Software Documentation
NetDCUx

Native SPI V1 – Software Interface for .NET

Version 1.00
2009-02-28

© F&S Elektronik Systeme GmbH
Untere Waldplätze 23
D-70569 Stuttgart
Fon: +49(0)711-123722-0
Fax: +49(0)711 – 123722-99

History

Date	V	Platform	A,M,R	Chapter	Description	Au
2015-07-09	1.00	all	M	*	Changed to new corporate design	JG

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

Table of Contents

1 Introduction

Some of the NetDCU, PicoMOD and PicoCOM boards support the so-called Native SPI, or NSPI for short. This is an SPI bus directly implemented by some dedicated hardware of the board, usually the micro-controller itself. This document describes, how the appropriate device driver is installed and how this SPI bus can be used in applications written in a Microsoft `.NET` programming language like `C#` or `Visual Basic`.
The main device driver only provides a Win32 interface. To use this driver from `.NET`, an additional library called `NativeSPI-V1.dll` is required. This library provides some useful data types and classes to access the NSPI driver interface in a comfortable way from the `.NET` environment. For example we introduce the wrapper class `NspiPort` for access and a special exception class `NSpiPortException`, allowing easy error handling.
In the following chapters, the programming concept of NSPI, all functions and all data types provided by `NativeSPI-V1.dll` are explained. We also have included a sample program, showing the usage of the `NspiPortV1` class.

Remark
In the remaining document we'll use the term "NetDCU" as generic reference to all our Windows CE boards. This should also include PicoMOD and PicoCOM boards, even if they are not mentioned especially. `NativeSPI-V1.dll` can only access the V1.x interface of the NSPI driver. A driver for V2.x is available soon.

2 Installing the NSPI Software 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.

2.1 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.

2.2 Manual installation

You can also do the installation by hand. This requires setting some registry entries. Installation of the NSPI driver takes place in the registry under
`[HKLM\Drivers\BuiltIn\SPI1]`

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 clock mode
`Priority256`	DWORD	`103`	Thread priority
`DriverMethod`	DWORD	`0`	IRQ, Polling, DMA
`TxChannel`	DWORD	`3`	Transmit DMA channel *)
`RxChannel`	DWORD	`4`	Receive DMA channel *)
`Debug`	DWORD	`0`	Debug verbosity

*) Only on PicoMOD3 when using DriverMethod 2 (DMA)

Most of the values will get meaningful defaults if omitted, only those values highlighted in grey above really have to be given.
Please refer to the document “NetDCU/PicoMOD: NSPI – Native SPI Support” for further installation details of the driver.

2.3 Installing the .NET library NativeSPI-V1.dll

To use the `NativeSPI-V1.dll` library for `.NET`, you have to copy it to your PC, for example to your Visual Studio project directory, and add a reference to it in your project. This can be done in two ways:

1. In the solution explorer, right click on the “References” entry and select “Add Reference...”
2. In menu “Project” select “Add Reference...”

In both cases you will be presented with a dialog having several tabs. Click on the tab “Browse” and search for the `NativeSPI-V1.dll` in your project directory. After clicking OK, entry “NativeSPI-V1” will appear in the References section of the Solution Explorer.

If the `NspiPortV1` class is not automatically recognized in the editor immediately, close and re-open your solution. Now the new objects should be supported by the editor.

3 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 NetDCU will generate the clock and chip select signals.
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 NSPI driver support 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. The .NET driver exposes an overloaded version of each transmission function with and without command bytes.

4 The NspiPortV1 class

The `NspiPortV1` class defines all functions needed for accessing the NSPI bus, including some constants and enumerations. The class is embedded in the `FS.NetDCU` namespace, so the fully qualified name is `FS.NetDCU.NspiPortV1`.

Error Handling
As with most low-level Windows drivers written in C, it is common for a function to return an error or success value as the direct return value and return any requested data in data structures passed by reference as parameters. Contrary to this, modern languages like C# usually use asynchronous exceptions to report failure and therefore can use the return value directly to transfer the requested data, usually as objects.
With the `NspiPortV1` class, we let you choose what behaviour you want. By default any error in a `NspiPortV1` function will throw a `NspiPortV1Exception`. However you can change this behaviour by calling `HandleErrorsViaReturn()` immediately after constructing the `NspiPortV1` object. This switches this instance to the C style convention and then each function returns 0 for success and an error value different from 0 for failure.

