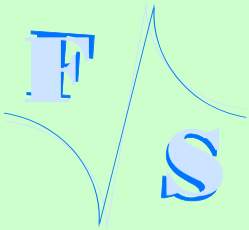
NSPI Driver

Native SPI Support

Version 3.1
(2013-03-13)

NetDCU
PicoMOD
PicoCOM

Windows CE



About This Document

This document describes how to install the Native SPI device driver (NSPI) and how to use it in own software applications. The driver is available for the PicoCOM, PicoMOD and Net-DCU series of boards from F&S under Windows Embedded CE. The latest version of this document can be found at http://www.fs-net.de.

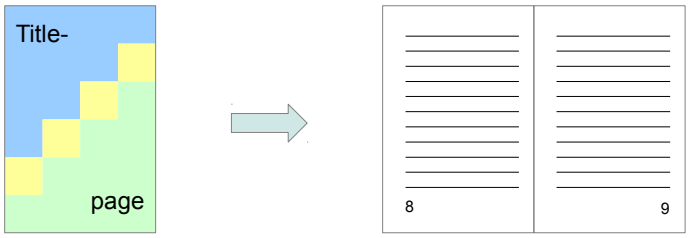
All available versions of the driver, V1.x, V2.x and V3.x are handled in this document.

Remark

The version number on the title page of this document is the version of the document. It is not directly related to the version number of the driver software described herein.

How To Print This Document

This document is designed to be printed double-sided (front and back) on A4 paper. If you want to read it with a PDF reader program, you should use a two-page layout where the title page is an extra single page. The settings are correct if the page numbers are at the outside of the pages, even pages on the left and odd pages on the right side. If it is reversed, then the title page is handled wrongly and is part of the first double-page instead of a single page.



Typographical Conventions

We use different fonts to emphasize the context of special terms:

`File names`

Menu entries

`Program code`

```
Listings
```

```
Board input/output
```


F&S Elektronik Systeme GmbH
Untere Waldplätze 23
D-70569 Stuttgart
Germany

Phone: +49(0)711-123722-0
Fax: +49(0)711-123722-99

History

Date	V	Platform	A,M,R	Chapter	Description	Au
2009-03-16	2.0		A, M		New document format, NSPI driver V2.0	HK
2011-11-08	V3.0		A, M	1-6, 8-9	New document format (ODF), NSPI driver V3.0, SPI explanation and introduction completely restructured	HK
2011-11-14	V3.1		M	3	PicoCOM4 added	MK
2013-03-13	V3.2		A	4.3	Supported Driver-Methods documented	MK

V Version
A,M,R Added, Modified, Removed
Au Author: CZ, DK, HF, HK, MK

Table of Contents

1 Introduction

SPI (Serial Peripheral Interface) is a bi-directional serial bus to connect a master device to one or more slave devices. The main communication takes place on a 1:1 basis as only one slave device is activated by a chip select signal. SPI can handle speeds of up to several megahertz.

F&S boards usually provide an SPI driver that uses General Purpose I/Os (GPIOs) for CLK, MISO, MOSI, and CS. This is rather flexible as most of the GPIOs provided by the board are supported and the user can decide which GPIO to use for each SPI signal. However as the SPI timing on this GPIO driver is done by software (“bit banging”), it uses quite a lot of CPU time and is still rather slow, at most several kilohertz.

Therefore F&S has developed another device driver that uses the native, dedicated SPI hardware available on all our micro controllers. This driver is called “Native SPI driver” or NSPI in short, and now actually allows for speeds of several megahertz.

The NSPI driver has now reached version 3.x. This document handles all three driver versions, V1.x, V2.x and V3.x. It does *not* handle the SPI over GPIO bit banging driver.

About this Document

After a short description of the SPI bus protocol, we'll introduce the Native SPI device driver available for the F&S Windows CE board families. We show how it is installed on the board, how it is configured, and how it is used in own applications by issuing transfer commands. We also show the difference between the available driver versions. The main part of the document is the application programming interface (API) reference that discusses all functions provided by the driver, including examples.

Remark

In this document, we will sometimes use the generic term “NetDCU” or simply “board” for all our boards and modules. It should also mean PicoMOD, PicoCOM or any other future board type or family, where the NSPI driver is available. We will also refer to the driver file generally as `nspi.dll`, even if the real name may have a board specific prefix added, e.g. `pm3_nspi.dll` on a PicoMOD3. We hope that this does not cause any inconvenience.

2 The Serial Peripheral Interface

2.1 Bus Topology

An SPI bus can be used to transfer data between a master device and one or more slave devices. It consists of the following signals:

1. A data line from the slave to the master, called MISO (Master In, Slave Out).
2. A data line from the master to the slave, called MOSI (Master Out, Slave In).
3. A clock signal CLK. The clock is always generated by the master.
4. A chip select signal CS generated by the master. The slave device reacts if CS goes low, otherwise it ignores everything that happens on the bus.

The simplest SPI bus therefore has four signal lines, where the slave drives one line (if selected) and the master the remaining three lines (all the time).

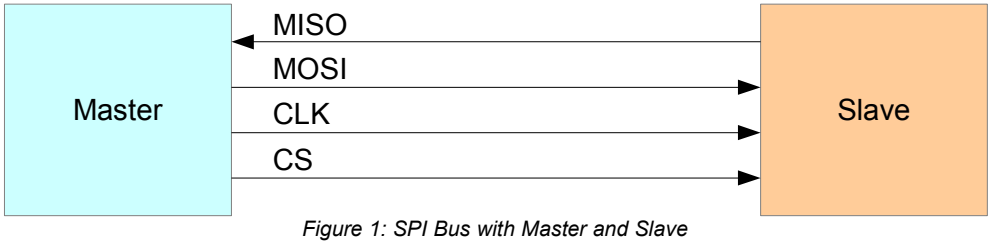


Figure 1: SPI Bus with Master and Slave

If more than one slave is connected to the bus, each slave has its own chip select signal, increasing the number of required lines by one per additional slave device. But notice that at most one chip select is allowed to be active at any time, so the communication still happens on a 1:1 basis between the master and the one slave selected by the active chip select signal. Of course only this slave is allowed to send data on the MISO line.

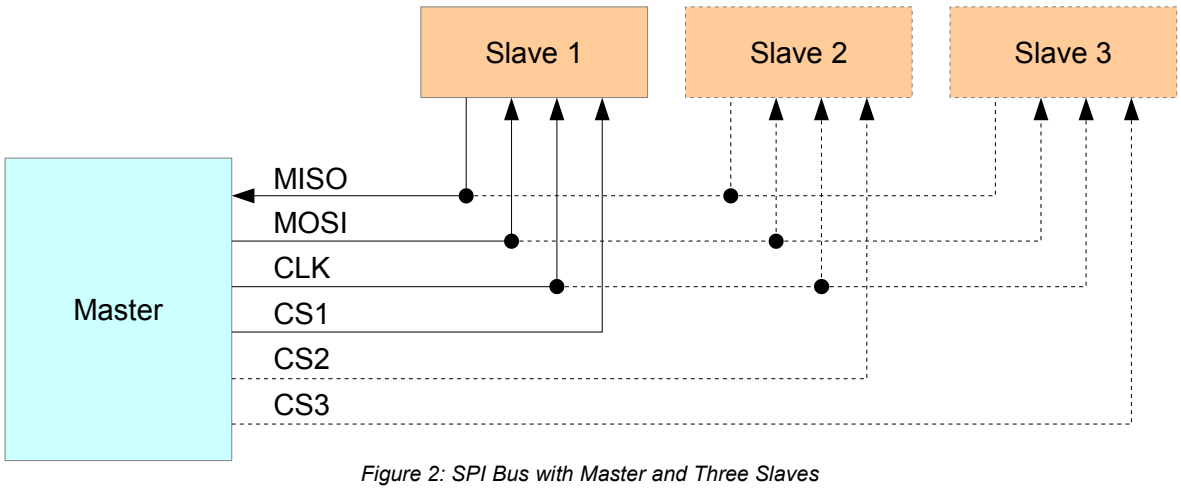


Figure 2: SPI Bus with Master and Three Slaves

2.2 SPI Commands

By design, the data transfer is bit-oriented. So theoretically it is possible to transfer in units of arbitrary length, for example 5 bits, 10 bits or even larger units like 300 bits or more. A 12-bit A/D converter for instance could return one converted value with every 12-bit SPI cycle. However nowadays most SPI devices handle data in multiple of bytes, i.e. 8 bits.

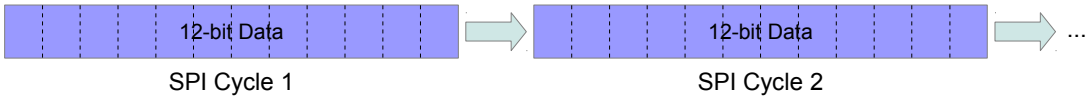


Figure 3: Arbitrary Number of Bits in SPI Cycle

Only very simple SPI devices have an SPI cycle where data is exchanged in the same way all the time. Most slaves are capable of doing different things. By sending a command, the master tells the slave what to do next. This command can consist of one or more bytes at the beginning of the SPI cycle. This command also determines whether there is further data and how the data is to be interpreted.

Figure 4: SPI Cycle with Command and Data Phase

For example the slave may be some memory device. Then the command could tell whether the memory should be read or written and it could also give a base address where to start. Assuming a 16-bit address, the command would consist of three bytes: one byte telling whether to read or write and two bytes for the address. The remaining bytes would be the data bytes to read or write. The transfer could have any length. Each new byte would access the next memory address. The transfer would end when the chip select is de-asserted.

Such devices actually exist. In Chapter 7 on Page 66 we have a sample program that accesses an FM25CL64 device. This is exactly such a memory device with 64KB of memory and it supports exactly such read and write commands. To prevent unintended memory modifications, this device also has an additional write-protect mechanism so that each write cycle must be preceded by a special write-enable cycle. And the device also has a status register that can be read and written with two additional commands. So in the end this SPI slave supports a set of five commands with different lengths and different transfer directions.

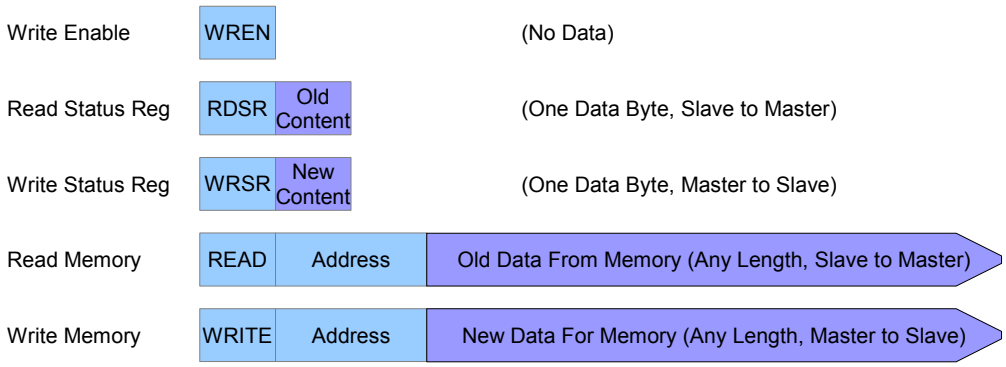


Figure 5: Sample Set of Slave Commands

Rather common are SPI devices that are based on registers. Again there is command and register address (register number). But as the set of registers is usually rather small, the register number can be encoded in the first command byte itself already. So these devices often use exactly one command byte.

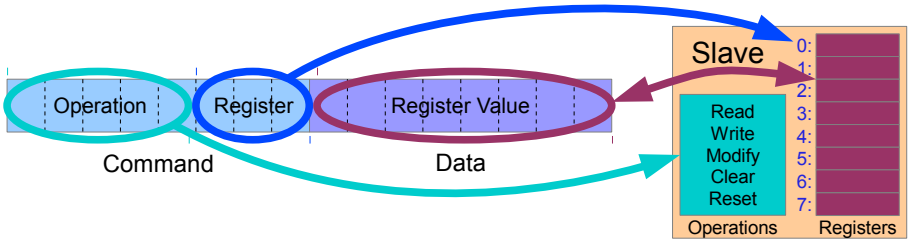


Figure 6: Register Based SPI Device

2.3 Transfer Direction

The SPI protocol is insofar interesting as it works bi-directional, i.e. always sends data in both directions at the same time. So with each clock cycle, one bit is sent on the MOSI line and at the same time one bit is received on the MISO line. If a transfer only receives, dummy bits must be sent on the MOSI line. If a transfer only sends, the received bits from the MISO line can be discarded.

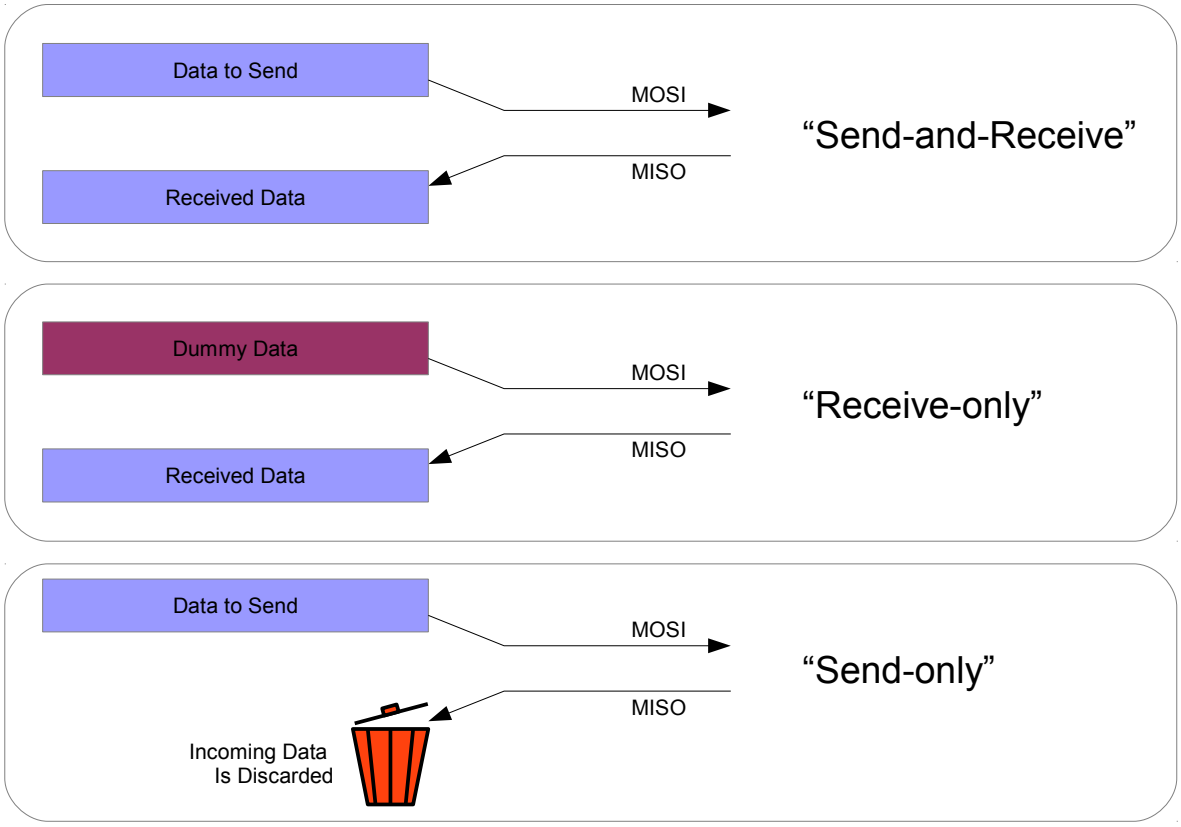


Figure 7: Different Transfer Directions

Switching the transfer direction within one SPI cycle is rather common. For example the command phase is always send-only, but the data phase can be any direction.

