Register	Value	Direction	Description
SPC_M2CMD	100	write	Command register of the M2i/M3i/M4i/M4x/M2p/M5i series cards.
M2CMD_CARD_ENABLETRIGGER	8h		Enables the trigger engine. Any trigger event will now be recognized.
M2CMD_CARD_DISABLETRIGGER	20h		Disables the trigger engine. No trigger events will be recognized, except force trigger.

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

```
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_ENABLETRIGGER); // Trigger engine is armed.
...
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_DISABLETRIGGER); // Trigger engine is disarmed.
```

Trigger delay

All of the Spectrum M4i/M4x series cards allow the user to program an additional trigger delay. As shown in the trigger overview section, this delay is the last element in the trigger chain. Therefore the user does not have to care for the sources when programming the trigger delay.

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

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

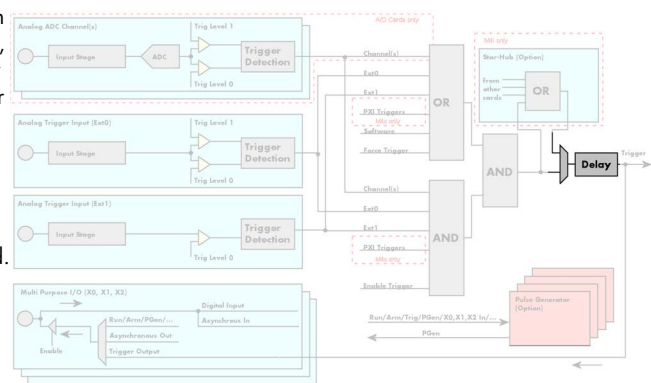


Image 58: trigger engine overview with marked trigger delay stage

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

Table 80: Spectrum API: trigger delay registers and available settings

Register	Value	Direction	Description
SPC_TRIG_AVAILDELAY	40800	read	Contains the maximum available delay as a decimal integer value.
SPC_TRIG_DELAY	40810	read/write	Defines the delay for the detected trigger events.
0			No additional delay will be added. The resulting internal delay is mentioned in the technical data section.
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 $(8G\text{Samples} - 16) = 8589934576$. 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 $(8G\text{Samples} - 32) = 8589934560$. Stepsize is 32 samples for 8 bit cards.

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

```
spcm_dwSetParam_i64 (hDrv, SPC_TRIG_DELAY, 1984); // A detected trigger event will be
                                                    // delayed for 1984 sample clocks.
```

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



Trigger Counter

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

Table 81: Spectrum API: trigger counter register and register return values

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

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



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



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

Main external window trigger (Ext0/Trg0)

The M4i/M4x 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.

The external trigger Ext0 is labelled Trg0 on the front-panel

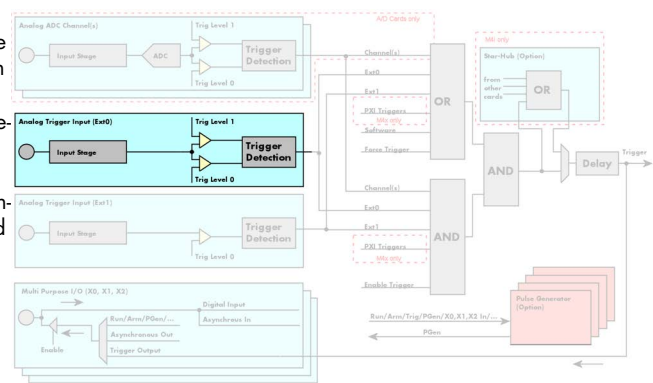


Image 59: trigger engine overview with marked main external trigger Ext0/Trg0

Trigger Mode

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

Table 82: Spectrum API: external trigger Ext0 registers and register settings

Register	Value	Direction	Description
SPC_TRIG_EXT0_AVAILMODES	40500	read	Bitmask showing all available trigger modes for external 0 (Ext0) = 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:

Table 83: Spectrum API: external trigger Ext0 OR mask settings

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

Trigger Input Termination

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

Table 84: Spectrum API: external trigger Ext0 input termination

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

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

Trigger 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.

Table 85: Spectrum API: external trigger Ext0 input coupling

Register	Value	Direction	Description
SPC_TRIG_EXT0_ACDC	40120	read/write	COUPLING_DC enables DC coupling, COUPLING_AC enables AC coupling for the external trigger input (AC coupling is the default).

Secondary external level trigger (Ext1/Trg1)

The M4i/M4x 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.

The secondary trigger input Ext1 is labelled Trg1 on the front-panel.

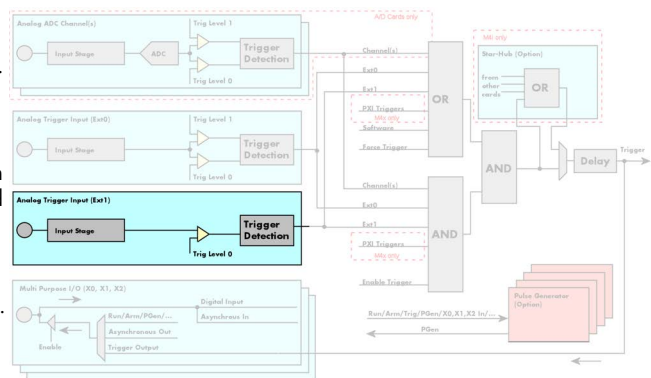


Image 60: trigger engine overview with external trigger Ext1 marked

Trigger Mode

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

Table 86: Spectrum API: external trigger Ext1 registers and register settings

Register	Value	Direction	Description
SPC_TRIG_EXT1_AVAILMODES	40501	read	Bit mask showing all available trigger modes for Ext1 (Trg1) = secondary 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:

Table 87: Spectrum API: external trigger Ext1 OR mask settings

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 detailed trigger mode description that follows.

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

Table 88: Spectrum API: external trigger available settings for trigger levels

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

Table 88: Spectrum API: external trigger available settings for trigger levels

Register	Value	Direction	Description	Range
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:.

Table 89: Spectrum API: external trigger OR mask and AND mask register and settings

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 (labelled Trg0 on front panel) for the mask.
SPC_TMASK_EXT1	4h		Enables the secondary external (analog) trigger 1 (labelled Trg1 on front panel) 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 (Trg0).

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.

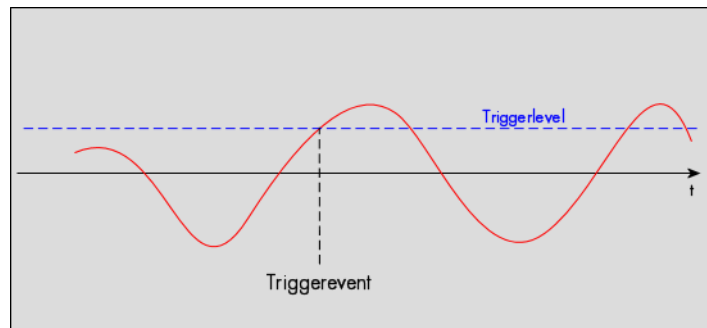


Table 90: Spectrum API: external register mode setup for trigger on positive edge

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.

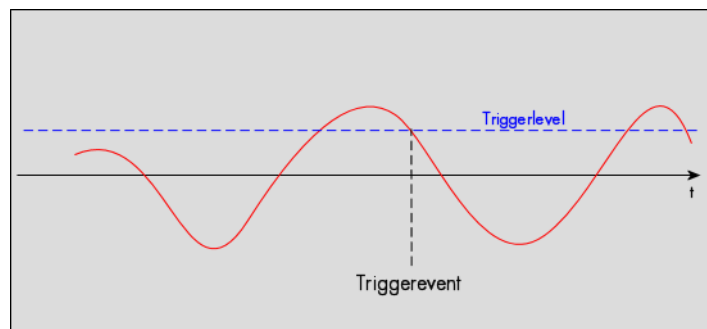


Table 91: Spectrum API: external register mode setup for trigger on negative edge

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.

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

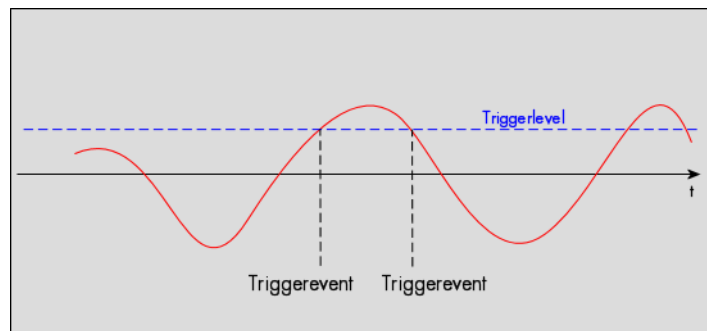


Table 92: Spectrum API: external trigger register mode setup for trigger on positive and negative edge

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.

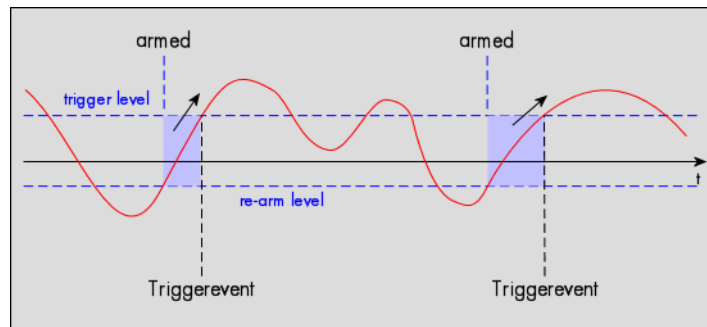


Table 93: Spectrum API: external trigger register mode setup for trigger re-arm on positive edge

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.

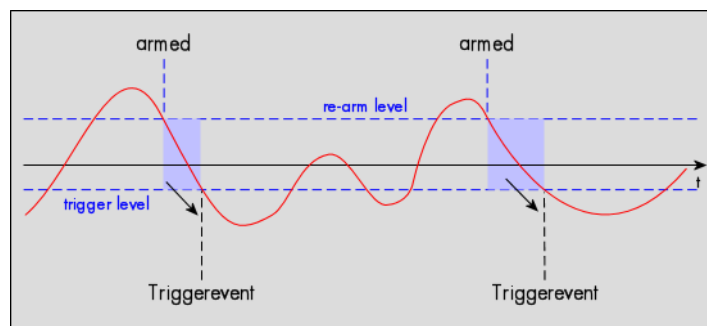


Table 94: Spectrum API: external trigger register mode setup for trigger re-arm on negative edge

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.

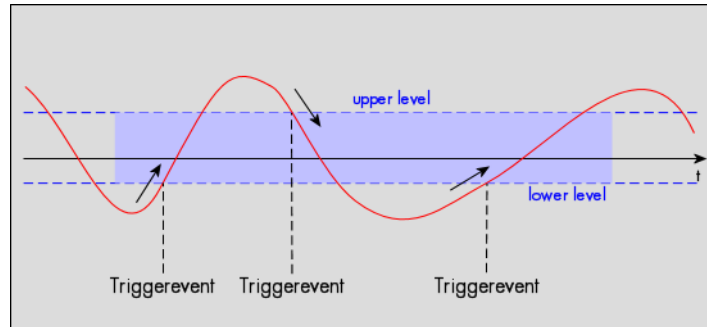


Table 95: Spectrum API: external trigger register mode setup for window trigger for entering signals

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.

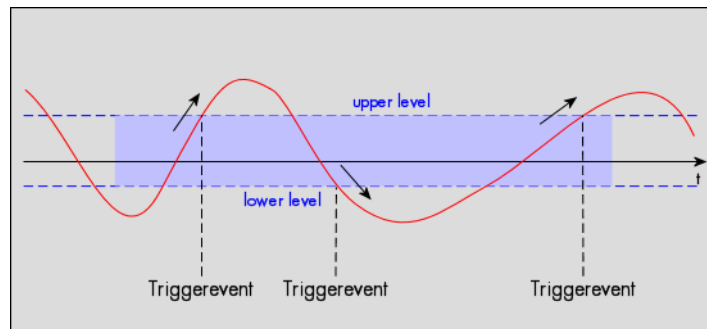


Table 96: Spectrum API: external trigger register mode setup for window trigger for leaving signals

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.

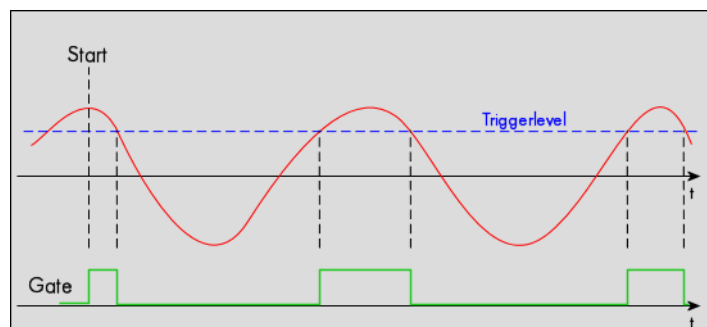


Table 97: Spectrum API: external trigger register mode setup for high level trigger

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.

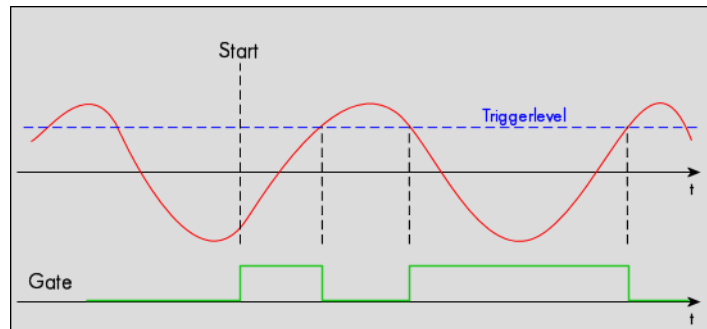


Table 98: Spectrum API: external trigger register mode setup for low level trigger

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.

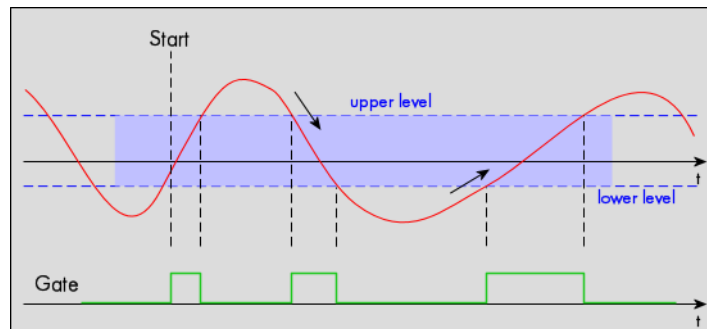


Table 99: Spectrum API: external trigger register mode setup for in window trigger

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 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 outside the programmed trigger window.

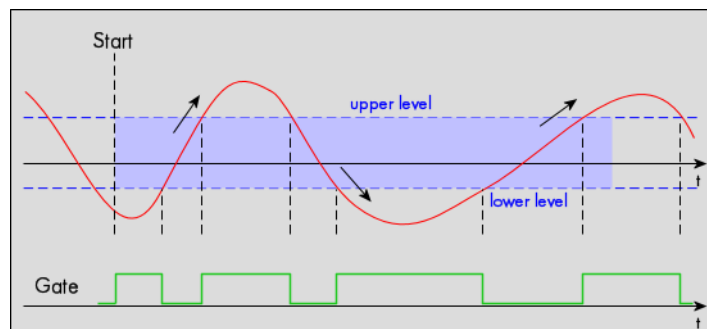


Table 100: Spectrum API: external trigger register mode setup for outside window trigger

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

PXI Trigger (M4x PXIe cards only)

The M4x PXIe cards can use the various PXI trigger sources for trigger detection and/or trigger and status distribution.

This includes the eight lines from the PXI trigger bus (PXI_TRIG[7] to PXI_TRIG[0]), as well as the „older“ single-ended PXI Star-Trigger line (PXI_STAR) and the „newer“ differential PXI_DSTARC (dedicated output) and PXI_DSTARB (dedicated input) lines, that have been introduced with the PXI Express standard.

All these lines can be included within the programmable masks on the cards, either within the OR mask as well within the AND mask, to form rather complex trigger conditions required to properly synchronize multiple M4x.xxxx cards to a single trigger event.

The following passage shows, how to program these lines for either input or output and how to properly use them for trigger synchronization between multiple M4x.xxxx cards inside a PXI or PXIe chassis.

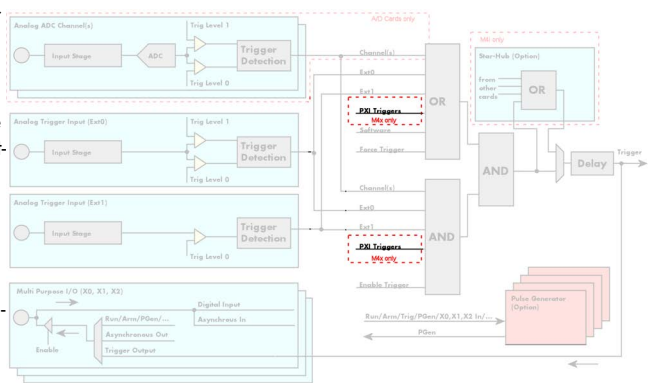


Table 101: trigger overview with PXI trigger lines marked

To set up PXI trigger conditions, the mode registers have to be set up properly, to define the direction of one or multiple PXI lines, as well as the trigger masks registers for including one or multiple PXI lines to generate a trigger event from.

PXI Trigger Mode Registers

Please find the list of the various modes the different PXI trigger lines can be used for below. The modes available for driving the PXI lines are very similar to those of the Multi Purpose I/O lines described earlier in this manual.

Table 102: Spectrum API: PXI trigger register and available register settings

Register	Value	Direction	Description
SPC_PXITRG0_AVAILMODES	47310	read	Bitmask showing all available PXI trigger modes usable with PXI_TRIG0 line.
SPC_PXITRG1_AVAILMODES	47311	read	Bitmask showing all available PXI trigger modes usable with PXI_TRIG1 line.
SPC_PXITRG2_AVAILMODES	47312	read	Bitmask showing all available PXI trigger modes usable with PXI_TRIG2 line.
SPC_PXITRG3_AVAILMODES	47313	read	Bitmask showing all available PXI trigger modes usable with PXI_TRIG3 line.
SPC_PXITRG4_AVAILMODES	47314	read	Bitmask showing all available PXI trigger modes usable with PXI_TRIG4 line.
SPC_PXITRG5_AVAILMODES	47315	read	Bitmask showing all available PXI trigger modes usable with PXI_TRIG5 line.
SPC_PXITRG6_AVAILMODES	47316	read	Bitmask showing all available PXI trigger modes usable with PXI_TRIG6 line.
SPC_PXITRG7_AVAILMODES	47317	read	Bitmask showing all available PXI trigger modes usable with PXI_TRIG7 line.
SPC_PXISTAR_AVAILMODES	47318	read	Bitmask showing all available PXI trigger modes usable with PXI_STAR line.
SPC_PXIDSTARC_AVAILMODES	47310	read	Bitmask showing all available PXI trigger modes usable with PXI_DSTARC line, to send information to a Startrigger card, installed in the System Timing Slot. The corresponding returned signal (from the System Timing Slot to the card) will be available on the DSTARB line, which then has to be properly included into the trigger source masks, as described later.
SPC_PXITRG0_MODE	47300	read/write	Defines the output mode for the PXI_TRIG0 line.
SPC_PXITRG1_MODE	47301	read/write	Defines the output mode for the PXI_TRIG1 line.
SPC_PXITRG2_MODE	47302	read/write	Defines the output mode for the PXI_TRIG2 line.
SPC_PXITRG3_MODE	47303	read/write	Defines the output mode for the PXI_TRIG3 line.
SPC_PXITRG4_MODE	47304	read/write	Defines the output mode for the PXI_TRIG4 line.
SPC_PXITRG5_MODE	47305	read/write	Defines the output mode for the PXI_TRIG5 line.
SPC_PXITRG6_MODE	47306	read/write	Defines the output mode for the PXI_TRIG6 line.
SPC_PXITRG7_MODE	47307	read/write	Defines the output mode for the PXI_TRIG7 line.
SPC_PXISTAR_MODE	47308	read/write	Defines the trigger mode for the PXI_STAR line, to send information to a Startrigger card, installed in the System Timing Slot.
SPC_PXIDSTARC_MODE	47309	read/write	Defines the trigger mode for the PXI_DSTARC line, to send information to a possible Startrigger card, installed in the System Timing Slot. The corresponding returned signal (from the System Timing Slot to the card) will be available on the DSTARB line, which then has to be properly included into the trigger source masks, as described later.

SPCM_PXITRGMODE_DISABLE	00000000h	The PXI line is neither used as input or output and is in high-impedance mode (tristate).
SPCM_PXITRGMODE_IN	00000001h	The PXI line is used as an input and can now be included in the trigger masks, as described below.
SPCM_PXITRGMODE_ASYNCOUT	00000002h	The PXI line is programmed for asynchronous output. Use SPC_PXITRG_ASYNCIO to write data asynchronously.
SPCM_PXITRGMODE_RUNSTATE	00000004h	The PXI line outputs 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_PXITRGMODE_ARMSTATE	00000008h	The PXI line outputs 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 either still acquiring pretrigger data or the trigger has already been detected the signal is LOW.
SPCM_PXITRGMODE_TRIGOUT	00000010h	The PXI line outputs a detected trigger and hence 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_PXITRGMODE_REFLCKOUT	00000020h	The PXI line reflects the internal generated 10 MHz reference clock signal generated from the PXI_CLK100 clock signal in conjunction with the PXI_SYNC100 signal. Can be used to provide other equipment with an additional clock signal via one of the trigger lines.
SPCM_PXITRGMODE_CONTOUTMARK	00000040h	Generator Cards only: the PXI line outputs a HIGH pulse as continuous marker signal for continuous replay mode. The marker signal length is ½ of the programmed memory size.

Depending on the used PXI/PXIe backplane, the PXI trigger bus (PXI_TRIG[7..0]) might be segmented and needs to be setup separately, to allow routing of trigger signals between different segments an even within one segment. Please consult your PXI chassis/backplane manual for details on the routing capabilities of these lines and the relating software interface.



Be aware not to enable outputs on the same PXI trigger line on multiple cards, or to enable a segmented backplane driver and a card's output on the same line within the same segment. If two or more outputs are working against each other the result is unpredictable and may even harm the hardware parts.



The PXI trigger lines cannot be used to phase-synchronize multiple PXIe cards. The PXIe trigger lines are asynchronous to the used sampling rate and thus may result in a phase difference of more than one sample between multiple cards.



PXI Trigger Sources within the Trigger Masks

To include PXI lines, whose mode has been set to „SPCM_PXITRGMODE_IN“, as shown above, the desired masks must contain the corresponding input, as the following table shows:

Table 103: Spectrum API: PXI trigger mask register and available register settings

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_PXI0	100000h		Enables the PXI_TRIG0 for the mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI1	200000h		Enables the PXI_TRIG1 for the mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI2	400000h		Enables the PXI_TRIG2 for the mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI3	800000h		Enables the PXI_TRIG3 for the mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI4	1000000h		Enables the PXI_TRIG4 for the mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI5	2000000h		Enables the PXI_TRIG5 for the mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI6	4000000h		Enables the PXI_TRIG6 for the mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXI7	8000000h		Enables the PXI_TRIG7 for the mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXISTAR	10000000h		Enables the PXISTAR line for the mask. The card will trigger when the signal on this input is HIGH.
SPC_TMASK_PXIDSTARB	20000000h		Enables the PXI_DSTARB for the mask. The card will trigger when the signal on this input is HIGH.

The use of multiple PXI_TRIG lines in either mask to combine status and trigger information is shown in the following example for clarification.

PXI Trigger Setup Example

Example of connecting three M4x.xxxx cards, card 0 is triggering all three cards:

```
drv_handle hDrv[3];

for (i = 0; i < 3; i++)
{
    sprintf (s, "/dev/spcm%d", i);
    hDrv[i] = spcm_hOpen (s); // open all three cards

    spcm_dwSetParam_i32 (hDrv[i], SPC_CLOCKMODE, SPC_CM_PXIREFCLOCK); // Use PXI reference clock on all cards
    spcm_dwGetParam_i64 (hDrv[i], SPC_SAMPLERATE, 100000000); // Use 100 MS/s as sample clock
}

// Slave card1 trigger setup
spcm_dwSetParam_i32 (hDrv[1], SPC_PXITRG0_MODE, SPCM_PXITRGMODE_IN); // set PXI_TRIG0 as input
spcm_dwSetParam_i32 (hDrv[1], SPC_PXITRG1_MODE, SPCM_PXITRGMODE_ARMSTATE); // Output my ARM state on PXI_TRIG1
spcm_dwSetParam_i32 (hDrv[1], SPC_TRIG_ORMASK, SPC_TMASK_PXI0); // trigger source: PXI_TRIG0

// Slave card2 trigger setup
spcm_dwSetParam_i32 (hDrv[2], SPC_PXITRG0_MODE, SPCM_PXITRGMODE_IN); // set PXI_TRIG0 as input
spcm_dwSetParam_i32 (hDrv[2], SPC_PXITRG2_MODE, SPCM_PXITRGMODE_ARMSTATE); // Output my ARM state on PXI_TRIG2
spcm_dwSetParam_i32 (hDrv[2], SPC_TRIG_ORMASK, SPC_TMASK_PXI0); // trigger source: PXI_TRIG0

// Master card0: Acts as a trigger master distributing External trigger 0 trigger
spcm_dwSetParam_i32 (hDrv[0], SPC_TRIG_ORMASK, SPC_TMASK_EXT0);

// Setting Ext0 trigger for rising edges (for AD/DA cards trigger levels might need to be adjusted)
spcm_dwSetParam_i32 (hDrv[0], SPC_TRIG_EXT0_MODE, SPC_TM_POS);

spcm_dwSetParam_i32 (hDrv[0], SPC_PXITRG0_MODE, SPCM_PXITRGMODE_TRIGOUT); // Output trigger on PXI_TRIG0
spcm_dwSetParam_i32 (hDrv[0], SPC_PXITRG1_MODE, SPCM_PXITRGMODE_IN); // Set PXI_TRIG1 as input
spcm_dwSetParam_i32 (hDrv[0], SPC_PXITRG2_MODE, SPCM_PXITRGMODE_IN); // Set PXI_TRIG2 as input

// Synchronize Pre-Trigger area of all cards to prevent unintended trigger detection, while the
// other card(s) are not ready yet. Therefore include PXI_TRIG1 and PXI_TRIG2 inputs in the AND mask,
// so that these lines both must be HIGH, to enable trigger detection on card0 and hence
// trigger distribution to the other cards.
spcm_dwSetParam_i32 (hDrv[0], SPC_TRIG_ANDMASK, SPC_TMASK_PXI2 | SPC_TMASK_PXI1);

// transfer setup to all cards to allow PXI lines to be activated (leave a possible high-impedance mode)
for (i = 0; i < 3; i++)
    spcm_dwSetParam_i32 (hCard[i], SPC_M2CMD, M2CMD_CARD_WRITESSETUP);

// Start the cards now in any order, as any triggering is now prevented as long no all the cards are armed.

// Wait for all cards to finish recording and then get data from all cards and do processing.
```

The above example assumes, that a possible PXI trigger bus segmentation of the used backplane has been properly set up, so that each card's output can reach the other card's inputs properly and that no two drivers on one segment are driving against each other.