4.1 NspiPortV1() (Construction)

Signature:
`NspiPortV1(string FileName, NspiAccess access)`

Parameters:

`FileName`	Name of the device (`SPI:`, `SPI2:`)
`access`	Access type: Device query access, read access, write access, or read-write access.

Description:
Open the device file. Throw a `NspiPortV1Exception` if it fails. The device file is automatically closed by the destructor when the object is destroyed.
The NSPI bus usually has the device name `SPI<n>:` where `<n>` is the number of the port, usually 1. The access defines whether you want to transmit or receive messages.
For the description of `NspiAccess` see page 18.

Example:

```
try
{
      // Create a NspiPortV1 object
      NspiPortV1 nspi =
                  new NspiPortV1("SPI1:", NspiPortV1.NspiAccess.READ);
}
catch (NspiPortV1Exception e)
{
      // Handle error according to e.Reason
}
```

4.2 HandleErrorsViaReturn()

Signature:
`void HandleErrorsViaReturn(bool bCStyle)`

Parameters:

`bCStyle`	`true`:	Return error as return value
	`false`:	Throw exception on error (default)

Description:
Determine how errors are reported. This can be either by returning an error value (like in C), or by throwing an exception. This function is usually used right after creating the `NspiPortV1` object.
Please note that the constructor of `NspiPortV1` itself will always throw an exception on error. There is no way of defining the behaviour before calling the constructor and there is no way to return an error value from a constructor.

Example 1:
```
// Create a NspiPortV1 object
NspiPortV1 nspi =
          new NspiPortV1("SPI1:", NspiPortV1.NspiAccess.READ_WRITE);
// Set error handling by return value
nspi.HandleErrorsViaReturn(true);
// Send some data and check for error
int err = nspi.Send(...);
if (err != 0)
     Console.WriteLine("Error {0} in Send()", err);
```

Example 2:
```
// Create a NspiPortV1 object
NspiPortV1 nspi =
          new NspiPortV1("SPI1:", NspiPortV1.NspiAccess.READ_WRITE);
// Set error handling by return value
nspi.HandleErrorsViaReturn(true);
// Set exception error handling
nspi.HandleErrorsViaReturn(false);
// Send some data and check for error
try
{
     nspi.Send(...);
}
catch (NspiPortV1Exception e)
{
     Console.WriteLine("Error {0} in Send()", e.Reason);
}
```

Both examples do exactly the same, however one uses the error reporting via return values and the other the exception mechanism for errors.

4.3 Exchange()

Signature 1:
`int Exchange(byte[] data)`

Parameters:
`data` Array of data bytes to send; will be replaced with received data

Return:
`0` Success
`!=0` Error from `GetLastWin32Error()`

Description:
Sends all the bytes of the `data` array to the SPI bus. At the same time receives the same number of bytes and stores them also in the `data` array, replacing the previous content. Therefore this array is an IN and OUT parameter.

Remark:
Please make sure that you don't need this data anymore as it will be overwritten by this call. If you need a non-destructive transmission method, use `Transfer()` instead.

Signature 2:
```
int Exchange(byte[] cmd, byte[] data)
```

Parameters:

`cmd`	Array of command bytes to send
`data`	Array of data bytes to send; will be replaced with received data

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
First sends all the given bytes of the `cmd` array. This is a send-only phase, the bytes received during this phase are discarded. Then sends all the bytes of the `data` array to the SPI bus. At the same time receives the same number of bytes and stores them also in the `data` array, replacing the previous content. Therefore this array is an IN and OUT parameter.

Remark:

Please make sure that you don't need this data anymore as it will be overwritten by this call. If you need a non-destructive transmission method, use `Transfer()` instead.