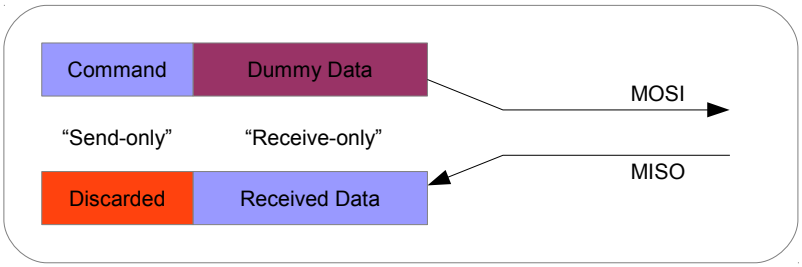


Figure 8: Change of Transfer Direction within an SPI Cycle

2.4 Protocol

It is the task of the device designer to provide a command set to allow efficient transfers. Using both directions as often as possible proves to be most efficient. For example in the memory device above, the MISO line is unused during memory writes. As an improvement, the WRITE command of this device could be modified in a way so that the old memory content is returned via the MISO line while the new data is sent, resulting in a READ-WRITE command. At least in some cases this improvement could avoid a separate read cycle before writing the new data. So the average performance of this device could be increased.

Figure 9: Improved Combined READ-WRITE Memory Command

As you can see from this memory device example, providing an SPI slave also requires setting up a command protocol. It is not possible to have a generic protocol that will work for all slaves. This is why the F&S Native SPI driver does not support being a slave. We simply can not define a protocol because we can't anticipate what function the board will serve in the customer's application in the end. Therefore the driver can act as an SPI master only.

2.5 SPI Mode

Unfortunately the clock signal CLK is not handled in the same way on all SPI devices. A bit cycle may start with CLK high or CLK low and the data may be latched on the falling or the rising edge of CLK. This results in four possible SPI modes (see Figure 10). At least there seems to be some agreement among device manufacturers how to number these modes.

- SPI Mode 0: CLK is active high, data is valid and latched on first (=rising) edge.
- SPI Mode 1: CLK is active high, data is valid and latched on second (=falling) edge.
- SPI Mode 2: CLK is active low, data is valid and latched on first (=falling) edge.
- SPI Mode 3: CLK is active low, data is valid and latched on second (=rising) edge.

Please note that SPI devices usually can not handle all modes. Most devices support only one or at most two modes. You have to check the specifications of the device to find out.

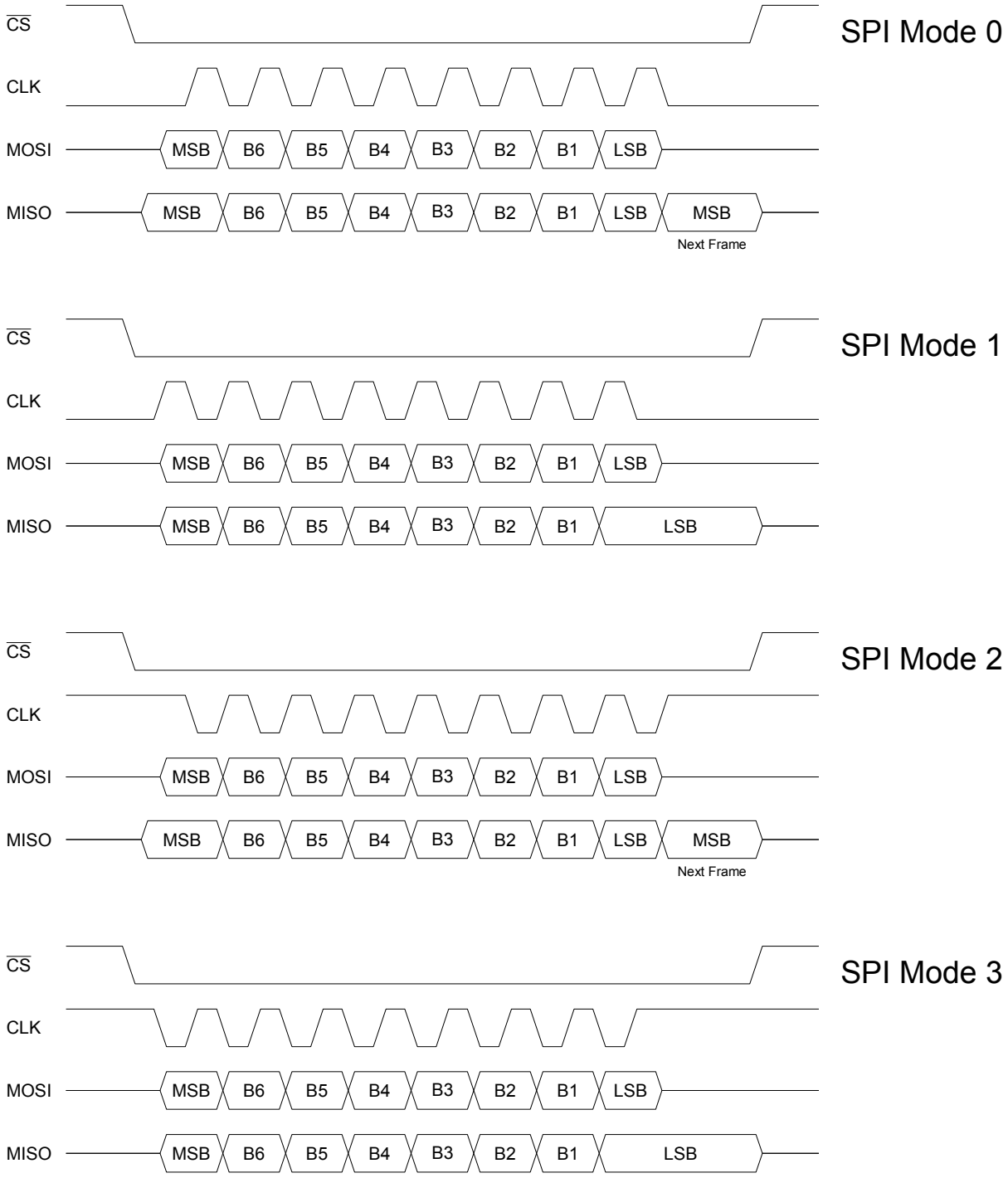


Figure 10: SPI Modes

2.6 Data Delay

Some SPI slaves need some time after the command to prepare the result, for example to read some internal register. When accessing such a slave, the master must insert a short pause after the command phase before continuing with the data phase.

Figure 11: Data Delay

2.7 Interrupt-Driven Communication

Some SPI devices provide features that can happen asynchonously to the normal workflow. These devices have an additional interrupt request signal IRQ to tell the master of this external event. For example an SPI device providing additional external I/Os could inform the master of a state change of one of the input lines. Or an A/D converter can inform the master of a completed complex conversion.

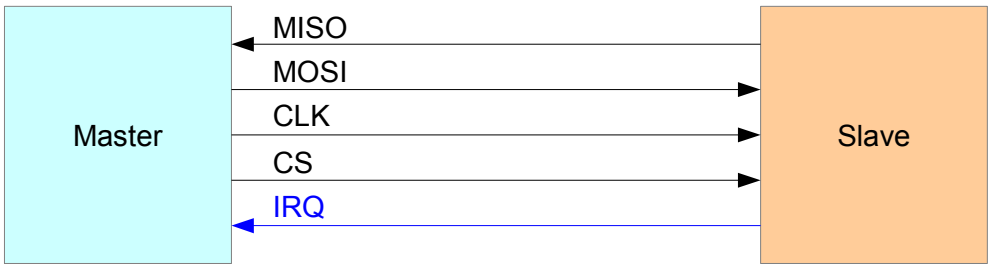


Figure 12: Interrupt Request Line

After asserting the IRQ line, the master should start an SPI cycle as soon as possible to react to this event. In our example, the master would either check the I/O line states or fetch the conversion result then.

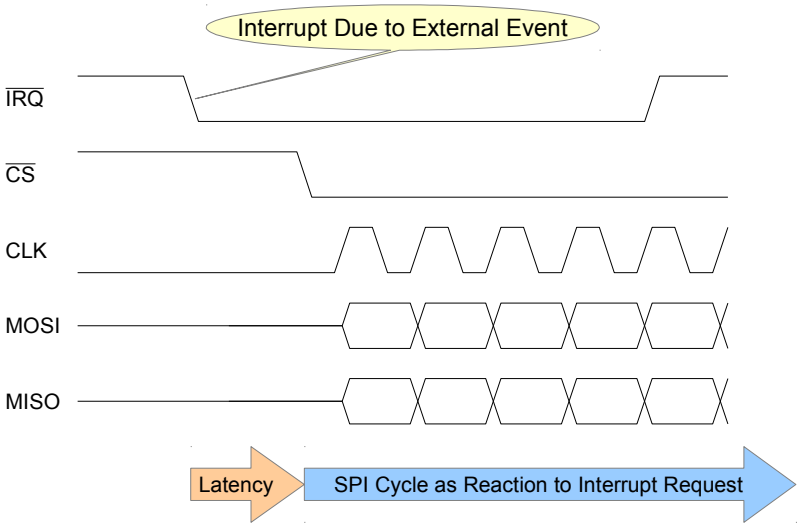


Figure 13: SPI Cycle after IRQ

2.8 Multiple Chip Selects

As an SPI device should ignore the bus if it is not selected by the CS signal, it does not matter what protocol other devices work with. It is perfectly OK that one device on the bus runs at one speed, mode and command set while another device runs at a different speed, mode and command set. The master just has to switch to the appropriate settings before accessing each device, i.e. before asserting the chip select.

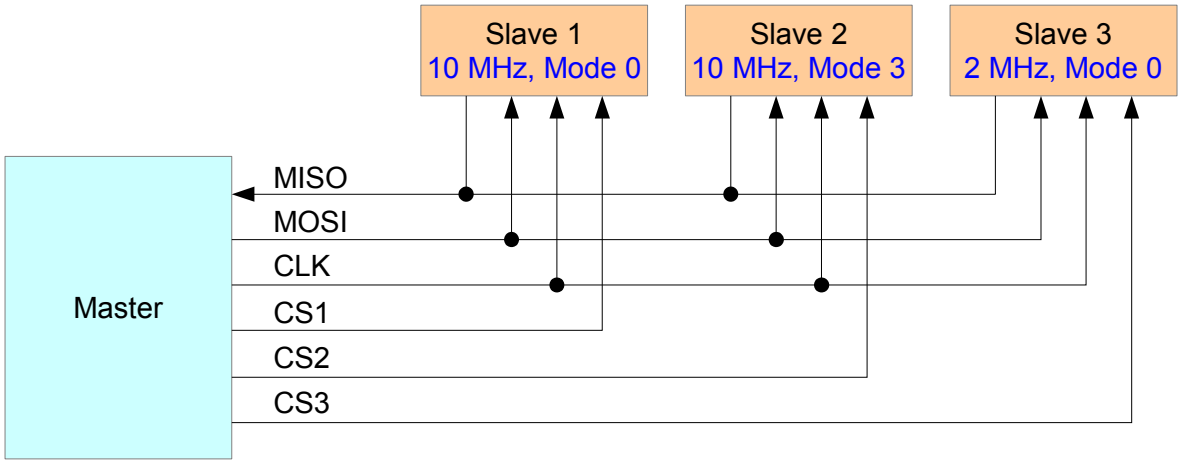


Figure 14: One SPI Bus with Arbitrary Devices

This allows combining arbitrary devices on one bus. The only drawback is that the bus is blocked for all other devices while one device is serviced, probably causing unwanted contentions.

3 The F&S Native SPI Driver

3.1 Driver Versions

The NSPI driver has now reached version 3.x. Each new major version represents major feature improvements that also affect the driver interface. Each minor version represents bug fixes or small improvements that usually do not change the driver interface. Here is a short overview of the features provided by each major version.

Features of V1.x

- Only one slave device supported (one chip select). The F&S board is always the master.
- Transfers can be handled with `DeviceIoControl()` only.
- Four different transfer commands: send-only, receive-only, send-and-receive with different buffers, send-and-receive with one buffer (incoming data replaces outgoing data).
- All settings have to be done via the registry, not changeable at runtime.

Features added in V2.0 (March 2009)

- Transfers with `ReadFile()` and `WriteFile()` to support the `Stream` class in `.NET`.
- Support for more than one SPI controller.
- Possibility to query and modify the driver method, bus speed and SPI mode at runtime.
- Possibility to determine driver version at runtime.

Features added in V3.0 (October 2011)

- Support for more than one chip select signal per SPI controller. Any I/O pin can be used.
- Additional transfer commands that wait for an interrupt before issuing the SPI cycle. This allows to react more promptly to an incoming signal. Any interrupt-capable I/O pin can be used. Interrupt type (edge or level) and timeout can be configured.
- The dummy byte that is sent in receive-only transfers can be configured.
- The driver can issue a short delay between command and data bytes.
- The general NSPI driver timeout can be configured.

Which driver version you have can be determined by looking at the debug output of the board. The first line starting with `NSPI:` shows the appropriate driver version. In V2.0, there was also an IOCTL command code added called `IOCTL_DRIVER_GETINFO`. If this command succeeds, it returns the appropriate driver version in a data structure. If it fails, it is a V1.x driver.

3.2 Possible interface conflicts

Please note that on some devices, the native SPI interface also might be used internally. Hence you might need to keep this into account for proper function of SPI interface. Here is a list of known issues:

Board	Competing interface
PicoCOM4	CAN interface

Table 1: List of possible mutual interferences

Please refer to the corresponding documentation for details on how to handle this mutual interference.

3.3 Pin Assignment

With the Native SPI driver, you are not free to choose the pins to use like with the GPIO SPI driver. Instead they are given by the SPI hardware on the board. The following table shows the dedicated SPI lines on the different boards. On the PicoMOD modules, we give the pin number of the module connector itself and of the connector on the starter interface board.

Board	Connector	MISO	MOSI	CLK	CS
NetDCU5.2	J5	Pin 10	Pin 11	Pin 15	Pin 13
NetDCU8	J5	Pin 4	Pin 3	Pin 2	Pin 6
NetDCU9	J5	Pin 10	Pin 11	Pin 15	Pin 13
NetDCU10	J5	Pin 4	Pin 3	Pin 2	Pin 6
NetDCU11	J5	Pin 10	Pin 11	Pin 15	Pin 13
PicoMOD3 Module PicoMOD3 Startinterface	J1 J5	Pin 3 Pin 4	Pin 4 Pin 3	Pin 2 Pin 2	Pin 1 Pin 6
PicoMOD6 Module PicoMOD6 Startinterface	J1 J5	Pin 3 Pin 4	Pin 4 Pin 3	Pin 2 Pin 2	Pin 1 Pin 6

PicoCOM4 Module PicoCOM4 Startinterface	J1 J10	Pin 26 Pin 3	Pin 27 Pin 4	Pin 28 Pin 5	Pin 29 Pin 6

Table 2: Pin Assignment of SPI Signals

You can use this driver in combination with the GPIO SPI driver, if both drivers are available on the platform. But please make sure that the other driver is not configured to use the above pins or otherwise the drivers will get into conflict.

Remark

As already mentioned, starting with V3.0, you can use more than one chip select signal to drive different SPI devices on one SPI bus. You can use any arbitrary I/O signal as chip select that is not occupied with other functions. But notice that MISO, MOSI and CLK are still fixed to the above pins, and if not specified, the driver still uses the above CS pin as default.

4 Installing the NSPI Driver

The NSPI driver is usually installed as `SPI1:`. We provide a special Windows Cabinet File ("CAB-File") for an automatic installation, but you can also do the installation manually.

4.1 Installation with the CAB file

The easiest way to install the driver is to use the provided Windows Cabinet File `nspi.cab`. Just copy this file to the board (e.g. to the root directory) and double click on it. This will automatically install the driver as `SPI1:`. When asked for a destination directory, just click *OK*. All registry settings will be done for the default values and the CAB file will vanish again when done.

