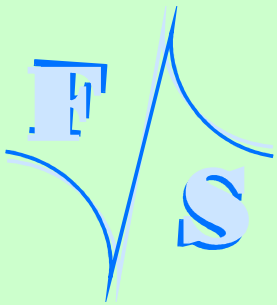
NSPI Driver

Native SPI Support

Version 1.7
(2009-07-13)

PicoCOM1
PicoCOM2

Windows CE



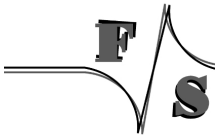
About This Document

This document describes how to configure the SPI driver and how to use it in own software applications. As the drivers for PicoCOM1 and PicoCOM2 are nearly identical this documentation is desired to be used for both boards. Platform specific differences are noted explicitly.

The latest version of this document can be found at http://www.fs-net.de.

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

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



History

Date	V	Platform	A,M,R	Chapter	Description	Au
2009-07-13	1.6	PicoCOM1/2	A, M	*	New document format A4. One documentation for Picoc-COM1 and PicoCOM2	MK
2009-07-16	1.7	PicoCOM1/2	A, M	4	Clock rate limitations added.	MK

V	Version
A,M,R	Added, Modified, Removed
Au	Author

Table of Contents

1 Introduction

Due to a lot of configuration values the SPI interface of the PicoCOM is very flexible, even up to high clock ranges. The PicoCOM features 3 Chip-Select lines directly (CS0-CS2). Additionally it offers the possibility to combine these lines and select up to 7 devices, using an external decoder logic.

Note - PicoCOM1:

Please note that CS3 is restricted for internal use only. For this reason its configuration should not be changed.

Note - PicoCOM2:

By default only CS0 can be used without restriction. CS1-CS2 are linked with some LCD signals. Please refer to the Hardware or Device Driver Documentation for more details.

2 Pin Assignment

The following table shows the dedicated SPI lines, located on the 80 pin main connector and connector on the PicoCOM Startinterface.

Board	Connec-tor	MISO	MOSI	CLK	CS0	CS1	CS2
PicoCOM1	Main	Pin 56	Pin 55	Pin 57	Pin 58	Pin 59	Pin 60
	J11	Pin 16	Pin 15	Pin 17	Pin 18	Pin 19	Pin 20
PicoCOM2	Main	Pin 26	Pin 27	Pin 28	Pin 29	Pin 53	Pin 54
	J10	Pin 3	Pin 4	Pin 5	Pin 6	n.c.	n..c.

Table 1: 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.

3 Installing the NSPI Driver

The NSPI driver is usually installed as `SPI0:`. We provide a special Windows Cabinet File ("CAB-File") for an automatic installation, but you can also do the installation manually.

Note – PicoCOM1:

In difference to the PicoCOM2 the SPI driver is included in each default kernel for the PicoCOM1

3.1 Installation with the CAB file

The easiest way to install the driver is to use the provided Windows Cabinet File `NSPI-PicoCOM2.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 `SPI0:`. 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 PicoCOM, 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-PicoCOM2.cab
```

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

4 Configuration

You can also do the installation by hand. This requires setting some registry values. The timing parameters of each Chip-Select line can be configured individually. These configuration values take place in the registry under