4.4 Receive()

Signature 1:
```
int Receive(int len, out byte[] data)
```

Parameters:
`len` Number of bytes to receive
`data` Array of data bytes that are received

Return:
`0` Success
`!=0` Error from `GetLastWin32Error()`

Description:
Receives the given number `len` of bytes from the SPI bus and returns them as an array in `data`.

Signature 2:

```
int Receive(byte[] cmd,
            int len, out byte[] data)
```

Parameters:

`cmd`	Array of command bytes to send
`len`	Number of bytes to receive
`data`	Array of data bytes that are received

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:

First sends all the given bytes of the `cmd` array. This is a send-only phase, the bytes received during this phase are discarded. Then receives the given number `len` of bytes from the SPI bus and returns them as an array in `data`.

4.5 Send()

Signature 1:
```
int Send(byte[] data)
```

Parameters:
`data` Array of data bytes to send

Return:
`0` Success
`!=0` Error from `GetLastWin32Error()`

Description:
Sends all the given bytes of the `data` array to the SPI bus.

Signature 2:
```
int Send(byte[] cmd, byte[] data)
```

Parameters:

`cmd`	Array of command bytes to send
`data`	Array of data bytes to send

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
First sends all the given bytes of the `cmd` array, then all the given bytes from the `data` array to the SPI bus.

4.6 Transfer()

Signature 1:
```
int Transfer(byte[] sdata, out byte[] rdata)
```

Parameters:

`sdata`	Array of data bytes to send
`rdata`	Array of data bytes that are received

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:
Sends all the bytes of the `sdata` array to the SPI bus. At the same time receives the same number of bytes and returns them as `rdata` array. The number of bytes to transfer is given by the length of the `sdata` array.

Signature 2:

```
int Transfer(byte[] cmd,
             byte[] sdata, out byte[] rdata)
```

Parameters:

`cmd`	Array of command bytes to send
`sdata`	Array of data bytes to send
`rdata`	Array of data bytes that are received

Return:

`0`	Success
`!=0`	Error from `GetLastWin32Error()`

Description:

First sends all the given bytes of the `cmd` array in a send-only phase. The bytes received during this phase are discarded. Then sends all the bytes of the `sdata` array to the SPI bus. At the same time receives the same number of bytes and returns them as `rdata` array. The number of bytes to transfer in this phase is given by the length of the `sdata` array.

Remark:

Please keep in mind that the underlying NSPI driver function can only handle two arrays. Therefore this call internally needs to copy the `cmd` and `sdata` arrays to one single new array, causing some overhead. Sometimes it is possible to use the more efficient `Exchange()` function instead.

4.7 enum NspiAccess

Values:

`QUERY`	Just open the device to check parameters
`WRITE`	Open the device for write access
`READ`	Open the device for read-only access
`READ_WRITE`	Open the device for read/write access

Description:
These values may be given when creating the `NspiPortV1` object (see page 8). Usually you would like to use `READ_WRITE` for access.

4.8 enum APIError

Values:

`ERROR_SUCCESS`	No error
`ERROR_INVALID_FUNCTION`	Function not implemented
`ERROR_FILE_NOT_FOUND`	Device not found
`ERROR_ACCESS_DENIED`	Access to device denied
`ERROR_INVALID_HANDLE`	Invalid handle
`ERROR_NOT_READY`	Device not ready
`ERROR_WRITE_FAULT`	Write fault
`ERROR_GEN_FAILURE`	Generic device error
`ERROR_DEV_NOT_EXIST`	Device does not exist
`ERROR_INVALID_PARAMETER`	Bad parameters
`ERROR_INVALID_NAME`	Invalid device name
`ERROR_TIMEOUT`	Device timed out

Description:
The most common values that are reported as errors when calling the `NspiPortV1` functions. For additional values see the Win32 API.
Especially if the NSPI device driver is not installer, you'll get `ERROR_DEV_NOT_EXIST` when trying to create the `NspiPortV1` object.

5 The NspiPortV1Exception class

The `NspiPortV1Exception` class defines an exception used in combination with the `NspiPortV1` class. When an error happens within a function of `NspiPortV1`, it throws this kind of exception, so you can react to it in a `try-catch` statement.