If you don't have access to a mouse or touch panel on the NetDCU, or if you even don't use a display at all, you can also do the CAB file installation on the command line. Just type the following command:

```
wceload /noui nspi.cab
```

If you need settings other than the defaults, you can edit the registry values anytime after installation is complete.

4.2 Manual installation

You can also do the installation by hand. To do this you first have to store the library file `nspi.dll` in flash memory into the `\FFSDISK` directory, if it is not already pre-loaded in the kernel. Then you have to set some registry values under registry key

`[HKLM\Drivers\BuiltIn\SPI`*n*`]`

where *n* is the SPI device to create. The possible values are described in Chapter 4.3 on Page 13.

Starting with driver version 2.0 it is also possible to configure some more controller specific values. This takes place under the registry key

`[HKLM\Drivers\SPIController`*X*`]`

where *X* is the zero-based index of the SPI controller (`0` if only one controller is available). The possible values are described in Chapter 4.4 on Page 18.

On some boards it is possible to have more than one SPI bus. Then you have to select the correct controller for each device that you define. Each controller will handle one SPI bus.

Starting with V3.0, you can use different chip select signals for each SPI controller to drive more than one device on the SPI bus. Then you have to create one such SPI key under `BuiltIn` for each chip select and define different indexes and different I/O pins to use as chip selects. Each device will then be available with its own `SPI`*n*`:` device name.

4.3 Registry Values in [HKLM\Drivers\BuiltIn\SPIn]

The following values can be configured in registry key

`[HKLM\Drivers\BuiltIn\SPIn]`

where *n* is the SPI device to create. Most of the values will get meaningful defaults if omitted, only those values in the first five rows highlighted in blue with shaded background really have to be given.

Entry	Type	Value	Description
Dll	*String*	*nspi.dll*	*Driver DLL*
FriendlyName	*String*	*Native SPI driver*	*Description*
Prefix	*String*	*SPI*	*For SPI1:*
Index	*DWORD*	*1*	*For SPI1:*
Order	*DWORD*	*101*	*Load sequence*
ClockFreq	DWORD	200000	in Hz
SPIMode	DWORD	0	SPI mode
DriverMethod	DWORD	0	IRQ, Polling, DMA
Priority256	DWORD	103	Thread priority
SPIController	DWORD	0	Index of SPI controller[1)]
DummyByte	DWORD	255	Byte sent in receive-only transfers[1)]
DataDelay	DWORD	0	Delay between command and data bytes (in ns!)[2)]
CsPin	DWORD	(depends on board)	Pin number to use for chip select[2)]
GenTimeout	DWORD	1000	Timeout for SPI functions (in ms)[2)]
IrqPin	DWORD	0xFFFFFFFF	Pin for WAITIRQ interrupt input[2)]
IrqCfg	DWORD	2	WAITIRQ edge/level configuration[2)]
IrqTimeout	DWORD	1000	WAITIRQ timeout (in ms)[2)]
Debug	DWORD	0	Debug verbosity

1) Introduced in V2.0 2) Introduced in V3.0

Table 3: NSPI Registry Values

The detailed meaning of these values is explained in the following sub-chapters.

4.3.1 ClockFreq

`ClockFreq` defines the frequency of the clock signal CLK. The clock is derived from a base frequency by dividing by an integer. Therefore not all frequencies can be achieved exactly. But for any given `ClockFreq`, the driver will automatically choose the divider for the nearest possible frequency.

Board	Base Frequency	Possible Dividers	Minimum Frequency	Maximum Frequency
NetDCU5.2	1.84 MHz	1 to 256	7.2 kHz	1.84 MHz
NetDCU8	25.5 MHz	1 to 256	99.6 kHz	25.5 MHz
NetDCU9	13 MHz	1 to 4096	3.174 kHz	13 MHz
NetDCU10	25.5 MHz	1 to 256	99.6 kHz	25.5 MHz
NetDCU11	13 MHz	1 to 4096	3.174 kHz	13 MHz
PicoMOD3	66.5 MHz	2 to 512	130 kHz	33.25 MHz
PicoMOD6	66.5 MHz	2 to 512	130 kHz	33.25 MHz
PicoCOM4	66.5 MHz	2 to 512	130 kHz	33.25 MHz

Table 4: NSPI Frequency Range

This value can also be modifed at runtime with `IOCTL_NSPI_SET_CLKFREQ` (only V2.0 and newer).

4.3.2 SPIMode

Entry `SPIMode` defines the active mode (polarity) and the active edge (phase) of the clock signal.

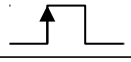
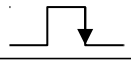
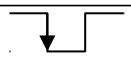
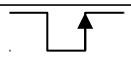
SPIMode	Description	Signal
0	Clock active high, data valid on 1st (=rising) edge	
1	Clock active high, data valid on 2nd (=falling) edge	
2	Clock active low, data valid on 1st (=falling) edge	
3	Clock active low, data valid on 2nd (=rising) edge	

Table 5: SPI Mode (Polarity and Phase)

This value can also be modified at runtime with `IOCTL_NSPI_SET_MODE` (only V2.0 and newer).

4.3.3 DriverMethod

Entry `DriverMethod` determines the method how bytes are transmitted. This has an influence on the speed, the latency and how much CPU load the driver takes.

DriverMethod	Method	Function
0	IRQ	The driver issues a byte to the SPI, then goes to sleep. After the byte is transmitted, the driver is woken up again by an SPI interrupt request to transfer the next byte.
1	Polling	The driver issues a byte to the SPI, then waits in a busy loop, polling the “Transmit-complete” flag. When the transmission is done, it issues the next byte.
2	DMA	The driver sets up the DMA-Controller to handle the transmission by direct memory access to the SPI. Then it goes to sleep and is woken up again by a DMA interrupt request when the transmission is completely done.

Table 6: NSPI Transfer Methods

This value can also be modified at runtime with `IOCTL_NSPI_SET_METHOD` (only V2.0 and newer).

Remark

`DriverMethod` is currently not implemented on the NetDCU5.2. This board automatically uses an IRQ mode combined with the hardware FIFO available on the SPI there. The NetDCU9, NetDCU11, PicoMOD3 and PicoMOD6 do support the `DriverMethod` entry, but they currently only support IRQ and Polling, not DMA.

Board	IRQ-Method (0)	POLLING-Method (1)	DMA-Method (2)
NetDCU5.2	✓	×	×
NetDCU8/9/10/11	✓	✓	×
PicoMOD3/4/6	✓	✓	×
PicoCOM4	✓	×	×

Table 7: Supporter Driver-Methods

4.3.4 Priority256

The actual transfer will take place with the Windows CE priority given in `Priority256`. Changing this value is only required if the NSPI driver does interfere with other drivers. A lower value means higher priority, a higher value means lower priority. The region is `0` to `255`.

Attention:

A value too small (= very high priority) may block other device drivers, resulting in sporadic malfunctions.

4.3.5 SPIController

This entry tells the driver which SPI controller to use. The appropriate controller information will be found under `[HKLM\Drivers\SPIControllerX]` (see Chapter 4.4 on Page 18).

4.3.6 DummyByte

This entry tells the driver what byte value to send in receive-only transfers. SPI always is a bi-directional transmission. That means for each bit that is received also one bit must be sent. In phases where only data is received, dummy bits must be sent. As the transfer with this driver is byte-based, we have to give a byte value here. V1.x and V2.x always used the constant value `0xFF`. Starting with V3.0, this value can be configured. This allows to set a different dummy value if an SPI device does interpret `0xFF` in an undesired way.

This value can also be modified at runtime with `IOCTL_NSPI_SET_DUMMYBYTE`.

4.3.7 DataDelay

Some SPI devices need some time after they receive the command until they can return the answer. So it is required that the master pauses for a short period of time before continuing the SPI cycle. Starting with V3.0, such a delay can be configured with this entry. The value is given in nanoseconds (!) and the delay is internally done by a busy-wait loop. Please note that even if the timing can be given in nanoseconds, reasonable values are in the micro-seconds region. This is due to implementation restrictions.

This value can also be modified at runtime with `IOCTL_NSPI_SET_DATADELAY`.

Remark

When using the DMA driver method, the `DataDelay` value is ignored. This method always combines command and data phase to one single DMA transfer that allows no pauses.

4.3.8 CsPin

Starting with V3.0, the chip select pin can be configured freely among all available I/O pins of the board. This is done by giving the pin number of the I/O pin to use here. Please refer to the Device Driver Documentation of your board for the possible values here. It is basically the same value that you would use when setting a pin value with `IOCTL_DIO_SET_PIN` in the DIO driver.

What is even more, you can use more than one chip select signal on an SPI bus. This is done by activating several instances of the NSPI driver in the registry where each instance uses a different `CsPin` value. You can even use different settings for all the other entries. The driver will automatically switch to the correct settings when talking to the device on this chip select. This explained in more detail in Chapter 4.5 on Page 20.

V1.x and V2.x only can use the one chip select that is given in 2 on Page 11.

4.3.9 GenTimeout

Some calls to the NSPI driver may block, for example because another thread already uses the SPI controller. By setting this value, you can tell the driver how long it should wait in such blocking situations until it gives up and returns with an `ERROR_TIMEOUT` error. This entry was added in V3.0. The older versions wait indefinitely.

This value can also be modified at runtime with `IOCTL_NSPI_SET_GENTIMEOUT`.

4.3.10 IrqPin

In V3.0, new transfer commands were introduced that wait for an interrupt before issuing the SPI cycle. By setting this value, you define what I/O pin to use as interrupt input. Any interrupt capable I/O pin of the board can be used. Please refer to the Device Driver Documentation of your board for the possible values here. It is basically the same value that you would use when requesting an interrupt in the DIO driver with `IOCTL_DIO_REQUEST_SYSINTR`.

V1.x and V2.x do not support this interrupt feature.

4.3.11 IrqCfg

This value defines the type of interrupt that must happen on the `IrqPin`. The possible values are shown in this table. They correspond to the binary value built by registry values `IRQCfg2` to `IRQCfg0` in the DIO driver.

IrqCfg	Interrupt Type
1	Rising edge
2	Falling edge
3	Rising and falling edge
5	High level
6	Low Level

Table 8: Possible Interrupt Types

4.3.12 IrqTimeout

This value defines how long the driver will wait at most for the interrupt on `IrqPin` before it gives up and returns with `WAIT_TIMEOUT` error. This `IrqTimeout` is independent from `GenTimeout` and only is valid when waiting for an interrupt with one of the `WAITIRQ` transfer functions.

This value can also be modified at runtime with `IOCTL_NSPI_SET_IRQTIMEOUT`.

4.3.13 Debug

If the `Debug` entry is set to a value different to zero, the driver will output additional information on the debug port. Each bit enables a different category of output. This information is usually not required and only necessary when looking for errors in the driver. Keep this value at zero to have the best possible performance.

4.4 Registry Values in [HKLM\Drivers\SPIControllerX]

Starting with driver version 2.0 it is also possible to configure some more controller specific values. This takes place under the registry key

`[HKLM\Drivers\SPIController`*X*`]`

where *X* is the zero-based index of the SPI controller (`0` if only one controller is available).

Entry	Type	Value	Description
`DmaBufferSize`	DWORD	4096	Size of the internal DMA buffer[1)]
`DmaTxChannel`	DWORD	3	DMA channel to use for sending[1)]
`DmaRxChannel`	DWORD	4	DMA channel to use for receiving[1)]
`DmaTriggerLevel`	DWORD	16	FIFO level required to trigger DMA request[1)]

IrqTriggerLevel	DWORD	16	FIFO level required to trigger IRQ[1)]
ThreadSync	DWORD	1	Use internal thread synchronization[2)]

1) Introduced in V2.0 2) Introduced in V3.0

Table 9: NSPI Registry Settings for the SPI Controller

The detailed meaning of these values is explained in the following sub-chapters.

4.4.1 DmaBufferSize

This entry is only valid in DMA mode. It determines the size of the internal buffer. To use DMA, the data has to be copied to and from a special consecutive uncached buffer that can also be accessed by the DMA controller hardware. Please note that this increases the transfer time in DMA mode. Default is to use one memory page, i.e. 4096 bytes.

4.4.2 DmaTxChannel and DmaRxChannel

These entries are only valid in DMA mode. They determine which DMA channel to use for sending (DmaTxChannel) and receiving (DmaRxChannel). Which channels are available depends on the other hardware in use. For example on the PicoMOD3, there are 32 channels available, but when using audio, channels 1 and 2 are already occupied.

The driver always uses both channels, even if you only send or only receive data with your call.

The NetDCU8 and NetDCU10 only support send-only or receive-only transfers via DMA and they use DMA channel 3 by default. This is a hardware restriction an can not be changed.

4.4.3 DmaTriggerLevel

This value is only valid when using DMA driver method on boards having a send and receive FIFO embedded in the SPI controller hardware. It defines how many bytes must have been received in the receive FIFO to trigger the next DMA request. Setting a small value increases DMA overhead as the FIFO is not used efficiently. Setting a too high value may cause a FIFO overrun if the DMA can not be handled fast enough due to DMA priorities. Default is to use half of the FIFO size.

4.4.4 IrqTriggerLevel

This value is valid only when using IRQ driver method on boards having a send and receive FIFO embedded in the SPI controller hardware. It defines how many bytes must have been received in the receive FIFO to trigger the next interrupt request. Setting a small value increases interrupt overhead as the FIFO is not used efficiently. Setting a too high value may

cause a FIFO overrun if the interrupt can not be handled fast enough due to thread priorities. Default is to use half of the FIFO size.

4.4.5 ThreadSync

In V1.x and V2.x, all thread synchronisation takes place within the driver. Starting with V3.0, you can tell the driver that you do not need this internal thread synchronisation. This may be the case if your application only uses one single thread to access the NSPI driver or that you handle all synchronisation yourself within the application.

ThreadSync	Meaning
0	Don't use any thread synchronisation
1	Use internal thread synchronisation

Table 10: Thread Synchronisation Values

This is just an optimisation setting. If no internal synchronisation is required, the driver will run a little bit faster. This may be of interest if you are using the `WAITIRQ` functions and want the shortest possible response time to the interrupt. Disabling internal synchronisation may save up to 20 µs then, depending on the board.

4.5 Using Different Chip Selects

Starting with V3.0, the NSPI driver is capable of driving different chip select lines on each SPI bus. This is done by configuring a separate SPI device for each chip select in the registry. Each of these devices uses a different value for the `CsPin` entry, but the same value for the `SPIController` entry (= same SPI bus).

Example

The SPI bus on controller 0 should serve three chip selects:

Device	I/O Pin	Speed	SPI Mode
`SPI1:`	Default	10 MHz	0
`SPI2:`	14	10 MHz	3
`SPI3:`	15	2 MHz	0

Table 11: Different Chip Selects

This would result in the following registry entries:

In `[HKLM\Drivers\BuiltIn\SPI1]`:

Entry	Type	Value
Dll	String	nspi.dll
FriendlyName	String	Native SPI, CS1
Prefix	String	SPI
Index	DWORD	1
Order	DWORD	101
ClockFreq	DWORD	10000000
SPIMode	DWORD	0
SPIController	DWORD	0

In `[HKLM\Drivers\BuiltIn\SPI2]`:

Entry	Type	Value
Dll	String	nspi.dll
FriendlyName	String	Native SPI, CS2
Prefix	String	SPI
Index	DWORD	2
Order	DWORD	102
ClockFreq	DWORD	10000000
SPIMode	DWORD	3
SPIController	DWORD	0
CsPin	DWORD	14

In `[HKLM\Drivers\BuiltIn\SPI3]`:

Entry	Type	Value
Dll	String	nspi.dll
FriendlyName	String	Native SPI, CS3
Prefix	String	SPI
Index	DWORD	3
Order	DWORD	103
ClockFreq	DWORD	2000000
SPIMode	DWORD	0
SPIController	DWORD	0
CsPin	DWORD	15

Each device holds its own set of driver settings. So not only speed and SPI mode can be different on different chip selects, but all settings under `[HKLM/Drivers/BuiltIn/SPI`*n*`]` can be configured individually. The driver will automatically switch to the appropriate settings before asserting one of the chip selects.