Multi Purpose I/O Lines

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

The M4i/M4x series cards and the based upon digitizerNETBOX, generatorNETBOX and hybridNETBOX products have 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 labeled with X0 (line 0), X1 (line 1) and X2 (line 2). As default these lines are switched off.

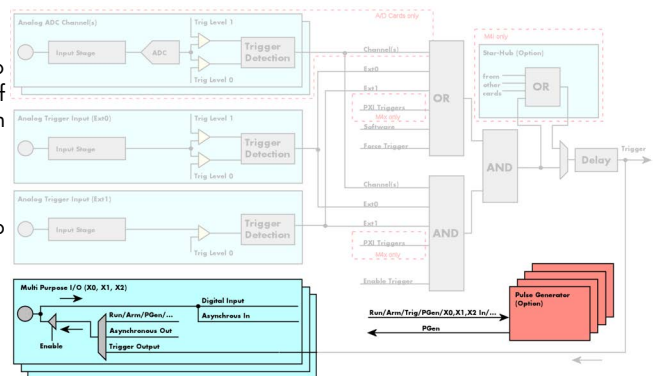


Image 61: trigger overview with multi-purpose lines marked



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



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

Programming the behavior

Each multi purpose I/O line can be individually programmed. Please check the available modes by reading the SPCM_X0_AVAILMODES, SPCM_X1_AVAILMODES 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.

Table 104: Spectrum API: multi-purpose I/O lines registers and available register settings

Register	Value	Direction	Description
SPCM_X0_AVAILMODES	47210	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X0)
SPCM_X1_AVAILMODES	47211	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X1)
SPCM_X2_AVAILMODES	47212	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X2)
SPCM_X0_MODE	47200	read/write	Defines the mode for (X0). Only one mode selection is possible to be set at a time
SPCM_X1_MODE	47201	read/write	Defines the mode for (X1). Only one mode selection is possible to be set at a time
SPCM_X2_MODE	47202	read/write	Defines the mode for (X2). Only one mode selection is possible to be set at a time
SPCM_XMODE_DISABLE	0000000h		No mode selected. Output is tristate (default setup)
SPCM_XMODE_ASYNCIN	00000001h		Connector is programmed for asynchronous input. Use SPCM_XX_ASYNCIO to read data asynchronous as shown in next chapter.
SPCM_XMODE_ASYNCOUT	00000002h		Connector is programmed for asynchronous output. Use SPCM_XX_ASYNCIO to write data asynchronous as shown in next chapter.
SPCM_XMODE_DIGIN	00000004h		A/D cards only: Connector is programmed for synchronous digital input. For each analog channel, one digital channel X0/X1/X2 is integrated into the ADC data stream. Depending on the ADC resolution of your card the resulting merged samples can have different formats. Please check the „Sample format“ chapter for more details. Please note that automatic sign extension of analog data is ineffective as soon as one digital input line is activated and the software must properly mask out the digital bits.
SPCM_XMODE_DIGOUT	00000008h		D/A cards only: Connector is programmed for synchronous digital output. Digital channels can be „included“ within the analog samples and synchronously replayed along. Requires additional MODE bits to be set along with this flag, as explained later on.
SPCM_XMODE_TRIGOUT	00000020h		Connector is programmed as trigger output and shows the trigger detection. The trigger output goes HIGH as soon as the trigger is recognized. After end of acquisition it is LOW again. In Multiple Recording/Gated Sampling/ABA mode it goes LOW after the acquisition of the current segment stops. In FIFO single mode the trigger output is HIGH until FIFO mode is stopped.
SPCM_XMODE_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 internally generated PLL reference clock in the range of 10 to 62.5 MHz.
SPCM_XMODE_CONTOUTMARK	00002000h		Generator Cards only: outputs a HIGH pulse as continuous marker signal for continuous replay mode. The marker signal length is 1/2 of the programmed memory size.

SPCM_XMODE_SYSCLOCKOUT	00004000h	Connector reflects the internally generated system clock in the range of 2.5 up to 156.25 MHz.
SPCM_XMODE_PULSEGEN	00080000h	A/D and D/A cards only (optional): Connector reflects the output of the same index pulse generator (X0 output from pulse generator 0, X1 from pulse generator 1 etc.). For details on the pulse generator option please consult the "Pulse Generator (Option)" chapter.



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

Using asynchronous I/O

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

Table 105: Spectrum API: asynchronous I/O register settings of the multi-purpose I/O registers

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

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

```

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

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

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

```

Special behavior of trigger output

As the driver of the M4i/M4x series is the same as the driver for the M2i/M3i series and some old software may rely on register structure of the M2i/M3i card series, there is a special compatible trigger output register that will work according to the M2i/M3i series style. It is not recommended to use this register unless you're writing software for multiple card series:

Table 106: Spectrum API: additional trigger output register for compatibility with older hardware

Register	Value	Direction	Description
SPC_TRIG_OUTPUT	40100	read/write	M2i style trigger output programming. Write a „1“ to enable: - X2 trigger output (SPCM_X2_MODE = SPCM_XMODE_TRIGOUT) - X1 arm state (SPCM_X1_MODE = SPCM_XMODE_ARMSTATE). - X0 run state (SPCM_X0_MODE = SPCM_XMODE_RUNSTATE). Write a „0“ to disable all three outputs: - SPCM_X0_MODE = SPCM_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. Do not use both methods together from within one program.

Using synchronous digital outputs

This mode allows the user to replay up to three additional digital channels that are synchronous and phase stable along with the analog data. To enable that mode for a particular Multi Purpose I/O line the the digital output mode must selected along with some additional information:

Table 107: Spectrum API: multi-purpose I/O registers and synchronous digital output settings

Register	Value	Direction	Description
SPCM_X0_AVAILMODES	47210	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X0)
SPCM_X1_AVAILMODES	47211	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X1)
SPCM_X2_AVAILMODES	47212	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X2)
SPCM_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_DIGOUT	00000008h	D/A cards only: Connector is programmed for synchronous digital output. Digital channels can be „included“ within the analog samples and synchronously replayed along. Requires additional MODE bits to be set along with this flag, as explained later on.	

Additional constants that must be combined together with SPCM_XMODE_DIGOUT to select the analog channel or channels containing the digital data information and also the bit of the combined data word to be used for digital output:

SPCM_XMODE_DIGOUTSRC_CH0	01000000h	Select channel 0 as source (channel 0 must be enabled for replay).
SPCM_XMODE_DIGOUTSRC_CH1	02000000h	Select channel 1 as source (channel 1 must be enabled for replay).
SPCM_XMODE_DIGOUTSRC_CH2	04000000h	Select channel 2 as source (channel 2 must be enabled for replay).
SPCM_XMODE_DIGOUTSRC_CH3	08000000h	Select channel 3 as source (channel 3 must be enabled for replay).
SPCM_XMODE_DIGOUTSRC_BIT15	00100000h	Use Bit15 of selected channel: channel's resolution will be reduced to 15 bit.
SPCM_XMODE_DIGOUTSRC_BIT14	00200000h	Use Bit14 of selected channel: channel's resolution will be reduced to 14 bit, even if bit 15 is not used for digital replay.
SPCM_XMODE_DIGOUTSRC_BIT13	00400000h	Use Bit13 of selected channel: channel's resolution will be reduced to 13 bit, even if bit 15 and/or bit 14 are not used for digital replay.

A channel's samples can contain also information for the synchronous digital output channels, with up to three digital channels combined with the analog sample within one data word. When extracting the digital channels from the data word, the analog data will automatically be shifted upwards, to not loose any gain information. The analog data is still in the same twos complement format.

Table 108: Spectrum API: data format and DAC resolution depending on selected mode and digital output modes

Data bit	Standard Mode No embedded digital Bit 16 bit DAC resolution	Digital outputs enabled 1 embedded digital Bit 15 bit DAC resolution	Digital outputs enabled 2 embedded digital Bits 14 bit DAC resolution	Digital outputs enabled 3 embedded digital Bits 13 bit DAC resolution
D15	DAx Bit 15 (MSB)	Digital „Bit15“ of channel x	Digital „Bit15“ of channel x	Digital „Bit15“ of channel x
D14	DAx Bit 14	DAx Bit 15 (MSB)	Digital „Bit14“ of channel x	Digital „Bit14“ of channel x
D13	DAx Bit 13	DAx Bit 14	DAx Bit 15 (MSB)	Digital „Bit13“ of channel x
D12	DAx Bit 12	DAx Bit 13	DAx Bit 14	DAx Bit 15 (MSB)
D11	DAx Bit 11	DAx Bit 12	DAx Bit 13	DAx Bit 14
D10	DAx Bit 10	DAx Bit 11	DAx Bit 12	DAx Bit 13
D9	DAx Bit 9	DAx Bit 10	DAx Bit 11	DAx Bit 12
D8	DAx Bit 8	DAx Bit 9	DAx Bit 10	DAx Bit 11
D7	DAx Bit 7	DAx Bit 8	DAx Bit 9	DAx Bit 10
D6	DAx Bit 6	DAx Bit 7	DAx Bit 8	DAx Bit 9
D5	DAx Bit 5	DAx Bit 6	DAx Bit 7	DAx Bit 8
D4	DAx Bit 4	DAx Bit 5	DAx Bit 6	DAx Bit 7
D3	DAx Bit 3	DAx Bit 4	DAx Bit 5	DAx Bit 6
D2	DAx Bit 2	DAx Bit 3	DAx Bit 4	DAx Bit 5
D1	DAx Bit 1	DAx Bit 2	DAx Bit 3	DAx Bit 4
D0	DAx Bit 0 (LSB)	DAx Bit 1 (LSB)	DAx Bit 2 (LSB)	DAx Bit 3 (LSB)

This very flexible routing allows the use of one up to three digital outputs, whose data is included in the samples of only one, two or three different channels. This allows to only enable as much digital channels as needed, whilst keeping the resolution of the analog channels as high as possible.

The following example shows the generation of analog data on four channels with two channels sourcing all three digital outputs:

```
uint32 dwXMode;

// enable all four channels
spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0 | CHANNEL1 | CHANNEL2 | CHANNEL3);

// X0 set to synchronous output Bit 15 of channel 0
dwXMode = (SPCM_XMODE_DIGOUT | SPCM_XMODE_DIGOUTSRC_CH0 | SPCM_XMODE_DIGOUTSRC_BIT15);
spcm_dwSetParam_i32 (hDrv, SPCM_X0_MODE, dwXMode);

// X1 set to synchronous output Bit 15 of channel 1
dwXMode = (SPCM_XMODE_DIGOUT | SPCM_XMODE_DIGOUTSRC_CH1 | SPCM_XMODE_DIGOUTSRC_BIT15);
spcm_dwSetParam_i32 (hDrv, SPCM_X1_MODE, dwXMode);

// X2 set to synchronous output Bit 14 of channel 1
dwXMode = (SPCM_XMODE_DIGOUT | SPCM_XMODE_DIGOUTSRC_CH1 | SPCM_XMODE_DIGOUTSRC_BIT14);
spcm_dwSetParam_i32 (hDrv, SPCM_X2_MODE, dwXMode);
```

The following example shows the generation of analog data on just one channel sourcing all three digital outputs:

```
uint32 dwXMode;

// enable only one channel
spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0);

// X0 set to synchronous output Bit 15 of channel 0
dwXMode = (SPCM_XMODE_DIGOUT | SPCM_XMODE_DIGOUTSRC_CH0 | SPCM_XMODE_DIGOUTSRC_BIT15);
spcm_dwSetParam_i32 (hDrv, SPCM_X0_MODE, dwXMode);

// X1 set to synchronous output Bit 14 of channel 0
dwXMode = (SPCM_XMODE_DIGOUT | SPCM_XMODE_DIGOUTSRC_CH0 | SPCM_XMODE_DIGOUTSRC_BIT14);
spcm_dwSetParam_i32 (hDrv, SPCM_X1_MODE, dwXMode);

// X2 set to synchronous output Bit 13 of channel 0
dwXMode = (SPCM_XMODE_DIGOUT | SPCM_XMODE_DIGOUTSRC_CH0 | SPCM_XMODE_DIGOUTSRC_BIT13);
spcm_dwSetParam_i32 (hDrv, SPCM_X2_MODE, dwXMode);
```

The following example shows the generation of analog data on two channels sourcing the one synchronous digital output:

```
uint32 dwXMode;

// enable two channels
spcm_dwSetParam_i32 (hDrv, SPC_CHENABLE, CHANNEL0 | CHANNEL1);

// X0 set to synchronous output Bit 15 of channel 1
dwXMode = (SPCM_XMODE_DIGOUT | SPCM_XMODE_DIGOUTSRC_CH1 | SPCM_XMODE_DIGOUTSRC_BIT15);
spcm_dwSetParam_i32 (hDrv, SPCM_X0_MODE, dwXMode);

// X1 set to trigger output
dwXMode = (SPCM_XMODE_DIGOUT | SPCM_XMODE_TRIGOUT);
spcm_dwSetParam_i32 (hDrv, SPCM_X1_MODE, dwXMode);
```

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.

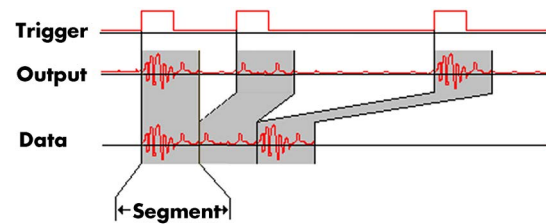


Image 62: Multiple Replay output and trigger timing diagram

Table 109: Spectrum API: segment size register for multiple replay mode

Register	Value	Direction	Description
SPC_SEGMENTSIZE	10010	read/write	Size of one Multiple Replay segment: the total number of samples to be replayed per channel after detection of one trigger event.

Trigger Modes

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

Programming examples

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

```

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

spcm_dwSetParam_i64 (hDrv, SPC_SEGMENTSIZE, 1024); // Set the segment size to 1024 samples
spcm_dwSetParam_i64 (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 trig mode 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_i64 (hDrv, SPC_SEGMENTSIZE, 2048); // Set the segment size to 2048 samples
spcm_dwSetParam_i64 (hDrv, SPC_LOOPS, 256); // 256 segments will be replayed

spcm_dwSetParam_i32 (hDrv, SPC_TRIG_EXT0_MODE, SPC_TM_NEG); // Set trig mode 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 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.

Table 110: Spectrum API: card mode register and multiple replay settings

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.

Table 111: Spectrum API: memory and loop registers with related multiple replay settings

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.
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.

Standard replay mode with the use of SPC_LOOPS

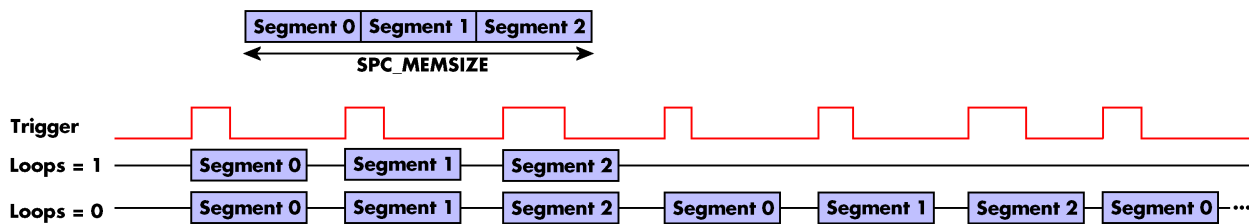


Image 63: timing diagram of multiple replay mode depending on loops settings

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 transferred 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.

Image 64: Spectrum API: card mode register and multiple replay FIFO mode settings

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode
SPC_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:

Table 112: Spectrum API: loops register settings when using Multiple Replay FIFO mode

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

Fifo replay mode with the use of SPC LOOPS

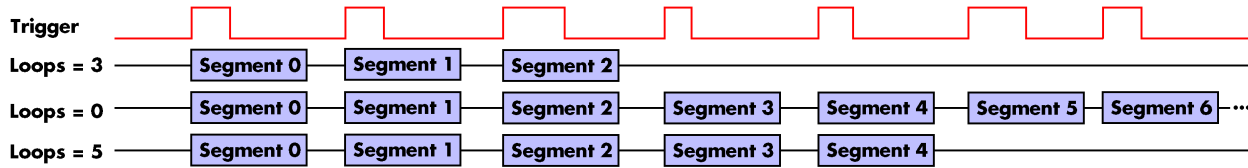


Image 65: timing diagram of Multiple Replay FIFO mode with different loops settings

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 sample 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:

Table 113: Spectrum API: limits of segment size, memory size and loops registers depending on selected mode

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	0 (∞)	1	1
	Standard Gate	32	Mem	32	not used			0 (∞)	1	1
	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/2	32	not used			0 (∞)	4G - 1	1
	Single Restart	32	Mem/2	32	not used			0 (∞)	4G - 1	1
	Standard Multi	32	Mem/2	32	16	Mem/4	16	0 (∞)	1	1
	Standard Gate	32	Mem/2	32	not used			0 (∞)	1	1
	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/4	32	not used			0 (∞)	4G - 1	1
	Single Restart	32	Mem/4	32	not used			0 (∞)	4G - 1	1
	Standard Multi	32	Mem/4	32	16	Mem/8	16	0 (∞)	1	1
	Standard Gate	32	Mem/4	32	not used			0 (∞)	1	1
	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 behavior 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 behavior after replay:

Table 114: Spectrum API: stop level register and register settings

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

Table 114: Spectrum API: stop level register and register settings

Register	Value	Direction	Description
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). When synchronous digital bits are replayed, these will be set to LOW state during pause.
SPCM_STOPLVL_LOW	2		Defines the analog output to enter maximum negative level (D/A converter is fed with most negative level). When synchronous digital bits are replayed, these will be set to LOW state during pause.
SPCM_STOPLVL_HIGH	4		Defines the analog output to enter maximum positive level (D/A converter is fed with most positive level). When synchronous digital bits are replayed, these will be set to HIGH state during pause.
SPCM_STOPLVL_HOLDLAST	8		Holds the last replayed sample on the analog output. When synchronous digital bits are replayed, their last state will also be hold.
SPCM_STOPLVL_CUSTOM	32		Allows to define a 16bit wide custom level per channel for the analog output to enter in pauses. The sample format is exactly the same as during replay, as described in the „sample format“ section. When synchronous digital bits are replayed along, the custom level must include these as well and therefore allows to set a custom level for each multi-purpose line separately.

When using SPCM_STOPLVL_CUSTOM, the sample value for the pauses must be defined via the following registers:

Table 115: Spectrum API: custom stop level registers

Register	Value	Direction	Description
SPC_CH0_CUSTOM_STOP	206050	read/write	Defines the custom stop level for channel 0 when using SPCM_STOPLVL_CUSTOM.
SPC_CH1_CUSTOM_STOP	206051	read/write	Defines the custom stop level for channel 1 when using SPCM_STOPLVL_CUSTOM.
SPC_CH2_CUSTOM_STOP	206052	read/write	Defines the custom stop level for channel 2 when using SPCM_STOPLVL_CUSTOM.
SPC_CH3_CUSTOM_STOP	206053	read/write	Defines the custom stop level for channel 3 when using SPCM_STOPLVL_CUSTOM.

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

Example showing how to set a custom stoplevel for channel 0:

```
// enable the use of custom stop level and use raw value 10487 as stop value
spcm_dwSetParam_i32 (stCard.hDrv, SPC_CH0_STOPLEVEL, SPCM_STOPLVL_CUSTOM);
spcm_dwSetParam_i32 (stCard.hDrv, SPC_CH0_CUSTOM_STOP, 10487);
```

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.

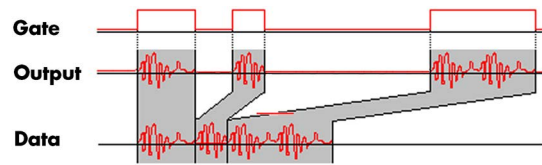


Image 66: Gated Replay timing diagram in relation to gate signal

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 fulfills 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.

The table below shows the register for enabling Gated Sampling. For detailed information on how to setup and start the standard acquisition mode please refer to the according chapter earlier in this manual.

Table 116: Spectrum API: card mode register and settings for Gated Replay standard mode

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.

Table 117: Spectrum API: memsize and loops register and register settings for Gated Replay mode

Register	Value	Direction	Description
SPC_MEMSIZE	10000	read/write	Defines the total number of samples to be 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 segment is cut off when end of memory is reached.

Examples of Standard 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.

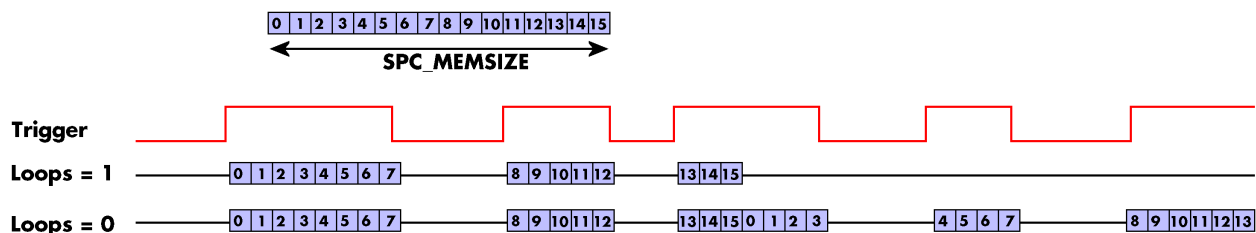


Image 67: timing diagram of Gated Replay mode depending on different loops settings

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.

Table 118: Spectrum API: card mode register and Gated Replay FIFO mode settings

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:

Table 119: Spectrum API: Gated Replay FIFO mode loops register settings

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.

Examples of Fifo 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.

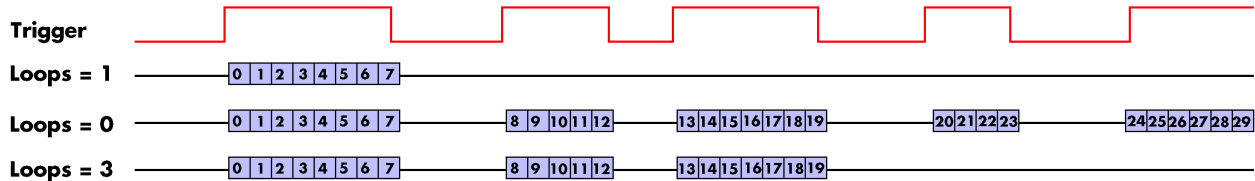


Image 68: timing diagram of Gated Replay FIFO mode depending on different loops settings

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 sample 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:

Table 120: Spectrum API: limits of segment size, memory size and loops registers depending on selected mode

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	0 (∞)	1	1
	Standard Gate	32	Mem	32	not used			0 (∞)	1	1
	FIFO Single	not used			16	8G - 16	16	0 (∞)	4G - 1	1
	FIFO Multi	not used			16	Mem/2	16	0 (∞)	4G - 1	1
2 channels	FIFO Gate	not used			not used			0 (∞)	4G - 1	1
	Standard Single	32	Mem/2	32	not used			0 (∞)	4G - 1	1
	Single Restart	32	Mem/2	32	not used			0 (∞)	4G - 1	1
	Standard Multi	32	Mem/2	32	16	Mem/4	16	0 (∞)	1	1
	Standard Gate	32	Mem/2	32	not used			0 (∞)	1	1
	FIFO Single	not used			16	8G - 16	16	0 (∞)	4G - 1	1
4 channels	FIFO Multi	not used			16	Mem/4	16	0 (∞)	4G - 1	1
	FIFO Gate	not used			not used			0 (∞)	4G - 1	1
	Standard Single	32	Mem/4	32	not used			0 (∞)	4G - 1	1
	Single Restart	32	Mem/4	32	not used			0 (∞)	4G - 1	1
	Standard Multi	32	Mem/4	32	16	Mem/8	16	0 (∞)	1	1
	Standard Gate	32	Mem/4	32	not used			0 (∞)	1	1
	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.

Trigger

Detailed description of the external analog trigger modes

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

Table 121: Spectrum API: trigger mask registers and available register settings

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 gate starts.

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

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

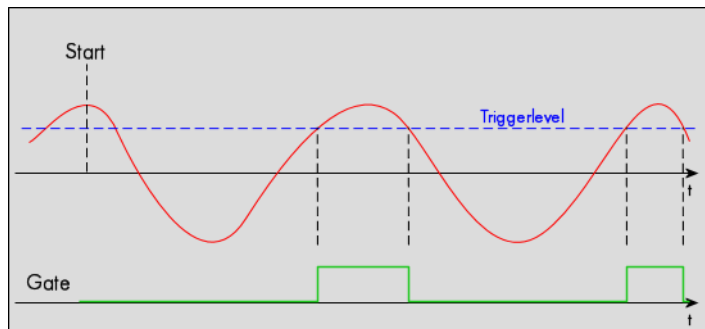


Table 122: Spectrum API: trigger register settings for trigger on positive edge

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 gate starts.

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

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

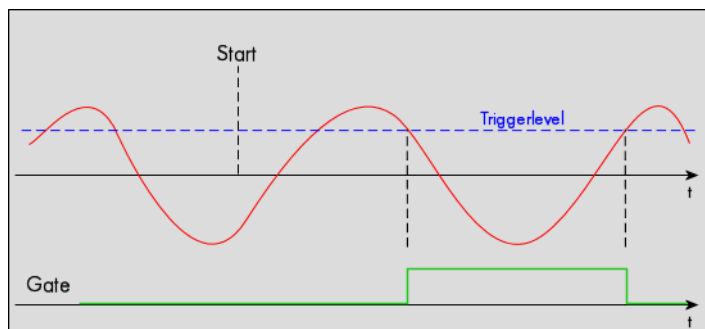


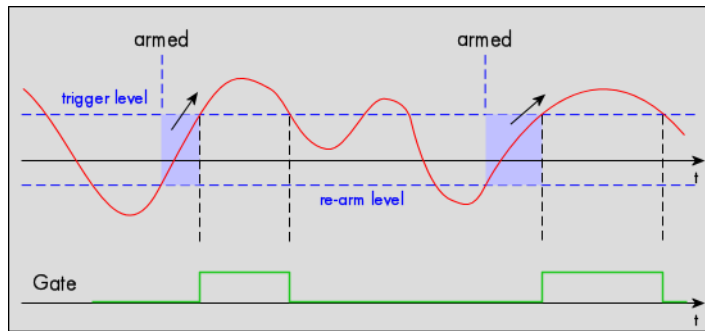
Table 123: Spectrum API: trigger register settings for trigger on negative edge

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

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 gate starts will be detected and the trigger engine will be disarmed. A new trigger event is only detected if the trigger engine is armed again.

If the programmed trigger level is crossed by the external signal from higher values to lower values (falling edge) the gate stops.



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

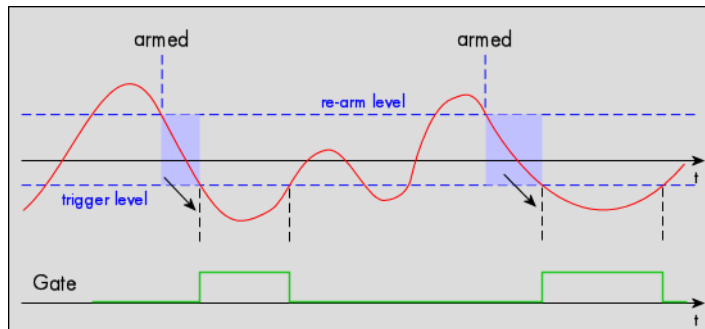
Table 124: Spectrum API: trigger register settings for re-arm trigger on positive edge

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 gate starts and the trigger engine will be disarmed. A new trigger event is only detected, if the trigger engine is armed again.

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



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

Table 125: Spectrum API: trigger register settings for re-arm trigger on negative edge

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.

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

As this mode is purely edge-triggered, the signal outside the window at the cards start time does not trigger the board.

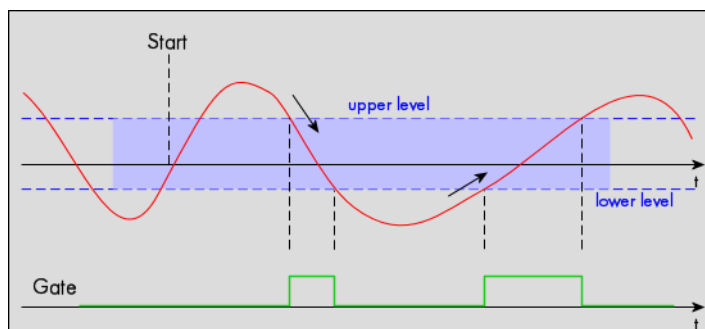


Table 126: Spectrum API: trigger register settings for window trigger on entering signals

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.

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

As this mode is purely edge-triggered, the signal within the window at the cards start time does not trigger the board.

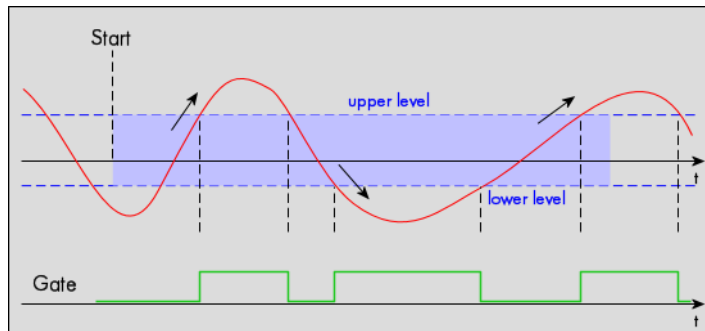


Table 127: Spectrum API: trigger register settings for window trigger on leaving signals

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

The external input is continuously sampled with the selected sample rate. If the signal is equal or higher than the programmed trigger level the gate starts.

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

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

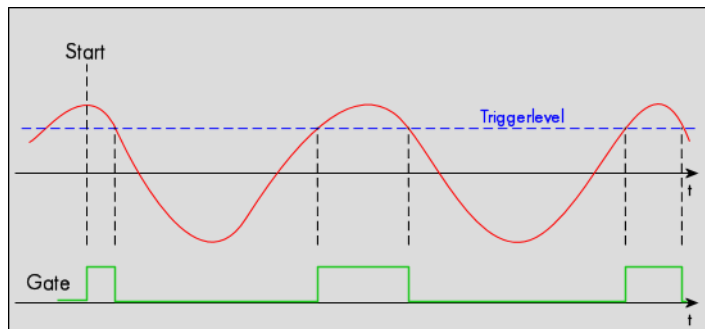


Table 128: Spectrum API: trigger register settings for high-level trigger

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

The external input is continuously sampled with the selected sample rate. If the signal is equal or lower than the programmed trigger level the gate starts.

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

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

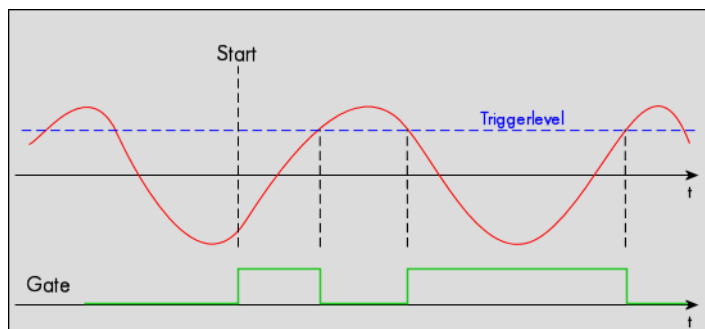


Table 129: Spectrum API: trigger register settings for low-level trigger

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

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

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

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

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

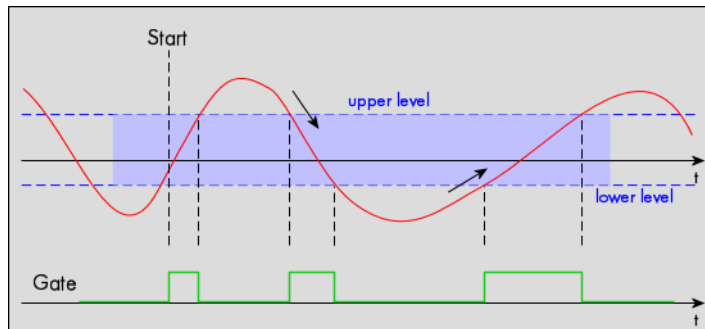


Table 130: Spectrum API: trigger register settings for in-window trigger

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

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

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

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

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

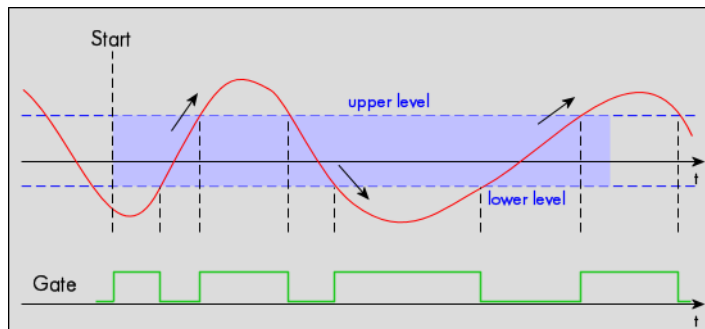


Table 131: Spectrum API: trigger register settings for outside-window trigger

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

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_i64 (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 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_i64 (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 falling edge
spcm_dwSetParam_i32 (hDrv, SPC_TRIG_ORMASK, SPC_TMASK_EXT0); // and enable it within the trigger OR-mask