```
[HKLM\Drivers\BuiltIn\SPICS0]
[HKLM\Drivers\BuiltIn\SPICS1]
[HKLM\Drivers\BuiltIn\SPICS2]
```

Entry	Type	Value	Description
Dll	*String*	*pc<n>_spi.dll*	*Driver DLL*
FriendlyName	*String*	*Native SPI driver*	*Description*
Prefix	*String*	*SPI*	*For SPI0:*
Index	*DWORD*	*0*	*For SPI0:*
Order	*DWORD*	*101*	*Load sequence*
SPIController	DWORD	1	Index of the SPI controller do be used (must be 1)
ClockFreq	DWORD	420000	SPI clock in Hz
ClockDelay	DWORD	0	Clock delay after Chip-Select signal
ByteDelay	DWORD	0	Minimum delay between each byte transfer
Mode	DWORD	0	SPI-Mode (See SPI-Modes)
SPICS	DWORD	0…3	Chip-Select of the corresponding device
Timeout	DWORD	2000	Number of milliseconds wait for end of transfer
PollingMode	DWORD	0	1: Polling-Mode 0: IRQ-Mode
Priority256	DWORD	103	Thread priority
Debug	DWORD	0	Debug verbosity

Table 2: NSPI Registry Values

Most of the values will get meaningful defaults if omitted, only those values highlighted in blue/grey and italics above in the first few rows really have to be given. The library pc<n>_spi.dll has to be stored in flash memory into the \FFSDISK directory, if it is not already pre-loaded in the kernel.

Additionally there are some more controller specific values. These takes place under the registry key

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

where *X* is the 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
`ModeFaultDetect`	DWORD	`1`	Mode-Fault detection 1: Enabled 0: Disabled
`LoopBackEnable`	DWORD	`0`	Loop-Back MOSI and MISO line 1: Enabled 0: Disabled
`CSDelay`	DWORD	`0`	Chip-Select delay (see Timing Paramters)
`CSDecode`	DWORD	`0`	Enable Chip-Select decoding 1: Enabled 0: Disabled

Table 3: NSPI Registry Settings for the SPI Controller

4.1 Description of the Available Registry Values

4.1.1 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.

4.1.2 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.

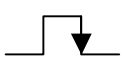
4.1.3 Polling-Mode

By default the SPI Driver runs in IRQ-mode. After establishing a DMA cycle the driver goes into sleep. After the DMA buffer is running out, the driver is woken up again by an SPI interrupt to start the next DMA transfer. This method is very efficient, as the driver doesn't block other processes on the PicoCOM. The DMA buffer in most cases will be big enough to transfer the data in one cycle. This causes the driver to release execution time directly and wait for the corresponding interrupt. In some cases, especially for very small data packages, waiting for an Interrupt could slow down the response time.

To finish transferring as fast as possible Polling-Mode can be enabled. This causes the driver to poll the transfer state directly after starting the DMA cycle. When sending a lot of small data packages this might be the preferred method.

4.1.4 Mode

The registry value `Mode` 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	

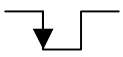
2	Clock active low, data valid on 1st (=falling) edge	
3	Clock active low, data valid on 2nd (=rising) edge	

Table 4: SPI Modes

4.1.5 Timing parameters (ClockDelay, ByteDelay, CSDelay)

Altogether there are three time parameters:

- **`ClockDelay:`** Time between triggering the Chip-Select line and starting data transfer.
  PicoCOM1: `<ClockDelay> = <ClockDelay value> x 100MHz`
  PicoCOM2: `<ClockDelay> = <ClockDelay value> x 120MHz`
- **`ByteDelay:`** Delay between consecutive transfers.
  PicoCOM1: `<ByteDelay> = <ByteDelay value> x 100MHz`
  PicoCOM2: `<ByteDelay> = <ByteDelay value> x 120MHz`
- **`CSDelay:`** Rest period between accessing different Slaves (Chip-Select switch).
  PicoCOM1: `<CSDelay> = <CSDelay value> x 100MHz / 32`
  PicoCOM2: `<CSDelay> = <CSDelay value> x 120MHz / 32`

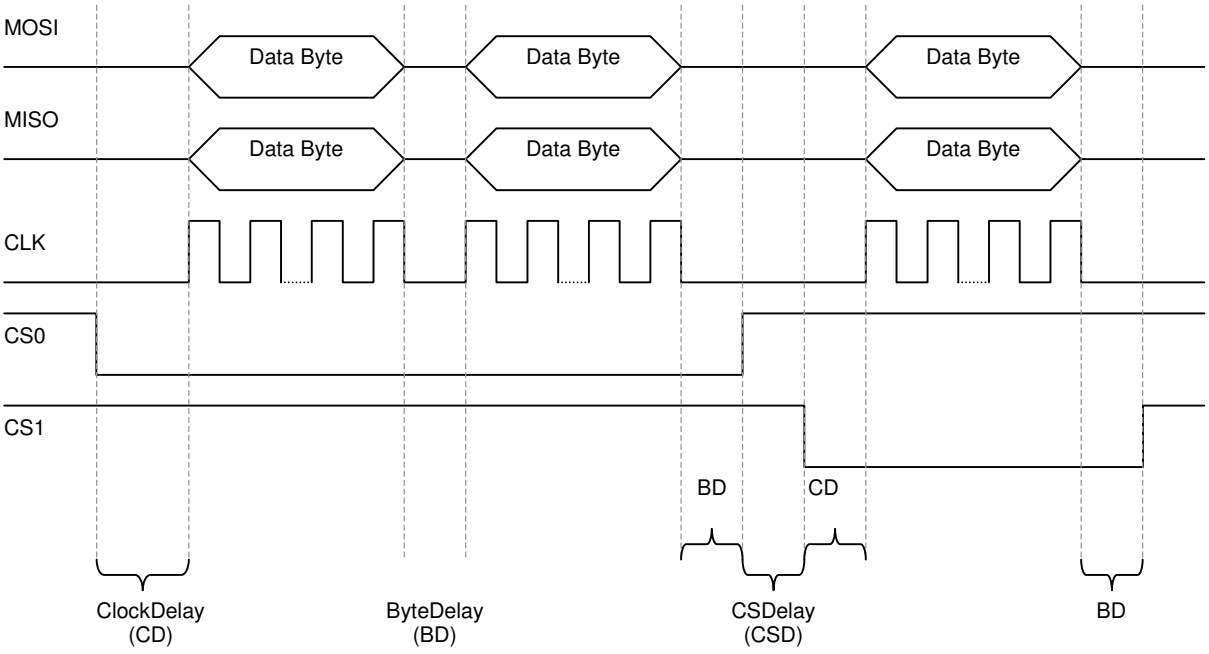


Figure 1: SPI timing parameters

4.1.6 ClockFreq

Due to dependencies of internal clocks the clock range of the SPI interface is limited. On PicoCOM1 for example the SPI-Clock can operate between ~390kHz and ~100MHz.

Additionally please note that only values devisable by the maximum clock-rate are possible. For example if a destination rate of 650kHz is desired on PicoCOM1, an essential clock of about 100MHz/154 ≈ 649kHz will be used.

Board	Min clock rate	Max clock rate
PicoCOM1	~390kHz	~100Mhz
PicoCOM2	~470KHz	~120MHz

Table 5: Possible clock rates

4.2 Chip-Select Decoding

The SPI-controller of the PicoCOM natively supports connecting a 4-16 bit decoder to generate up to 15 Chip-Select signals. As CS3 is unavailable on the PicoCOM, only CS0-CS2 can be used for decoding (up to 7 devices).

Chip-Select decoding is enabled by setting the value `CSDecode` in the controller registry key to `1`. By enabling this feature there can be created one SPI registry entry for each physically available device. This means that there will be one device (`SPIX:`) in WindowsCE for each device connected to the Chip-Select decoder:

```
[HKLM\Drivers\BuiltIn\SPICS0]
[HKLM\Drivers\BuiltIn\SPICS1]
                …
[HKLM\Drivers\BuiltIn\SPICS6]
```

When doing so the `SPICS` configuration value defines the combined output-value of the three Chip-Select lines. When no transfer is in process all CS lines are high.

SPICS	CS2	CS1	CS0	configuration set
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	2
5	1	0	1	2
6	1	1	0	2
7	idle state			

Table 6: Chip-Select decoding

Please note that the configuration for each Chip-Select output value can not be configured separately. As shown in the table above, the Chip-Select coded values 0 – 3 and 4 – 6 each share one configuration set. It is recommended to keep the corresponding parameters consistent within its configuration set.

Remark – PicoCOM1:

When Chip-Select decoding is activated, it must be taken into account that CS3 may not be used for decoding (compare table above). Additionally there are some adaptations for the CAN driver required. To avoid malfunctions the easiest way will be deactivating the CAN driver completely. This can be done by settings the Flags value in the CAN driver registry key to 4.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CAN1]
    "Flags"=dword:4
```

If you need both interfaces (CAN and SPI) and the CS-Decoding functionality is required, please contact the F&S support. We will give you the configuration set, fitting your needs.

5 The NSPI Driver in Applications

The NSPI driver is designed to work as master, therefore the connected device must be slave. This means that MOSI, CLK and CS are output signals and MISO is an input signal. The PicoCOM will generate the clock and chip select signals. The driver uses the common file interface (stream interface) to talk to the SPI bus.