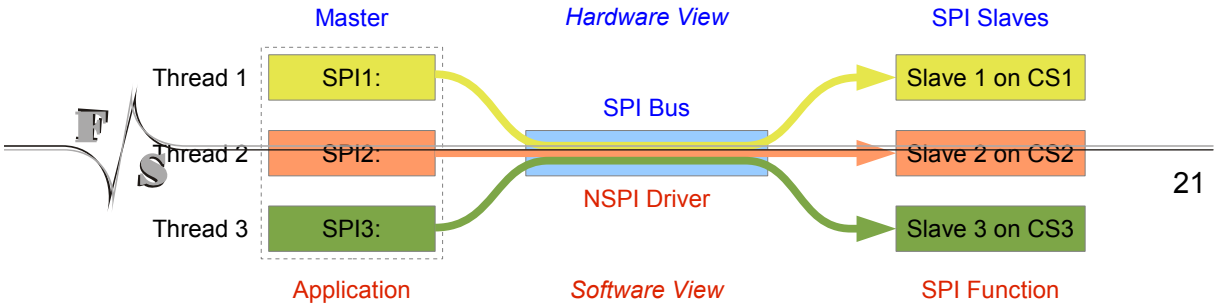


Figure 15: Virtual Connection Between Devices and Slaves

Now if the software wants to talk to the SPI slave with CS1, it has to use device `SPI1:`. Accordingly if it wants to talk to CS2, it has to use `SPI2:`, and if it wants to talk to CS3, it has to use `SPI3:`. In fact these devices could also be used by different threads or even processes. The NSPI driver automatically serialises the accesses to the SPI bus in a first come first served manner. So from the view of the software, the SPI bus is more or less invisible, and it looks as if it can talk directly to the slave via a virtual direct connection just by using the appropriate device.

The only thing to consider are contention issues that may arise if different threads want to access the SPI bus at the same time.

4.6 Choosing the Driver Method

The SPI transfer speed is not only determined by the clock frequency. There are also latencies at the beginning and the end of each transfer and even between the bytes, reducing the average data rate.

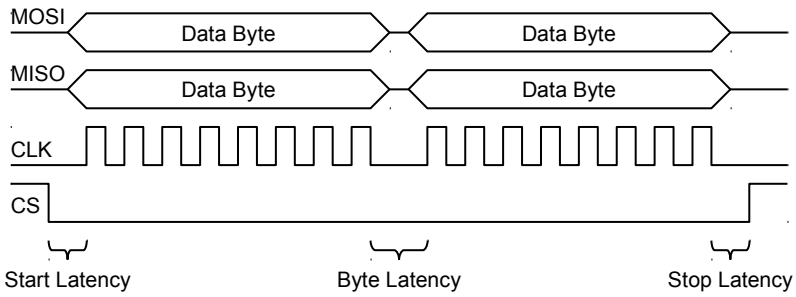


Figure 16: SPI Latencies

The start and stop latencies are implicitly given by the overhead of the driver to set up and shut down the SPI and to toggle the CS signal. The byte transfer itself is done by the SPI hardware at full speed. But the latency between the bytes depends largely on the software driver method.

For example on one hand, when polling the completion flag, the driver can issue the next byte way faster than by reacting to an interrupt request in IRQ mode, which might involve a Windows task switch. But on the other hand, the busy-wait of the Polling method blocks the CPU completely while the IRQ method allows the driver to go to sleep, freeing the CPU to work on other tasks in the meantime. Therefore by choosing between Polling and IRQ method, you choose between low byte latency and low CPU load.

Using the DMA method seems to solve this dilemma, as it has a very low byte latency and does not need any help of the driver at all during the whole transmission. However there are other restrictions here, too.

- Not enough DMA channels available. For example the NetDCU8 only has four DMA channels. Due to hardware restrictions, SPI can only be done using DMA channel 3. Depending on your software configuration and hardware periphery needs, this chan-

nel may already be occupied by another high speed device driver, e.g. USB-Device or SD-Card.

- DMA may have restricted access. For example DMA on the NetDCU8 does only support send-only and receive-only transfers, not any combined versions. The driver will return an error when trying to call these combined functions.
- When sending/receiving via DMA, the data bytes have to be copied to/from an internal buffer. This buffer is restricted in size (currently 4096 bytes) and the copying takes additional time, increasing the start/stop latency even more.
- The additional overhead for setting up the DMA controller registers adds quite a lot to the start and stop latencies. Therefore DMA is not well suited for short communications but gains on communications with several hundred bytes in one go.

We have done some timing measurements on a NetDCU8 (300 MHz) by reading from an SPI memory chip at a rather high clock speed of 8.5 MHz. This would theoretically be the nominal data rate of 8.5 Mbit/s. The reading sequence was to send three command bytes and then read an arbitrary number of data bytes. We tested for 100 and 1000 data bytes and measured the time for the whole transmission (from CS low to CS high again) and the byte latency between two adjacent bytes. Then we computed the effective data rate for this transmission.

The remaining latency, which consists of the start and stop latencies as well as any other delays during transmission, was computed from these numbers, too, and is not very accurate, just a hint of the magnitude.

Send 3 bytes, receive 100 bytes @ 8.5 MHz:

Driver Method	Measured Time	Effective Data Rate	Byte Latency	Remaining Latency
IRQ	750 µs	1.1 Mbit/s	5500 ns	106 µs
Polling	210 µs	3.8 Mbit/s	910 ns	25 µs
DMA	258 µs	3.1 Mbit/s	150 ns	149 µs

Table 12: Transfer times with short transmissions

Send 3 bytes, receive 1000 bytes @ 8.5 MHz:

Driver Method	Measured Time	Effective Data Rate	Byte Latency	Remaining Latency
IRQ	6550 µs	1.2 Mbit/s	5500 ns	109 µs
Polling	1880 µs	4.3 Mbit/s	910 ns	29 µs
DMA	1250 µs	6.4 Mbit/s	150 ns	159 µs

Table 13: Transfer times with long transmissions

As we can see, DMA benefits from long transmissions, and IRQ is very slow at high data rates. Nonetheless, IRQ is very well suited for low SPI speeds, when the transmission of every single byte takes long enough so that switching tasks in the meantime really makes sense. Here, Polling has a big disadvantage as it takes all CPU time when waiting for every byte transmission.

However Polling gains with high SPI speeds. If the transmitted messages are short, the total transmission time is low compared to the overhead involved with DMA and IRQ, making up for the busy wait involved. Or put in other words: when DMA and IRQ are still actively setting up everything for transmission (start latency), Polling has already transferred the whole message, taking less CPU time than the other two methods.

This results in the following recommendations:

- For low data rates use DMA or IRQ.
- For high data rates and long transmissions use DMA.
- For high data rates and short transmissions use Polling.

This is only true for boards without a FIFO in the SPI hardware. If a FIFO is available, IRQ method approves to be rather universally suited: it fills the FIFO (like in polling mode) and only waits for an interrupt if no more bytes fit into the FIFO. The `IrqTriggerLevel` can be set to a level that starts the IRQ before the FIFO empties. So we have low latency like in the Polling method, but also low CPU load like always with IRQ method. This works in almost all situations.

Starting with Version 2.0 of the NSPI driver, it is possible to switch the driver method at runtime. This allows to choose the best method individually for the following transfers.

5 The NSPI Driver in Applications

The F&S NSPI driver is designed to support most features available on SPI buses as described in Chapter 2. However the following points must be noted.

- The driver can only work as a master. Being a slave would involve defining a transfer protocol, which is not possible without knowing the purpose of the board in advance.
- Transfer is byte oriented. All values are given as bytes or multiple of bytes.
- Send-and-receive transmissions are further split into *Transfers* and *Exchanges*. Transfers send data from one place and receive data to another place. Exchanges receive data to the same place as the data was sent. So the old data is replaced (exchanged) with the new data.
- Modifying speed, SPI mode and driver method at runtime is possible since V2.0.
- Multiple chip selects are possible since V3.0. Each chip select is handled by a separate SPI device.
- Interrupt-driven communication is possible since V3.0.
- A data delay can be defined since V3.0.
- Defining a dummy byte other than `0xFF` is possible since V3.0.

As a master, MOSI, CLK and CS are output signals, MISO (and probably IRQ) are input signals. The F&S board will generate the clock and chip select signals.

With the introduction of Transfers and Exchanges, we have the following transmission commands.

Transmission	Description
Send-only	Meaningful data is only transferred via the MOSI line. The received bytes are discarded.
Receive-only	Meaningful data is only transferred via the MISO line. The data sent on the MOSI line is ignored by the device and does not matter. The NSPI driver sends the `DummyByte` value in this case (usually `0xFF`).
Transfer (Send-and-Receive)	Both directions carry meaningful data. The received data is stored at a *place different* to the sent data.
Exchange (Send-and-Receive)	Both directions carry meaningful data. The received data is stored at the *same* place as the sent data, replacing it.

Table 14: SPI Transmission types

The driver uses the common file interface (stream interface) to talk to the SPI bus. This means devices are opened with `CreateFile()`, closed with `CloseHandle()`, and communication is done with `ReadFile()`, `WriteFile()` and `DeviceIoControl()`.

6 NSPI Reference

6.1 CreateFile()

Signature

```
HANDLE CreateFile(
      LPCTSTR lpFileName, DWORD dwAccess, DWORD dwShareMode,
      LPSECURITY_ATTRIBUTES lpSecurity, DWORD dwCreation,
      DWORD dwFlags, HANDLE hTemplate
);
```

Parameters

`lpFileName`.....Device file name, usually `SPI1:`
`dwAccess`.....Device access (see below)
`dwShareMode`.....File share mode (see below)
`lpSecurity`.....Ignored, set to `NULL`
`dwCreation`.....Set to `OPEN_EXISTING`
`dwFlags`.....Set to `FILE_FLAG_WRITE_THROUGH`
`hTemplate`.....Ignored, set to `0`

Device access `dwAccess`

`0`.....Device query mode
`GENERIC_READ`.....Open device file read-only (receive)
`GENERIC_WRITE`.....Open device file write-only (send)
`GENERIC_READ | GENERIC_WRITE`
Open device file in read-write mode (send & receive)

File share mode `dwShareMode`

`FILE_SHARE_READ`.....Subsequent open operations succeed only if read access
`FILE_SHARE_WRITE`.....Subsequent open operations succeed only if write access

Return

`INVALID_HANDLE_VALUE`.....Failure, see `GetLastError()` for details
Otherwise.....File handle

Description

Opens the `SPI`*`n`*`:` device file for access. This is required for all other functions using this SPI bus. If the file handle is not required anymore, you have to call function `CloseHandle()`.

Example

```
HANDLE hSpi;

hSpi = CreateFile(_T("SPI1:"), GENERIC_READ | GENERIC_WRITE, 0,
                  NULL, OPEN_EXISTING, FILE_FLAG_WRITE_THROUGH, 0);
if (hSpi == INVALID_HANDLE_VALUE)
{
     DWORD dwError = GetLastError();

     /* Handle error in dwError */
}
```

Listing 1: Example CreateFile()

6.2 WriteFile()

Signature

```
BOOL WriteFile(
      HANDLE hDevice, LPCVOID lpBuffer, DWORD dwLen,
      LPDWORD dwActuallySent, LPOVERLAPPED lpOverlapped
);
```

Parameters

`hDevice`.....Handle to already open device file

`lpBuffer`.....Pointer to the buffer with data to send

`dwLen`.....Number of bytes to send

`dwActuallySent`.....Pointer to a `DWORD` where the number of actually sent bytes is returned

`lpOverlapped`.....Ignored, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details

`!=0`.....Success

Description

Sends `dwLen` bytes that are stored at `lpBuffer` to the SPI device. This is a send-only transfer. Any command bytes that are required must be included in the data to send. As the function can not distinguish between command and data bytes, no `DataDelay` is possible.

If you want to have separate command data, consider to use `IOCTL_NSPI_SEND` instead. If you want to wait for an interrupt request on the IRQ line before sending data, use command `IOCTL_NSPI_WAITIRQ_SEND` instead.

This function is not available in driver version 1.x.

Example

Send data bytes 0x01, 0x02, 0x03, 0x04, 0x05 to the SPI device.

```
DWORD dwWritten;
BYTE data[] =
{
      0x01, 0x02, 0x03, 0x04, 0x05
};
WriteFile(hSpi, data, sizeof(data), &dwWritten, NULL);
```

Listing 2: Example WriteFile()

6.3 ReadFile()

Signature

```
BOOL ReadFile(
      HANDLE hDevice, LPCVOID lpBuffer, DWORD dwLen,
      LPDWORD dwRead, LPOVERLAPPED lpOverlapped
);
```

Parameters

`hDevice`.....Handle to already open device file

`lpBuffer`.....Pointer to the buffer where the received data is stored

`dwLen`.....Number of bytes to receive

`dwRead`.....Pointer to a `DWORD` where the number of actually received bytes is returned

`lpOverlapped`.....Ignored, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details

`!=0`.....Success

Description

Receives `dwLen` bytes from the SPI device and stores the data at `lpBuffer`. This is a receive-only transfer, hence this function can not be used if a command is required prior to receiving the data. While receiving, the value of `DummyByte` is sent on the MOSI line.

If you want to have a separate command phase, consider to use `IOCTL_NSPI_RECEIVE` instead. If you want to wait for an interrupt request on the IRQ line before receiving data, use `IOCTL_NSPI_WAITIRQ_RECEIVE` instead.

This function is not available in driver version 1.x.

Example

Read 7 data bytes from the SPI device.

```
DWORD dwReceived;
BYTE data[7];

ReadFile(hSpi, data, 7, &dwReceived, NULL);
```

Listing 3: Example ReadFile()

6.4 CloseHandle()

Signature

```
BOOL CloseHandle(HANDLE hDevice);
```

Parameters

`hDevice`.....Handle to open device file

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

Closes the device file that was opened with `CreateFile()`.

Example

```
HANDLE hSpi;
hSpi = CreateFile(...);
/* Handle SPI transmissions */
CloseHandle(hSpi);
```

Listing 4: Example CloseHandle()

6.5 DeviceIoControl()

Signature

```
int DeviceIoControl(
      HANDLE hDevice, DWORD dwIoControlCode,
      LPVOID lpInBuffer, DWORD dwInBufferSize,
      LPVOID lpOutBuffer, DWORD dwOutBufferSize,
      LPDWORD lpReturned, LPOVERLAPPED lpOverlapped
);
```

Parameters

`hDevice`.....Handle to already open device file

`dwIoControlCode`.....Control code specifying the device specific command to execute

`lpInBuffer`.....Pointer to the data going into the command (`IN` data)

`dwInBufferSize`.....Size of the `IN` data (in bytes)

`lpOutBuffer`.....Pointer to a buffer where data coming out of the command can be stored (`OUT` data)

`dwOutBufferSize`.....Number of bytes available for the `OUT` data

`lpReturned`.....Number of bytes actually written to the `OUT` data buffer

`lpOverlapped`.....Unused, set to `NULL`

Description

Executes a device specific command. The type of function is given by a control code in parameter `dwIoControlCode`. Each command has a specific set of parameters. Usually there is some data going into the command (`IN` data) and some data is returned out of the command (`OUT` data).

The following table lists all control codes (commands) recognised by the NSPI driver V1.x.

Control Code	Function
`IOCTL_NSPI_SEND`	Sends command and data to the SPI device
`IOCTL_NSPI_RECEIVE`	Sends command to and then receives data from the SPI device
`IOCTL_NSPI_TRANSFER`	Sends command and data to the SPI device and receives data from the device
`IOCTL_NSPI_EXCHANGE`	Sends command and data to the SPI device and receives data from the device. The received data replaces the sent data.