The `NspiPortV1Exception` extends `ApplicationException` by a read-only property `int Reason`, showing the error code why the exception was thrown. This is usually the value returned by the Win32 API via `GetLastWin32Error()`. A typical piece of code would look like this.

```
try
{
      NspiPortV1 nspi = new NspiPortV1("SPI1:", ...);
      ...  // Use nspi
}
catch (NspiPortV1Exception e)
{
      switch (e.Reason)
      {
      case NspiPortV1.APIError.ERROR_DEV_NOT_EXIST:
            ...  // Handle error
      case NspiPortV1.APIError.ERROR_ACCESS_DENIED:
            ...  // Handle error
      }
}
```

When examining the reason, `NspiPortV1.APIError` (see page 18) may be of some help to check for possible error sources.

5.1 NspiPortV1Exception() (Construction)

Signature 1:

```
NspiPortV1Exception(string text, int reason)
```

Parameters:

`text` Error text
`reason` Error number

Description:
Store given error value as Reason. The error text is automatically completed with "`: Error code <reason>`" where <reason> is the given error number.

Signature 2:

```
NspiPortV1Exception(string text, int reason,
                    Exception inner)
```

Parameters:

`text` Error text
`reason` Error number
`inner` Inner exception

Description:
Same as above, but with inner exception.

Signature 3:

```
NspiPortV1Exception(string text)
```

Parameters:

`text` Error text

Description:

Same as above, but automatically use the result of `GetLastWin32Error()` as error number.

Signature 4:

```
NspiPortV1Exception(string text,
                    Exception inner)
```

Parameters:

`text` Error text

`inner` Inner exception

Description:

Same as above, with inner exception.

Signature 5:
```
NspiPortV1Exception(int reason)
```

Parameters:

`reason`	Error number

Description:
Use given error number and "`System error`" as error text.

Signature 6:
```
NspiPortV1Exception(int reason,
                    Exception inner)
```

Parameters:

`reason`	Error number
`inner`	Inner exception

Description:
Same as above, but with inner exception.

Signature 7:
`NspiPortV1Exception()`

Description:
Use `GetLastWin32Error()` as error number and string "`System error`" as error text.

Signature 8:
`NspiPortV1Exception(Exception inner)`

Parameters:
`inner` Inner exception

Description:
Same as above, but with inner exception.