```

Programming the behavior 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 behavior after replay:

Table 132: Spectrum API: stop level register and register settings

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). When synchronous digital bits are replayed, these will be set to LOW state during pause.
SPCM_STOPLVL_LOW	2		Defines the analog output to enter maximum negative level (D/A converter is fed with most negative level). When synchronous digital bits are replayed, these will be set to LOW state during pause.
SPCM_STOPLVL_HIGH	4		Defines the analog output to enter maximum positive level (D/A converter is fed with most positive level). When synchronous digital bits are replayed, these will be set to HIGH state during pause.
SPCM_STOPLVL_HOLDLAST	8		Holds the last replayed sample on the analog output. When synchronous digital bits are replayed, their last state will also be hold.
SPCM_STOPLVL_CUSTOM	32		Allows to define a 16bit wide custom level per channel for the analog output to enter in pauses. The sample format is exactly the same as during replay, as described in the „sample format“ section. When synchronous digital bits are replayed along, the custom level must include these as well and therefore allows to set a custom level for each multi-purpose line separately.

When using SPCM_STOPLVL_CUSTOM, the sample value for the pauses must be defined via the following registers:

Table 133: Spectrum API: custom stop level registers

Register	Value	Direction	Description
SPC_CH0_CUSTOM_STOP	206050	read/write	Defines the custom stop level for channel 0 when using SPCM_STOPLVL_CUSTOM.
SPC_CH1_CUSTOM_STOP	206051	read/write	Defines the custom stop level for channel 1 when using SPCM_STOPLVL_CUSTOM.
SPC_CH2_CUSTOM_STOP	206052	read/write	Defines the custom stop level for channel 2 when using SPCM_STOPLVL_CUSTOM.
SPC_CH3_CUSTOM_STOP	206053	read/write	Defines the custom stop level for channel 3 when using SPCM_STOPLVL_CUSTOM.

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

Example showing how to set a custom stoplevel for channel 0:

```
// enable the use of custom stop level and use raw value 10487 as stop value
spcm_dwSetParam_i32 (stCard.hDrv, SPC_CH0_STOPLEVEL, SPCM_STOPLVL_CUSTOM);
spcm_dwSetParam_i32 (stCard.hDrv, SPC_CH0_CUSTOM_STOP, 10487);
```

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 or spikes. 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 $[\text{Installed Memory}] / [\text{Number of Segments}]$. 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.

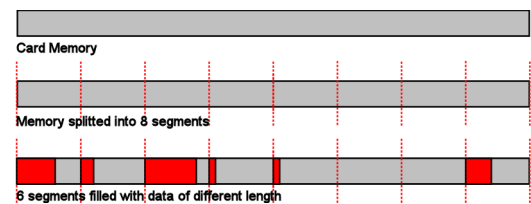


Image 69: Sequence Mode: Segment definition in card memory

Each data segment is filled with data for all active channels in a multiplexed way. Please check the chapter "data organization" for details of how to organize the data inside each segment.

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.

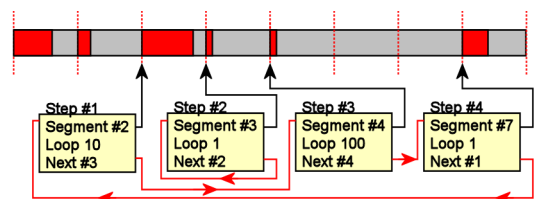


Image 70: Sequence mode: steps and step looping

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:

Table 134: Spectrum API: sequence mode registers and register settings

Register			
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			
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:

Table 135: Spectrum API: card mode register with Sequence Mode setup

Register			
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:

Table 136: Spectrum API: sequence mode registers for segment handling

Register			
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. 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 in samples per channel.

Each data segment is filled with data for all active channels in a multiplexed way. Please check the chapter "data organization" for details of how to organize the data inside each segment.

Due to the internal organization of the card memory there is a certain minimum, maximum and stepsize when setting the segment size 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.

Table 137: Spectrum API: sequence mode step registers and register setup

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	000FFFFh		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:

Table 138: Spectrum API: sequence mode start register

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 number of the sequence step that is currently at the output connector of the card. This could be extremely useful if external equipment has to be changed after a dedicated sequence has been replayed or if the AWG is changing between different patterns in automatic test environment.

Table 139: Spectrum API: sequence mode segment status register

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



Due to the internal structure of the sequencer, the delay between a trigger event and the change in the sequence, when using the SPCSEQ_ENDLOOPONTRIG feature, is not a fixed value but rather varies with the current fill-size of the Output FIFO. Please see „Output latency“ section in this manual for the size of the Output FIFO on your card and take half the value given there as the max. value when using the Sequence mode. In Sequence mode only half the value is involved, since in this mode only the FIFO between the on-board memory and the DAC is relevant and the FIFO between the PCIe receiver and the on-board memory not in use in Sequence mode.

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:

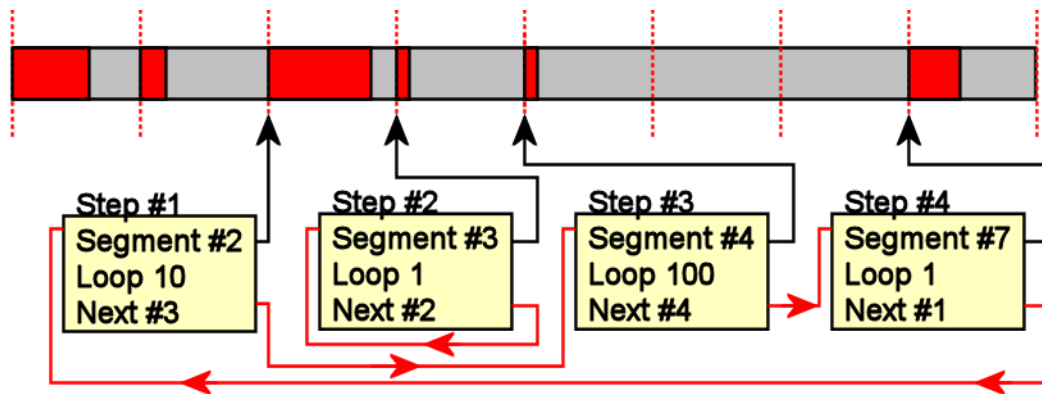


Image 71: sequence mode changing sequence on-the-fly

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 to 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 of the associated segment #3. As step #2 links back to itself it will generate data segment #3 in an endless loop until being either stopped by a software command or another change in the sequence is applied.

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 useful 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, so that the driver can automatically re-calculate the internal start-addresses of the segments.

Synchronization



Please note that the sequence mode is NOT fully synchronized using the star-hub. This also relates to generatorNETBOX products with an internal star-hub.

Using sequence mode together with star-hub, it is still possible to

- synchronize the clock
- synchronize the start of the cards

However, with star-hub synchronization, it is NOT possible to

- synchronize any changes inside the step memory
- synchronize software commands that change the step memory order
- synchronize a trigger that ends a steps loop

The above mentioned restrictions are also valid with any other setup of synchronization apart from star-hub, be it via external clock or internal SH-direct clock.

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)
int32 lStep = 0; // current step is Step#0
int64 llSegment = 0; // associated with data memory segment 0
int64 llLoop = 10; // Pattern will be repeated 10 times
int64 llNext = 1; // Next step is Step#1
int64 llCondition = SPCSEQ_ENDLOOPALWAYS; // Unconditionally leave current step

// combine all the parameters to one int64 bit value
int64 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 data memory segment 1
llLoop = 1; // Pattern will be repeated once before condition is checked
llNext = 0; // Next step is Step#0
llCondition = SPCSEQ_ENDLOOPONTRIG; // 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);
```

Mode DDS (Firmware Option)

General Information

DDS – Direct Digital Synthesis – is a method for generating arbitrary periodic waves from a single, fixed-frequency reference clock and is widely used in signal generation applications. The DDS functionality implemented on Spectrum Instrumentation's AWGs is based on the principle of adding multiple "DDS cores" to generate a multi-carrier (multi-tone) signal, with each carrier having its own well-defined frequency, amplitude and phase. In addition to these static parameters, there are also build in dynamic parameters like frequency and amplitude slope to allow for intrinsic linear changes for multiple cores.

In the simplest case, the user writes the commands frequency and amplitude for a specific DDS core to the card. The card will then output a single periodic sine wave continuously until the user writes a change to the card. These changes are written to the card in the form of commands (see below tables for a list of the available commands) that are added to a First-In-First-Out (FIFO) buffer. These commands are then executed in the order in which they were written to the card.

Feature Overview

The DDS firmware allows the user a certain maximum number of DDS cores (see register SPC_DDS_NUM_CORES), that each on its own generates a sine wave with the following parameters:

Static Parameters:

- Frequency (e.g., SPC_DDS_CORE0_FREQ)
- Amplitude (e.g., SPC_DDS_CORE0_AMP)
- Phase (e.g., SPC_DDS_CORE0_PHASE)

Dynamic Parameters:

- Frequency slope (e.g., SPC_DDS_CORE0_FREQ_SLOPE)
changes the active frequency of the DDS core with a linear slope
- Amplitude slope (e.g., SPC_DDS_CORE0_AMP_SLOPE)
changes the active amplitude of the DDS core with a linear slope.

Each of these cores can either be added together and outputted, or specific groups of cores can be added together and outputted on a specific hardware output channel. See drawing below for details.

Controlling the DDS functionality

The whole card needs to be switched to DDS mode to allow programming of the DDS functionality. The below table only shows the DDS mode. All other modes are shown in the "Generation modes" chapter of this manual:

Table 140: Spectrum API: card mode and read out of available card mode software registers

Register	Value	Direction	Description
SPC_CARDMODE	9500	read/write	Defines the used operating mode, a read command will return the currently used mode.
SPC_AVAILCARDMODES	9501	read	Returns a bitmap with all available modes on your card. Please see the general chapter "Generation modes" for a list of all supported modes.
Mode	Value	Description	
SPC_REP_STD_DDS	4000000h	DDS replay mode functionality available	

Command FIFO

The DDS functionality is controlled through commands that are written to a driver-internal list and then written to the card when the command SPCM_DDS_CMD_WRITE_TO_CARD is sent. These lists of commands are put onto a First-In-First-Out (command queue) buffer and executed one after the other.

The right hand command queue overview gives an idea how commands are used to generate the different output states of a single DDS core. Commands in the command queue are executed from top to bottom.

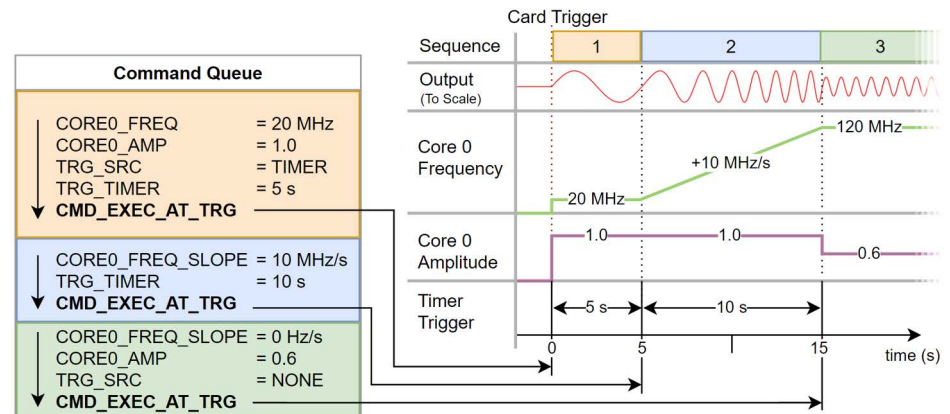


Image 72: command queue, trigger and timer interaction

The settings are first written to a set of “shadow registers” that are a separate set of registers in parallel to the active DDS configuration registers. One command after the other manipulates the shadow registers until the command SPCM_DDS_CMD_EXEC_AT_TRIG is received, then writing from the FIFO to the shadow registers is stopped and the card starts waiting for the next internal trigger. After a trigger is received the shadow registers are transferred to the active registers.

Hence, if two commands writing to the same register are sent before an SPCM_DDS_CMD_EXEC_AT_TRIG command was sent, then the second command overwrites the first.

DDS Command buffer fill size, overrun and underrun

The card internally has a command buffer (called “hardware DDS command buffer” in the drawing) that hold all single DDS commands that have been sent to the card. The size of this command buffer is fixed and can be read out using the SPC_DDS_QUEUE_CMD_MAX register. The current fill size of the command buffer can be read out by reading the SPC_DDS_QUEUE_CMD_COUNT register.

Inside the API there is a second command buffer (“software DDS command buffer”) which is handled by the API. The size of this buffer is automatically adjusted by the API as needed. All single commands sent by the application are placed in this software buffer. Each time a SPCM_DDS_CMD_WRITE_TO_CARD is performed, the whole software buffer is written to the hardware buffer. In case there are too many commands to fit in the hardware buffer, an overrun error is thrown.

On a trigger/timer/EXEC_NOW event, all content of the shadow registers is written to the registers. Directly after that the shadow registers are filled from the hardware buffer up to the next SPCM_DDS_CMD_EXEC_AT_TRIG. In case the hardware buffer doesn't hold a set of commands finalized by the SPCM_DDS_CMD_EXEC_AT_TRIG, an underrun error is thrown.

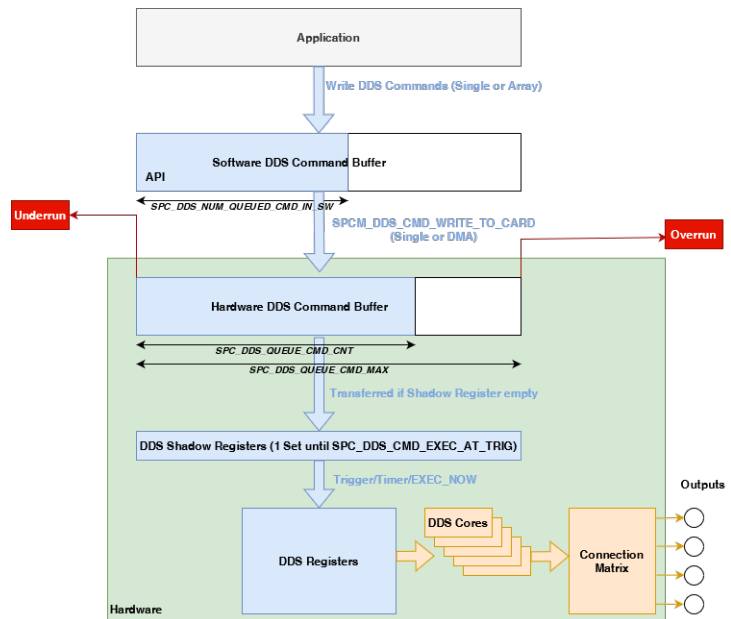


Image 73: Overview of the DDS buffer structure, data flow and size registers

Table 141: Spectrum API: DDS information on the command queue

Register	Value	Direction	Description
SPC_DDS_QUEUE_CMD_MAX	608005I	read	Length of the DDS command queue.
SPC_DDS_QUEUE_CMD_COUNT	608006I	read	Current number of entries in the DDS command queue.
SPC_DDS_NUM_QUEUED_CMD_IN_SW	608011I	read	Returns the number of commands that are currently in the command queue in the library and not yet written to the card. Writing to the card is done with the SPCM_DDS_CMD_WRITE_TO_CARD command.

DDS Command

Table 142: Spectrum API: DDS command register

Register	Value	Direction	Description
SPC_DDS_CMD	608003	write	Register for DDS-specific commands.
SPCM_DDS_CMD_RESET	1h		Resets all DDS-specific firmware registers to default state
SPCM_DDS_CMD_EXEC_AT_TRG	2h		Apply the current changes at DDS trigger. DDS trigger can be card trigger or DDS timer
SPCM_DDS_CMD_EXEC_NOW	4h		Apply the current changes immediately.
SPCM_DDS_CMD_WRITE_TO_CARD	8h		The driver accumulates all DDS internally settings. This command transfers them to the card.
SPCM_DDS_CMD_NO_NOP_FILL	10h		Special flag for DMA mode, please see separate chapter

Before starting the DDS output, the channels need to be enabled and output level needs to be defines and finally the card itself needs to be started using the start and enable trigger command. This is necessary to write all general settings to the hardware and to initialize trigger and timing engine. Without the card start command, timing is unpredictable.

When writing the first bunch of settings to the DDS engine, they are loaded into the shadow registers. This first bunch of settings need to at least define the DDS trigger source (please see below) for the following commands. After receiving a SPCM_DDS_CMD_EXEC_AT_TRG command, the shadow registers are transferred into the active registers as soon as a trigger is received. The trigger is defined in the general card trigger registers (see separate chapter) and can be a hardware trigger, a software trigger of a separate force trigger command.

Please see below example for the starting sequence of the DDS output with a single hardware channel active:

```

spcm_dwSetParam_i32 (hCard, SPC_CHENABLE, CHANNEL0);           // enable channel 0
spcm_dwSetParam_i32 (hCard, SPC_ENABLEOUT0, 1);                // output enabled (default)
spcm_dwSetParam_i32 (hCard, SPC_AMP0, 1000);                   // output amplitude 1000 mV
spcm_dwSetParam_i32 (hCard, SPC_FILTER0, 0);                    // full bandwidth, no filter
spcm_dwSetParam_i32 (hCard, SPC_CARDMODE, SPC_REP_STD_DDS);     // DDS mode
spcm_dwSetParam_i32 (hCard, SPC_CLOCKMODE, SPC_CM_INTPLL);     // clock mode internal PLL

// card start, writing of all paramters
spcm_dwSetParam_i32 (hCard, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);

// write DDS setup for first sine signal
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP, 1);              // 100% amplitude
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ, MEGA(100));    // 100 MHz sine signal
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG); // execute at trigger
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_WRITE_TO_CARD); // write all dds settings to card

// start the output by initiating a force trigger
spcm_dwSetParam_i32 (hCard, SPC_M2CMD, M2CMD_CARD_FORCE_TRIGGER);

```

Internal trigger sources

There are three different internal trigger sources, that can be set with the register entry SPC_DDS_TRG_SRC. The DDS can be triggered by an internal timer or use the card's trigger event.:

Table 143: Spectrum API: DDS trigger sources

Register	Value	Direction	Description
SPC_DDS_TRG_SRC	608000l	write	Register for DDS-specific trigger condition.
SPCM_DDS_TRG_SRC_NONE	0h		No special DDS trigger is used. To change a setting SPCM_DDS_CMD_EXEC_NOW needs to be used (see above).
SPCM_DDS_TRG_SRC_TIMER	1h		The firmware itself generates a repeating trigger event. The triggers are generated on a timed grid with a period that can be set by the register SPC_DDS_TRG_TIMER
SPCM_DDS_TRG_SRC_CARD	2h		The trigger source uses the cards internal trigger logic. For more information, see the product's user manual on how to set up the different trigger conditions. In the DDS-mode, multiple triggers are processed, with each trigger received advancing to the next command step.

The selection of a trigger source is also one of the commands that is executed on receipt of a trigger. That means, changing the trigger source happens only when a SPCM_DDS_CMD_EXEC_AT_TRG command was sent and a next trigger/timer event was received.

DDS status register

The status register reports whether the DDS command queue is waiting for a trigger or has missed a trigger:

Table 144: Spectrum API: DDS trigger status register

Register	Value	Direction	Description
SPC_DDS_STATUS	608004l	read	Register for reading out the DDS status
SPCM_DDS_STAT_WAITING_FOR_TRG	1h		DDS is waiting for a trigger event
SPCM_DDS_STAT_QUEUE_UNDERRUN	2h		The DDS detected a trigger or timer event prior to receiving the EXEC_AT_TRG command. The trigger source was set to SRC_TIMER or SRC_CARD and a trigger or timer event was detected but the finalizing command EXEC_AT_TRG was not sent in time.

General DDS information register

These registers can be used to generalize software. Using these information registers, one can read out the different available settings. This could be helpful to utilize software to different hardware versions or firmware release versions:

Table 145: Spectrum API: DDS information registers

Register	Value	Direction	Description
SPC_DDS_QUEUE_CMD_MAX	608005I	read	Length of the DDS command queue. Read-only.
SPC_DDS_QUEUE_CMD_COUNT	608006I	read	Current number of entries in the DDS command queue. Read-only.
SPC_DDS_NUM_CORES	608007I	read	Number of available DDS cores. Depends on the specific card. Read-only.
SPC_DDS_NUM_QUEUED_CMD_IN_SW	608011I	read	Returns the number of commands that are currently in the command queue in the library and not yet written to the card. Writing to the card is done with the SPCM_DDS_CMD_WRITE_TO_CARD command.
SPC_DDS_TRG_COUNT	608013I	read	Returns the number of trigger events that have been received since the last SPCM_DDS_CMD_RESET. A trigger event is any event that executes a DDS command, including external triggers, the timer or manual events (SPCM_DDS_CMD_EXEC_NOW)

Trigger timer

The internal trigger timer defines an interval after which the DDS section automatically generates a trigger signal to advance to the next sequence of DDS settings. Please note that this interval needs to be long enough to allow to execute all DDS settings. Only used if SPC_DDS_TRG_SRC is set to SPCM_DDS_TRG_SRC_TIMER:

Table 146: Spectrum API: DDS trigger timer

Register	Value	Direction	Description
SPC_DDS_TRG_TIMER	608001I	read/write	The interval of the timer can be set in seconds or fraction of seconds as double value using the spcm_dwGetParam_d64 function. The minimum value, maximum value and resolution of the timer depends on an internal system clock which is a fraction of the output clock. Please see the table below for available settings for the different product families.

Each timer interval corresponds to a number of output clocks depending on the platform and type of product used. The driver finds the nearest matching timer interval. It is therefore recommended to read-back the timer to determine the exact programmed timer value.

Table 147: Spectrum API: DDS trigger timer resolution depending on product

Products	DAC output rate	Timer clocks	Min timer	Max timer	Timer resolution	DDS update rate
DN2.66x-xx DN6.66x-xx M4i.66xx M4x.66xx	1.25 GS/s	8	83.2 ns	27.48 s	6.4 ns	6.4 ns

As an example, writing a 0.00001 (10 us) to the SPC_DDS_TRG_TIMER register on a M4i.66xx AWG card sets the DDS trigger timer to precisely 0.0000100032 s (10.0032 us or 1563 x 6.4 ns):