Table 15: IOCTL command codes for V1.x

In V2.0, the set of control codes was extended. The following table lists all control codes that were newly added to the NSPI driver.

Control Code	Function
IOCTL_NSPI_GET_CLOCKFREQ	Get the current transfer speed
IOCTL_NSPI_SET_CLOCKFREQ	Set a new transfer speed
IOCTL_NSPI_GET_MODE	Get the current SPI mode
IOCTL_NSPI_SET_MODE	Set a new SPI mode
IOCTL_NSPI_GET_METHOD	Get the current driver method
IOCTL_NSPI_SET_METHOD	Set a new driver method
IOCTL_DRIVER_GETINFO	Get the driver version

Table 16: Additional IOCTL command codes for V2.x

In V3.0, the set of control codes was again extended. The following table lists all control codes that were newly added to the NSPI driver.

Control Code	Function
IOCTL_NSPI_GET_DUMMYBYTE	Get the current dummy byte (receive-only transfers)
IOCTL_NSPI_SET_DUMMYBYTE	Set a new dummy byte (receive-only transfers)
IOCTL_NSPI_GET_DATADELAY	Get the current delay between command and data
IOCTL_NSPI_SET_DATADELAY	Set a new delay between command and data
IOCTL_NSPI_GET_GENTIMEOUT	Get the current generic timeout
IOCTL_NSPI_SET_GENTIMEOUT	Set a new generic timeout
IOCTL_NSPI_WAITIRQ_SEND	Wait for IRQ, then send to SPI device
IOCTL_NSPI_WAITIRQ_RECEIVE	Wait for IRQ, then receive from SPI device
IOCTL_NSPI_WAITIRQ_TRANSFER	Wait for IRQ, then send to and receive from SPI
IOCTL_NSPI_WAITIRQ_EXCHANGE	Wait for IRQ, then send to and receive from SPI (new data replacing old data)
IOCTL_NSPI_GET_IRQTIMEOUT	Get current timeout for waiting for IRQ
IOCTL_NSPI_SET_IRQTIMEOUT	Set a new timeout for waiting for IRQ
IOCTL_NSPI_CLEAR_IRQ	Clear all pending interrupts for IRQ pin

Table 17: Additional IOCTL command codes for V3.x

6.6 IOCTL_NSPI_SEND

Parameters

hDevice	Handle to already open device file
dwIoControlCode	IOCTL_NSPI_SEND
lpInBuffer	Pointer to command bytes; can be NULL if no command is required
dwInBufferSize	Number of command bytes
lpOutBuffer	Pointer to the data bytes to send; can be NULL if no data is required
dwOutBufferSize	Number of data bytes
lpReturned	Unused, set to NULL
lpOverlapped	Unused, set to NULL

Return

0	Error, see GetLastError() for details
!=0	Success

Description

This command sends the command bytes and then the data bytes to the SPI device. All received bytes are discarded.

If a DataDelay is set either in the registry or with IOCTL_NSPI_SET_DATADELAY, a short pause of this length is inserted between command and data phase. See Chapter 2.6 on Page 7 for a description of the DataDelay feature.

If not using a DataDelay, there is no difference between command and data bytes. So if you like, you can append command and data to one buffer and use it either as IN or as OUT array. For example this can be done when the command and data bytes are well known. In this case this is the same as using WriteFile().

However this function makes more sense when command and data already arrive as two different entities, for example when the command is known, but the data is some variable parameter of the surrounding function. Then the possibility to pass these on as two different arrays avoids having to copy command and data bytes to a common buffer.

If you want to wait for an interrupt request on the IRQ line before sending data, use command IOCTL_NSPI_WAITIRQ_SEND instead.

The DataDelay feature is not available in V1.x and V2.x.

Remarks

- In the split version, this function needs two arrays going in: the command bytes and the data bytes to send. Therefore this call uses both data pointers of the `DeviceIoControl()` as `IN` pointers, `lpInBuffer` and `lpOutBuffer`. This is a little bit unusual, but works nonetheless.
- When using the DMA method, the number of bytes to send (command+data) is restricted to the value set in registry value `DmaBufferSize`, usually 4096 bytes. When trying to send more data in one go, the driver will return `ERROR_INVALID_PARAMETER` without transmitting anything. Because command and data are sent as one combined DMA transfer then, it is not possible to have a pause between command and data phase. So any `DataDelay` value is ignored in this case.

Example 1

Send command bytes 0x12, 0x34, 0x56 and data bytes 0x01, 0x02, 0x03, 0x04, 0x05 to the SPI device. Here we can combine command and data bytes in one array.

```
BYTE chCmdData[8] =
{
     0x12, 0x34, 0x56,              /* command */
     0x01, 0x02, 0x03, 0x04, 0x05   /* data */
};

DeviceIoControl(hSpi, IOCTL_NSPI_SEND, chCmdData, sizeof(chCmdData),
                NULL, 0, NULL, NULL);
```

Listing 5: Example IOCTL_NSPI_SEND: One Array

Example 2

Function for sending command bytes 0x12, 0x34, 0x56 and some data given as function parameter to the SPI device. To avoid having to copy the data bytes behind the command bytes into a temporary array, it is better to use the 2-array version.

```
BYTE chCmd[3] =
{
     0x12, 0x34, 0x56
};

void Send(BYTE *pData, DWORD dwDataLen)
{
     DeviceIoControl(hSpi, IOCTL_NSPI_SEND, chCmd, sizeof(chCmd),
                     pData, dwDataLen, NULL, NULL);
}
```

Listing 6: Example IOCTL_NSPI_SEND: Two Arrays

6.7 IOCTL_NSPI_RECEIVE

Parameters

`hDevice`.....Handle to already open device file

`dwIoControlCode`.....`IOCTL_NSPI_RECEIVE`

`lpInBuffer`.....Pointer to command bytes; can be `NULL` if no command is required

`dwInBufferSize`.....Number of command bytes

`lpOutBuffer`.....Pointer to the byte array where the received data bytes will be stored

`dwOutBufferSize`.....Number of data bytes to receive

`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`

`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details

`!=0`.....Success

Description

This command sends the command bytes to the SPI device. Any bytes received during this phase are discarded. Then it receives the given number of data bytes from the SPI device. As the driver has to send dummy data while receiving, it will use the value of `DummyByte` for this purpose which can be set in the registry or with `IOCTL_NSPI_SET_DUMMYBYTE`.

If a `DataDelay` is set either in the registry or with `IOCTL_NSPI_SET_DATADELAY`, a short pause of this length is inserted between command and data phase. See Chapter 2.6 on Page 7 for a description of the `DataDelay` feature.

If you want to wait for an interrupt request on the IRQ line before receiving data, use command `IOCTL_NSPI_WAITIRQ_RECEIVE` instead.

The `DataDelay` feature is not available in V1.x and V2.x and the `DummyByte` is fixed to `0xFF`.

Remark

When using the DMA method, the number of bytes to transfer (command+data) is restricted to the value set in registry value `DmaBufferSize`, usually 4096 bytes. When trying to transfer more bytes in one go, the driver will return `ERROR_INVALID_PARAMETER` without transferring anything. Because command and data are sent as one combined DMA transfer then, it is not possible to have a pause between command and data phase. So any `DataDelay` value is ignored in this case.

Example

Send command 0x98 0x76 to the device and receive 10 bytes.

```
BYTE chCmd[2] = {0x98, 0x76};
BYTE chData[10];

DeviceIoControl(hSpi, IOCTL_NSPI_RECEIVE, chCmd, sizeof(chCmd),
                chData, sizeof(chData), NULL, NULL);
```

Listing 7: Example IOCTL_NSPI_RECEIVE

6.8 IOCTL_NSPI_TRANSFER

Parameters

`hDevice`.....Handle to already open device file

`dwIoControlCode`.....`IOCTL_NSPI_TRANSFER`

`lpInBuffer`.....Pointer to command bytes and data bytes to send

`dwInBufferSize`.....Number of command plus data bytes

`lpOutBuffer`.....Pointer to the byte array where the received data bytes will be stored

`dwOutBufferSize`.....Number of data bytes to receive

`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`

`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details

`!=0`.....Success

Description

This command first sends the command bytes to the SPI device. The bytes received during this phase are discarded. Then it sends the given data bytes to the device and at the same time receives data bytes from the device.

The number of command bytes is determined as the difference of `dwInBufferSize` and `dwOutBufferSize`.

If a `DataDelay` is set either in the registry or with `IOCTL_NSPI_SET_DATADELAY`, a short pause of this length is inserted between command and data phase. See Chapter 2.6 on Page 7 for a description of the `DataDelay` feature.

If you want to wait for an interrupt request on the IRQ line before transferring data, use command `IOCTL_NSPI_WAITIRQ_TRANSFER` instead.

The `DataDelay` feature is not available in V1.x and V2.x.

Remarks

- As here both, the `IN` and the `OUT` array are used for different data, the data to send must be appended to the command data. This may be a little inconvenient as command bytes and data may need to be copied to a temporary array first. However most cases using simultaneous send and receive transmissions allow direct exchanging of the data, so you can use `IOCTL_NSPI_EXCHANGE` instead.

- When using the DMA method, the number of bytes (command+data) is restricted to the value set in registry value `DmaBufferSize`, usually 4096 bytes. When trying to transfer more data in one go, the driver will immediately return without transmitting anything. Because command and data are sent as one combined DMA transfer then, it is not possible to have a pause between command and data phase. So any `Data-Delay` value is ignored in this case.
- When using the DMA method on the NetDCU8 or NetDCU10, this command is not allowed. These boards only support send-only and receive-only DMA transfers and this command would be a combined send-and-receive transfer.

Example

Send command 0x55 0x66 and the four data bytes 0x01, 0x02, 0x03, 0x04 to the device and receive four bytes in return.

```
BYTE chCmdSendData[6] =
{
     0x55, 0x66,                        /* command */
     0x01, 0x02, 0x03, 0x04             /* send data */
};
BYTE chReceiveData[4];                  /* receive data */

DeviceIoControl(hSpi, IOCTL_NSPI_TRANSFER,
                chCmdSendData, sizeof(chCmdSendData),
                chReceiveData, sizeof(chReceiveData), NULL, NULL);
```

Listing 8: Example IOCTL_NSPI_TRANSFER

6.9 IOCTL_NSPI_EXCHANGE

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_EXCHANGE`
`lpInBuffer`.....Pointer to command bytes
`dwInBufferSize`.....Number of command bytes
`lpOutBuffer`.....Pointer to the byte array with the data bytes to send *and* where the received data bytes will be stored
`dwOutBufferSize`.....Number of bytes to send and receive
`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command first sends the command bytes to the SPI device. The bytes received during this phase are discarded. Then it sends the given data bytes to the device and at the same time receives data bytes from the device. The received data bytes will replace byte after byte the sent data. After return, the old data is completely overwritten with the new data.

If a `DataDelay` is set either in the registry or with `IOCTL_NSPI_SET_DATADELAY`, a short pause of this length is inserted between command and data phase. See Chapter 2.6 on Page 7 for a description of the `DataDelay` feature.

If you want to wait for an interrupt request on the IRQ line before exchanging data, use command `IOCTL_NSPI_WAITIRQ_EXCHANGE` instead.

The `DataDelay` feature is not available in V1.x and V2.x.

Remarks

- This function needs one array going in, and a second array with data going in and out. Therefore this call uses both data pointers of the `DeviceIoControl()` for providing `IN` data, `lpInBuffer` and `lpOutBuffer`. This is a little bit unusual, but works nonetheless.
- When using the DMA method, the number of bytes (command+data) is restricted to the value set in registry value `DmaBufferSize`, usually 4096 bytes. When trying to transfer more data in one go, the driver will immediately return without transmitting anything. Because command and data are sent as one combined DMA transfer then,

it is not possible to have a pause between command and data phase. So any `DataDelay` value is ignored in this case.

- When using the DMA method on the NetDCU8 or NetDCU10, this command is not allowed. These boards only support send-only and receive-only DMA transfers and this command would be a combined send-and-receive transfer.

Example

Send the command 0x55 0x66 and the four data bytes 0x01, 0x02, 0x03, 0x04 to the device and receive four bytes in return, replacing the four data bytes.

```
BYTE chCmd[] = {0x55, 0x66};
BYTE chSendReceiveData[] =
{
     0x01, 0x02, 0x03, 0x04     /* initialise with send data */
};

DeviceIoControl(hSpi, IOCTL_NSPI_EXCHANGE, chCmd, sizeof(chCmd),
                chSendReceiveData, sizeof(chSendReceiveData),
                NULL, NULL);

/* Now array chSendReceiveData contains the received data */
```

Listing 9: Example IOCTL_NSPI_EXCHANGE

6.10 IOCTL_NSPI_WAITIRQ_SEND

Parameters

hDevice	Handle to already open device file
dwIoControlCode	IOCTL_NSPI_WAITIRQ_SEND
lpInBuffer	Pointer to command bytes; can be NULL if no command is required
dwInBufferSize	Number of command bytes
lpOutBuffer	Pointer to the data bytes to send; can be NULL if no data is required
dwOutBufferSize	Number of data bytes
lpReturned	Unused, set to NULL
lpOverlapped	Unused, set to NULL

Return

0	Error, see GetLastError() for details
!=0	Success

Description

This command first waits for an interrupt request on the IRQ pin of this device (=chip select). Then it works exactly like IOCTL_NSPI_SEND, i.e. it sends the command bytes and then the data bytes to the SPI device and all received bytes are discarded.

For a detailed description of the send function, its restrictions and possible parameter combinations, see Page 33.

To be able to use this command, you have to configure a pin for the interrupt request. See registry settings IrqPin in Chapter 4.3.10 on Page 17 and IrqCfg in Chapter 4.3.11 on Page 17 respectively.

To discard any pending interrupts, use command IOCTL_NSPI_CLEAR_IRQ before calling IOCTL_NSPI_WAITIRQ_SEND. This makes sure that the call reacts to the next interrupt request that happens in the future and not to some interrupt request from the past.

The driver also honours the registry value IrqTimeout that can also be modified at runtime with IOCTL_NSPI_SET_IRQTIMEOUT. This means if there happens no interrupt within this timeout period, the driver gives up and returns with WAIT_TIMEOUT (in contrast to ERROR_TIMEOUT that is returned if the call is blocked for other reasons).

If you just want to send data without waiting for an interrupt request first, consider to use WriteFile() or IOCTL_NSPI_SEND instead.

This command is not available in V1.x and V2.x.

Example

After waiting for an interrupt request, send command bytes 0x12, 0x34, 0x56 and data bytes 0x01, 0x02, 0x03, 0x04, 0x05 to the SPI device. The interrupt must happen within the next 300 ms, any other blocking must not exceed 2 s.

```
DWORD dwIrqTimeout = 300;
DWORD dwGenTimeout = 2000;
DWORD dwError;
BYTE chCmd[] = {0x12, 0x34, 0x56};
BYTE chData[] = {0x01, 0x02, 0x03, 0x04, 0x05};

/* Set timeouts */
DeviceIoControl(hSpi, IOCTL_NSPI_SET_IRQTIMEOUT,
                &dwIrqTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);
DeviceIoControl(hSpi, IOCTL_NSPI_SET_GENTIMEOUT,
                &dwGenTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send command and data */
dwError = DeviceIoControl(hSpi, IOCTL_NSPI_WAITIRQ_SEND,
                          chCmd, sizeof(chCmd),
                          chData, sizeof(chData), NULL, NULL);

/* Handle timeout errors */
if (dwError == ERROR_TIMEOUT)
{
     /* Handle general timeout */
}
else if (dwError == WAIT_TIMEOUT)
{
     /* Handle interrupt timeout */
}
```

Listing 10: Example IOCTL_NSPI_WAITIRQ_SEND

6.11 IOCTL_NSPI_WAITIRQ_RECEIVE

Parameters

`hDevice`.....Handle to already open device file

`dwIoControlCode`.....`IOCTL_NSPI_WAITIRQ_RECEIVE`

`lpInBuffer`.....Pointer to command bytes; can be `NULL` if no command is required

`dwInBufferSize`.....Number of command bytes

`lpOutBuffer`.....Pointer to the byte array where the received data bytes will be stored

`dwOutBufferSize`.....Number of data bytes to receive

`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`

`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details

`!=0`.....Success

Description

This command first waits for an interrupt request on the IRQ pin of this device (=chip select). Then it works exactly like `IOCTL_NSPI_RECEIVE`, i.e. it first sends the command bytes to the SPI device. Any bytes received during this phase are discarded. Then it receives the given number of data bytes from the SPI device. As the driver has to send dummy data while receiving, it will use the value of `DummyByte` for this purpose which can be set in the registry or with `IOCTL_NSPI_SET_DUMMYBYTE`.