Before the actual data transmission, most devices require some command to determine what to do with the data. For example a memory device will require information whether to read or write and also an address where to start. This command part is a send-only phase, i.e. the bytes received during this phase are discarded. This phase is optional. If the device does not require this command phase, you can leave it empty, i.e. use a length of 0 bytes.

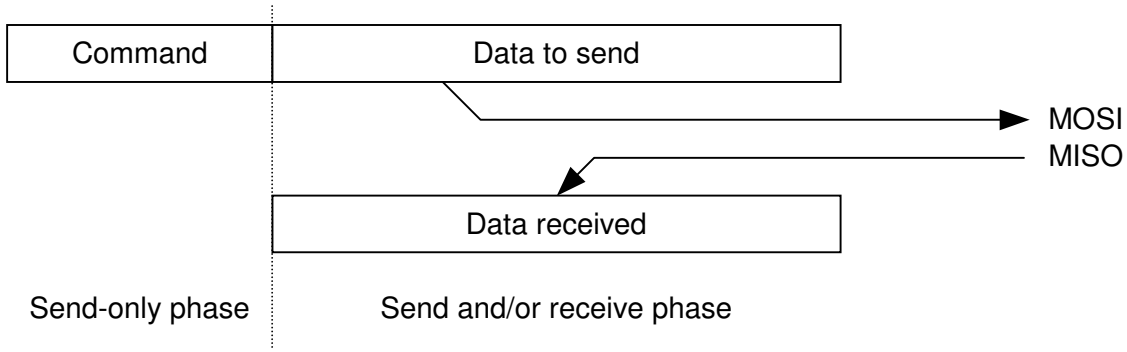


Figure 2: SPI Transfer Cycle

6 NSPI Reference

The driver uses the common file interface, and there mostly the `DeviceIoControl()` function to talk to the SPI bus.

When communicating to an SPI device, the transmission always goes in both directions at the same time. With every clock cycle, one bit is sent via the MOSI line to the device and one bit is received via the MISO line from the device. Therefore after one byte is sent, also one byte is received. This allows for the following transmission types.

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. We send `0xFF` as dummy values.
Send and receive	Both directions carry meaningful data. The received data is stored at a *different* place than the sent data.
Exchange	Both directions carry meaningful data. The received data is stored at the *same* place as the sent data, replacing it.

Before the actual data transmission, most devices require some command to determine what to do with the data. For example a memory device will require information whether to read or write and an address where to start. This command part is a send-only phase, i.e. the bytes received during this phase are discarded.

Therefore all transmission functions of the SPI driver also contain a command phase, that is performed before the actual data transfer takes place. If the device does not require this command phase, you can leave it empty, i.e. use 0 bytes.

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

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
Otherwise	File handle

Description:

Opens the `SPIx:` 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()`.

6.2 CloseHandle()

Signature:

```
BOOL CloseHandle(HANDLE hFileHandle);
```

Parameters:

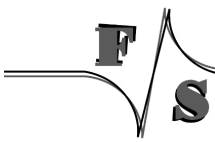
`hFileHandle`	Handle to device file

Return:

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

Description:

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



6.3 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 function to execute
`lpInBuffer`	Pointer to the data going into the function (IN data)
`dwInBufferSize`	Size of the IN data (in bytes)
`lpOutBuffer`	Pointer to a buffer where data coming out of the function 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 function. The type of function is given by a control code in parameter `dwIoControlCode`. Each function has a specific set of parameters. Usually there is some data going into the function (`IN` data) and some data is returned out of the function (`OUT` data).

The following table lists all control codes 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 7: IOCTL command codes for V1.x

6.3.1 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:

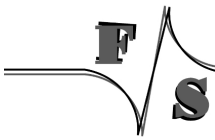
This command sends the command bytes and then the data bytes to the SPI device. All received bytes are discarded.

In fact 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.

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.



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

DeviceIoControl(hDevice, IOCTL_NSPI_SEND, data, 8,
                NULL, 0, NULL, NULL);
```

Listing 1: 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 command[3] =
{
     0x12, 0x34, 0x56
};

void Send(BYTE *data, int len)
{
     DeviceIoControl(hDevice, IOCTL_NSPI_SEND, command, 3,
                     data, len, NULL, NULL);
}
```

Listing 2: Example IOCTL_NSPI_SEND: Two Arrays

6.3.2 IOCTL_SPI_EXCHANGE

Parameters:

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_SPI_EXCHANGE`
`lpInBuffer`	Pointer to command bytes
`nInBufferSize`	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
`nOutBufferSize`	Number of bytes to send and receive
`lpReturned`	The referenced value will be set to `nOutBufferSize` 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.