```
double dTimer_s = 0;

spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_TIMER, 0.00001);
spcm_dwGetParam_d64 (hCard, SPC_DDS_TRG_TIMER, &dTimer_s);
printf ("Programmed Trigger Timer: %lf s\n", dTimer_s);
```

DDS core connections

The DDS_CORE_ON_CHx registers defines the setup of the different multiplexers that form the connections of the different DDS cores on hardware channels. If a hardware channel is not present on your specific hardware, programming this connection does not have any function. In that case, dedicated DDS cores that are routed to a fixed hardware channel (like Core 22 being solely connected to hardware channel 3) cannot be used.

Each block represents a single DDS core or a group of DDS cores where the following parameters can be individually programmed:

- Amplitude
- Frequency
- Phase
- Amplitude Slope
- Frequency Slope

The cores on channel registers combine the setup of all shown multiplexers. Each core is represented by a single bit. All cores that should be routed to the dedicated channel needs to be combined by a logical OR. Only settings that match the shown possible connections in the right-hand drawing can be selected:

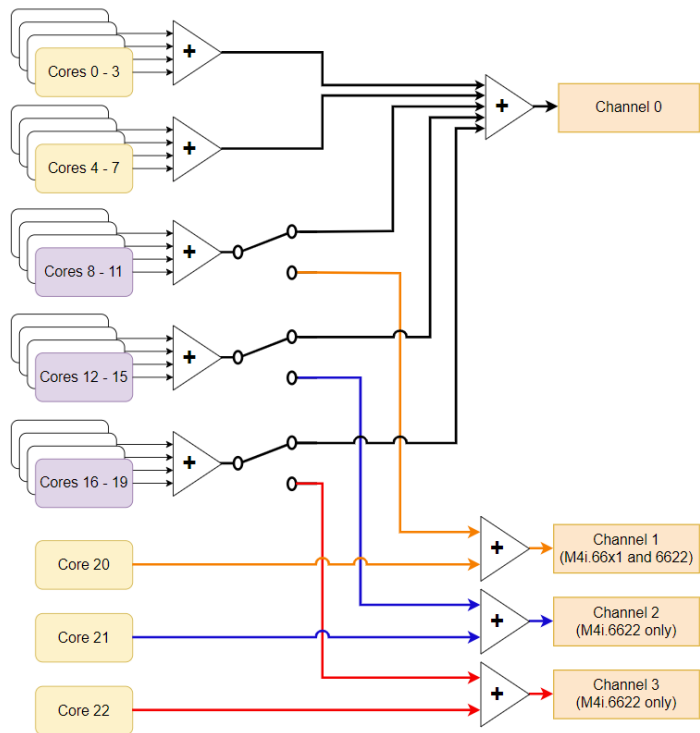


Image 74: block diagram of M4i.66xx DDS option showing the DDS cores and connections options.

Table 148: Spectrum API: DDS connection register

Register	Value	Direction	Description
SPC_DDS_CORES_ON_CH0	608040I	read/write	Setup of the DDS core routed to channel 0.
SPC_DDS_CORES_ON_CH1	608041I	read/write	Setup of the DDS core routed to channel 1.
SPC_DDS_CORES_ON_CH2	608042I	read/write	Setup of the DDS core routed to channel 2.
SPC_DDS_CORES_ON_CH3	608043I	read/write	Setup of the DDS core routed to channel 3.
SPCM_DDC_CORE_NONE	00000h		no core connected to channel
SPCM_DDC_CORE0	000001h		core 0 connected to channel
SPCM_DDC_CORE1	000002h		core 1 connected to channel
SPCM_DDC_CORE2	000004h		core 2 connected to channel
...

The following example for a 4-channel product connects the maximum number of cores to channel 0 and the minimum number of cores to channel 1, 2 and 3. Please note that we mix numbers and constants here to keep the example readable.

This setup is also the default setup after reset:

```
// connect the maximum number of cores to channel 0 and the minimum to channel 1, 2 and 3

spcm_dwSetParam_i64 (hCard, SPC_DDS_CORES_ON_CH0, 0x0FFFFFF);
spcm_dwSetParam_i64 (hCard, SPC_DDS_CORES_ON_CH1, SPCM_DDS_CORE20);
spcm_dwSetParam_i64 (hCard, SPC_DDS_CORES_ON_CH2, SPCM_DDS_CORE21);
spcm_dwSetParam_i64 (hCard, SPC_DDS_CORES_ON_CH3, SPCM_DDS_CORE22);
```

Programming the DDS cores

The DDS cores are individually programmed. The phase and frequency settings of each DDS core is determining which position of a programmed memory look-up-table (LUT) is used for this DDS output.

The amplitude value is used to attenuate the output of the DDS core before addition of multiple cores takes place.

All below mentioned settings can be programmed in parallel on each core. There is no specific "mode" register for the core. It is recommend to use the API functions writing and reading "double" values to program these settings:

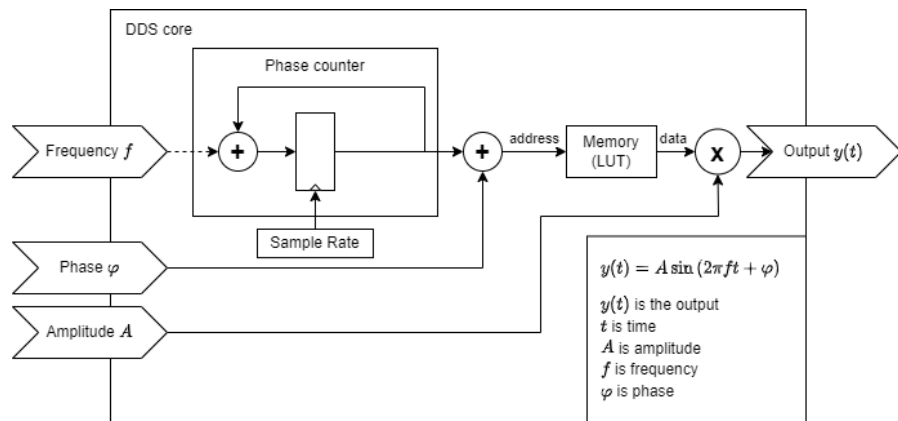


Image 76: Details of the DDS core

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

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

Frequency

The frequency settings are done in Hertz. The driver re-calculates the given frequency to the frequency best matching by taking the output rate and the restrictions of the DDS-core internal settings. It is advised to read the register after writing to determine which exact frequency has been calculated. Please also take into account the output filter of the AWG section. While large frequencies can be programmed for the DDS core, the output filter may fully filter them off the signal if they're above the filter frequency.

Table 149: Spectrum API: DDS core frequency settings

Register	Value	Direction	Description
SPC_DDS_CORE0_FREQ	603000I	read/write	Set up the frequency of DDS core 0 in Hertz. Read back to get the exact frequency settings. The setting can be programmed in the range from 0 Hz to [output rate] where frequencies above [output rate/2] are mirrored at the [output rate/2] and 180° phase shifted
SPC_DDS_CORE1_FREQ	603001I DDS core 1
...

The available programmable frequency minimum, maximum and stepsize can be read from the library:

Table 150: Spectrum API: DDS programmable frequency range

Register	Value	Direction	Description
SPC_DDS_AVAIL_FREQ_MIN	608500I	read	Minimum programmable frequency in Hz
SPC_DDS_AVAIL_FREQ_MAX	608501I	read	Maximum programmable frequency in Hz
SPC_DDS_AVAIL_FREQ_STEP	608502I	read	Stepsize of frequency programming in Hz

Amplitude

The amplitude of the DDS core is programmed as a fraction of the output level. Please check the chapter "Setting up the outputs" in the "Analog Outputs" section of the manual to see the different settings of the output amplifier. Writing a "1" in this register uses the full programmed output level of the output amplifier. Using multiple DDS cores that are accumulated, it may be a good idea to attenuate each DDS core to avoid overranging the used DAC. If the sum of all DDS core outputs exceeds the "1" at any time, the DAC will go into overflow and the output signal is clipped. This leads to heavy distortion in the signal.

Table 151: Spectrum API: DDS core amplitude settings

Register	Value	Direction	Description
SPC_DDS_CORE0_AMP	605000I	read/write	Set up the relative amplitude of DDS core on a scale between -1.0 and +1.0. Some examples: 0.0: no output at all 1.0: full amplitude output -0.5: inverted output with half of maximum
SPC_DDS_CORE1_AMP	605001I DDS core 1
...

The available programmable amplitude minimum, maximum and stepsize can be read from the library:

Table 152: Spectrum API: DDS programmable amplitude range

Register	Value	Direction	Description
SPC_DDS_AVAIL_AMP_MIN	608506I	read	Minimum programmable amplitude as scale between -1.0 and 1.0
SPC_DDS_AVAIL_AMP_MAX	608507I	read	Maximum programmable amplitude as scale between -1.0 and 1.0
SPC_DDS_AVAIL_AMP_STEP	608508I	read	Stepsize of programmable amplitude as scale between 0.0 and 1.0

Phase

The phase for each DDS core in relation to other DDS cores is set in degree. Valid values range from -360° to +360° with a resolution of $360/4096 = 0.088^\circ$.

Table 153: Spectrum API: DDS core phase settings

Register	Value	Direction	Description
SPC_DDS_CORE0_PHASE	607000I	read/write	Set up the phase of the DDS core in degree.
SPC_DDS_CORE1_PHASE	607001I DDS core 1
...

The available programmable phase minimum, maximum and stepsize can be read from the library:

Table 154: Spectrum API: DDS programmable phase range

Register	Value	Direction	Description
SPC_DDS_AVAIL_PHASE_MIN	608512I	read	Minimum programmable phase in degree
SPC_DDS_AVAIL_PHASE_MAX	608513I	read	Maximum programmable phase in degree
SPC_DDS_AVAIL_PHASE_STEP	608514I	read	Stepsize of programmable phase in degree

Simple example for fixed frequency output

The following example programs core 0 for a 50 MHz output, with full amplitude and 90° phase shift. The setup is written immediately without waiting for a trigger or a timer:

```
// ... card initialisation and output amplifier settings

spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP, 1);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_PHASE, 90);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ, MEGA(50));

// read back the exact signal frequency
double dFreq_Hz = 0;
spcm_dwGetParam_d64 (hCard, SPC_DDS_CORE0_FREQ, &dFreq_Hz);
printf ("Generated signal frequency: %lf Hz\n", dFreq_Hz);

spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_NOW);
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_WRITE_TO_CARD);
```

Slope update rate

Slopes are programmed with the above mentioned registers. The slope functionality automatically adds a calculated increment or decrement to either amplitude or frequency or both. The time resolution for the increment/decrement is the same as the above mentioned DDS update rate. For a M4i.6622-x8 (6.4 ns timer resolution) that means that the slope functionality smoothly changes amplitude and/or frequency every 6.4 ns. There are no jumps in the slope functionality.

Frequency Slope

The frequency slope settings are done in Hertz per second, meaning how much change of frequency should be done within one second. The figure can be entered as positive (increasing frequency value) or negative (decreasing frequency value) value. The driver re-calculates the given frequency slope to the frequency slope best matching by taking the output rate, ramp stepsize and the restrictions of the DDS-core internal settings into account. It is advised to read the register after writing to determine which exact frequency slope has been calculated.

Table 155: Spectrum API: DDS core frequency slope settings

Register	Value	Direction	Description
SPC_DDS_CORE0_FREQ_SLOPE	604000I	read/write	Set up the frequency slope of DDS core 0 in Hertz per second
SPC_DDS_CORE1_FREQ_SLOPE	604001I DDS core 1
...	...		

The available programmable frequency slope minimum, maximum and stepsize can be read from the library:

Table 156: Spectrum API: DDS programmable frequency slope range

Register	Value	Direction	Description
SPC_DDS_AVAIL_FREQ_SLOPE_MIN	608503I	read	Minimum programmable frequency slope in Hertz per second
SPC_DDS_AVAIL_FREQ_SLOPE_MAX	608504I	read	Maximum programmable frequency slope in Hertz per second
SPC_DDS_AVAIL_FREQ_SLOPE_STEP	608505I	read	Stepsize of programmable frequency slope in Hertz per second

Example for Frequency Slope

The following example should generate a 110 MHz signal for 100 ms, do a ramp from 110 MHz to 120 MHz within the next 100 ms and then keep the 120 MHz until stopped.

```
// ... card initialisation and output amplifier settings

// set 100 ms timer and execute steps on timer
spcm_dwSetParam_i32 (hCard, SPC_DDS_TRG_SRC, SPCM_DDS_TRG_SRC_TIMER);
spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_TIMER, 0.1);

// Initial 110 MHz frequency
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP, 1);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_PHASE, 0);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ, MEGA(110));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);

// slope from 110 MHz to 120 MHz (10 MHz change in 100 ms = 100 MHz change in 1 second)
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, MEGA(100));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);

// Final 120 MHz frequency
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ, MEGA(120));
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, 0);
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);

spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_WRITE_TO_CARD);
```

Amplitude Slope

The amplitude slope settings are done in level of output amplitude per second, meaning how much change of amplitude should be done within one second. The figure can be entered as positive (increasing amplitude) or negative (decreasing amplitude) value. The driver recalculates the given amplitude slope to the amplitude slope best matching by taking the output rate, ramp stepsize and the restrictions of the DDS-core internal settings into account. It is advised to read the register after writing to determine which exact amplitude slope has been calculated. The amplitude slope is only available for dedicated DDS cores.

Table 157: Spectrum API: DDS core amplitude slope settings

Register	Value	Direction	Description
SPC_DDS_CORE0_AMP_SLOPE	606000I	read/write	Set up the amplitude slope of DDS core 0 in level of amplitude per second
SPC_DDS_CORE1_AMP_SLOPE	606001I DDS core 1
...

The available programmable amplitude slope minimum, maximum and stepsize can be read from the library:

Table 158: Spectrum API: DDS programmable amplitude slope range

Register	Value	Direction	Description
SPC_DDS_AVAIL_AMP_SLOPE_MIN	608509I	read	Minimum programmable amplitude slope in level of amplitude per second
SPC_DDS_AVAIL_AMP_SLOPE_MAX	608510I	read	Maximum programmable amplitude slope in level of amplitude per second
SPC_DDS_AVAIL_AMP_SLOPE_STEP	608511I	read	Stepsize of programmable amplitude slope in level of amplitude per second

Example for Amplitude Slope

The following example generates a 120 MHz signal which is ramped up for 100 ms, run at full amplitude for 500 ms and ramped down again within 100 ms.

```
// ... card initialisation and output amplifier settings

// execute steps on timer
spcm_dwSetParam_i32 (hCard, SPC_DDS_TRG_SRC, SPCM_DDS_TRG_SRC_TIMER);

// slope from zero to full amplitude in 100 ms (+1.0 change in 100 ms = +10.0 change in 1 second)
spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_TIMER, 0.1);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP, 0);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_PHASE, 0);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ, MEGA(120));
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP_SLOPE, +10.0);
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);

// keep amplitude for 500 ms
spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_TIMER, 0.5);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP, 1.0);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP_SLOPE, 0.0);
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);

// slope from full amplitude to zero in 100 ms (-1.0 change in 100 ms = -10.0 change in 1 second)
spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_TIMER, 0.1);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP_SLOPE, -10.0);
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);

spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_WRITE_TO_CARD);
```

Modifying stepsize of slope

For very slow slopes of frequency or amplitude, the resolution of the internal firmware registers may not be large enough. For these cases, there are two additional global stepsize registers which allow to work with even smaller slopes:

Table 159: Spectrum API: DDS core slope stepsize settings

Register	Value	Direction	Description
SPC_DDS_FREQ_RAMP_STEPSIZE	608008I	read/write	Valid range: 1 to 4096: The stepsize setting increases the number of clock cycles between two changes and thus allows smaller frequency changes during ramps
SPC_DDS_AMP_RAMP_STEPSIZE	608009I	read/write	Valid range: 1 to 4096: The stepsize setting increases the number of clock cycles between two changes and thus allows smaller amplitude changes during ramps

Defining the Phase Behaviour

The phase behaviour defines how the sine phase behaves with the next trigger event. There are two different modes available that define the behaviour for all cores:

Table 160: Spectrum API: DDS Phase Behaviour

Register	Value	Direction	Description
SPC_DDS_PHASE_BEHAVIOUR	608002I	read/write	Setup of the phase behaviour. One of the below settings is available

SPCM_DDS_PHASE_JUMP	0	phase jumps to the next phase value at the next trigger
SPCM_DDS_PHASE_SHIFT	1	phase value is added to the phase counter output

Mode Phase Jump

The phase jumps to the defined phase value at the next trigger or timer. The phase counter is reset and the prior phase is irrelevant. This can be used for phase coherent frequency multiplexing.

The example on the right shows a phase jump to 90° phase when the 2nd trigger signal arrives. Below are the commands that are written to the library to receive this behaviour.

Command List

```
// 1. Sequence
SPC_DDS_PHASE_BEHAVIOUR: SPCM_DDS_PHASE_JUMP
SPC_DDS_CORE0_PHASE: 0°
SPC_DDS_CORE0_AMP: 1.0
SPC_DDS_CORE0_FREQ: 1 Hz
SPC_DDS_CMD: SPCM_DDS_CMD_EXEC_AT_TRG

// 2. Sequence
SPC_DDS_CORE0_PHASE: 90°
SPC_DDS_CMD: SPCM_DDS_CMD_EXEC_AT_TRG
```

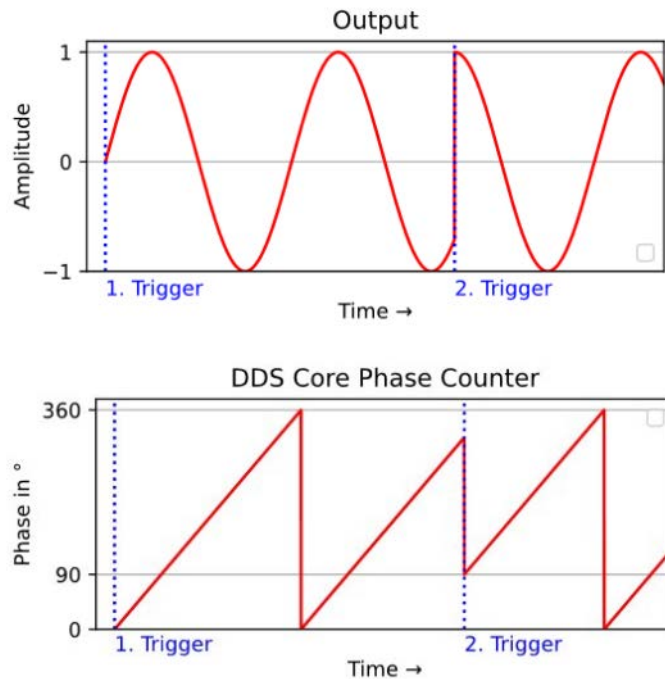


Image 77: Phase Jumps to 90° instantly when the 2nd trigger arrives, regardless of prior state.

Mode Phase Shift

The phase shifts to a new phase by adding the phase value to the current phase counter at the moment of the trigger event. Advantage here is that the phase relations are guaranteed even if the trigger is not 100% time synchronized. This mode can be used for phase modulation and phase shifts between hardware channels during runtime.

The example in the right shows a phase shift. The signal phase of the first signal part is 180°. At the 2nd trigger the signal shifts to a signal phase of 270°. This phase is in relation to the original phase, thus the shift of phase is $270^\circ - 180^\circ = 90^\circ$. At the position of the 2nd trigger the signal therefore moves from 270° phase to $270^\circ + 90^\circ = 0^\circ$ phase. The dotted blue signal shows the phase counter of the second signal part how it would have been if started with that phase immediately at the beginning of the output.

Command List

```
// 1. Sequence
SPC_DDS_PHASE_BEHAVIOUR: SPCM_DDS_PHASE_SHIFT
SPC_DDS_CORE0_PHASE: 180°
SPC_DDS_CORE0_AMP: 1.0
SPC_DDS_CORE0_FREQ: 1 Hz
SPC_DDS_CMD: SPCM_DDS_CMD_EXEC_AT_TRG

// 2. Sequence
SPC_DDS_CORE0_PHASE: 270°
SPC_DDS_CMD: SPCM_DDS_CMD_EXEC_AT_TRG
```

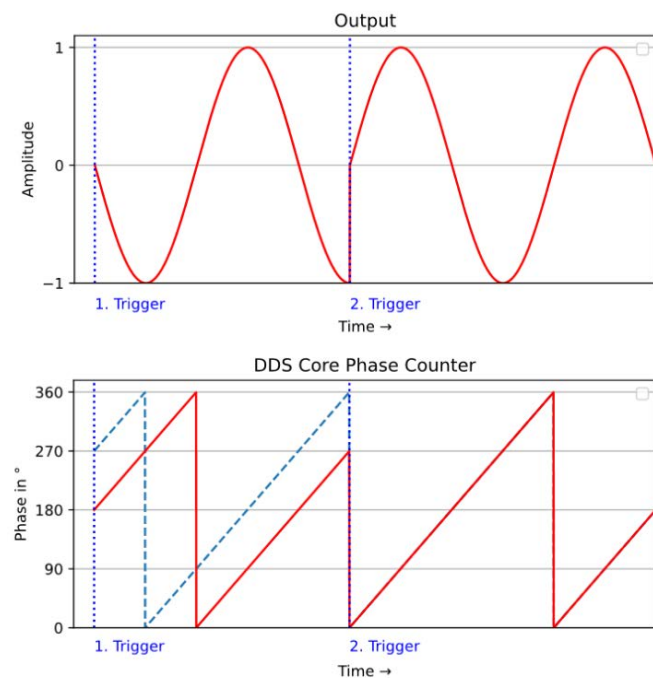


Image 78: Phase shifts forward by 90° relative to current state at 2nd Trigger

Additional functions in DDS core

Using the multi-purpose I/O lines within the DDS core command structure allows to manipulate external equipment within the exact timing of the DDS functionality. Please note that the general SPCM_Xx_MODE registers need to be set to SPCM_MODE_DDS first to be under control of the DDS functionality.

Table 161: Spectrum API: multi-purpose I/O lines registers and available register settings

Register	Value	Direction	Description
SPCM_X0_AVAILMODES	47210	read	Bit mask with all bits of the below mentioned modes showing the available modes for (X0)
SPCM_X1_AVAILMODES	47211	read	Bit mask with all bits of the below mentioned modes showing the available modes for (X1)
SPCM_X2_AVAILMODES	47212	read	Bit mask 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_DDS	0000004h		Switch the output to DDS mode

Once switched to DDS mode, you can use an additional DDS-MODE register, specialized to DDS functionality:

Table 162: Spectrum API: DDS multi-purpose I/O additional registers

Register	Value	Direction	Description
SPC_DDS_X0_MODE	608011	read/write	After setting SPCM_X0_MODE to SPCM_XMODE_DDS this register defines the DDS-specific usage of the multi purpose line X0.
SPC_DDS_X1_MODE	608012	read/write	After setting SPCM_X1_MODE to SPCM_XMODE_DDS this register defines the DDS-specific usage of the multi purpose line X1.
SPC_DDS_X2_MODE	608013	read/write	After setting SPCM_X2_MODE to SPCM_XMODE_DDS this register defines the DDS-specific usage of the multi purpose line X2.
SPCM_DDS_XMODE_MANUAL	1		Set the XIO-line to High or low with a software command SPC_DDS_X_MANUAL_OUTPUT. This command is part of the DDS command queue and executed following the DDS trigger and timer settings in parallel to changes of the DDS core.
SPCM_DDS_XMODE_WAITING_FOR_TRG	2		Generates a high signal on the multi-purpose output as soon as the DDS engine is waiting for the next trigger event (DDS engine is armed for trigger). This mode can be used to interact with external hardware and issue changes and the next trigger event.
SPCM_DDS_XMODE_EXEC	3		Generates a high pulse on the multi-purpose output with the length of DDS timer resolution when a trigger or timer event occurs. This output shows the loading of a new set of DDS parameters from shadow registers to DDS registers.

The additional register below defines the manual output if the DDS_XMODE of that specific output is set to manual:

Table 163: Spectrum API: DDS multi-purpose I/O manual output register

Register	Value	Direction	Description
SPC_DDS_X_MANUAL_OUTPUT	608014	write	If SPCM_DDS_XMODE_MANUAL is used this register sets the XIO line to high if the corresponding bit is set.
SPCM_DDS_X0	1h		Sets output X0 to high
SPCM_DDS_X1	2h		Sets output X1 to high
SPCM_DDS_X2	4h		Sets output X2 to high

Example for multi-purpose DDS output

The following example should generate constantly a 110 MHz signal. After 100 ms, it switches X1 to high for another 200 ms and switches back to 0.