For a detailed description of the receive function, its restrictions and possible parameter combinations, see Page 35.

To be able to use this command, you have to configure a pin for the interrupt request. See registry settings `IrqPin` in Chapter 4.3.10 on Page 17 and `IrqCfg` in Chapter 4.3.11 on Page 17 respectively.

To discard any pending interrupts, use command `IOCTL_NSPI_CLEAR_IRQ` before calling `IOCTL_NSPI_WAITIRQ_RECEIVE`. This makes sure that the call reacts to the next interrupt request that happens in the future and not to some interrupt request from the past.

The driver also honours the registry value `IrqTimeout` that can also be modified at runtime with `IOCTL_NSPI_SET_IRQTIMEOUT`. This means if there happens no interrupt within this timeout period, the driver gives up and returns with `WAIT_TIMEOUT` (in contrast to `ERROR_TIMEOUT` that is returned if the call is blocked for other reasons).

If you just want to receive data without waiting for an interrupt request first, consider to use `ReadFile()` or `IOCTL_NSPI_RECEIVE` instead.

This command is not available in V1.x and V2.x.

Example

Wait for interrupt, then send command 0x98 0x76 to the device and receive 10 bytes. The interrupt must happen within the next 300 ms, any other blocking must not exceed 2 s.

```
DWORD dwIrqTimeout = 300;
DWORD dwGenTimeout = 2000;
DWORD dwError;
BYTE chCmd[] = {0x98, 0x76};
BYTE chData[10];

/* Set timeouts */
DeviceIoControl(hSpi, IOCTL_NSPI_SET_IRQTIMEOUT,
                &dwIrqTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);
DeviceIoControl(hSpi, IOCTL_NSPI_SET_GENTIMEOUT,
                &dwGenTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send command and receive data */
dwError = DeviceIoControl(hSpi, IOCTL_NSPI_WAITIRQ_RECEIVE,
                          chCmd, sizeof(chCmd),
                          chData, sizeof(chData), NULL, NULL);

/* Handle timeout errors */
if (dwError == ERROR_TIMEOUT)
{
     /* Handle general timeout */
}
else if (dwError == WAIT_TIMEOUT)
{
     /* Handle interrupt timeout */
}
```

Listing 11: Example IOCTL_NSPI_WAITIRQ_RECEIVE

6.12 IOCTL_NSPI_WAITIRQ_TRANSFER

Parameters

hDevice	Handle to already open device file
dwIoControlCode	IOCTL_NSPI_WAITIRQ_TRANSFER
lpInBuffer	Pointer to command bytes and data bytes to send
dwInBufferSize	Number of command plus data bytes
lpOutBuffer	Pointer to the byte array where the received data bytes will be stored
dwOutBufferSize	Number of data bytes to receive
lpReturned	The referenced value will be set to dwOutBufferSize if pointer is not NULL
lpOverlapped	Unused, set to NULL

Return

0	Error, see GetLastError() for details
!=0	Success

Description

This command first waits for an interrupt request on the IRQ pin of this device (=chip select). Then it works exactly like IOCTL_NSPI_TRANSFER, i.e. it sends the command bytes to the SPI device. The bytes received during this phase are discarded. Then it sends the given data bytes to the device and at the same time receives data bytes from the device.

The number of command bytes is determined as the difference of dwInBufferSize and dwOutBufferSize. For a detailed description of the transfer function, its restrictions and possible parameter combinations, see Page 37.

To be able to use this command, you have to configure a pin for the interrupt request. See registry settings IrqPin in Chapter 4.3.10 on Page 17 and IrqCfg in Chapter 4.3.11 on Page 17 respectively.

To discard any pending interrupts, use command IOCTL_NSPI_CLEAR_IRQ before calling IOCTL_NSPI_WAITIRQ_TRANSFER. This makes sure that the call reacts to the next interrupt request that happens in the future and not to some interrupt request from the past.

The driver also honours the registry value IrqTimeout that can also be modified at runtime with IOCTL_NSPI_SET_IRQTIMEOUT. This means if there happens no interrupt within this timeout period, the driver gives up and returns with WAIT_TIMEOUT (in contrast to ERROR_TIMEOUT that is returned if the call is blocked for other reasons).

If you just want to transfer data without waiting for an interrupt request first, consider to use IOCTL_NSPI_TRANSFER instead.

This command is not available in V1.x and V2.x.

Example

Wait for interrupt, then send command 0x55 0x66 and the four data bytes 0x01, 0x02, 0x03, 0x04 to the device and receive four bytes in return. The interrupt must happen within the next 300 ms, any other blocking must not exceed 2 s.

```
DWORD dwIrqTimeout = 300;
DWORD dwGenTimeout = 2000;
DWORD dwError;
BYTE chCmdSendData[6] =
{
     0x55, 0x66,                        /* command */
     0x01, 0x02, 0x03, 0x04             /* send data */
};
BYTE chReceiveData[4];                  /* receive data */

/* Set timeouts */
DeviceIoControl(hSpi, IOCTL_NSPI_SET_IRQTIMEOUT,
                dwIrqTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);
DeviceIoControl(hSpi, IOCTL_NSPI_SET_GENTIMEOUT,
                dwGenTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send command, send and receive data */
dwError = DeviceIoControl(hSpi, IOCTL_NSPI_WAITIRQ_TRANSFER,
                          chCmdSendData, sizeof(chCmdSendData),
                          chReceiveData, sizeof(chReceiveData),
                          NULL, NULL);

/* Handle timeout errors */
if (dwError == ERROR_TIMEOUT)
{
     /* Handle general timeout */
}
else if (dwError == WAIT_TIMEOUT)
{
     /* Handle interrupt timeout */
}
```

Listing 12: Example IOCTL_NSPI_WAITIRQ_TRANSFER

6.13 IOCTL_NSPI_WAITIRQ_EXCHANGE

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_WAITIRQ_EXCHANGE`
`lpInBuffer`.....Pointer to command bytes
`dwInBufferSize`.....Number of command bytes
`lpOutBuffer`.....Pointer to the byte array with the data bytes to send *and* where the received data bytes will be stored
`dwOutBufferSize`.....Number of bytes to send and receive
`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command first waits for an interrupt request on the IRQ pin of this device (=chip select). Then it works exactly like `IOCTL_NSPI_EXCHANGE`, i.e. it sends the command bytes to the SPI device. The bytes received during this phase are discarded. Then it sends the given data bytes to the device and at the same time receives data bytes from the device. The received data bytes will replace byte after byte the sent data. After return, the old data is completely overwritten with the new data.

For a detailed description of the exchange function, its restrictions and possible parameter combinations, see Page 39.

To be able to use this command, you have to configure a pin for the interrupt request. See registry settings `IrqPin` in Chapter 4.3.10 on Page 17 and `IrqCfg` in Chapter 4.3.11 on Page 17 respectively.

To discard any pending interrupts, use command `IOCTL_NSPI_CLEAR_IRQ` before calling `IOCTL_NSPI_WAITIRQ_EXCHANGE`. This makes sure that the call reacts to the next interrupt request that happens in the future and not to some interrupt request from the past.

The driver also honours the registry value `IrqTimeout` that can also be modified at runtime with `IOCTL_NSPI_SET_IRQTIMEOUT`. This means if there happens no interrupt within this timeout period, the driver gives up and returns with `WAIT_TIMEOUT` (in contrast to `ERROR_TIMEOUT` that is returned if the call is blocked for other reasons).

If you just want to exchange data without waiting for an interrupt request first, consider to use `IOCTL_NSPI_EXCHANGE` instead.

This command is not available in V1.x and V2.x.

Example

Wait for interrupt, then send command 0x55 0x66 and the four data bytes 0x01, 0x02, 0x03, 0x04 to the device and receive four bytes in return, replacing the four data bytes. The interrupt must happen within the next 300 ms, any other blocking must not exceed 2 s.

```
DWORD dwIrqTimeout = 300;
DWORD dwGenTimeout = 2000;
DWORD dwError;
BYTE chCmd[] = {0x55, 0x66};
BYTE chSendReceiveData[] =
{
     0x01, 0x02, 0x03, 0x04     /* initialise with send data */
};

/* Set timeouts */
DeviceIoControl(hSpi, IOCTL_NSPI_SET_IRQTIMEOUT,
                &dwIrqTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);
DeviceIoControl(hSpi, IOCTL_NSPI_SET_GENTIMEOUT,
                &dwGenTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send command, send and receive data */
dwError = DeviceIoControl(hSpi, IOCTL_NSPI_WAITIRQ_EXCHANGE,
                          chCmd, sizeof(chCmd), chSendReceiveData,
                          sizeof(chSendReceiveData), NULL, NULL);

/* Handle timeout errors */
if (dwError == ERROR_TIMEOUT)
{
     /* Handle general timeout */
}
else if (dwError == WAIT_TIMEOUT)
{
     /* Handle interrupt timeout */
}

/* Now chSendReceiveData contains the received data */
```

Listing 13: Example IOCTL_NSPI_WAITIRQ_EXCHANGE

6.14 IOCTL_NSPI_GET_CLOCKFREQ

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_GET_CLOCKFREQ`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Pointer to a `DWORD` receiving the current clock frequency
`dwOutBufferSize`.....`sizeof(DWORD)`
`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command retrieves the current transfer data rate in Hz or Bit/s. The default value is taken from registry entry `ClockFreq`. A new data rate value can be set with command `IOCTL_NSPI_SET_CLOCKFREQ`.

This command is not available in driver V1.x.

Example

Transfer some data with double data rate, then return to previous rate.

```
DWORD dwOldFreq;
DWORD dwNewFreq;

/* Get current data rate */
DeviceIoControl(hSpi, IOCTL_NSPI_GET_CLOCKSPEED, NULL, 0,
                &dwOldFreq, sizeof(DWORD), NULL, NULL);

/* Set double data rate */
dwNewFreq = 2*dwOldFreq;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_CLOCKSPEED,
                &dwNewFreq, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send some data */
WriteFile(hSpi, ...);

/* Return to previous data rate */
DeviceIoControl(hSpi, IOCTL_NSPI_SET_CLOCKSPEED,
                &dwOldFreq, sizeof(DWORD), NULL, 0, NULL, NULL);
```

Listing 14: Example IOCTL_NSPI_GET_CLOCKFREQ

6.15 IOCTL_NSPI_SET_CLOCKFREQ

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_SET_CLOCKFREQ`
`lpInBuffer`.....Pointer to a `DWORD` with the new clock frequency
`dwInBufferSize`.....`sizeof(DWORD)`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command sets a new transfer data rate. The current data rate can always be determined with command `IOCTL_NSPI_GET_CLOCKFREQ`.

This command `IOCTL_NSPI_SET_CLOCKFREQ` is not available in driver V1.x. There the clock frequency can only be set via the registry and it is not possible to change it at runtime.

Example

Transfer some data with 400 kHz, then with 2 MHz.

```
DWORD dwClockFreq;

/* Set 400 kHz */
dwClockFreq = 400000;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_CLOCKSPEED,
                &dwClockFreq, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send some data */
WriteFile(hSpi, ...);

/* Set 2 MHz */
dwClockFreq = 2000000;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_CLOCKSPEED,
                &dwClockFreq, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send some data */
WriteFile(hSpi, ...);
```

Listing 15: Example IOCTL_NSPI_SET_CLOCKFREQ

6.16 IOCTL_NSPI_GET_MODE

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_GET_MODE`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Pointer to a `DWORD` receiving the current SPI mode
`dwOutBufferSize`.....`sizeof(DWORD)`
`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command retrieves the current transfer SPI mode, i.e. 0, 1, 2 or 3. A new mode can be set with command `IOCTL_NSPI_SET_MODE`. The default value for the SPI mode is taken from registry entry `SPIMode`. Please refer to 5 on Page 14 for a description of the SPI modes.

This command is not available in driver V1.x.

Example

Get SPI mode and print description to `stdout`.

```
DWORD dwCurrentMode;
DeviceIoControl(hSpi, IOCTL_NSPI_GET_MODE, NULL, 0,
                &dwCurrentMode, sizeof(DWORD), NULL, NULL);
switch (dwCurrentMode)
{
  case 0: printf("Clock active high, latch on rising edge"); break;
  case 1: printf("Clock active high, latch on falling edge"); break;
  case 2: printf("Clock active low, latch on falling edge"); break;
  case 3: printf("Clock active low, latch on rising edge"); break;
}
```

Listing 16: Example IOCTL_NSPI_GET_MODE

6.17 IOCTL_NSPI_SET_MODE

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_SET_MODE`
`lpInBuffer`.....Pointer to a `DWORD` with the new SPI mode
`dwInBufferSize`.....`sizeof(DWORD)`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command sets a new SPI mode. The current SPI mode can always be determined with command `IOCTL_NSPI_GET_MODE`. Please refer to 5 on Page 14 for a description of the SPI modes.

This command is not available in driver V1.x. There the SPI mode can only be set via the registry and it is not possible to change it at runtime.

Example

Transfer some data with SPI mode 0, then some data with SPI mode 3.

```
DWORD dwMode;

/* Set SPI mode 0 */
dwMode = 0;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_MODE,
                &dwMode, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send some data */
WriteFile(hSpi, ...);

/* Set SPI mode 3 */
dwMode = 3;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_MODE,
                &dwMode, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send some data */
WriteFile(hSpi, ...);
```

Listing 17: Example IOCTL_NSPI_SET_MODE

6.18 IOCTL_NSPI_GET_METHOD

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_GET_METHOD`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Pointer to a `DWORD` receiving the current driver method
`dwOutBufferSize`.....`sizeof(DWORD)`
`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command retrieves the current transfer driver method, i.e. 0 for IRQ, 1 for Polling and 2 for DMA. A new driver method can be set with command `IOCTL_NSPI_SET_METHOD`. The default value for the driver method is taken from registry entry `DriverMethod`. Please refer to 6 on Page 15 for a description of the driver methods.

This command is not available in driver V1.x.

Example

Get the driver method and print description to `stdout`.

```
DWORD dwCurrentMethod;

DeviceIoControl(hSpi, IOCTL_NSPI_GET_METHOD, NULL, 0,
                &dwCurrentMethod, sizeof(DWORD), NULL, NULL);

switch (dwCurrentMethod)
{
  case 0: printf("IRQ"); break;

  case 1: printf("Polling"); break;

  case 2: printf("DMA"); break;
}
```

Listing 18: Example IOCTL_NSPI_GET_METHOD

6.19 IOCTL_NSPI_SET_METHOD

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_SET_METHOD`
`lpInBuffer`.....Pointer to a `DWORD` with the new driver method
`dwInBufferSize`.....`sizeof(DWORD)`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command sets a new driver method. The current driver method can always be determined with command `IOCTL_NSPI_GET_METHOD`. Please refer to 6 on Page 15 for a description of the driver methods.

This command is not available in driver V1.x. There the driver method can only be set via the registry and can not be modified at runtime.