Example:

Send command 0x55 0x66 and data bytes 0x01, 0x02, 0x03, 0x04 to the device and receive 4 bytes in return.

```
BYTE command[2] = {0x55, 0x66};
BYTE xdata[4] =
{
      0x01, 0x02, 0x03, 0x04      // send data
};

DeviceIoControl(hDevice, IOCTL_SPI_TRANSMIT,
                 command, 2, xdata, 4, NULL, NULL);
```

Listing 3: Example IOCTL_SPI_EXCHANGE

Remarks:

- This function needs one array going in, and one array going in and returning data 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.

6.3.3 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;
```

Listing 4: DRIVER_INFO structure

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.

Example:

Get the driver version and print it to stdout.

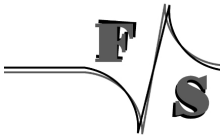
```
#include <fs_driverinfo.h>

...

DRIVER_INFO cInfo;

if (!DeviceIoControl(hDevice, 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 5: Example IOCTL_DRIVER_GETINFO

7 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, constants and data structures required to call the driver.

```
/*****************************************************************************/
/***                                                                       ***/
/***                                                                       ***/
/***                   S P I   D e v i c e   D r i v e r                   ***/
/***                                                                       ***/
/***                              f o r                                    ***/
/***                                                                       ***/
/***                           P i c o C O M                               ***/
/***                                                                       ***/
/***                                                                       ***/
/*****************************************************************************/
/*** File:     spiio.h                                                     ***/
/*** Authors:  Hartmut Keller                                              ***/
/*** Created:  14.03.2005                                                  ***/
/*** Modified: 07.07.2008 12:21:06 (MK)                                    ***/
/***                                                                       ***/
/*** Description:                                                          ***/
/*** Include file for SPI IOCTLs. You have to include this file to use the ***/
/*** SPI driver in your own applications.                                  ***/
/***                                                                       ***/
/*** Modification History:                                                 ***/
/*** 03.07.08 MK: Ported to PicoCOM1                                       ***/
/*****************************************************************************/

/*---------------------------------------------------------------------------
  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) 2005 F&S Elektronik Systeme GmbH
---------------------------------------------------------------------------*/

#ifndef __SPIIO_H__
#define __SPIIO_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 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_SPI_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_SPI_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_SPI_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_SPI_EXCHANGE

   Table of parameters for DeviceIoControl():
```



```
   Transfer type        IN-data               OUT-data before / after
   ------------------------------------------------------------------
   IOCTL_SPI_SEND      Command & send data   (unused)  / (unused)
   IOCTL_SPI_SEND      Command               Send data / Send data
   IOCTL_SPI_RECEIVE   Command               (unused)  / Received data
   IOCTL_SPI_TRANSFER  Command & send data   (unused)  / Received data
   IOCTL_SPI_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_SPI_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.

   When using IOCTL_SPI_SEND, you can either send the data as part of the
   command or as separate data in the OUT-array. This will generate the same
   output to the device, but it allows easier data handling in some cases.*/

/* New IOControlCode values */
#define FILE_DEVICE_SPI    0x0000800A

/* Send command and data to SPI device */
#define IOCTL_SPI_SEND \
    CTL_CODE(FILE_DEVICE_SPI, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Send command to SPI device and then receive data from SPI device */
#define IOCTL_SPI_RECEIVE \
    CTL_CODE(FILE_DEVICE_SPI, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Send command and data to SPI device, receive data from SPI device */
#define IOCTL_SPI_TRANSFER \
    CTL_CODE(FILE_DEVICE_SPI, 0x802, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Like IOCTL_SPI_TRANSFER, but replace send data with receive data */
#define IOCTL_SPI_EXCHANGE \
    CTL_CODE(FILE_DEVICE_SPI, 0x803, METHOD_BUFFERED, FILE_ANY_ACCESS)


#endif /*!__SPIIO_H__*/
```

Listing 6: Header File spiio.h

8 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 de-

sign 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.