```
// ... card initialisation and X-mode settings

// set XIO 0 to DDS, manual output mode and execute steps on timer
spcm_dwSetParam_i32 (hCard, SPC_DDS_TRG_SRC, SPCM_DDS_TRG_SRC_TIMER);
spcm_dwSetParam_i32 (hCard, SPCM_X0_MODE, SPCM_XMODE_DDS);
spcm_dwSetParam_i32 (hCard, SPC_DDS_X0_MODE, SPCM_DDS_XMODE_MANUAL);

// Initial 110 MHz frequency for 100 ms
spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_TIMER, 0.1);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP, 1);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ, MEGA(110));
spcm_dwSetParam_i32 (hCard, SPC_DDS_X_MANUAL, 0);
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);

// set X0 to high for 200 ms
spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_TIMER, 0.2);
spcm_dwSetParam_i32 (hCard, SPC_DDS_X_MANUAL, SPCM_DDS_X1);
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);

// and back to zero
spcm_dwSetParam_i32 (hCard, SPC_DDS_X_MANUAL, 0);
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);

spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_WRITE_TO_CARD);
```

Benefits of using the parameter array function

All above examples have show how to program parameters with single commands. This is an easy to understand approach which can be extended at any time by new commands. However, this approach executes a library call for each single command. Each library call has some internal overhead for the call itself, for handle routing, for error checking and for miscellaneous others steps that are done. That overhead adds up to some latency for every call, limiting the speed one can send commands to the library. This is especially limiting when a lot of commands should be send in a short time with the intention to get a fast update rate. To enhance the command speed, one can also send a number of commands as a parameter array with a single library call using the `dwSetParam_ptr` function (introduced with library V7.x). Below are two examples showing the two different approaches:

Single Commands Example (6 library calls)

```
spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_TIMER, 0.1);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP, 0);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_PHASE, 0);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ, MEGA(120));
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP_SLOPE, +10.0);
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD, SPCM_DDS_CMD_EXEC_AT_TRG);
```

Parameter Array Example (1 library call)

```
// setting up the list
ST_LIST_PARAM astList[6];
astList[0].lReg = SPC_DDS_TRG_TIMER; astList[0].llType= TYPE_DOUBLE; astList[0].dValue= 0.1;
astList[1].lReg = SPC_DDS_CORE0_AMP; astList[1].llType= TYPE_DOUBLE; astList[1].dValue= 0.0;
astList[2].lReg = SPC_DDS_CORE0_PHASE; astList[2].llType= TYPE_DOUBLE; astList[2].dValue= 0.0;
astList[3].lReg = SPC_DDS_CORE0_FREQ; astList[3].llType= TYPE_DOUBLE; astList[3].dValue= MEGA(120);
astList[4].lReg = SPC_DDS_CORE0_AMP_SLOPE; astList[4].llType= TYPE_DOUBLE; astList[4].dValue= +10.0;
astList[5].lReg = SPC_DDS_CMD; astList[5].llType= TYPE_INT64; astList[5].dValue= SPCM_DDS_CMD_EXEC_AT_TRG;

// and transferring the list with one single library call
spcm_dwSetParam_ptr (hCard, SPC_REGISTER_LIST, astList, 6 * sizeof(ST_LIST_PARAM));
```

Transfer Mode Specification

There are two different modes to transfer data from software buffer to hardware buffer. Both modes are completely handled by the API and don't need any interaction with the application.

Table 164: Spectrum API: data transfer mode definition

Register	Value	Direction	Description
SPC_DDS_DATA_TRANSFER_MODE	608012I	read/write	defining the data transfer mode that is performed when executing the SPCM_DDS_CMD_WRITE_TO_CARD command
SPCM_DDS_DTM_SINGLE	0		commands are transferred by single calls from library buffer to card
SPCM_DDS_DTM_DMA	1		commands are transferred by DMA from library buffer to card

As mentioned before, all transfers start as soon as a `SPCM_DDS_CMD_WRITE_TO_CARD` command is issued. If `DTM_SINGLE` mode is selected, the API is transferring single commands from software buffer to hardware buffer. If `DTM_DMA` is selected, data is transferred using DMA.

The Single Data Transfer Mode features the lowest latency but has a low continues Command Rate and a limited Hardware DDS command buffer of 4k Entries. DMA Mode has a higher latency for single commands but much higher speed and is practically only limited by user software speed. In a standard setup, a streaming speed of 10 million commands/second has been reached without special optimizations using C++. As it utilizes the whole on-board memory it can store billions of commands. Below is a comparison of the two different modes in terms of advantages and disadvantages:

Table 165: Comparison of single and DMA transfer mode

	DTM_SINGLE	DTM_DMA
Min. user software to analog output latency	10 us	20 us
Max continuous command rate	400 kHz	10 MHz
On-board command buffer	4k commands	2G commands
CPU load	High	Low

Combining the parameter array function with the DMA transfer mode allows to transfer DDS commands with a very high speed, making advanced waveforms available. Using this mode it is easy to perform different kinds of modulation or different slope shapes like s-form shape. Please see the examples below.

Example of s-shape frequency slope

The built-in slope function is a linear ramp and is performed by a single command. Other slopes can be performed using multiple linear ramps. Below is an example of a s-shape slope using 7 single linear ramps:

```
// basic setup of starting frequency 100 MHz, full amplitude with a timing of 100 ms
spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_SOURCE,          SPCM_TG_SRC_TIMER);
spcm_dwSetParam_d64 (hCard, SPC_DDS_TRG_TIMER,           0.1);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_AMP,           1);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_PHASE,        0);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ,         MEGA(100));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_EXEC_AT_TRG);

// s-form slope in 7 steps (700 ms)
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, +MEGA(10));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_EXEC_AT_TRG);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, +MEGA(20));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_EXEC_AT_TRG);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, +MEGA(40));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_EXEC_AT_TRG);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, +MEGA(80));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_EXEC_AT_TRG);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, +MEGA(40));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_EXEC_AT_TRG);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, +MEGA(20));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_EXEC_AT_TRG);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, +MEGA(10));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_EXEC_AT_TRG);

// final frequency 122 MHz
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ_SLOPE, 0);
spcm_dwSetParam_d64 (hCard, SPC_DDS_CORE0_FREQ,         MEGA(122));
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_EXEC_AT_TRG);

// writing the whole setup to the card
spcm_dwSetParam_i32 (hCard, SPC_DDS_CMD,                 SPCM_DDS_CMD_WRITE_TO_CARD);
```

Execute Now, Timer and Trigger timing behaviour

This chapter explains the timing behaviour of the three different command to change the output. It is possible to mix Execute Now (SPCM_DDS_CMD_EXEC_NOW) with trigger (SPCM_DDS_CMD_EXEC_AT_TRG) and also with timer but depending on the combination there are some additional time delays:

- If an SPCM_DDS_CMD_EXEC_NOW command is queued directly after a SPCM_DDS_CMD_EXEC_AT_TRG, there is a time delay of roughly 51 ns (8 times the time resolution of 6.4 ns) between the trigger event and the execute of the command(s)
- If SPCM_DDS_CMD_EXEC_NOW commands are queued directly after each other, the delay between the command execution is 1 time the time resolution of 6.4 ns.
- If the SPCM_DDS_CMD_EXEC_NOW command is sent from software together with a SPCM_DDS_CMD_WRITE_TO_CARD command with the command FIFO being empty, there is a highly operating system depending delay until the output is changed. The delay is in the region of a few micro seconds up to milli seconds.
- Using a timer together with a SPCM_DDS_CMD_EXEC_NOW resets the timer to zero at the execution time of the command.

Pulse Generator (Firmware Option)

General Information

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

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

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

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

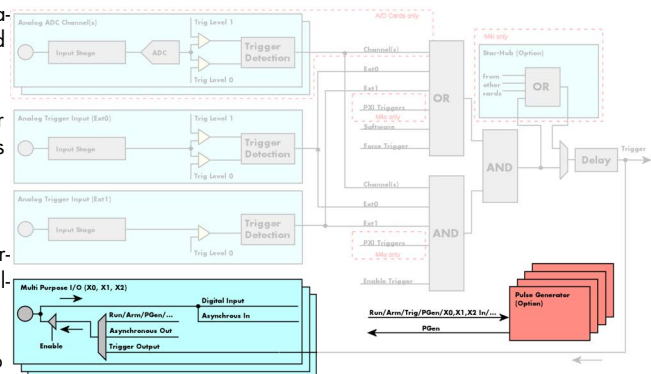


Image 79: overview block diagram of multi-purpose I/O lines and pulse generators

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

Feature Overview

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

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

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



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



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

Principle of Operation

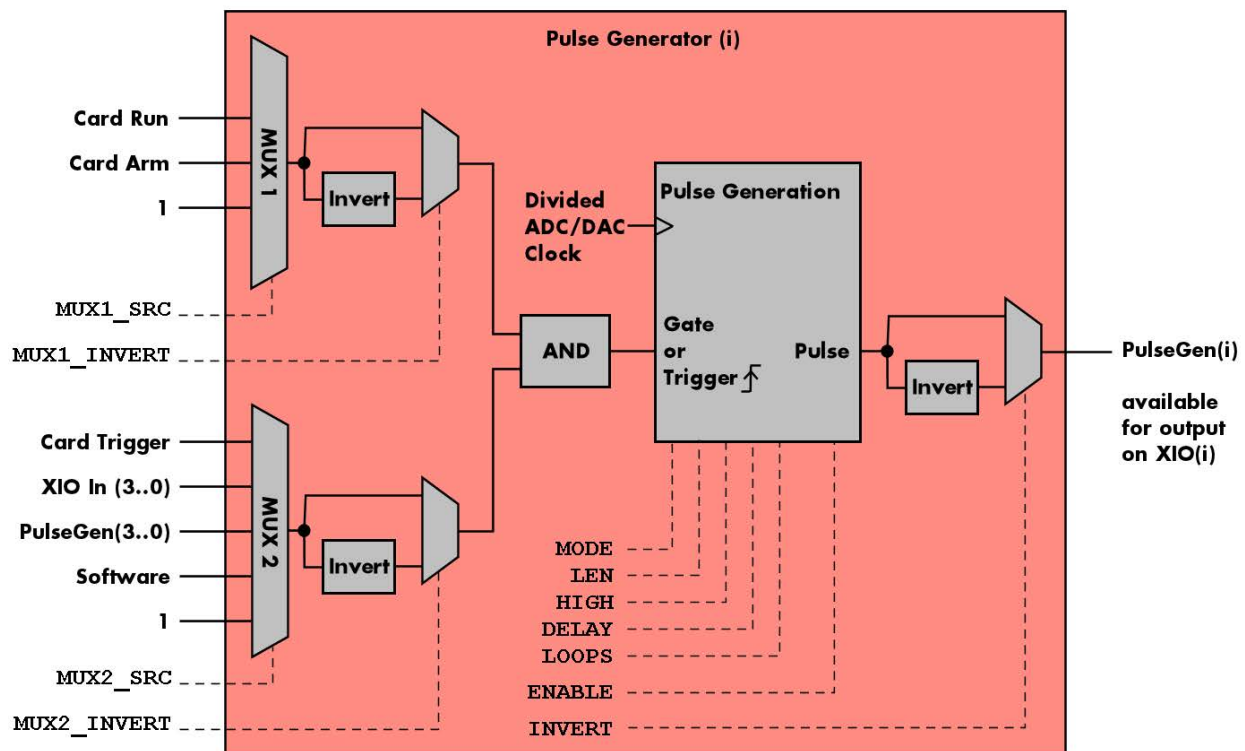


Image 80: overview block diagram of the pulse generator

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

As shown above, each unit consists of:

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

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

Table 166: Spectrum API: pulse generator clock frequency read register

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

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

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

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



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

Setting up the Pulse Generator

Enabling, disabling and resetting a pulse generator

Each pulse generator unit can be enabled and disabled separately:

Table 167: Spectrum API: pulse generator enable registers

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

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

Defining the basic pulse parameters

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

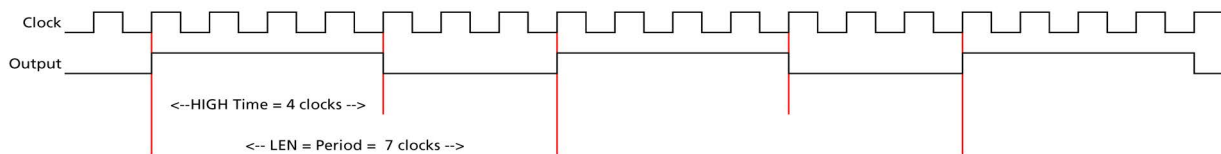


Image 81: timing diagram illustrating the basic pulse parameters

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

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

Table 168: Spectrum API: pulse generator length/period register

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

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

Table 169: Spectrum API: pulse generator HIGH time registers

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

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

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

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

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

Table 170: Spectrum API: pulse generator loops/pulse repetition registers

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

Delaying (phase shifting) the Outputs

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

When using the delay, it is possible to delay the initial HIGH portion of the pulse generator(s) by a defined amount of clock cycles. This in combination with a common starting point (start/trigger) allows for the generation of phase shifted signals as shown below for two of the pulse generators. Both are set up with identical LEN and HIGH parameters, but the additional delay for pulse generator 0 (PGen0) is kept at the default of zero clock cycles, whilst PGen1 is delayed by 5 clock cycles:

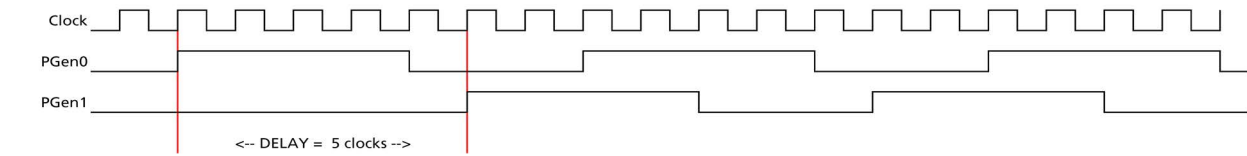


Image 82: timing diagram illustrating delaying a pulse generator output

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

Table 171: Spectrum API: pulse generator delay/phase shift registers

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

Defining the trigger behavior

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

Table 172: Spectrum API: pulse generator mode registers with their available settings

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

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

Continuously triggered output

After enabling the pulse generator, it will detect trigger events. Upon each trigger, the programmed number of pulses are generated, as defined by the LEN, HIGH, DELAY and LOOPS parameters explained above. After finishing the programmed number of triggers, it will automatically arm itself again and wait for the next trigger.

In contrast to the Gated mode (see below), once a trigger has been detected the trigger input is ignored and the pulse train will finish independent from any activity on the trigger input. Only when it has finished the current generation, a new trigger will be detected:

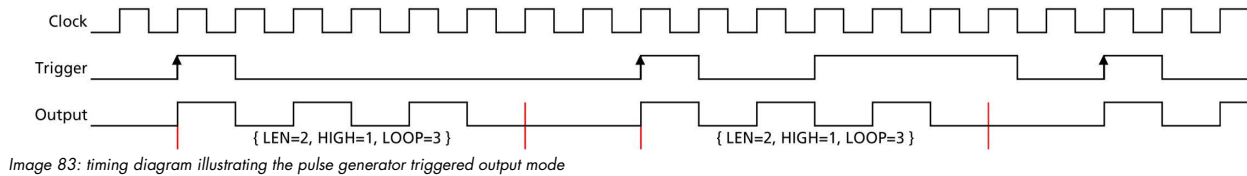


Image 83: timing diagram illustrating the pulse generator triggered output mode

Single Shot triggering

This mode is similar to the triggered mode, but after enabling the pulse generator it will only detect one single trigger. Upon that trigger, the programmed number of pulses are generated, as defined by the LEN, HIGH, DELAY and LOOPS parameters explained above:

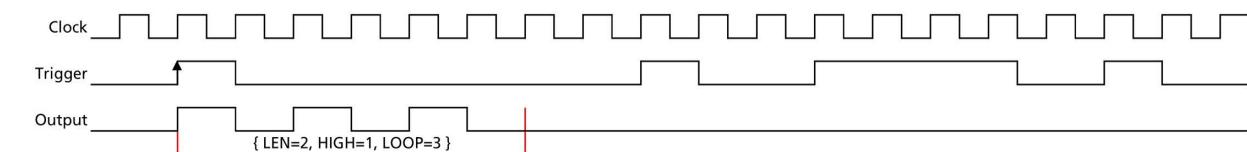


Image 84: timing diagram illustrating the pulse generator single-shot triggered output mode

Afterwards the pulse generator will not detect any further triggers, until being reset by re-enabling:

Continuously gated Output

After enabling the pulse generator, it will detect trigger events. Upon each trigger, the programmed number of pulses are generated, as defined by the LEN, HIGH, DELAY and LOOPS parameters explained above and as long as the trigger condition or gate is still valid (HIGH). If the gate ends, this will stop the output and reset all internal counters back to start. So, each time the gate turns HIGH, the sequence (number of pulses as defined by the LEN, HIGH, DELAY and LOOPS) starts again from its beginning:

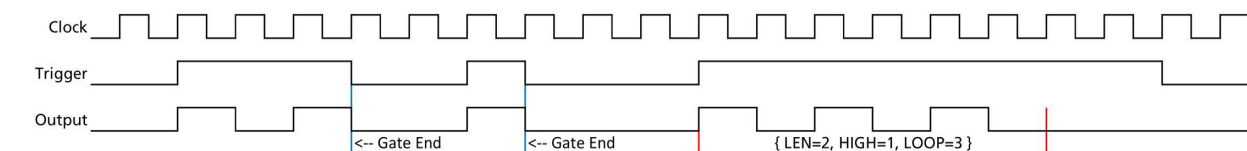


Image 85: timing diagram illustrating the pulse generator gated output mode

Configuring the pulse generator's trigger source

The various possible signals that can logically be combined to form a trigger event for a pulse generator are split up into two portions each consisting of a multiplexer (MUX).

Multiplexer 1

The first multiplexer, MUX1, selects between two different sources and also allows to be completely unused by utilizing a logical '1' or HIGH level, being transparent to the following AND condition combining the two multiplexers:

Table 173: Spectrum API: pulse generator trigger MUX1 registers with their available settings

Register	Value	Direction	Description
SPC_XIO_PULSEGEN0_MUX1_SRC	601005	read/write	Selects the input source for MUX1 for pulse generator 0.
SPC_XIO_PULSEGEN1_MUX1_SRC	601105	read/write	Selects the input source for MUX1 for pulse generator 1.
SPC_XIO_PULSEGEN2_MUX1_SRC	601205	read/write	Selects the input source for MUX1 for pulse generator 2.
SPC_XIO_PULSEGEN3_MUX1_SRC	601305	read/write	Selects the input source for MUX1 for pulse generator 3.
SPCM_PULSEGEN_MUX1_SRC_UNUSED	0		Inputs of MUX1 are not used in creating the trigger condition and instead a static logic HIGH is used for MUX1.
SPCM_PULSEGEN_MUX1_SRC_RUN	1		This input of MUX1 reflects the current run state of the card. If acquisition/output is running the signal is HIGH. If card has stopped the signal is LOW. The signal is identical to XIO output using SPCM_XMODE_RUNSTATE.
SPCM_PULSEGEN_MUX1_SRC_ARM	2		This input of MUX1 reflects the current ARM state of the card. If the card is armed and ready to receive a trigger the signal is HIGH. If the card isn't running or the card is still acquiring pretrigger data or the trigger has already been detected, the signal is LOW. The signal is identical to XIO output using SPCM_XMODE_ARMSTATE.

By having the two status lines ARM and RUN available as input, it is either possible to generate pulses depending only on the card's RUN or ARM state (e.g., currently running or currently not running enabling the inverter of MUX1 output) or to mask other trigger conditions from MUX2 to only be passed upon the card's acquisition/replay RUN or ARM state.

Multiplexer 2

The second multiplexer can be transparent and hence unused or allows to select various sources for starting the pulse creation:

- Allowing a start command issued by the application software by issuing a force trigger command
- Any one of the other pulse generator unit outputs to create pulses or pulse trains with up to four repetition time scales
- The card's acquisition or replay trigger output
- An external logic signal coming in from any of the multi-purpose XIO input capable lines

Table 174: Spectrum API: pulse generator trigger MUX2 registers with their available settings

Register	Value	Direction	Description
SPC_XIO_PULSEGEN0_MUX2_SRC	601006	read/write	Selects the input source for MUX2 for pulse generator 0.
SPC_XIO_PULSEGEN1_MUX2_SRC	601106	read/write	Selects the input source for MUX2 for pulse generator 1.
SPC_XIO_PULSEGEN2_MUX2_SRC	601206	read/write	Selects the input source for MUX2 for pulse generator 2.
SPC_XIO_PULSEGEN3_MUX2_SRC	601306	read/write	Selects the input source for MUX2 for pulse generator 3.
SPCM_PULSEGEN_MUX2_SRC_UNUSED	0		No input of MUX2 is used in creating the trigger condition for the pulse generator. A static logic HIGH is used, so that the MUX output is transparent for the following AND gate.
SPCM_PULSEGEN_MUX2_SRC_SOFTWARE	1		This input reflects the positive edge generated by issuing the SPCM_PULSEGEN_CMD_FORCE command.
SPCM_PULSEGEN_MUX2_SRC_CARDTRIGGER	2		This input of MUX2 reflects the trigger detection of the acquisition/replay. The trigger output goes HIGH as soon as the card's main trigger is recognized. After end of acquisition/replay it is LOW again. In Multiple Recording/Gated Sampling/ABA mode it goes LOW after the acquisition of the current segment stops. In FIFO single mode the trigger output is HIGH until FIFO mode is stopped. The signal is identical to what a XIO output is providing when using SPCM_XMODE_TRIGOUT.
SPCM_PULSEGEN_MUX2_SRC_PULSEGEN0	3		Input to MUX2 is set to output of pulse generator 0/1/2 or 3.
SPCM_PULSEGEN_MUX2_SRC_PULSEGEN1	4		This can be used to cascade pulse generators for creating up to four pulse repetition time scales. Each pulse generator can select to be triggered by any of the other pulse generator's output.
SPCM_PULSEGEN_MUX2_SRC_PULSEGEN2	5		Selecting its own pulse generator's output as a trigger (loopback) is not allowed and will lead to a driver error.
SPCM_PULSEGEN_MUX2_SRC_PULSEGEN3	6		
SPCM_PULSEGEN_MUX2_SRC_XIO0	7		Input to MUX2 is set to the input signal coming in from multi-purpose line of X0. M2p: Since X0 is an output only, it therefore is not allowed to be used as an input.
SPCM_PULSEGEN_MUX2_SRC_XIO1	8		Input to MUX2 is set to the input signal coming in from multi-purpose line of X1.
SPCM_PULSEGEN_MUX2_SRC_XIO2	9		Input to MUX2 is set to the input signal coming in from multi-purpose line of X2.
SPCM_PULSEGEN_MUX2_SRC_XIO3	10		Input to MUX2 is set to the input signal coming in from multi-purpose line of X3. M4i/M4x: Since X3 is not available, it therefore is not allowed to be used as an input.

The output of the following command register is connected to all pulse generator units in parallel in a synchronous fashion:

Table 175: Spectrum API: pulse generator command register for trigger forcing by software

Register	Value	Direction	Description
SPC_XIO_PULSEGEN_COMMAND	601501	write only	Executes a command for the pulse generator option.
SPCM_PULSEGEN_CMD_FORCE	1h		Generate a single rising edge, that is common for all pulse generator engines. This allows to start/trigger the output of all enabled pulse generators synchronously by issuing a software command.

This allows to start any number of pulse generators set to MUX2_SRC_SOFTWARE to be started at the same instant even from software, useful when requiring pulses with a known and static phase relation.

Additional trigger configuration (changing the active edge or level)



Please note that the Trigger/Gate input to the "Pulse Generation" portion is always HIGH-active. Depending on the selected pulse generator configuration it is triggering on the rising edge or the logic HIGH state. The two programmable inverters at the multiplexer outputs can be used to trigger on the falling edge or a logical LOW instead.

To access the three programmable inverters and to optionally change whether triggering on a rising edge (the trigger signal changing its state from LOW to HIGH) or on the valid level (the trigger being logically HIGH), following registers can be used:

Table 176: Spectrum API: pulse generator additional configuration registers with the available settings

Register	Value	Direction	Description
SPC_XIO_PULSEGEN0_CONFIG	601007	read/write	Bitmask with additional configuration for pulse generator 0.
SPC_XIO_PULSEGEN1_CONFIG	601107	read/write	Bitmask with additional configuration for pulse generator 1.
SPC_XIO_PULSEGEN2_CONFIG	601207	read/write	Bitmask with additional configuration for pulse generator 2.
SPC_XIO_PULSEGEN3_CONFIG	601307	read/write	Bitmask with additional configuration for pulse generator 3.
SPCM_PULSEGEN_CONFIG_MUX1_INVERT	1h		When bit is set, the output of MUX1 is logically inverted.
SPCM_PULSEGEN_CONFIG_MUX2_INVERT	2h		When bit is set, the output of MUX2 is logically inverted.
SPCM_PULSEGEN_CONFIG_INVERT	4h		When bit is set, the output of the pulse generator is logically inverted.
SPCM_PULSEGEN_CONFIG_HIGH	8h		As default the pulse generator's trigger input is sensitive only to a rising edge. When using this configuration, the input will not look for an active edge, but rather detect a HIGH level. This is similar to the distinction of the card's main trigger modes, when choosing between SPC_TM_POS and SPC_TM_HIGH.

Since the register is implemented as a bitmask, any combination of the above configuration flags is possible.