Example

Transfer some data with IRQ, then some data with Polling.

```
DWORD dwMethod;

/* Set IRQ mode */
dwMethod = 0;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_METHOD,
                &dwMethod, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send some data */
WriteFile(hSpi, ...);

/* Set Polling mode */
dwMethod = 1;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_METHOD,
                &dwMethod, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send some data */
WriteFile(hSpi, ...);
```

Listing 19: Example IOCTL_NSPI_SET_METHOD

6.20 IOCTL_NSPI_GET_DUMMYBYTE

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_GET_DUMMYBYTE`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Pointer to a `DWORD` receiving the current dummy byte
`dwOutBufferSize`.....`sizeof(DWORD)`
`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command retrieves the current dummy byte that is transferred on the MOSI line in case of receive-only transmissions. A new dummy byte value can be set with command `IOCTL_NSPI_SET_DUMMYBYTE`. The default value for the dummy byte is taken from registry entry `DummyByte`.

This command is not available in driver V1.x and V2.x. There the dummy byte is always fix the value `0xFF`.

Example

Get the current dummy byte and print value to `stdout`.

```
DWORD dwCurrentDummyByte;

DeviceIoControl(hSpi, IOCTL_NSPI_GET_DUMMYBYTE, NULL, 0,
                &dwCurrentDummyByte, sizeof(DWORD), NULL, NULL);

printf("Current dummy byte is 0x%02x", dwCurrentDummyByte);
```

Listing 20: Example IOCTL_NSPI_GET_DUMMYBYTE

6.21 IOCTL_NSPI_SET_DUMMYBYTE

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_SET_DUMMYBYTE`
`lpInBuffer`.....Pointer to a `DWORD` with the new dummy byte
`dwInBufferSize`.....`sizeof(DWORD)`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command sets a new dummy byte that is transferred on the MOSI line in case of receive-only transmissions. The current dummy byte can always be determined with command `IOCTL_NSPI_GET_DUMMYBYTE`.

This command is not available in driver V1.x or V2.x. There the dummy byte is always fix the value `0xFF`.

Example

Receive some data while sending 0x00, then read some data while sending 0xFF.

```
DWORD dwDummyByte;

/* Set dummy byte to 0x00 */
dwDummyByte = 0x00;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_DUMMYBYTE,
                &dwDummyByte, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Receive some data */
ReadFile(hSpi, ...);

/* Set dummy byte to 0xFF */
dwDummyByte = 0xFF;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_DUMMYBYTE,
                &dwDummyByte, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Receive some data */
ReadFile(hSpi, ...);
```

Listing 21: Example IOCTL_NSPI_SET_DUMMYBYTE

6.22 IOCTL_NSPI_GET_DATADELAY

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_GET_DATADELAY`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Pointer to a `DWORD` receiving the current data delay (in ns)
`dwOutBufferSize`.....`sizeof(DWORD)`
`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command retrieves the current data delay, i.e. the delay that is inserted by the master between command and data phase (see Chapter 2.6 on Page 7). A new data delay value can be set with command `IOCTL_NSPI_SET_DATADELAY`. The default value for the data delay is taken from registry entry `DataDelay`.

This command is not available in driver V1.x and V2.x. There the `DataDelay` is always zero.

Example

Get the current data delay and print value to `stdout`.

```
DWORD dwCurrentDataDelay;

DeviceIoControl(hSpi, IOCTL_NSPI_GET_DATADELAY, NULL, 0,
                &dwCurrentDataDelay, sizeof(DWORD), NULL, NULL);

printf("Current data delay is %d ns", dwCurrentDataDelay);
```

Listing 22: Example IOCTL_NSPI_GET_DATADELAY

6.23 IOCTL_NSPI_SET_DATADELAY

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_SET_DATADELAY`
`lpInBuffer`.....Pointer to a `DWORD` with the new data delay (in ns)
`dwInBufferSize`.....`sizeof(DWORD)`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command sets a new data delay value. This is a delay that is inserted by the driver between command and data phase (see Chapter 2.6 on Page 7). The value is given in nanoseconds, reasonable values are 20000 (for 20 µs) and up. The current data delay can always be determined with command `IOCTL_NSPI_GET_DATADELAY`.

This command is not available in driver V1.x and V2.x. There the `DataDelay` is always zero.

Example

Function to send command with given data delay, then reset data delay back to 0.

```
void SendWithDelay(DWORD dwDataDelay, BYTE *pCmd, DWORD dwCmdLen,
                   BYTE *pData, DWORD dwDataLen)
{
     /* Set given data delay */
     DeviceIoControl(hSpi, IOCTL_NSPI_SET_DATADELAY, &dwDataDelay,
                     sizeof(DWORD), NULL, 0, NULL, NULL);

     /* Send command */
     DeviceIoControl(hSpi, IOCTL_NSPI_SEND, pCmd, dwCmdLen,
                     pData, dwDataLen, NULL, NULL);

     /* Set data delay back to 0 */
     dwDataDelay = 0;
     DeviceIoControl(hSpi, IOCTL_NSPI_SET_DATADELAY, &dwDataDelay
                     sizeof(DWORD), NULL, 0, NULL, NULL);
}
```

Listing 23: Example IOCTL_NSPI_SET_DATADELAY

6.24 IOCTL_NSPI_GET_GENTIMEOUT

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_GET_GENTIMEOUT`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Pointer to a `DWORD` receiving the current general timeout value (in ms)
`dwOutBufferSize`.....`sizeof(DWORD)`
`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command retrieves the current general timeout. This is the time the driver waits in blocking situations until it gives up and returns with `ERROR_TIMEOUT`. A new general timeout value can be set with command `IOCTL_NSPI_SET_GENTIMEOUT`. The default value for the general timeout is taken from registry entry `GenTimeout`. The special value `0xFFFFFFFF` means infinite.

This command is not available in driver V1.x and V2.x. There the timeout in blocking situations was given by the program (different values in different places).

Example

Get the current general timeout and print value to `stdout`.

```
DWORD dwCurrentGenTimeout;

DeviceIoControl(hSpi, IOCTL_NSPI_GET_GENTIMEOUT, NULL, 0,
                &dwCurrentGenTimeout, sizeof(DWORD), NULL, NULL);

printf("Current general timeout is %d ms", dwCurrentGenTimeout);
```

Listing 24: Example IOCTL_NSPI_GET_GENTIMEOUT

6.25 IOCTL_NSPI_SET_GENTIMEOUT

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_SET_GENTIMEOUT`
`lpInBuffer`.....Pointer to a `DWORD` with the new general timeout (in ms)
`dwInBufferSize`.....`sizeof(DWORD)`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command sets a new general timeout value. This is the time the driver waits in blocking situations until it gives up and returns with `ERROR_TIMEOUT`. The value is given in micro-seconds. A value of `0xFFFFFFFF` means wait infinitely. The current general timeout value can always be determined with command `IOCTL_NSPI_GET_GENTIMEOUT`.

This command is not available in driver V1.x or V2.x. There the timeout in blocking situations was given by the program (different values in different places).

Example

Allow send function to block for 5 s, then set general timeout to 1 s.

```
DWORD dwGenTimeout;

/* Set general timeout to 5s */
dwGenTimeout = 5000;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_GENTIMEOUT,
                &dwGenTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send something */
DeviceIoControl(hSpi, IOCTL_NSPI_SEND, ...);

/* Set general timeout back to 1s */
dwGenTimeout = 1000;
DeviceIoControl(hSpi, IOCTL_NSPI_SET_GENTIMEOUT,
                &dwGenTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);
```

Listing 25: Example IOCTL_NSPI_SET_GENTIMEOUT

6.26 IOCTL_NSPI_GET_IRQTIMEOUT

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_GET_IRQTIMEOUT`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Pointer to a `DWORD` receiving the current interrupt timeout value (in ms)
`dwOutBufferSize`.....`sizeof(DWORD)`
`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command retrieves the current interrupt timeout. This is the time the driver waits in `WAITIRQ` transmission commands for the interrupt to occur until it gives up and returns with `WAIT_TIMEOUT` (as opposed to `ERROR_TIMEOUT` if the general timeout expires). A new interrupt timeout value can be set with command `IOCTL_NSPI_SET_IRQTIMEOUT`. The default value for the interrupt timeout is taken from registry entry `IrqTimeout`. The special value `0xFFFFFFFF` means infinite.

This command is not available in driver V1.x and V2.x.

Example

Get the current interrupt timeout and print value to `stdout`.

```
DWORD dwCurrentIrqTimeout;

DeviceIoControl(hSpi, IOCTL_NSPI_GET_IRQTIMEOUT, NULL, 0,
                &dwCurrentIrqTimeout, sizeof(DWORD), NULL, NULL);

printf("Current interrupt timeout is %d ms", dwCurrentIrqTimeout);
```

Listing 26: Example IOCTL_NSPI_GET_IRQTIMEOUT

6.27 IOCTL_NSPI_SET_IRQTIMEOUT

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_SET_IRQTIMEOUT`
`lpInBuffer`.....Pointer to a `DWORD` with the new interrupt timeout (in ms)
`dwInBufferSize`.....`sizeof(DWORD)`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

This command sets a new interrupt timeout value. This is the time the driver waits in `WAITIRQ` transmission commands for the interrupt to occur until it gives up and returns with `WAIT_TIMEOUT` (as opposed to `ERROR_TIMEOUT` if the general timeout expires). The value is given in microseconds. A value of `0xFFFFFFFF` means wait infinitely. The current interrupt timeout value can always be determined with command `IOCTL_NSPI_GET_IRQTIMEOUT`.

The interrupt can of course already happen before the call to the driver with the `WAITIRQ` transmission command. In this case the `WAITIRQ` command will continue immediately. If this behaviour is not what you want, then you can clear all pending interrupts with command `IOCTL_NSPI_CLEAR_IRQ`.

This command is not available in driver V1.x or V2.x.

Example

Allow `WAITIRQ` send function to wait for interrupt for at most 300 ms.

```
DWORD dwIrqTimeout = 300;

DeviceIoControl(hSpi, IOCTL_NSPI_SET_IRQTIMEOUT,
                &dwIrqTimeout, sizeof(DWORD), NULL, 0, NULL, NULL);

/* Send something */
dwError = DeviceIoControl(hSpi, IOCTL_NSPI_WAITIRQ_SEND, ...);
```

Listing 27: Example IOCTL_NSPI_SET_IRQTIMEOUT

6.28 IOCTL_NSPI_CLEAR_IRQ

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NSPI_CLEAR_IRQ`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details
`!=0`.....Success

Description

When calling a `WAITIRQ` transmission command, i.e. one of `IOCTL_NSPI_WAITIRQ_SEND`, `IOCTL_NSPI_WAITIRQ_RECEIVE`, `IOCTL_NSPI_WAITIRQ_TRANSFER`, or `IOCTL_NSPI_WAITIRQ_EXCHANGE`, the interrupt may already have happened before. By calling this command immediately before the `WAITIRQ` transmission command, all pending `WAITIRQ` interrupts for this SPI device (= chip select) are cleared. So the `WAITIRQ` transmission command will only return if a *new* interrupt occurs after the call of `IOCTL_NSPI_CLEAR_IRQ`.

This command is not available in driver V1.x or V2.x.

Example

Make sure that `IOCTL_NSPI_WAITIRQ_SEND` waits for the *next* interrupt.

```
/* Clear any pending interrupts */
DeviceIoControl(hSpi, IOCTL_NSPI_CLEAR_IRQ,
                NULL, 0, NULL, 0, NULL, NULL);

/* Wait for IRQ, then send something */
dwError = DeviceIoControl(hSpi, IOCTL_NSPI_WAITIRQ_SEND, ...);
```

Listing 28: Example IOCTL_NSPI_CLEAR_IRQ

6.29 IOCTL_DRIVER_GETINFO

Parameters

`hDevice`.....Handle to already open device file

`dwIoControlCode`.....`IOCTL_DRIVER_GETINFO`

`lpInBuffer`.....Unused, set to `NULL`

`dwInBufferSize`.....Unused, set to `0`

`lpOutBuffer`.....Pointer to a `DRIVER_INFO` structure receiving the driver version (see below)

`dwOutBufferSize`.....`sizeof(DRIVER_INFO)`

`lpReturned`.....The referenced value will be set to `dwOutBufferSize` if pointer is not `NULL`

`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Error, see `GetLastError()` for details

`!=0`.....Success

Description

This command retrieves the version information of the NSPI driver.

```
typedef struct tagDRIVER_INFO
{
      WORD wVerMajor;
      WORD wVerMinor;
      DWORD dwTemp[15];
} DRIVER_INFO, *PDRIVER_INFO;
```

Entry `dwTemp[]` in this structure is reserved for future extensions and is currently unused. Just ignore it.

Please note, as this command is also available for other F&S drivers, `DRIVER_INFO` and `IOCTL_DRIVER_GETINFO` are defined in a separate header file `fs_driverinfo.h`, that should be available in the newest SDK for your board.

This command is not available in NSPI V1.x. If the call fails, then the NSPI driver version is V1.x. Otherwise the correct version information is returned in the `DRIVER_INFO` structure.

Example

Get the driver version and print it to `stdout`.

```
#include <fs_driverinfo.h>

...

DRIVER_INFO cInfo;

if (!DeviceIoControl(hSpi, IOCTL_DRIVER_GETINFO, NULL, 0,
                     &cInfo, sizeof(cInfo), NULL, NULL))
{
     cInfo.wVerMajor = 1;       /* Command failed: this is V1.x */
     cInfo.wVerMinor = 0;
}

printf("NSPI driver V%d.%d", cInfo.wVerMajor, cInfo.wVerMinor);
```

Listing 29: Example IOCTL_DRIVER_GETINFO

7 Sample Program

The following program communicates to an FM25CL64 FRAM device. This is a type of non-volatile memory. Each transmission has to start with a command.

Command	Parameter	Description
MEM_READ	16-bit address	Read data from address
MEM_WREN	-	Write enable
MEM_WRITE	16-bit address	Write data to address
MEM_RDSR	-	Read status register
MEM_WRSR	-	Write status register

Table 18: Commands of FRAM FM25CL64

Before being able to write any data to the memory, it must first be unlocked with the MEM_WREN command. After every completed write command, writing is automatically disabled again.

In addition, the memory can be completely protected to avoid any change at all. This protection flag is part of a status register that can be read and written with the separate commands MEM_RDSR and MEM_WRSR.

The sample program reads the first 100 bytes of the memory. Then it unprotects the memory, writes some sample data starting at address 0 and protects the memory again. Finally the first 100 bytes are read once more.

```
/*************************************************************/
/*** File:        nspi-example.c                           ***/
/*** Author:      Hartmut Keller, (C) F&S 2006             ***/
/*** Description: NSPI communication with FM25CL64 FRAM    ***/
/*************************************************************/

#include "windows.h"            /* BYTE, WORD, ... */
#include "nspiio.h"             /* IOCTL_NSPI_* */

#define MEM_WRSR  0x01          /* Write Status Register */
#define MEM_WRITE 0x02          /* Write Memory Data */
#define MEM_READ  0x03          /* Read Memory Data */
#define MEM_RDSR  0x05          /* Read Status Register */
#define MEM_WREN  0x06          /* Set Write Enable Latch */

BYTE DEMODATA[]="Hello World!";
HANDLE hSpi;

/* Set or clear protection flag in the status register */
void Protect(BOOL bOn)
{
      BYTE command;
      BYTE sr;

      command = MEM_WREN;
      DeviceIoControl(hSpi, IOCTL_SPI_SEND, &command, 1,
                      NULL, 0, NULL, NULL);

      command = MEM_RDSR;
      DeviceIoControl(hSpi, IOCTL_SPI_RECEIVE, &command, 1,
                      &sr, 1, NULL, NULL);

      if (bOn)
            sr |= 0x0C;           /* protect */
      else
            sr &= ~0x0C;          /* unprotect */

      command = MEM_WRSR;
      DeviceIoControl(hSpi, IOCTL_SPI_SEND, &command, 1,
                      &sr, 1, NULL, NULL);
}

/* Read wCount bytes from the memory, starting at wOffset */
void ReadMem(WORD wOffset, WORD wCount, BYTE *pData)
{
      BYTE command[3];

      command[0] = MEM_READ;
      command[1] = wOffset / 256;
      command[2] = wOffset % 256;
      DeviceIoControl(hSpi, IOCTL_SPI_RECEIVE, command, 3,
                      pData, wCount, NULL, NULL);
}

/* Write wCount bytes to the memory, starting at wOffset */
void WriteMem(WORD wOffset, WORD wCount, BYTE *pData)
```

```
{
      BYTE command[3];

      command[0] = MEM_WREN;
      DeviceIoControl(hSpi, IOCTL_SPI_SEND, command, 1,
                      NULL, 0, NULL, NULL);

      command[0] = MEM_WRITE;
      command[1] = wOffset / 256;
      command[2] = wOffset % 256;
      DeviceIoControl(hSpi, IOCTL_SPI_SEND, command, 3,
                      pData, wCount, NULL, NULL);
}

/* Main program: read data then store new data */
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrev,
                   LPTSTR lpCmdLine, int nCmdShow)
{
      BYTE Buffer[100];

      /* Open the device file */
      hSpi = CreateFile(TEXT("SPI1:"), GENERIC_READ | GENERIC_WRITE,
                        0, NULL, OPEN_EXISTING,
                        FILE_FLAG_WRITE_THROUGH, 0);

      /* Read memory */
      ReadMem(0, 100, Buffer);

      /* Store new data */
      Protect(FALSE);
      WriteMem(0, strlen((PCHAR)DEMODATA)+1, DEMODATA);
      Protect(TRUE);

      /* Read back memory */
      ReadMem(0, 100, Buffer);

      /* Close device file and return */
      CloseHandle(hSpi);

      return 0;
}
```

Listing 30: Sample Program accessing FRAM FM25CL46

8 Header File nspiio.h

The following listing shows the contents of the header file `nspiio.h`. This file must be included in all applications that want to use the NSPI driver. It contains all IOCTL values required to call the driver.

```
/*****************************************************************************/
/***                                                                       ***/
/***                _   _        _   _____   ____ _    _                   ***/
/***               | \ | |      | | |  __ \ / ___/ |  | |                  ***/
/***               |  \| | ___ _| |_| |  \ | |    | |  | |                 ***/
/***               |   \ |/ _ \_   _| |  | | |    | |  | |                 ***/
/***               | |\  | |/_/ | | | |  | | |    | |  | |                 ***/
/***               | | \ | |__  | | | |__/ | |____| |__| |                 ***/
/***               |_|  \_|\___\ \_\ |_____/ \_____\______/                ***/
/***                                                                       ***/
/*****************************************************************************/
/***                                                                       ***/
/***                                                                       ***/
/***  N a t i v e   S P I   D e v i c e   D r i v e r   I n t e r f a c e ***/
/***                                                                       ***/
/***                                                                       ***/
/*****************************************************************************/
/*** File:     nspiio.h                                                    ***/
/*** Authors:  Hartmut Keller                                              ***/
/*** Created:  14.03.2005                                                  ***/
/*** Modified: 21.09.2011 15:56:15 (HK)                                    ***/
/***                                                                       ***/
/*** Description:                                                          ***/
/*** Include file for SPI IOCTLs. You have to include this file to use the ***/
/*** NSPI driver in your own applications.                                 ***/
/***                                                                       ***/
/*** Modification History:                                                 ***/
/*** 07.04.2006 HK: Change from SPI to NSPI                                ***/
/*** 19.02.2009 HK: V2.x: Added new IOCTL commands: IOCTL_DRIVER_GETINFO,  ***/
/***                IOCTL_NSPI_GET_CLOCKFREQ, IOCTL_NSPI_SET_CLOCKFREQ,    ***/
/***                IOCTL_NSPI_GET_METHOD, IOCTL_NSPI_SET_METHOD,          ***/
/***                IOCTL_NSPI_GET_MODE, IOCTL_NSPI_SET_MODE.              ***/
/*** 21.09.2011 HK: V3.x: New IOCTL commands: IOCTL_NSPI_WAITIRQ_EXCHANGE, ***/
/***                IOCTL_NSPI_WAITIRQ_TRANSFER, IOCTL_NSPI_WAITIRQ_SEND,  ***/
/***                IOCTL_NSPI_WAITIRQ_RECEIVE, IOCTL_NSPI_CLEAR_IRQ,      ***/
/***                IOCTL_NSPI_GET_IRQTIMEOUT, IOCTL_NSPI_SET_IRQTIMEOUT,  ***/
/***                IOCTL_NSPI_GET_GENTIMEOUT, IOCTL_NSPI_SET_GENTIMEOUT,  ***/
/***                IOCTL_NSPI_GET_DATADELAY, IOCTL_NSPI_SET_DATADELAY,    ***/
/***                IOCTL_NSPI_GET_DUMMYBYTE, IOCTL_NSPI_SET_DUMMYBYTE.    ***/
/*****************************************************************************/

/*---------------------------------------------------------------------------
  THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
  KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED
  WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.

  Copyright (c) 2011 F&S Elektronik Systeme GmbH
---------------------------------------------------------------------------*/

#ifndef __NSPIIO_H__
#define __NSPIIO_H__

#include "WINIOCTL.h"                    /* CTL_CODE(), METHOD_BUFFERED,... */

/* -------------------- Exported Definitions ------------------------------ */
```

```
/* With each clock cycle, the SPI transfer sends one bit on the MOSI line and
   receives one bit on the MISO line. Therefore sending and receiving of data
   on SPI is done at the same time. After one byte is sent, also a byte is
   received. This allows the following transmissions:

   1. Send-only: the received bytes are meaningless and therefore discarded.
          --> IOCTL_NSPI_SEND

   2. Receive-only: the sent data bytes are ignored at the device, therefore
      don't matter. Usually the value 0xFF is used as dummy value.
          --> IOCTL_NSPI_RECEIVE

   3. Send and receive: both data directions carry meaningful data.

      3a. Independent transfer: the data bytes to send are taken from one
          place and the received data bytes are stored at a different place.
          --> IOCTL_NSPI_TRANSFER

      3b. Replacing transfer: the received data bytes are stored at the same
          loaction as the bytes to send, replacing them one after the other.
          --> IOCTL_NSPI_EXCHANGE

   Table of parameters for DeviceIoControl():

   Transfer type                IN-data              OUT-data before / after
   ------------------------------------------------------------------------
   IOCTL_NSPI_SEND              Command              Send data / Send data
   IOCTL_NSPI_RECEIVE           Command              (unused)  / Received data
   IOCTL_NSPI_TRANSFER          Command & send data  (unused)  / Received data
   IOCTL_NSPI_EXCHANGE          Command              Send data / Received data

   Most SPI devices require some command bytes to determine what to do before
   transmitting the actual data. This is a send-only phase, i.e. the bytes
   received during this phase are discarded. If the device does not require
   command bytes, the command part may be left empty.

   When using IOCTL_NSPI_TRANSFER, the command size is determined by the
   difference of the IN-data and OUT-data array sizes. For example if 10 bytes
   go in and 8 bytes go out, the command size is 2 bytes.

   Some SPI devices require a short delay between command and data bytes. This
   delay can be configured starting with V3.0 of this driver. As it is
   usually only a short delay, it is given in nanoseconds and implemented as a
   busy-wait loop.

   Some SPI devices issue an interrupt when data is available to transfer. For
   these devices we have introduced new transfer controls in V3.0 that first
   wait for the occurance of an interrupt on a configurable I/O pin and only
   then start the SPI cycle:

   Transfer type                IN-data              OUT-data before / after
   ------------------------------------------------------------------------
   IOCTL_NSPI_WAITIRQ_SEND      Command              Send data / Send data
   IOCTL_NSPI_WAITIRQ_RECEIVE   Command              (unused)  / Received data
   IOCTL_NSPI_WAITIRQ_TRANSFER  Command + Send data  (unused)  / Received data
   IOCTL_NSPI_WAITIRQ_EXCHANGE  Command              Send data / Received data

   Remark:
   When using IOCTL_NSPI_SEND or IOCTL_NSPI_WAITIRQ_SEND, you can either send
   the data as part of the command in the IN-array or as separate data in the
   OUT-array. This can make handling of data easier in some cases. However
   please note that a delay between command and data bytes can only be used
   when the data is provided in the OUT-array because otherwise the driver
   does not know how many bytes are command bytes and how many bytes are data
   bytes. */
/* New IOControlCode values */
#define FILE_DEVICE_NSPI    0x0000800A
```

```
/* Send command to SPI device. After an optional delay, send data to SPI
   device. Any data that is received during this time is discarded. */
#define IOCTL_NSPI_SEND \
    CTL_CODE(FILE_DEVICE_NSPI, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Send command to SPI device. Any data received during this phase is
   discarded. After an optional delay, send dummy bytes to SPI device while
   receiving data from the device at the same time. Store received data at
   given place. */
#define IOCTL_NSPI_RECEIVE \
    CTL_CODE(FILE_DEVICE_NSPI, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Send command to SPI device. Any data received during this phase is
   discarded. After an optional delay, send data to SPI device while receiving
   data from the device at the same time. Store received data at new place. */
#define IOCTL_NSPI_TRANSFER \
    CTL_CODE(FILE_DEVICE_NSPI, 0x802, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Send command to SPI device. Any data received during this phase is
   discarded. After an optional delay, send data to SPI device while receiving
   data from the device at the same time. Replace old sent data with the newly
   received data. */
#define IOCTL_NSPI_EXCHANGE \
    CTL_CODE(FILE_DEVICE_NSPI, 0x803, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Get current clock frequency */
#define IOCTL_NSPI_GET_CLOCKFREQ \
    CTL_CODE(FILE_DEVICE_NSPI, 0x804, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Set new clock frequency */
#define IOCTL_NSPI_SET_CLOCKFREQ \
    CTL_CODE(FILE_DEVICE_NSPI, 0x805, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Get current SPI mode */
#define IOCTL_NSPI_GET_MODE \
    CTL_CODE(FILE_DEVICE_NSPI, 0x806, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Set new SPI mode */
#define IOCTL_NSPI_SET_MODE \
    CTL_CODE(FILE_DEVICE_NSPI, 0x807, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Get current driver method */
#define IOCTL_NSPI_GET_METHOD \
    CTL_CODE(FILE_DEVICE_NSPI, 0x808, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Set new driver method */
#define IOCTL_NSPI_SET_METHOD \
    CTL_CODE(FILE_DEVICE_NSPI, 0x809, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Get current dummy send byte */
#define IOCTL_NSPI_GET_DUMMYBYTE \
    CTL_CODE(FILE_DEVICE_NSPI, 0x80A, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Set new dummy send byte */
#define IOCTL_NSPI_SET_DUMMYBYTE \
    CTL_CODE(FILE_DEVICE_NSPI, 0x80B, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Get current delay between command and data bytes (in ns!!!) */
#define IOCTL_NSPI_GET_DATADELAY \
    CTL_CODE(FILE_DEVICE_NSPI, 0x80C, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Set new delay between command and data bytes (in ns!!!) */
#define IOCTL_NSPI_SET_DATADELAY \
    CTL_CODE(FILE_DEVICE_NSPI, 0x80D, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Get current general timeout (in ms) */
#define IOCTL_NSPI_GET_GENTIMEOUT \
    CTL_CODE(FILE_DEVICE_NSPI, 0x80E, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

```
/* Set new general timeout (in ms) */
#define IOCTL_NSPI_SET_GENTIMEOUT \
    CTL_CODE(FILE_DEVICE_NSPI, 0x80F, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Wait for IRQ, then send command to SPI device. After an optional delay,
   send data to SPI device. Any data that is received during this time is
   discarded. */
#define IOCTL_NSPI_WAITIRQ_SEND \
    CTL_CODE(FILE_DEVICE_NSPI, 0x810, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Wait for IRQ, then send command to SPI device. Any data received during
   this phase is discarded. After an optional delay, send dummy bytes to SPI
   device while receiving data from the device at the same time. Store
   received data at given place. */
#define IOCTL_NSPI_WAITIRQ_RECEIVE \
    CTL_CODE(FILE_DEVICE_NSPI, 0x811, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Wait for IRQ, then send command to SPI device. Any data received during
   this phase is discarded. After an optional delay, send data to SPI device
   while receiving data from the device at the same time. Store received data
   at new place. */
#define IOCTL_NSPI_WAITIRQ_TRANSFER \
    CTL_CODE(FILE_DEVICE_NSPI, 0x812, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Wait for IRQ, then send command to SPI device. Any data received during
   this phase is discarded. After an optional delay, send data to SPI device
   while receiving data from the device at the same time. Replace old sent
   data with the newly received data. */
#define IOCTL_NSPI_WAITIRQ_EXCHANGE \
    CTL_CODE(FILE_DEVICE_NSPI, 0x813, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Get current IRQ timeout (in ms) */
#define IOCTL_NSPI_GET_IRQTIMEOUT \
    CTL_CODE(FILE_DEVICE_NSPI, 0x814, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Set new IRQ timeout (in ms) */
#define IOCTL_NSPI_SET_IRQTIMEOUT \
    CTL_CODE(FILE_DEVICE_NSPI, 0x815, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Clear any pending interrupt, i.e. the WAITIRQ functions will only be
   triggered by an interrupt that happens after this call here. */
#define IOCTL_NSPI_CLEAR_IRQ \
    CTL_CODE(FILE_DEVICE_NSPI, 0x816, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Set/get device ID, ID=1 CS=SPI0_CS, ID=2 CS=Pin70*/
#define IOCTL_NSPI_ASSERT_CS \
    CTL_CODE(FILE_DEVICE_NSPI, 0x817, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define NSPI_CS_NORMAL 0
#define NSPI_CS_ASSERT 1
#define NSPI_CS_RELEASE 2

#define IOCTL_NSPI_GET_DEVICEID \
    CTL_CODE(FILE_DEVICE_NSPI, 0x818, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_NSPI_SET_DEVICEID \
    CTL_CODE(FILE_DEVICE_NSPI, 0x819, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* In addition, you can use the IOCTL_DRIVER_GETINFO call to get the driver
   version. Please include <fs_driverinfo.h> to use this call. */

#endif /*!__NSPIIO_H__*/
```

Listing 31: Header File nspiio.h

9 Appendix

Listings

List of Figures

List of Tables

Important Notice

The information in this publication has been carefully checked and is believed to be entirely accurate at the time of publication. F&S Elektronik Systeme assumes no responsibility, however, for possible errors or omissions, or for any consequences resulting from the use of the information contained in this documentation.

F&S Elektronik Systeme reserves the right to make changes in its products or product specifications or product documentation with the intent to improve function or design at any time and without notice and is not required to update this documentation to reflect such changes.

F&S Elektronik Systeme makes no warranty or guarantee regarding the suitability of its products for any particular purpose, nor does F&S Elektronik Systeme assume any liability arising out of the documentation or use of any product and specifically disclaims any and all liability, including without limitation any consequential or incidental damages.

Products are not designed, intended, or authorised for use as components in systems intended for applications intended to support or sustain life, or for any other application in which the failure of the product from F&S Elektronik Systeme could create a situation where personal injury or death may occur. Should the Buyer purchase or use a F&S Elektronik Systeme product for any such unintended or unauthorised application, the Buyer shall indemnify and hold F&S Elektronik Systeme and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, expenses, and reasonable attorney fees arising out of, either directly or indirectly, any claim of personal injury or death that may be associated with such unintended or unauthorised use, even if such claim alleges that F&S Elektronik Systeme was negligent regarding the design or manufacture of said product.