```
// enable the inverters on MUX1 and MUX2 outputs for pulse generator 2
int32 lPulseGenConfig = (SPCM_PULSEGEN_CONFIG_MUX1_INVERT | SPCM_PULSEGEN_CONFIG_MUX2_INVERT);

spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN2_CONFIG, lPulseGenConfig);
```

Configuring Multi Purpose lines to output generated pulses

Each of the up to four on-board multi purpose I/O lines can be programmed to output the pulses generated by its corresponding pulse generator unit, making it available for any external devices.

Please check the available modes by reading the SPCM_X0_AVAILMODES, SPCM_X1_AVAILMODES, SPCM_X2_AVAILMODES and SPCM_X3_AVAILMODES register first. The available modes may differ from card to card and may be enhanced with new driver/firmware versions to come.

Table 177: Spectrum API: XIO lines and mode software registers with their reduced to the settings required for outputting pulses

Register	Value	Direction	Description
SPCM_X0_AVAILMODES	600300	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X0)
SPCM_X1_AVAILMODES	600301	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X1)
SPCM_X2_AVAILMODES	600302	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X2)
SPCM_X3_AVAILMODES	600303	read	Bitmask with all bits of the below mentioned modes showing the available modes for (X3)
SPCM_X0_MODE	600200	read/write	Defines the mode for (X0). Only one mode selection is possible to be set at a time
SPCM_X1_MODE	600201	read/write	Defines the mode for (X1). Only one mode selection is possible to be set at a time
SPCM_X2_MODE	600202	read/write	Defines the mode for (X2). Only one mode selection is possible to be set at a time
SPCM_X3_MODE	600203	read/write	Defines the mode for (X3). Only one mode selection is possible to be set at a time
SPCM_XMODE_DISABLE	00000000h		No mode selected. Output is tristate (default setup)
...	...		For all other modes please see chapter "Multi Purpose I/O Lines".
SPCM_XMODE_PULSEGEN	00080000h		A/D and D/A cards only (optional): Connector reflects the output of the same index pulse generator (X1 can output pulses from pulse generator 1, X2 can output pulses from pulse generator 2, ... etc.). On M4i/M4x cards with three XIO lines (X0, X1, X2) and four pulse generators, pulses from pulse generator 3 cannot be output, but can still be used in cascading configurations to trigger another pulse generator.



Please note that a change to the SPCM_X0_MODE, SPCM_X1_MODE, SPCM_X2_MODE or SPCM_X3_MODE will only be updated with the next call to either the M2CMD_CARD_START or M2CMD_CARD_WRITESETUP register. For further details please see the relating chapter on the M2CMD_CARD registers.

Programming Example

The following example shows in principle, the steps required for generating a single, repetitive pulse with one of the pulse generators and how to output that pulse on the matching multi-purpose I/O line:

```
// First we set up the channel selection and the clock.
// For this example we enable only one channel to be able to use max sampling rate on all card types.
spcm_dwSetParam_i32 (hCard, SPC_CHENABLE, CHANNEL0);

// Read out the max. supported sampling rate ...
int64 llMaxSR = 0;
spcm_dwGetParam_i64 (hCard, SPC_PCISAMPLERATE, &llMaxSR);

// ... and use this as the card's sampling rate
spcm_dwSetParam_i64 (hCard, SPC_SAMPLERATE, llMaxSR);

// Read out the clock, at which the pulse generator will run with the above set sampling rate.
int64 llPulseGenClock_Hz = 0;
spcm_dwGetParam_i64 (hCard, SPC_XIO_PULSEGEN_CLOCK, &llPulseGenClock_Hz);

// Configure X0 to output signal from corresponding pulse generator 0
spcm_dwSetParam_i32 (hCard, SPCM_X0_MODE, SPCM_XMODE_PULSEGEN);

// Setup pulse generator 0 (output on X0)
// to generate a continuous signal with 1 MHz and ~50% duty-cycle
int32 lLenFor1MHz = static_cast < int32 > (llPulseGenClock_Hz / MEGA(1));
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_MODE, SPCM_PULSEGEN_MODE_TRIGGERED);
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_LEN, lLenFor1MHz);

// An integer division by 2 will be truncated if lLenFor1MHz is an odd number,
// resulting in a slightly shorter HIGH than LOW time.
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_HIGH, lLenFor1MHz / 2);

// Set LOOPS to 0: repeat infinitely
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_LOOPS, 0);

// Configure pulse generator to be triggered/started by software force command
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_MUX1_SRC, SPCM_PULSEGEN_MUX1_SRC_UNUSED);
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN0_MUX2_SRC, SPCM_PULSEGEN_MUX2_SRC_SOFTWARE);

// Enable the selected pulse generator and hence arm its trigger detection
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN_ENABLE, SPCM_PULSEGEN_ENABLE0);

// Write the settings to the card:
// This will update the clock section to generate the programmed frequencies
// (SPC_SAMPLERATE) and also write the pulse generator settings to the card.
spcm_dwSetParam_i32 (hCard, SPC_M2CMD, M2CMD_CARD_WRITESETUP);

// Start all armed pulse generators (in this case just one) by a software command
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN_COMMAND, SPCM_PULSEGEN_CMD_FORCE);

// Wait until a key is pressed
printf ("\nPress a key to stop the pulse generator(s) ");
cGetch ();

// Stop all running pulse generators
spcm_dwSetParam_i32 (hCard, SPC_XIO_PULSEGEN_ENABLE, 0);
spcm_dwSetParam_i32 (hCard, SPC_M2CMD, M2CMD_CARD_WRITESETUP);
```



Spectrum provides a dedicated programming example for the pulse generator feature as part of the standard example package. This example is showing different and more complex configurations than shown above, e.g., cascading of multiple pulse generators for more complex pulse generation time scales.

Option Star-Hub (M3i and M4i only)

Star-Hub introduction

The purpose of the Star-Hub is to extend the number of channels available for acquisition or generation by interconnecting multiple cards and running them simultaneously.

The Star-Hub option allows to synchronize several cards of the same M3i/M4i series that are mounted within one host system (PC):

- For the M3i series there are the two different versions available: a small version with 4 connectors (option SH4) for synchronizing up to four cards and a big version with 8 connectors (option SH8) for synchronizing up to eight cards.
- For the M4i series there are the two different mechanical versions available, with 8 connectors for synchronizing up to eight cards.

The Star-Hub allows synchronizing cards of the same family only. It is not possible to synchronize cards of different families!



Both versions are implemented as a piggy-back module that is mounted to one of the cards. For details on how to install several cards including the one carrying the Star-Hub module, please refer to the section on hardware installation.

Either which of the two available Star-Hub options is used, there will be no phase delay between the sampling clocks of the synchronized cards and either no delay between the trigger events. The card holding the Star-Hub is automatically also the clock master. Any one of the synchronized cards can be part of the trigger generation.

Star-Hub trigger engine

The trigger bus between an M3i/M4i card and the Star-Hub option consists of several lines. Some of them send the trigger information from the card's trigger engine to the Star-Hub and some receives the resulting trigger from the Star-Hub. All trigger events from the different cards connected are combined with OR on the Star-Hub.

While the returned trigger is identical for all synchronized cards, the sent out trigger of every single card depends on their trigger settings.

Star-Hub clock engine

The card holding the Star-Hub is the clock master for the complete system. If you need to feed in an external clock to a synchronized system the clock has to be connected to the master card. Slave cards cannot generate a Star-Hub system clock. As shown in the drawing on the right the clock master can use either the programmable quartz 1 or the external clock input to be broadcast to all other cards.

All cards including the clock master itself receive the distributed clock with equal phase information. This makes sure that there is no phase delay between the cards.

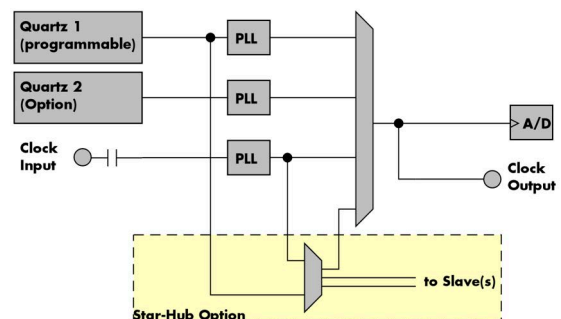


Table 178: star-hub clock overview diagram

Software Interface

The software interface is similar to the card software interface that is explained earlier in this manual. The same functions and some of the registers are used with the Star-Hub. The Star-Hub is accessed using its own handle which has some extra commands for synchronization setup. All card functions are programmed directly on card as before. There are only a few commands that need to be programmed directly to the Star-Hub for synchronization.

The software interface as well as the hardware supports multiple Star-Hubs in one system. Each set of cards connected by a Star-Hub then runs totally independent. It is also possible to mix cards that are connected with the Star-Hub with other cards that run independent in one system.

Star-Hub Initialization

The interconnection between the Star-Hubs is probed at driver load time and does not need to be programmed separately. Instead the cards can be accessed using a logical index. This card index is only based on the ordering of the cards in the system and is not influenced by the current cabling. It is even possible to change the cable connections between two system starts without changing the logical card order that is used for Star-Hub programming.

The Star-Hub initialization must be done AFTER initialization of all cards in the system. Otherwise the inter-connection won't be received properly.



The Star-Hubs are accessed using a special device name „sync“ followed by the index of the star-hub to access. The Star-Hub is handled completely like a physical card allowing all functions based on the handle like the card itself.

Example with 4 cards and one Star-Hub (no error checking to keep example simple)

```
drv_handle hSync;
drv_handle hCard[4];

for (i = 0; i < 4; i++)
{
    sprintf (s, "/dev/spcm%d", i);
    hCard[i] = spcm_hOpen (s);
}
hSync = spcm_hOpen ("sync0");

...

spcm_vClose (hSync);
for (i = 0; i < 4; i++)
    spcm_vClose (hCard[i]);
```

Example for a digitizerNETBOX or generatorNETBOX with two internal digitizer/generator modules, This example is also suitable for accessing a remote server with two cards installed:

```
drv_handle hSync;
drv_handle hCard[2];

for (i = 0; i < 2; i++)
{
    sprintf (s, "TCPIP::192.168.169.14::INST%d::INSTR", i);
    hCard[i] = spcm_hOpen (s);
}
hSync = spcm_hOpen ("sync0");

...

spcm_vClose (hSync);
for (i = 0; i < 2; i++)
    spcm_vClose (hCard[i]);
```

When opening the Star-Hub the cable interconnection is checked. The Star-Hub may return an error if it sees internal cabling problems or if the connection between Star-Hub and the card that holds the Star-Hub is broken. It can't identify broken connections between Star-Hub and other cards as it doesn't know that there has to be a connection.

The synchronization setup is done using bit masks where one bit stands for one recognized card. All cards that are connected with a Star-Hub are internally numbered beginning with 0. The number of connected cards as well as the connections of the star-hub can be read out after initialization. For each card that is connected to the star-hub one can read the index of that card:

Table 179: Spectrum API: star-hub related registers for reading detected connections

Register	Value	Direction	Description
SPC_SYNC_READ_NUMCONNECTORS	48991	read	Number of connectors that the Star-Hub offers at max. (available with driver V5.6 or newer)
SPC_SYNC_READ_SYNCCOUNT	48990	read	Number of cards that are connected to this Star-Hub
SPC_SYNC_READ_CARDIDX0	49000	read	Index of card that is connected to star-hub logical index 0 (mask 0x0001)
SPC_SYNC_READ_CARDIDX1	49001	read	Index of card that is connected to star-hub logical index 1 (mask 0x0002)
...		read	...
SPC_SYNC_READ_CARDIDX7	49007	read	Index of card that is connected to star-hub logical index 7 (mask 0x0080)
SPC_SYNC_READ_CARDIDX8	49008	read	M2i only: Index of card that is connected to star-hub logical index 8 (mask 0x0100)
...		read	...
SPC_SYNC_READ_CARDIDX15	49015	read	M2i only: Index of card that is connected to star-hub logical index 15 (mask 0x8000)
SPC_SYNC_READ_CABLECON0		read	Returns the index of the cable connection that is used for the logical connection 0. The cable connections can be seen printed on the PCB of the star-hub. Use these cable connection information in case that there are hardware failures with the star-hub cabling.
...	49100	read	...
SPC_SYNC_READ_CABLECON15	49115	read	Returns the index of the cable connection that is used for the logical connection 15.

In standard systems where all cards are connected to one star-hub reading the star-hub logical index will simply return the index of the card again. This results in bit 0 of star-hub mask being 1 when doing the setup for card 0, bit 1 in star-hub mask being 1 when setting up card 1

and so on. On such systems it is sufficient to read out the SPC_SYNC_READ_SYNCCOUNT register to check whether the star-hub has found the expected number of cards to be connected.

```
spcm_dwGetParam_i32 (hSync, SPC_SYNC_READ_SYNCCOUNT, &lSyncCount);
for (i = 0; i < lSyncCount; i++)
{
    spcm_dwGetParam_i32 (hSync, SPC_SYNC_READ_CARDIDX0 + i, &lCardIdx);
    printf ("star-hub logical index %d is connected with card %d\n", i, lCardIdx);
}
```

In case of 4 cards in one system and all are connected with the star-hub this program excerpt will return:

```
star-hub logical index 0 is connected with card 0
star-hub logical index 1 is connected with card 1
star-hub logical index 2 is connected with card 2
star-hub logical index 3 is connected with card 3
```

Let's see a more complex example with two Star-Hubs and one independent card in one system. Star-Hub A connects card 2, card 4 and card 5. Star-Hub B connects card 0 and card 3. Card 1 is running completely independent and is not synchronized at all:

card	Star-Hub connection	card handle	star-hub handle	card index in star-hub	mask for this card in star-hub
card 0	-	/dev/spcm0		0 (of star-hub B)	0x0001
card 1	-	/dev/spcm1			-
card 2	star-hub A	/dev/spcm2	sync0	0 (of star-hub A)	0x0001
card 3	star-hub B	/dev/spcm3	sync1	1 (of star-hub B)	0x0002
card 4	-	/dev/spcm4		1 (of star-hub A)	0x0002
card 5	-	/dev/spcm5		2 (of star-hub A)	0x0004

Now the program has to check both star-hubs:

```
for (j = 0; j < lStarhubCount; j++)
{
    spcm_dwGetParam_i32 (hSync[j], SPC_SYNC_READ_SYNCCOUNT, &lSyncCount);
    for (i = 0; i < lSyncCount; i++)
    {
        spcm_dwGetParam_i32 (hSync[j], SPC_SYNC_READ_CARDIDX0 + i, &lCardIdx);
        printf ("star-hub %c logical index %d is connected with card %d\n", (!j ? 'A' : 'B'), i, lCardIdx);
    }
    printf ("\n");
}
```

In case of the above mentioned cabling this program excerpt will return:

```
star-hub A logical index 0 is connected with card 2
star-hub A logical index 1 is connected with card 4
star-hub A logical index 2 is connected with card 5

star-hub B logical index 0 is connected with card 0
star-hub B logical index 1 is connected with card 3
```

For the following examples we will assume that 4 cards in one system are all connected to one star-hub to keep things easier.

Setup of Synchronization

The synchronization setup only requires one additional register to enable the cards that are synchronized in the next run

Table 180: Spectrum API: synchronization enable mask register

Register	Value	Direction	Description
SPC_SYNC_ENABLEMASK	49200	read/write	Mask of all cards that are enabled for the synchronization

The enable mask is based on the logical index explained above. It is possible to just select a couple of cards for the synchronization. All other cards then will run independently. Please be sure to always enable the card on which the star-hub is located as this one is a must for the synchronization.

In our example we synchronize all four cards. The star-hub is located on card #2 and is therefore the clock master

```
spcm_dwSetParam_i32 (hSync, SPC_SYNC_ENABLEMASK, 0x000F); // all 4 cards are masked

// set the clock master to 100 MS/s internal clock
spcm_dwSetParam_i32 (hCard[2], SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hCard[2], SPC_SAMPLERATE, MEGA(100));

// set all the slaves to run synchronously with 100 MS/s
spcm_dwSetParam_i32 (hCard[0], SPC_SAMPLERATE, MEGA(100));
spcm_dwSetParam_i32 (hCard[1], SPC_SAMPLERATE, MEGA(100));
spcm_dwSetParam_i32 (hCard[3], SPC_SAMPLERATE, MEGA(100));
```

Setup of Trigger

Setting up the trigger does not need any further steps of synchronization setup. Simply all trigger settings of all cards that have been enabled for synchronization are connected together. All trigger sources and all trigger modes can be used on synchronization as well.

Having positive edge of external trigger on card 0 to be the trigger source for the complete system needs the following setup:

```
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_ORMASK, SPC_TMASK_EXT0);
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_EXT0_MODE, SPC_TM_POS);

spcm_dwSetParam_i32 (hCard[1], SPC_TRIG_ORMASK, SPC_TM_NONE);
spcm_dwSetParam_i32 (hCard[2], SPC_TRIG_ORMASK, SPC_TM_NONE);
spcm_dwSetParam_i32 (hCard[3], SPC_TRIG_ORMASK, SPC_TM_NONE);
```

Assuming that the 4 cards are analog data acquisition cards with 4 channels each we can simply setup a synchronous system with all channels of all cards being trigger source. The following setup will show how to set up all trigger events of all channels to be OR connected. If any of the channels will now have a signal above the programmed trigger level the complete system will do an acquisition:

```
for (i = 0; i < lSyncCount; i++)
{
    int32 lAllChannels = (SPC_TMASK0_CH0 | SPC_TMASK0_CH1 | SPC_TMASK0_CH2 | SPC_TMASK0_CH3);
    spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_CH_ORMASK0, lAllChannels);
    for (j = 0; j < 2; j++)
    {
        // set all channels to trigger on positive edge crossing trigger level 100
        spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_CH0_MODE + j, SPC_TM_POS);
        spcm_dwSetParam_i32 (hCard[i], SPC_TRIG_CH0_LEVEL0 + j, 100);
    }
}
```

Run the synchronized cards

Running of the cards is very simple. The star-hub acts as one big card containing all synchronized cards. All card commands have to be omitted directly to the star-hub which will check the setup, do the synchronization and distribute the commands in the correct order to all synchronized cards. The same card commands can be used that are also possible for single cards:

Table 181: Spectrum API: star-hub synchronization commands

Register	Value	Direction	Description
SPC_M2CMD	100	write only	Executes a command for the card or data transfer
M2CMD_CARD_RESET	1h		Performs a hard and software reset of the card as explained further above
M2CMD_CARD_WRITESSETUP	2h		Writes the current setup to the card without starting the hardware. This command may be useful if changing some internal settings like clock frequency and enabling outputs.
M2CMD_CARD_START	4h		Starts the card with all selected settings. This command automatically writes all settings to the card if any of the settings has been changed since the last one was written. After card has been started 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 haven't reached this level.



SH-Direct: using the Star-Hub clock directly without synchronization

Starting with driver version 1.26 build 1754 it is possible to use the clock from the Star-Hub just like an external clock and running one or more cards totally independent of the synchronized card. The mode is by example useful if one has one or more output cards that run continuously in a loop and are synchronized with Star-Hub and in addition to this one or more acquisition cards should make multiple acquisitions but using the same clock.

For all M2i cards is also possible to run the „slave“ cards with a divided clock. Therefore please program a desired divided sampling rate in the SPC_SAMPLERATE register (example: running the Star-Hub card with 10 MS/s and the independent cards with 1 MS/s). The sampling rate is automatically adjusted by the driver to the next matching value.

What is necessary?

- All cards need to be connected to the Star-Hub
- The card(s) that should run independently can not hold the Star-Hub
- The card(s) with the Star-Hub must be setup to synchronization even if it's only one card
- The synchronized card(s) have to be started prior to the card(s) that run with the direct Star-Hub clock

Setup

At first all cards that should run synchronized with the Star-Hub are set-up exactly as explained before. The card(s) that should run independently and use the Star-Hub clock need to use the following clock mode:

Table 182: Spectrum API: clock mode register and settings for SH Direct mode

Register	Value	Direction	Description
SPC_CLOCKMODE	20200	read/write	Defines the used clock mode
SPC_CM_SHDIRECT	128		Uses the clock from the Star-Hub as if this was an external clock

When using SH_Direct mode, the register call to SPC_CLOCKMODE enabling this mode must be written before initiating a card start command to any of the connected cards. Also it is not allowed to be modified later in the programming sequence to prevent the driver from calculating wrong sample rates.



Example

In this example we have one generator card with the Star-Hub mounted running in a continuous loop and one acquisition card running independently using the SH-Direct clock.

```
// setup of the generator card
spcm_dwSetParam_i32 (hCard[0], SPC_CARDMODE, SPC_REC_STD_SINGLE);
spcm_dwSetParam_i32 (hCard[0], SPC_LOOPS, 0); // infinite data replay
spcm_dwSetParam_i32 (hCard[0], SPC_CLOCKMODE, SPC_CM_INTPLL);
spcm_dwSetParam_i32 (hCard[0], SPC_SAMPLERATE, MEGA(1));
spcm_dwSetParam_i32 (hCard[0], SPC_TRIG_ORMASK, SPC_TM_SOFTWARE);

spcm_dwSetParam_i32 (hSync, SPC_SYNC_ENABLEMASK, 0x0001); // card 0 is the generator card
spcm_dwSetParam_i32 (hSync, SPC_SYNC_CLKMASK, 0x0001); // only for M2i/M3i cards: set ClkMask

// Setup of the acquisition card (waiting for external trigger)
spcm_dwSetParam_i32 (hCard[1], SPC_CARDMODE, SPC_REC_STD_SINGLE);
spcm_dwSetParam_i32 (hCard[1], SPC_CLOCKMODE, SPC_CM_SHDIRECT);
spcm_dwSetParam_i32 (hCard[1], SPC_SAMPLERATE, MEGA(1));
spcm_dwSetParam_i32 (hCard[1], SPC_TRIG_ORMASK, SPC_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
spcm_dwSetParam_i32 (hSync, SPC_TIMEOUT, 1000);
spcm_dwSetParam_i32 (hSync, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER);

// start first acquisition
spcm_dwSetParam_i32 (hCard[1], SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_CARD_WAITREADY);

// process data

// start next acquisition
spcm_dwSetParam_i32 (hCard[1], SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_CARD_WAITREADY);

// process data
```

Error Handling

The Star-Hub error handling is similar to the card error handling and uses the function `spcm_dwGetErrorInfo_i32`. Please see the example in the card error handling chapter to see how the error handling is done.

Option Remote Server

Introduction

Using the Spectrum Remote Server (order code -SPc-RServer) it is possible to access the M2i/M3i/M4i/M4x/M2p/M5i card(s) installed in one PC (server) from another PC (client) via local area network (LAN), similar to using a digitizerNETBOX, generatorNETBOX or hybridNETBOX.

It is possible to use different operating systems on both server and client. For example the Remote Server is running on a Linux system and the client is accessing them from a Windows system.

The Remote Server software requires, that the option „-SPc-RServer“ is installed on at least one card installed within the server side PC. You can either check this with the Control Center in the "Installed Card features" node or by reading out the feature register, as described in the „Installed features and options“ passage, earlier in this manual.

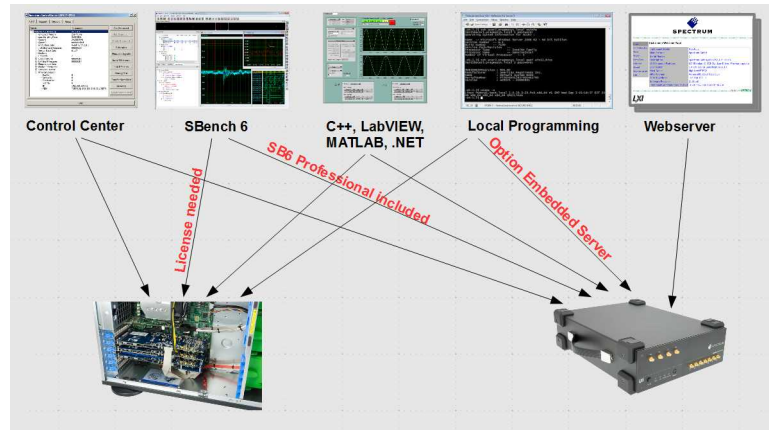


Image 86: Overview of remote server option interaction in comparison to NETBOX devices

To run the Remote Server software, it is required to have least version 3.18 of the Spectrum SPCM driver installed. Additionally at least on one card in the server PC the feature flag SPCM_FEAT_REMOTESERVER must be set.



Installing and starting the Remote Server

Windows

Windows users find the Control Center installer on the USB-Stick under „Install\win\spcm_remote_install.exe“. After the installation has finished there will be a new start menu entry in the Folder "Spectrum GmbH" to start the Remote Server. To start the Remote Server automatically after login, just copy this shortcut to the Autostart directory.

Linux

Linux users find the versions of the installer for the different StdC libraries under under /Install/linux/spcm_control_center/ as RPM packages.

To start the Remote Server type "spcm_remote_server" (without quotation marks). To start the Remote Server automatically after login, add the following line to the .bashrc or .profile file (depending on the used Linux distribution) in the user's home directory:

```
spcm_remote_server&
```

```
Spectrum Remote Server (32-bit)
SPCM Remote Server 1.3.17767
Providing access to the following cards:
M2p.5968-x4 sn13573
Listening on the following IPs:
192.168.56.1
192.168.169.42
Remote Server started successfully.
Press q + Enter to stop the Remote Server and close the program.
```

Detecting the digitizerNETBOX/generatorNETBOX/hybridNETBOX

Before accessing the digitizerNETBOX/generatorNETBOX/hybridNETBOX one has to determine the IP address of the device. Normally that can be done using one of the two methods described below:

Discovery Function

The digitizerNETBOX/generatorNETBOX/hybridNETBOX responds to the VISA described Discovery function. The next chapter will show how to install and use the Spectrum control center to execute the discovery function and to find the Spectrum hardware. As the discovery function is a standard feature of all LXI devices there are other software packages that can find the device using the discovery function:

- Spectrum control center (limited to Spectrum remote products)
- free LXI System Discovery Tool from the LXI consortium (www.lxistandard.org)
- Measurement and Automation Explorer from National Instruments (NI MAX)
- Keysight Connection Expert from Keysight Technologies

Additionally the discovery procedure can also be started from ones own specific application:

```
#define TIMEOUT_DISCOVERY 5000 // timeout value in ms

const uint32 dwMaxNumRemoteCards = 50;

char* pszVisa[dwMaxNumRemoteCards] = { NULL };
char* pszIdn[dwMaxNumRemoteCards] = { NULL };

const uint32 dwMaxIdnStringLen = 256;
const uint32 dwMaxVisaStringLen = 50;

// allocate memory for string list
for (uint32 i = 0; i < dwMaxNumRemoteCards; i++)
{
    pszVisa[i] = new char [dwMaxVisaStringLen];
    pszIdn[i] = new char [dwMaxIdnStringLen];
    memset (pszVisa[i], 0, dwMaxVisaStringLen);
    memset (pszIdn[i], 0, dwMaxIdnStringLen);
}

// first make discovery - check if there are any LXI compatible remote devices
dwError = spcm_dwDiscovery ((char**)pszVisa, dwMaxNumRemoteCards, dwMaxVisaStringLen, TIMEOUT_DISCOVERY);

// second: check from which manufacturer the devices are
spcm_dwSendIDNRequest ((char**)pszIdn, dwMaxNumRemoteCards, dwMaxIdnStringLen);

// Use the VISA strings of these devices with Spectrum as manufacturer
// for accessing remote devices without previous knowledge of their IP address
```

Finding the digitizerNETBOX/generatorNETBOX/hybridNETBOX in the network

As the digitizerNETBOX/generatorNETBOX/hybridNETBOX is a standard network device it has its own IP address and host name and can be found in the computer network. The standard host name consist of the model type and the serial number of the device. The serial number is also found on the type plate on the back of the digitizerNETBOX/generatorNETBOX/hybridNETBOX chassis.

As default DHCP (IPv4) will be used and an IP address will be automatically set. In case no DHCP server is found, an IP will be obtained using the AutoIP feature. This will lead to an IPv4 address of 169.254.x.y (with x and y being assigned to a free IP in the network) using a subnet mask of 255.255.0.0.

The default IP setup can also be restored, by using the „LAN Reset“ button on the device.

If a fixed IP address should be used instead, the parameters need to be set according to the current LAN requirements.

Windows 7, Windows 8, Windows 10 and Windows 11

Under Windows 7, Windows 8, Windows 10 and Windows 11 the digitizerNETBOX, generatorNETBOX and hybridNETBOX devices are listed under the „other devices“ tree with their given host name.

A right click on the digitizerNETBOX or generatorNETBOX device opens the properties window where you find further information on the device including the IP address.

From here it is possible to go the website of the device where all necessary information are found to access the device from software.

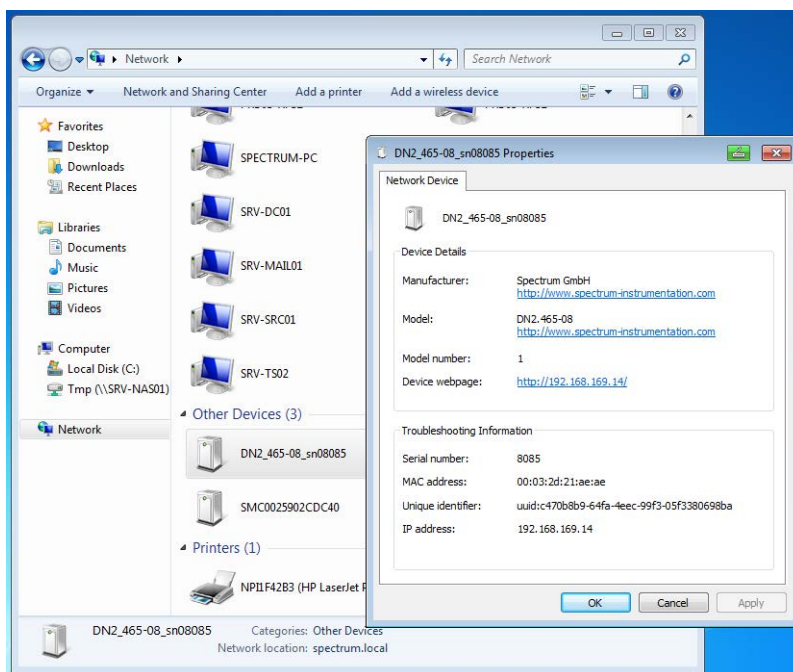


Image 87: Windows screenshot: finding a remote Spectrum device like digitizerNETBOX

Troubleshooting

If the above methods do not work please try one of the following steps:

- Ask your network administrator for the IP address of the digitizerNETBOX/generatorNETBOX and access it directly over the IP address.
- Check your local firewall whether it allows access to the device and whether it allows to access the ports listed in the technical data section.
- Check with your network administrator whether the subnet, the device and the ports that are listed in the technical data section are accessible from your system due to company security settings.

Accessing remote cards

To detect remote card(s) from the client PC, start the Spectrum Control Center on the client and click "Netbox Discovery". All discovered cards will be listed under the "Remote" node.

Using remote cards instead of using local ones is as easy as using a digitizerNETBOX and only requires a few lines of code to be changed compared to using local cards.

Instead of opening two locally installed cards like this:

```
hDrv0 = spcm_hOpen ("/dev/spcm0"); // open local card spcm0
hDrv1 = spcm_hOpen ("/dev/spcm1"); // open local card spcm1
```

one would call `spcm_hOpen()` with a VISA string as a parameter instead:

```
hDrv0 = spcm_hOpen ("TCPIP::192.168.1.2::inst0::INSTR"); // open card spcm0 on a Remote Server PC
hDrv1 = spcm_hOpen ("TCPIP::192.168.1.2::inst1::INSTR"); // open card spcm1 on a Remote Server PC
```

to open cards on the Remote Server PC with the IP address 192.168.1.2. The driver will take care of all the network communication.

Appendix

Error Codes

The following error codes could occur when a driver function has been called. Please check carefully the allowed setup for the register and change the settings to run the program.

Table 183: Spectrum API: driver error codes and error description

error name	value (hex)	value (dec.)	error description
ERR_OK	0h	0	Execution OK, no error.
ERR_INIT	1h	1	An error occurred when initializing the given card. Either the card has already been opened by another process or an hardware error occurred.
ERR_TYP	3h	3	Initialization only: The type of board is unknown. This is a critical error. Please check whether the board is correctly plugged in the slot and whether you have the latest driver version.
ERR_FNCNOTSUPPORTED	4h	4	This function is not supported by the hardware version.
ERR_BRDREMAP	5h	5	The board index re map table in the registry is wrong. Either delete this table or check it carefully for double values.
ERR_KERNELVERSION	6h	6	The version of the kernel driver is not matching the version of the DLL. Please do a complete re-installation of the hardware driver. This error normally only occurs if someone copies the driver library and the kernel driver manually.
ERR_HWDRVVERSION	7h	7	The hardware needs a newer driver version to run properly. Please install the driver that was delivered together with the card.
ERR_ADRRANGE	8h	8	One of the address ranges is disabled (fatal error), can only occur under Linux.
ERR_INVALIDHANDLE	9h	9	The used handle is not valid.
ERR_BOARDNOTFOUND	Ah	10	A card with the given name has not been found.
ERR_BOARDINUSE	Bh	11	A card with given name is already in use by another application.
ERR_EXPHW64BITADR	Ch	12	Express hardware version not able to handle 64 bit addressing -> update needed.
ERR_FWVERSION	Dh	13	Firmware versions of synchronized cards or for this driver do not match -> update needed.
ERR_SYNCPROTOCOL	Eh	14	Synchronization protocol versions of synchronized cards do not match -> update needed
ERR_LASTERR	10h	16	Old error waiting to be read. Please read the full error information before proceeding. The driver is locked until the error information has been read.
ERR_BOARDINUSE	11h	17	Board is already used by another application. It is not possible to use one hardware from two different programs at the same time.
ERR_ABORT	20h	32	Abort of wait function. This return value just tells that the function has been aborted from another thread. The driver library is not locked if this error occurs.
ERR_BOARDLOCKED	30h	48	The card is already in access and therefore locked by another process. It is not possible to access one card through multiple processes. Only one process can access a specific card at the time.
ERR_DEVICE_MAPPING	32h	50	The device is mapped to an invalid device. The device mapping can be accessed via the Control Center.
ERR_NETWORKSETUP	40h	64	The network setup of a digitizerNETBOX has failed.
ERR_NETWORKTRANSFER	41h	65	The network data transfer from/to a digitizerNETBOX has failed.
ERR_FWPOWERCYCLE	42h	66	Power cycle (PC off/on) is needed to update the card's firmware (a simple OS reboot is not sufficient !)
ERR_NETWORKTIMEOUT	43h	67	A network timeout has occurred.
ERR_BUFFERSIZE	44h	68	The buffer size is not sufficient (too small).
ERR_RESTRICTEDACCESS	45h	69	The access to the card has been intentionally restricted.
ERR_INVALIDPARAM	46h	70	An invalid parameter has been used for a certain function.
ERR_TEMPERATURE	47h	71	The temperature of at least one of the card's sensors measures a temperature, that is too high for the hardware.
ERR_REG	100h	256	The register is not valid for this type of board.
ERR_VALUE	101h	257	The value for this register is not in a valid range. The allowed values and ranges are listed in the board specific documentation.
ERR_FEATURE	102h	258	Feature (option) is not installed on this board. It's not possible to access this feature if it's not installed.
ERR_SEQUENCE	103h	259	Command sequence is not allowed. Please check the manual carefully to see which command sequences are possible.
ERR_READABORT	104h	260	Data read is not allowed after aborting the data acquisition.
ERR_NOACCESS	105h	261	Access to this register is denied. This register is not accessible for users.
ERR_TIMEOUT	107h	263	A timeout occurred while waiting for an interrupt. This error does not lock the driver.
ERR_CALLTYPE	108h	264	The access to the register is only allowed with one 64 bit access but not with the multiplexed 32 bit (high and low double word) version.
ERR_EXCEEDSINT32	109h	265	The return value is int32 but the software register exceeds the 32 bit integer range. Use double int32 or int64 accesses instead, to get correct return values.
ERR_NOWRITEALLOWED	10Ah	266	The register that should be written is a read-only register. No write accesses are allowed.
ERR_SETUP	10Bh	267	The programmed setup for the card is not valid. The error register will show you which setting generates the error message. This error is returned if the card is started or the setup is written.
ERR_CLOCKNOTLOCKED	10Ch	268	Synchronization to external clock failed: no signal connected or signal not stable. Please check external clock or try to use a different sampling clock to make the PLL locking easier.
ERR_MEMINIT	10Dh	269	On-board memory initialization error. Power cycle the PC and try another PCIe slot (if possible). In case that the error persists, please contact Spectrum support for further assistance.
ERR_POWERSUPPLY	10Eh	270	On-board power supply error. Power cycle the PC and try another PCIe slot (if possible). In case that the error persists, please contact Spectrum support for further assistance.
ERR_ADCCOMMUNICATION	10Fh	271	Communication with ADC failed. Power cycle the PC and try another PCIe slot (if possible). In case that the error persists, please contact Spectrum support for further assistance.
ERR_CHANNEL	110h	272	The channel number may not be accessed on the board: Either it is not a valid channel number or the channel is not accessible due to the current setup (e.g. Only channel 0 is accessible in interlace mode)

Table 183: Spectrum API: driver error codes and error description

error name	value (hex)	value (dec.)	error description
ERR_NOTIFYSIZE	111h	273	The notify size of the last spcm_dwDefTransfer call is not valid. The notify size must be a multiple of the page size of 4096. For data transfer it may also be a fraction of 4k in the range of 16, 32, 64, 128, 256, 512, 1k or 2k. For ABA and timestamp the notify size can be 2k as a minimum.
ERR_RUNNING	120h	288	The board is still running, this function is not available now or this register is not accessible now.
ERR_ADJUST	130h	304	Automatic card calibration has reported an error. Please check the card inputs.
ERR_PRETRIGGERLEN	140h	320	The calculated pretrigger size (resulting from the user defined posttrigger values) exceeds the allowed limit.
ERR_DIRMISMATCH	141h	321	The direction of card and memory transfer mismatch. In normal operation mode it is not possible to transfer data from PC memory to card if the card is an acquisition card nor it is possible to transfer data from card to PC memory if the card is a generation card.
ERR_POSTEXCDSEGMENT	142h	322	The posttrigger value exceeds the programmed segment size in multiple recording/ABA mode. A delay of the multiple recording segments is only possible by using the delay trigger!
ERR_SEGMENTINMEM	143h	323	Memsizes is not a multiple of segment size when using Multiple Recording/Replay or ABA mode. The programmed segment size must match the programmed memory size.
ERR_MULTIPLEPW	144h	324	Multiple pulsewidth counters used but card only supports one at the time.
ERR_NOCHANNELPWOR	145h	325	The channel pulsewidth on this card can't be used together with the OR conjunction. Please use the AND conjunction of the channel trigger sources.
ERR_ANDORMASKOVLAP	146h	326	Trigger AND mask and OR mask overlap in at least one channel. Each trigger source can only be used either in the AND mask or in the OR mask, no source can be used for both.
ERR_ANDMASKEDGE	147h	327	One channel is activated for trigger detection in the AND mask but has been programmed to a trigger mode using an edge trigger. The AND mask can only work with level trigger modes.
ERR_ORMASKLEVEL	148h	328	One channel is activated for trigger detection in the OR mask but has been programmed to a trigger mode using a level trigger. The OR mask can only work together with edge trigger modes.
ERR_EDGEPERMOD	149h	329	This card is only capable to have one programmed trigger edge for each module that is installed. It is not possible to mix different trigger edges on one module.
ERR_DOLEVELMINDIFF	14Ah	330	The minimum difference between low output level and high output level is not reached.
ERR_STARHUBENABLE	14Bh	331	The card holding the star-hub must be enabled when doing synchronization.
ERR_PATPWSMALLEDDGE	14Ch	332	Combination of pattern with pulsewidth smaller and edge is not allowed.
ERR_XMODESETUP	14Dh	333	The chosen setup for (SPCM_X0_MODE .. SPCM_X19_MODE) is not valid. See hardware manual for details.
ERR_AVRG_LSA	14Eh	334	Setup for Average LSA Mode not valid. Check Threshold and Replacement values for chosen AVRGMODE.
ERR_PCICHECKSUM	203h	515	The check sum of the card information has failed. This could be a critical hardware failure. Restart the system and check the connection of the card in the slot.
ERR_MEMALLOC	205h	517	Internal memory allocation failed. Please restart the system and be sure that there is enough free memory.
ERR_EEPROMLOAD	206h	518	Timeout occurred while loading information from the on-board EEPROM. This could be a critical hardware failure. Please restart the system and check the PCI connector.
ERR_CARDNOSUPPORT	207h	519	The card that has been found in the system seems to be a valid Spectrum card of a type that is supported by the driver but the driver did not find this special type internally. Please get the latest driver from www.spectrum-instrumentation.com and install this one.
ERR_CONFIGACCESS	208h	520	Internal error occurred during config writes or reads. Please contact Spectrum support for further assistance.
ERR_FIFOHWOVERRUN	301h	769	FIFO acquisition: Hardware buffer overrun in FIFO mode. The complete on-board memory has been filled with data and data wasn't transferred fast enough to PC memory. FIFO replay: Hardware buffer underrun in FIFO mode. The complete on-board memory has been replayed and data wasn't transferred fast enough from PC memory. If acquisition or replay throughput is lower than the theoretical bus throughput, check the application buffer setup.
ERR_FIFOFINISHED	302h	770	FIFO transfer has been finished, programmed data length has been transferred completely.
ERR_TIMESTAMP_SYNC	310h	784	Synchronization to timestamp reference clock failed. Please check the connection and the signal levels of the reference clock input.
ERR_STARHUB	320h	800	The auto routing function of the Star-Hub initialization has failed. Please check whether all cables are mounted correctly.
ERR_INTERNAL_ERROR	FFFFh	65535	Internal hardware error detected. Please check for driver and firmware update of the card.

Spectrum Knowledge Base

You will also find additional help and information in our knowledge base available on our website:

<https://spectrum-instrumentation.com/support/knowledgebase/index.php>

Temperature sensors

The M4i/M4x card series has integrated temperature sensors that allow to read out different internal temperatures. These functions are also available for the internal M4i cards inside the digitizerNETBOX, generatorNETBOX or hybridNETBOX series. In here the temperature can be read out for every internal card separately.

Temperature read-out registers

Up to three different temperature sensors can be read-out for each M4i and M4x card. Depending on the specific card type not all of these temperature sensors are used. The temperature can be read in different temperature scales at any time:

Table 184: Spectrum API: temperature read-out registers of internal temperature sensors

Register	Value	Direction	Description
SPC_MON_TK_BASE_CTRL	500022	read	Base card temperature in Kelvin
SPC_MON_TK_MODULE_0	500023	read	Module temperature 0 in Kelvin
SPC_MON_TK_MODULE_1	500024	read	Module temperature 1 in Kelvin
SPC_MON_TC_BASE_CTRL	500025	read	Base card temperature in degrees Celsius
SPC_MON_TC_MODULE_0	500026	read	Module temperature 0 in degrees Celsius
SPC_MON_TC_MODULE_1	500027	read	Module temperature 1 in degrees Celsius
SPC_MON_TF_BASE_CTRL	500028	read	Base card temperature in degrees Fahrenheit
SPC_MON_TF_MODULE_0	500029	read	Module temperature 0 in degrees Fahrenheit
SPC_MON_TF_MODULE_1	500030	read	Module temperature 1 in degrees Fahrenheit

Temperature hints

- Monitoring of the temperature figures is recommended for environments where the operating temperature can reach or even exceed the specified operating temperature. Please see technical data section for specified operating temperatures.
- The temperature sensors can be used to optimize the system cooling.

66xx temperatures and limits

The following description shows the meaning of each temperature figure on the 66xx series and also gives maximum ratings that should not be exceeded. All figures given in degrees Celsius:

Table 185: Spectrum API: temperature limits

Sensor Name	Sensor Location	Typical figure at 25°C environment temperature	Maximum temperature
BASE_CTRL	Inside FPGA	50°C ±5°C	80°C
MODULE_0	not used	n.a.	n.a.
MODULE_1	Amplifier Front-End	50°C ±5°C	80°C

Details on M4i/M4x cards I/O lines

Multi-Purpose I/O Lines

The MMCX Multi Purpose I/O connectors (X0, X1 and X2) of the M4i/M4x 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.

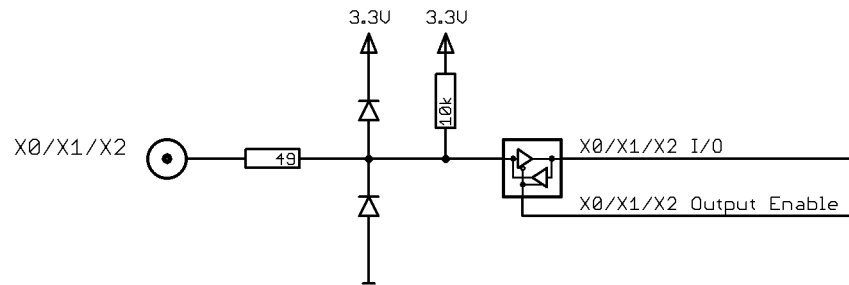


Image 88: electrical structure of multi-purpose I/O lines

The maximum forward current limit for the used CD1005 diodes is 100 mA, which is effectively limited by the used series resistor for logic levels up to 5.0V. To avoid floating levels with unconnected inputs, a pull up resistor of 10 kOhm to 3.3V is used on each line.

Interfacing with clock input

The clock input of the M4i/M4x cards is AC-coupled, single-ended PECL type. Due to the internal biasing and a relatively high maximum input voltage swing, it can be directly connected to various logic standards, without the need for external level converters.

Single-ended LVTTTL sources

All LVTTTL sources, be it 2.5V LVTTTL or 3.3V LVTTTL 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

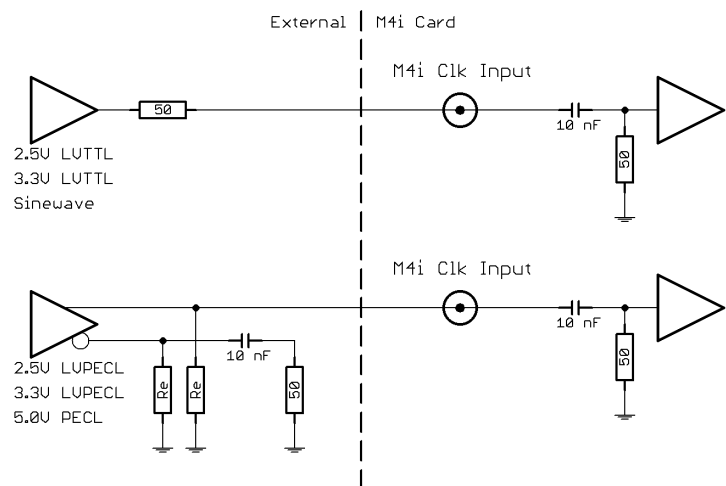


Image 89: electrical structure of clock inputs and potential interfacing circuits

Interfacing with clock output

The clock output of the M4i/M4x cards is AC-coupled, single-ended PECL type. The output swing of the M4i/M4x clock output is approximately 800 mV_{pp}.

Internal biased single-ended receivers

Because of the AC coupling of the M4i/M4x clock output, the signal must be properly re-biased for the receiver. Receivers that provide an internal re-bias only require the signal to be terminated to ground by a 50 Ohm resistor.

Differential (LV)PECL receivers

Differential receivers require proper re-biasing and likely a small minimum difference between the true and the inverting input to avoid ringing with open receiver inputs. Therefore a Thevenin-equivalent can be used, with receiver-type dependent values for $R1$, $R2$, $R1'$ and $R2'$.

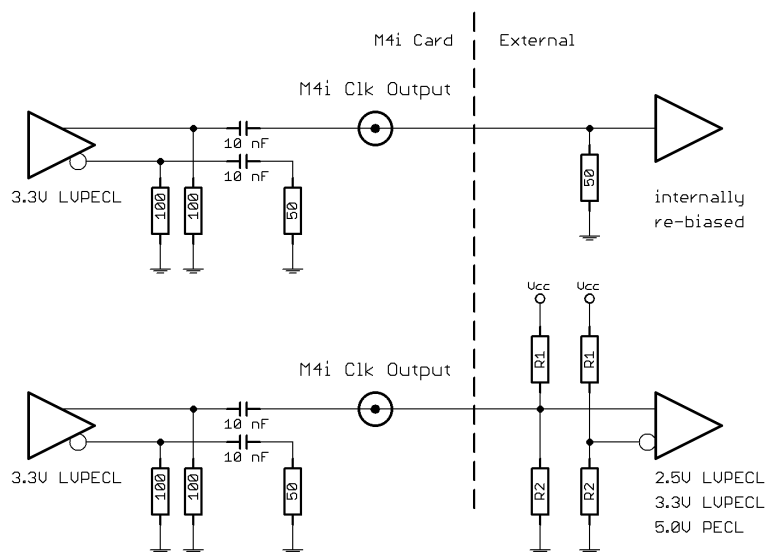


Image 90: electrical structure of clock outputs and potential interfacing circuits

Details on M4i/M4x cards status LED

Every M4i card has a two-color status LED mounted within the multi-purpose I/O connector field on the card bracket.



The same two-color LED is located on the bracket of the M4x cards as well.



This chapter explains the different color codes and offers some possible solutions in case of an error condition.

Table 186: card status LED color and blink coding

Condition	LED color	Status	Solution
Off	Off	Card not powered	Power on the PC.
Error	Static: red	Power supply error	Restart the PC. In case that the error persists, please contact Spectrum support for further assistance.
	Fast blinking (aprox. 4 Hz): red - green - red - green ...	Power supply error	Restart the PC. In case that the error persists, please contact Spectrum support for further assistance.
	Blinking: red - off - red - off ...	Over temperature error	Power down the PC, let the card cool down and restart the system. Please make sure that you have a proper cooling fan installed to supply the M4i card in the PCIe slot with a constant air flow.
	Strobed blinking: long red - short off - short red - short off ...	FPGA boot error	This error code is available with Power Firmware V1.8 or newer. Power down the PC and restart the system. In case that this error is occurring after a firmware update please contact Spectrum support for assistance on how to boot the card's golden recovery image.
	Slow blinking (aprox. 1 Hz): red - green - red - green ...	PCI Express link training has not finished	1) Power down the PC, un-plug and re-plug the card to verify that there is a proper contact between the card and the slot. 2) Try another PCI Express slot, maybe the currently used one is not properly working. In case that the above steps did not help, please contact Spectrum support for assistance.
O.K.	Static: green	Card is ready for operation (at full PCIe speed)	A full speed PCIe link has been established (PCIe x8, Gen 2) and the card is ready for operation.
	Slow blinking (approx. 1 Hz): green - off - green - off ...	Indicator mode on (at full PCIe speed)	To ease the identification of a specific card in a multi-card system without un-installing the card it is possible to activate the card identification status by software. This mode changes the static „Ready for Operation“ green into a blinking state.
	Static: green & red (yellow)	Card is ready for operation (at reduced PCIe speed)	A reduced speed PCIe link has been established either with less than all of the possible 8 lanes and/or the card is installed in a PCIe Gen 1 slot. The card is ready for operation, but the data transfer throughput over the PCI Express bus is reduced. For getting the highest PCIe performance please consult your PC's or motherboard's manual for details on the PCI Express slots of your system.
	Slow blinking (approx. 1 Hz): yellow - off - yellow - off ...	Indicator mode on (at reduced PCIe speed)	To ease the identification of a specific card in a multi-card system without un-installing the card it is possible to activate the card identification status by software. This mode changes the static „Ready for Operation“ yellow into a blinking state.

Turning on card identification LED

To enable/disable the cards LED indicator mode or to read out the current setting, please use the following register:

Table 187: Spectrum API: card identification LED register

Register	Value	Direction	Description
SPC_CARDIDENTIFICATION	201500	read/write	Writing a '1' turns on the LED card indicator mode, writing a '0' turns off the LED indicator mode.

The default for the card identification register is the OFF state.

Continuous memory for increased data transfer rate



The continuous memory buffer has been added to the driver version 1.36. The continuous buffer is not available in older driver versions. Please update to the latest driver if you wish to use this function.

Background

All modern operating systems use a very complex memory management strategy that strictly separates between physical memory, kernel memory and user memory. The memory management is based on memory pages (normally 4 kByte = 4096 Bytes). All software only sees virtual memory that is translated into physical memory addresses by a memory management unit based on the mentioned pages.

This will lead to the circumstance that although a user program allocated a larger memory block (as an example 1 MByte) and it sees the whole 1 MByte as a virtually continuous memory area this memory is physically located as spread 4 kByte pages all over the physical memory. No problem for the user program as the memory management unit will simply translate the virtual continuous addresses to the physically spread pages totally transparent for the user program.

When using this virtual memory for a DMA transfer things become more complicated. The DMA engine of any hardware can only access physical addresses. As a result the DMA engine has to access each 4 kByte page separately. This is done through the Scatter-Gather list. This list is simply a linked list of the physical page addresses which represent the user buffer. All translation and set-up of the Scatter-Gather list is done inside the driver without being seen by the user. Although the Scatter-Gather DMA transfer is an advanced and powerful technology it has one disadvantage: For each transferred memory page of data it is necessary to also load one Scatter-Gather entry (which is 16 bytes on 32 bit systems and 32 bytes on 64 bit systems). The little overhead to transfer (16/32 bytes in relation to 4096 bytes, being less than one percent) isn't critical but the fact that the continuous data transfer on the bus is broken up every 4096 bytes and some different addresses have to be accessed slows things down.

The solution is very simple: everything works faster if the user buffer is not only virtually continuous but also physically continuous. Unfortunately it is not possible to get a physically continuous buffer for a user program. Therefore the kernel driver has to do the job and the user program simply has to read out the address and the length of this continuous buffer. This is done with the function `spcm_dwGetContBuf` as already mentioned in the general driver description. The desired length of the continuous buffer has to be programmed to the kernel driver for load time and is done differently on the different operating systems. Please see the following chapters for more details.

Next we'll see some measuring results of the data transfer rate with/without continuous buffer. You will find more results on different motherboards and systems in the application note number 6 „Bus Transfer Speed Details“. Also with newer M5i/M4i/M4x/M2p cards the gain in speed is not as impressive, as it is for older cards, but can be useful in certain applications and settings. As this is also system dependent, your improvements may vary. This can not only depend on the system hardware but also on the used operating system, as in some cases Linux does seem to benefit more than Windows for newer cards.

Bus Transfer Speed Details (M2i/M3i cards in an example system)

Mode	PCI 33 MHz slot		PCI-X 66 MHz slot		PCI Express x1 slot	
	read	write	read	write	read	write
User buffer	109 MB/s	107 MB/s	195 MB/s	190 MB/s	130 MB/s	138 MB/s
Continuous kernel buffer	125 MB/s	122 MB/s	248 MB/s	238 MB/s	160 MB/s	170 MB/s
Speed advantage	15%	14%	27%	25%	24%	23%

Bus Transfer Standard Read/Write Transfer Speed Details (M4i.44xx card in an example system)

Mode	Notifysize 16 kByte		Notifysize 64 kByte		Notifysize 512 kByte		Notifysize 2048 kByte		Notifysize 4096 kByte	
	read	write	read	write	read	write	read	write	read	write
User buffer	243 MB/s	132 MB/s	793 MB/s	464 MB/s	2271 MB/s	1352 MB/s	2007 MB/s	1900 MB/s	2687 MB/s	2284 MB/s
Continuous kernel buffer	239 MB/s	133 MB/s	788 MB/s	457 MB/s	2270 MB/s	1470 MB/s	2555 MB/s	2121 MB/s	2989 MB/s	2549 MB/s
Speed advantage	-1.6%	+0.7%	-0.6%	-1.5%	0%	+8.7%	+27.3%	+11.6%	+11.2%	+11.6%

Bus Transfer FIFO Read Transfer Speed Details (M4i.44xx card in an example system)

Mode	Notifysize 4 kByte	Notifysize 8 kByte	Notifysize 16 kByte	Notifysize 32 kByte	Notifysize 64 kByte	Notifysize 256 kByte	Notifysize 1024 kByte	Notifysize 2048 kByte	Notifysize 4096 kByte
	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read
User buffer	455 MB/s	858 MB/s	1794 MB/s	2005 MB/s	3335 MB/s	3386 MB/s	3369 MB/s	3331 MB/s	3335 MB/s
Continuous kernel buffer	540 MB/s	833 MB/s	1767 MB/s	1965 MB/s	3216 MB/s	3386 MB/s	3389 MB/s	3388 MB/s	3389 MB/s
Speed advantage	+18.6%	-2.9%	-1.5%	-2.0%	-3.5%	0%	+0.6%	+1.7%	+1.6%

Bus Transfer FIFO Read Transfer Speed Details (M2p.5942 card in an example system)

Mode	Notifysize 4 kByte	Notifysize 8 kByte	Notifysize 16 kByte	Notifysize 32 kByte	Notifysize 64 kByte	Notifysize 256 kByte	Notifysize 1024 kByte	Notifysize 2048 kByte	Notifysize 4096 kByte
	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read	FIFO read
User buffer	282 MB/s	462 MB/s	597 MB/s	800 MB/s	800 MB/s	799 MB/s	799 MB/s	799 MB/s	797 MB/s
Continuous kernel buffer	279 MB/s	590 MB/s	577 MB/s	800 MB/s	800 MB/s	800 MB/s	800 MB/s	800 MB/s	799 MB/s
Speed advantage	-1.1%	+27.7%	-3.4%	+0.0%	+0.0%	0%	+0.1%	+0.1%	+0.3%

Setup on Linux systems

On Linux systems the continuous buffer setting is done via the command line argument `contmem_mb` when loading the kernel driver module:

```
insmod spcm.ko contmem_mb=4
```

As memory allocation is organized completely different compared to Windows the amount of data that is available for a continuous DMA buffer is unfortunately limited to a maximum of 8 MByte. On most systems it will even be only 4 MBytes.

To use a larger continuous buffer you can use the Continuous Memory Allocator (CMA). To allocate continuous memory this way you pass „cma=xyz“ as kernel boot parameter, with xyz being the size of the continuous memory, e.g. „cma=128M“ for 128 MByte.



Your kernel needs to have CMA support enabled to use this.

You can check this with „grep CONFIG_CMA /boot/config-\$(uname -r)“.

To enable CMA in our `spcm4` kernel driver module edit the Makefile for the kernel driver module and uncomment the line `#EXTRA_CFLAGS += -DSPCM4_USE_CMA` by removing the # in front. Then recompile the kernel module and load it as described above, like so as example:

```
insmod spcm4.ko contmem_mb=128
```



Using a continuous buffer of this size will need root privileges for the using program on most systems!

Setup on Windows systems

The continuous buffer settings is done with the Spectrum Control Center using a setup located on the „Support“ page. Please fill in the desired continuous buffer settings as MByte. After setting up the value the system needs to be restarted as the allocation of the buffer is done during system boot time.

If the system cannot allocate the amount of memory it will divide the desired memory by two and try again. This will continue until the system can allocate a continuous buffer. Please note that this try and error routine will need several seconds for each failed allocation try during boot up procedure. During these tries the system will look like being crashed. It is then recommended to change the buffer settings to a smaller value to avoid the long waiting time during boot up.

Continuous buffer settings should not exceed 1/4 of system memory. During tests the maximum amount that could be allocated was 384 MByte of continuous buffer on a system with 4 GByte memory installed.

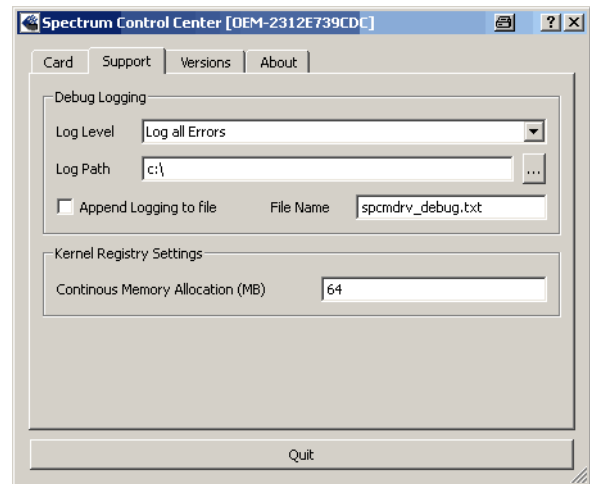


Image 91: setting up continuous memory buffer in Spectrum Control Center

Usage of the buffer

The usage of the continuous memory is very simple. It is just necessary to read the start address of the continuous memory from the driver and use this address instead of a self allocated user buffer for data transfer.

Function `spcm_dwGetContBuf`

This function reads out the internal continuous memory buffer (in bytes) if one has been allocated. If no buffer has been allocated the function returns a size of zero and a NULL pointer.

```
uint32 _stdcall spcm_dwGetContBuf_i64 ( // Return value is an error code
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType,                  // type of the buffer to read as listed above under SPCM_BUF_XXXX
    void** ppvDataBuffer,              // address of available data buffer
    uint64* pqwContBufLen);            // length of available continuous buffer

uint32 _stdcall spcm_dwGetContBuf_i64m ( // Return value is an error code
    drv_handle hDevice,                // handle to an already opened device
    uint32 dwBufType,                  // type of the buffer to read as listed above under SPCM_BUF_XXXX
    void** ppvDataBuffer,              // address of available data buffer
    uint32* pdwContBufLenH,            // high part of length of available continuous buffer
    uint32* pdwContBufLenL);           // low part of length of available continuous buffer
```

Please note that it is not possible to free the continuous memory for the user application.

Example

The following example shows a simple standard single mode data acquisition setup (for a card with 12/14/16 bit per resolution one sample equals 2 bytes) with the read out of data afterwards. To keep this example simple there is no error checking implemented.

```
int32 lMemsize = 16384; // recording length is set to 16 kSamples

spcm_dwSetParam_i64 (hDrv, SPC_CHENABLE, CHANNEL0); // only one channel activated
spcm_dwSetParam_i32 (hDrv, SPC_CARDMODE, SPC_REC_STD_SINGLE); // set the standard single recording mode
spcm_dwSetParam_i64 (hDrv, SPC_MEMSIZE, lMemsize); // recording length in samples
spcm_dwSetParam_i64 (hDrv, SPC_POSTTRIGGER, 8192); // samples to acquire after trigger = 8k

// now we start the acquisition and wait for the interrupt that signalizes the end
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_CARD_START | M2CMD_CARD_ENABLETRIGGER | M2CMD_CARD_WAITREADY);

// we now try to use a continuous buffer for data transfer or allocate our own buffer in case there's none
spcm_dwGetContBuf_i64 (hDrv, SPCM_BUF_DATA, &pvData, &qwContBufLen);
if (qwContBufLen < (2 * lMemsize))
    pvData = pvAllocMemPageAligned (lMemsize * 2); // assuming 2 bytes per sample

// read out the data
spcm_dwDefTransfer_i64 (hDrv, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, 0, pvData, 0, 2 * lMemsize);
spcm_dwSetParam_i32 (hDrv, SPC_M2CMD, M2CMD_DATA_STARTDMA | M2CMD_DATA_WAITDMA);

// ... Use the data here for analysis/calculation/storage

// delete our own buffer in case we have created one
if (qwContBufLen < (2 * lMemsize))
    vFreeMemPageAligned (pvData, lMemsize * 2);
```


Abbreviations

Table 188: Abbreviations used throughout the Spectrum documents

Abbreviation	Long Name	Description
s	Second	
ms	Milli Second	1/1000 second; 1 ms is the time between two samples when running at 1 kS/s
us (µs)	Micro Second	1/1000000 second or 1/1000 milli second; 1 ms is the time between two samples when running at 1 MS/s
ns	Nano Second	1/1000000000 second or 1/1000 micro second; 1 ns is the time between two samples when running at 1 GS/s
ps	Pico Second	1/1000000000000 second or 1/1000 nano second
Sample		One sample represents one data word that has been acquired on the same time position. Each sample consist of either one byte (8 bit resolution) or two bytes (12, 14 and 16 bit resolution)
Byte		The smallest storage unit
kB	Kilo Bytes	1024 (2 ¹⁰) Bytes
MB	Mega Bytes	1024 x 1024 (2 ²⁰) Bytes
GB	Giga Bytes	1024 x 1024 x 1024 (2 ³⁰) Bytes
Hz	Hertz	1 Hertz is one event/sample per second
kHz	Kilo Hertz	1000 Hertz
MHz	Mega Hertz	1000000 Hertz or 1000 kHz
GHz	Giga Hertz	1000000000 Hertz or 1000 MHz
kS/s	kilo Samples per Second	1000 samples per second
MS/s	Mega Samples per Second	1000 kilo samples (1000000 samples) per second
GS/s	Giga Samples per Second	1000 Mega samples (1000000000 samples) per second
PCIe	PCI Express	The PCI Express bus is a point to point connection allowing full speed for every single slot. The Express bus is freely scaling and is available with 1 lane (x1), 4 lanes (x4), 8 lanes (x8) and 16 lanes (x16)
PXI	PCI eXtensions for Instrumentation	Based on the CompactPCI 3U standard the PXI (PCI eXtensions for Instrumentation) enhancement was defined especially for the measurement user. In this specification additional lines for measurement purposes are defined.
PXIe	PXI Express	PXI Express or PXIe is a subset of the PXI standard that replaces PXI's parallel data bus with a high speed serial interface.
PLL	Phase Lock Loop	A clock device which generates a new clock phase-locked to a given reference clock.
LED	Light-Emitting Diode	A semiconductor device that emits light and is often used as a status light or indicator.
API	Application Programming Interface	A type of software interface, offering a service to access/control specific hardware or other pieces of software.
CPU	Central Processing Unit	The central processor of a computer/PC system.
GPU	Graphics Processing Unit	An co-processor specifically tailored for fast and efficient and massively parallel calculations of certain data structures. Often, but not exclusively, located on a separate PCIe graphics card or co-processing card. s
CUDA	Compute Unified Device Architecture	A proprietary API for Nvidia GPUs to perform "general purpose" as in non-graphic related processing on GPUs rather than the CPU.
DMA	Direct Memory Access	A method to transfer data directly between a device (card) and PC memory.
RDMA	Remote Direct Memory Access	A method to transfer data directly between two devices (cards).
RMA	Return Manufacturer Authorization	
WEEE	Waste Electrical and Electronic Equipment)	

List of Figures

Image 1: M4i card showing mounted star-hub and sync bus connector.....	17
Image 2: M4i card showing mounted digital option	17
Image 3: M4i.66xx PCI Express series hardware block diagram	18
Image 4: M4x.66xx PXI Express series hardware block diagram	18
Image 5: Mounting M4i PCIe card into connector	28
Image 6: M4i card slot retention with perforation.....	29
Image 7: M4i card additional power connection usage	29
Image 8: M4i card with digital option mechanical installation and position of screws	30
Image 9: M4i cards with star-hub ex installed and connecting cables	30
Image 10: Installation of PXIe cards with connector handling and mounting screws	31
Image 11: Windows Device Manager showing a new Spectrum card	32
Image 12: Spectrum Driver Installer Welcome Screen.....	32
Image 13: Spectrum Driver Installer - Progress	33
Image 14: Spectrum Driver Installer - finished	33
Image 15: Windows Device Manager showing properly installed Spectrum card	33
Image 16: Device Manager showing a new Spectrum card	37
Image 17: Spectrum Kernel Driver, API Library and Software structure	39
Image 18: Spectrum Control Center Installer.....	39
Image 19: Spectrum Control Center showing detail card information	40
Image 20: Spectrum Control Center - entering an IP address for a NETBOX	40
Image 21: Spectrum Control Center: wake on LAN for a cached card.....	40
Image 22: Netbox Monitor activation.....	41
Image 23: Spectrum Control Center: detailed hardware information on installed card	42
Image 24: Spectrum Control Center - showing firmware information of an installed card	42
Image 25: Spectrum Control Center - showing firmware information of an installed card	43
Image 26: Spectrum Control Center - showing driver information details.....	43
Image 27: Spectrum Control Center - adding a demo card to the system	44
Image 28: Spectrum Control Center - feature update, code entry	44
Image 29: Spectrum Control Center - software license install	44
Image 30: Spectrum Control Center - running an on-board calibration.....	44
Image 31: Spectrum Control Center - performing memory test	45
Image 32: Spectrum Control Center - running a transfer speed test of one card	45
Image 33: Spectrum Control Center - activate debug logging for support cases	45
Image 34: Spectrum Control Center - using device mapping	46
Image 35: Spectrum Control Center - doing a firmware update for one device	46
Image 36: SBench 6 overview of main functionality with demo data	47
Image 37: Structure of the Delphi examples	56
Image 38: LabVIEW driver oscilloscope example.....	63
Image 39: Spectrum MATLAB driver structure	63
Image 40: GPU usage with SCAPP SDK: data transfer options	64
Image 41: Scaling the output swing using the output amplitude registers.....	75
Image 42: output stage showing amplifier and filters.....	76
Image 43: schematics of double output mode	77
Image 44: Acquisition cards: graphical overview of acquisition status and card command interaction.....	81
Image 45: Generation cards: graphical overview of generation status and card command interaction	81
Image 46: timing diagram of single replay mode with commands and status changes	84
Image 47: timing diagram of single replay mode with two loops with commands and status changes	85
Image 48: timing diagram of continuous replay mode stopped by user with commands and status changes.....	85
Image 49: timing diagram of single restart mode with commands and status changes	85
Image 50: Overview of buffer handling for DMA transfers showing and the interaction with the DMA engine	89
Image 51: output latency involved components.....	92
Image 52: M4i/M4x clock section overview	96
Image 53: Trigger Engine Overview. Red marked parts not available on all card types	100
Image 54: trigger engine overview with trigger OR mask shown.....	101
Image 55: trigger engine OR mask details	101
Image 56: trigger engine overview with trigger AND mask shown	102
Image 57: trigger engine AND mask details.....	103
Image 58: trigger engine overview with marked trigger delay stage	105
Image 59: trigger engine overview with marked main external trigger Ext0/Trg0	106
Image 60: trigger engine overview with external trigger Ext1 marked	107
Image 61: trigger overview with multi-purpose lines marked.....	115
Image 62: Multiple Replay output and trigger timing diagram	119
Image 63: timing diagram of multiple replay mode depending on loops settings	120
Image 64: Spectrum API: card mode register and multiple replay FIFO mode settings.....	120
Image 65: timing diagram of Multiple Replay FIFO mode with different loops settings	121
Image 66: Gated Replay timing diagram in relation to gate signal.....	123
Image 67: timing diagram of Gated Replay mode depending on different loops settings	123
Image 68: timing diagram of Gated Replay FIFO mode depending on different loops settings.....	124
Image 69: Sequence Mode: Segment definition in card memory.....	130
Image 70: Sequence mode: steps and step looping	130

Image 71: sequence mode changing sequence on-the-fly	133
Image 72: command queue, trigger and timer interaction	136
Image 73: Overview of the DDS buffer structure, data flow and size registers	136
Image 74: block diagram of M4i.66xx DDS option showing the DDS cores and connections options.	139
Image 75: The shown multiplexers can be switched using the DDS_CONNECTION register	139
Image 76: Details of the DDS core	140
Image 77: Phase Jumps to 90° instantly when the 2nd trigger arrives, regardless of prior state.....	144
Image 78: Phase shifts forward by 90° relative to current state at 2nd Trigger.....	144
Image 79: overview block diagram of multi-purpose I/O lines and pulse generators	148
Image 80: overview block diagram of the pulse generator	149
Image 81: timing diagram illustrating the basic pulse parameters	150
Image 82: timing diagram illustrating delaying a pulse generator output	151
Image 83: timing diagram illustrating the pulse generator triggered output mode.....	152
Image 84: timing diagram illustrating the pulse generator single-shot triggered output mode	152
Image 85: timing diagram illustrating the pulse generator gated output mode	152
Image 86: Overview of remote server option interaction in comparison to NETBOX devices.....	162
Image 87: Windows screenshot: finding a remote Spectrum device like digitizerNETBOX	163
Image 88: electrical structure of multi-purpose I/O lines.....	168
Image 89: electrical structure of clock inputs and potential interfacing circuits.....	168
Image 90: electrical structure of clock outputs and potential interfacing circuits.....	168
Image 91: setting up continuous memory buffer in Spectrum Control Center	172

List of Tables

Table 1: Symbols and Safety Labels	9
Table 2: Packing List M4i card (PCI Express)	12
Table 3: Packing List M4x card (PXI Express)	12
Table 4: list of C/C++ header files in driver	47
Table 5: C/C++ type declarations for drivers and examples	49
Table 6: C/C++ type naming convention throughout drivers and examples	49
Table 7: Spectrum driver API functions overview and differentiation between 32 bit and 64 bit registers	52
Table 8: Spectrum API: Command register and basic commands	65
Table 9: Spectrum API: Card Type Register	68
Table 10: Spectrum API: list of card type codes for M4i.66xx series	68
Table 11: Spectrum API: list of card type codes for M4x.66xx series	68
Table 12: Spectrum API: hardware and PCB version register overview	68
Table 13: Spectrum API: extension module hardware and PCB version register	68
Table 14: Spectrum API: register for reading back the PXIe card slot number	69
Table 15: Spectrum API: Register overview of firmware versions	69
Table 16: Spectrum API: Register overview of reading current firmware	69
Table 17: Spectrum API: production date register	69
Table 18: Spectrum API: calibration date register	70
Table 19: Spectrum API: hardware serial number register	70
Table 20: Spectrum API: maximum sampling rate register	70
Table 21: Spectrum API: installed memory registers. 32 bit read is limited to a maximum of 1 GByte	70
Table 22: Spectrum API: Feature Register and available feature flags	70
Table 23: Spectrum API: Extended feature register and available extended feature flags	71
Table 24: Spectrum API: register overview of miscellaneous cards information	71
Table 25: Spectrum API: register card function type and possible types	72
Table 26: Spectrum API: register driver type information and possible driver types	72
Table 27: Spectrum API: driver version read register	72
Table 28: Spectrum API: kernel driver version read register	72
Table 29: Spectrum API: custom modification register and different bitmasks to split the register in various hardware parts	73
Table 30: Spectrum API: command register and reset command	73
Table 31: Spectrum API: channel enable register and register settings	74
Table 32: Spectrum API: channel count register	74
Table 33: Spectrum API: output enable register and register settings	75
Table 34: Spectrum API: output amplitude registers and register settings depending on board type	75
Table 35: Spectrum API: output filter registers and register settings	76
Table 36: output filter specifications depending on card version	76
Table 37: Spectrum API: differential output register and register settings	77
Table 38: availability of differential output mode depending on AWG model	77
Table 39: Spectrum API: double output mode registers	77
Table 40: availability of double output mode depending on AWG model	77
Table 41: Spectrum API: stop level register and register settings	78
Table 42: Spectrum API: custom stop level registers	78
Table 43: Spectrum API: card mode and read out of available card mode software registers	79
Table 44: Spectrum API: card command register and different commands with descriptions	79
Table 45: Spectrum API: timeout definition register	80
Table 46: Spectrum API: card status register and possible status values with descriptions of the status	80
Table 47: Spectrum API: memory test register	83
Table 48: Spectrum API: Command register and commands for DMA transfers	83
Table 49: Spectrum API: status register and status codes for DMA data transfer	83
Table 50: Spectrum API: card mode register and single mode settings	84
Table 51: Spectrum API: memory and loop settings	84
Table 52: Spectrum API: overview of mode settings in relation to loops settings and resulting behaviour	85
Table 53: Spectrum API: FIFO single replay mode register and settings	86
Table 54: Spectrum API: FIFO mode length settings registers	86
Table 55: Spectrum API: limits of segment size, memory size and loops registers depending on selected mode	88
Table 56: Spectrum API: registers for DMA buffer handling	89
Table 57: Spectrum API: content of DMA buffer handling registers for different use cases	89
Table 58: Spectrum API: output buffer size register and register settings	92
Table 59: output latency depending on channel settings, buffer settings and output FIFO	93
Table 60: M4i and M4x cards data organization	94
Table 61: Spectrum API: data format and DAC resolution depending on selected mode and digital output modes	94
Table 62: Spectrum API: hardware data conversion registers and available register settings	94
Table 63: Spectrum API: clock mode register and available clock modes	96
Table 64: Spectrum API: clock mode register and internal clock mode	97
Table 65: Spectrum API: samplerate register	97
Table 66: Spectrum API: clock output and clock output frequency register	97
Table 67: Spectrum API: clock mode register and quartz 2 settings	97
Table 68: Spectrum API: clock output and clock output frequency register	97
Table 69: Spectrum API: clock mode register and external reference clock setup	98
Table 70: Spectrum API: reference clock register and available settings	98

Table 71: Spectrum API: clock output and clock output frequency register	98
Table 72: Spectrum API: clock mode register and PXI reference clock usage	99
Table 73: Spectrum API: general trigger OR mask register and available settings	101
Table 74: Spectrum API: channel trigger OR mask registers and available settings	102
Table 75: Spectrum API: general trigger AND mask registers and available settings	103
Table 76: Spectrum API: channel trigger AND mask registers and available settings	103
Table 77: Spectrum API: software register and register setting for software trigger	104
Table 78: Spectrum API: command register and force trigger command	104
Table 79: Spectrum API: command register and trigger enable/disable command	104
Table 80: Spectrum API: trigger delay registers and available settings	105
Table 81: Spectrum API: trigger counter register and register return values	105
Table 82: Spectrum API: external trigger Ext0 registers and register settings	106
Table 83: Spectrum API: external trigger Ext0 OR mask settings	106
Table 84: Spectrum API: external trigger Ext0 input termination	106
Table 85: Spectrum API: external trigger Ext0 input coupling	107
Table 86: Spectrum API: external trigger Ext1 registers and register settings	107
Table 87: Spectrum API: external trigger Ext1 OR mask settings	107
Table 88: Spectrum API: external trigger available settings for trigger levels	107
Table 89: Spectrum API: external trigger OR mask and AND mask register and settings	108
Table 90: Spectrum API: external register mode setup for trigger on positive edge	108
Table 91: Spectrum API: external register mode setup for trigger on negative edge	108
Table 92: Spectrum API: external trigger register mode setup for trigger on positive and negative edge	109
Table 93: Spectrum API: external trigger register mode setup for trigger re-arm on positive edge	109
Table 94: Spectrum API: external trigger register mode setup for trigger re-arm on negative edge	109
Table 95: Spectrum API: external trigger register mode setup for window trigger for entering signals	110
Table 96: Spectrum API: external trigger register mode setup for window trigger for leaving signals	110
Table 97: Spectrum API: external trigger register mode setup for high level trigger	110
Table 98: Spectrum API: external trigger register mode setup for low level trigger	111
Table 99: Spectrum API: external trigger register mode setup for in window trigger	111
Table 100: Spectrum API: external trigger register mode setup for outside window trigger	111
Table 101: trigger overview with PXI trigger lines marked	112
Table 102: Spectrum API: PXI trigger register and available register settings	112
Table 103: Spectrum API: PXI trigger mask register and available register settings	113
Table 104: Spectrum API: multi-purpose I/O lines registers and available register settings	115
Table 105: Spectrum API: asynchronous I/O register settings of the multi-purpose I/O registers	116
Table 106: Spectrum API: additional trigger output register for compatibility with older hardware	116
Table 107: Spectrum API: multi-purpose I/O registers and synchronous digital output settings	117
Table 108: Spectrum API: data format and DAC resolution depending on selected mode and digital output modes	117
Table 109: Spectrum API: segment size register for multiple replay mode	119
Table 110: Spectrum API: card mode register and multiple replay settings	120
Table 111: Spectrum API: memory and loop registers with related multiple replay settings	120
Table 112: Spectrum API: loops register settings when using Multiple Replay FIFO mode	120
Table 113: Spectrum API: limits of segment size, memory size and loops registers depending on selected mode	121
Table 114: Spectrum API: stop level register and register settings	121
Table 115: Spectrum API: custom stop level registers	122
Table 116: Spectrum API: card mode register and settings for Gated Replay standard mode	123
Table 117: Spectrum API: memsize and loops register and register settings for Gated Replay mode	123
Table 118: Spectrum API: card mode register and Gated Replay FIFO mode settings	123
Table 119: Spectrum API: Gated Replay FIFO mode loops register settings	124
Table 120: Spectrum API: limits of segment size, memory size and loops registers depending on selected mode	124
Table 121: Spectrum API: trigger mask registers and available register settings	125
Table 122: Spectrum API: trigger register settings for trigger on positive edge	125
Table 123: Spectrum API: trigger register settings for trigger on negative edge	125
Table 124: Spectrum API: trigger register settings for re-arm trigger on positive edge	126
Table 125: Spectrum API: trigger register settings for re-arm trigger on negative edge	126
Table 126: Spectrum API: trigger register settings for window trigger for entering signals	126
Table 127: Spectrum API: trigger register settings for window trigger on leaving signals	127
Table 128: Spectrum API: trigger register settings for high-level trigger	127
Table 129: Spectrum API: trigger register settings for low-level trigger	127
Table 130: Spectrum API: trigger register settings for in-window trigger	128
Table 131: Spectrum API: trigger register settings for outside-window trigger	128
Table 132: Spectrum API: stop level register and register settings	129
Table 133: Spectrum API: custom stop level registers	129
Table 134: Spectrum API: sequence mode registers and register settings	131
Table 135: Spectrum API: card mode register with Sequence Mode setup	131
Table 136: Spectrum API: sequence mode registers for segment handling	131
Table 137: Spectrum API: sequence mode step registers and register setup	132
Table 138: Spectrum API: sequence mode start register	132
Table 139: Spectrum API: sequence mode segment status register	132
Table 140: Spectrum API: card mode and read out of available card mode software registers	135
Table 141: Spectrum API: DDS information on the command queue	136
Table 142: Spectrum API: DDS command register	137

Table 143: Spectrum API: DDS trigger sources	137
Table 144: Spectrum API: DDS trigger status register	137
Table 145: Spectrum API: DDS information registers	138
Table 146: Spectrum API: DDS trigger timer	138
Table 147: Spectrum API: DDS trigger timer resolution depending on product	138
Table 148: Spectrum API: DDS connection register	139
Table 149: Spectrum API: DDS core frequency settings	141
Table 150: Spectrum API: DDS programmable frequency range	141
Table 151: Spectrum API: DDS core amplitude settings	141
Table 152: Spectrum API: DDS programmable amplitude range	141
Table 153: Spectrum API: DDS core phase settings	141
Table 154: Spectrum API: DDS programmable phase range	141
Table 155: Spectrum API: DDS core frequency slope settings	142
Table 156: Spectrum API: DDS programmable frequency slope range	142
Table 157: Spectrum API: DDS core amplitude slope settings	143
Table 158: Spectrum API: DDS programmable amplitude slope range	143
Table 159: Spectrum API: DDS core slope stepsize settings	143
Table 160: Spectrum API: DDS Phase Behaviour	143
Table 161: Spectrum API: multi-purpose I/O lines registers and available register settings	145
Table 162: Spectrum API: DDS multi-purpose I/O additional registers	145
Table 163: Spectrum API: DDS multi-purpose I/O manual output register	145
Table 164: Spectrum API: data transfer mode definition	146
Table 165: Comparison of single and DMA transfer mode	146
Table 166: Spectrum API: pulse generator clock frequency read register	149
Table 167: Spectrum API: pulse generator enable registers	150
Table 168: Spectrum API: pulse generator length/period register	150
Table 169: Spectrum API: pulse generator HIGH time registers	150
Table 170: Spectrum API: pulse generator loops/pulse repetition registers	151
Table 171: Spectrum API: pulse generator delay/phase shift registers	151
Table 172: Spectrum API: pulse generator mode registers with their available settings	151
Table 173: Spectrum API: pulse generator trigger MUX1 registers with their available settings	152
Table 174: Spectrum API: pulse generator trigger MUX2 registers with their available settings	153
Table 175: Spectrum API: pulse generator command register for trigger forcing by software	153
Table 176: Spectrum API: pulse generator additional configuration registers with the available settings	153
Table 177: Spectrum API: XIO lines and mode software registers with their reduced to the settings required for outputting pulses	154
Table 178: star-hub clock overview diagram	156
Table 179: Spectrum API: star-hub related registers for reading detected connections	157
Table 180: Spectrum API: synchronization enable mask register	158
Table 181: Spectrum API: star-hub synchronization commands	159
Table 182: Spectrum API: clock mode register and settings for SH Direct mode	160
Table 183: Spectrum API: driver error codes and error description	165
Table 184: Spectrum API: temperature read-out registers of internal temperature sensors	167
Table 185: Spectrum API: temperature limits	167
Table 186: card status LED color and blink coding	169
Table 187: Spectrum API: card identification LED register	169
Table 188: Abbreviations used throughout the Spectrum documents	174



Spectrum Instrumentation GmbH

Ahrensfelder Weg 13-17 | 22927 Grosshansdorf | Germany

Phone +49 (0)4102-69 56-0 | Fax +49 (0)4102-69 56-66

info@spec.de



Spectrum Instrumentation Corp

401 Hackensack Ave, 4th Floor | Hackensack, NJ 07601 | USA

Phone +1 (201) 562-1999 | Fax +1 (201) 342-7598

sales@spectrum-instrumentation.com



spectrum-instrumentation.com