6 Sample Program

The following program exchanges some data with an FRAM FM24CL64 with 64 Kbyte connected on `SPI1:`. The is a non-volatile RAM. Accessing it always requires a command phase.
Reading can be done anytime. A read command consists of a command byte and two bytes with the start address where to read from. Then any number of bytes can be read.
Writing is only possible if the device is unprotected and a special write latch is enabled. Then the write command also consists of a command byte and two bytes with the start address where to start writing. Then any number of bytes can be written.
The write protection is handled by a bit in the status register. Setting the write enable latch is a separate command.

```
/*********************************************************/
/*** File:         fm25cl64.cs                           ***/
/*** Author:       Hartmut Keller, (C) F&S 2009          ***/
/***                                                     ***/
/*** Description: Example using the NspiPortV1 class     ***/
/*********************************************************/
using FS.NetDCU;
namespace FS.NetDCU
{
      class Program
      {
            /* Command bytes for FRAM FM25CL64 */
            const byte MEM_WREN = 0x06;  /* Write Enable Latch */
            const byte MEM_WRDI = 0x04;  /* Write Disable */
            const byte MEM_RDSR = 0x05;  /* Read Status Reg */
            const byte MEM_WRSR = 0x01;  /* Write Status Reg */
            const byte MEM_READ = 0x03;  /* Read Memory Data */
            const byte MEM_WRITE = 0x02; /* Write Memory Data */
            /* The NativeSPI object */
            static NspiPortV1 nspi;
            /* Read some bytes from the FRAM */
            static void ReadMem(int addr, int count,
                                out byte [] data)
            {
                  /* Give read command and read bytes */
                  byte[] command =
                  {
                        MEM_READ,
                        (byte)(addr/256),
                        (byte)(addr%256)
                  };
                  nspi.Receive(command, count, out data);
            }
            /* Write some data to the FRAM */
            static void WriteMem(int addr, byte [] data)
            {
                  /* Set write enable latch */
                  byte[] command1 =
                  {
                        MEM_WREN
                  };
                  nspi.Send(command1);
```

```
/* Protect or unprotect the FRAM */
static void Protect(bool bOn)
{
      byte[] command = new byte[1];
      /* Set write enable latch */
      command[0] = MEM_WREN;
      nspi.Send(command);
      /* Read status register */
      byte[] sr;
      command[0] = MEM_RDSR;
      nspi.Receive(command, 1, out sr);
      /* Set/clear protect flag */
      if (bOn)
            sr[0] |= 0x0C; /* protect */
      else
            sr[0] &= 0xFF-0x0C; /* unprotect */
      /* Write back to status register */
      command[0] = MEM_WRSR;
      nspi.Send(command, sr);
}
/* Write some data to the FRAM */
static void WriteMem(int addr, byte [] data)
{
      /* Set write enable latch */
      byte[] command1 =
      {
            MEM_WREN
      };
      nspi.Send(command1);
      /* Give write command and write bytes */
      byte[] command2 =
      {
            MEM_WRITE,
            (byte)(addr/256),
            (byte)(addr%256)
      };
      nspi.Send(command2, data);
}
/* Main program: Do some FRAM transfers */
static void Main(string[] args)
{
      /* Create the NspiPortV1 object */
      nspi = new NspiPortV1("SPI1:",
                        NspiPortV1.NspiAccess.READ_WRITE);
      /* Read 100 bytes from the FRAM at address 0 */
      byte[] buffer;
      ReadMem(0, 100, out buffer);
      /* Unprotect the FRAM */
      Protect(false);
      /* Write "xxxxxx" at address 0 */
      byte[] cleardata =
      {
            (byte)'x', (byte)'x', (byte)'x',
            (byte)'x', (byte)'x', (byte)'x', 0
      };
      WriteMem(0, cleardata);
```

```
                    /* Read back data, should start with "xxxxxx" */
                    ReadMem(0, 100, out buffer);
                    /* Write "Hello world!" at address 0 */
                    byte[] demodata =
                    {
                          (byte)'H', (byte)'e', (byte)'l',
                          (byte)'l', (byte)'o', (byte)' ',
                          (byte)'w', (byte)'o', (byte)'r',
                          (byte)'l', (byte)'d', (byte)'!', 0
                    };
                    WriteMem(0, demodata);
                    /* Write protect FRAM */
                    Protect(true);
                    /* Read back data, starts with "Hello world!" */
                    ReadMem(0, 100, out buffer);
              }
        }
}
```

7 Appendix

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.
Specific testing of all parameters of each device is not necessarily performed unless required by law or regulation.
Products are not designed, intended, or authorized 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 unauthorized 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 unauthorized use, even if such claim alleges that F&S Elektronik Systeme was negligent regarding the design or manufacture of said product.
Specifications are subject to change without notice.

Warranty Terms

Hardware Warranties
F&S guarantees hardware products against defects in workmanship and material for a period of one (2) year from the date of shipment. Your sole remedy and F&S's sole liability shall be for F&S, at its sole discretion, to either repair or replace the defective hardware product at no charge or to refund the purchase price. Shipment costs in both directions are the responsibility of the customer. This warranty is void if the hardware product has been altered or damaged by accident, misuse or abuse.

Software Warranties
Software is provided "AS IS". F&S makes no warranties, either express or implied, with regard to the software object code or software source code either or with respect to any third party materials or intellectual property obtained from third parties. F&S makes no warranty that the software is useable or fit for any particular purpose. This warranty replaces all other warranties written or unwritten. F&S expressly disclaims any such warranties. In no case shall F&S be liable for any consequential damages.

Disclaimer of Warranty

THIS WARRANTY IS MADE IN PLACE OF ANY OTHER WARRANTY, WHETHER EXPRESSED, OR IMPLIED, OF MERCHANTABILITY, FITNESS FOR A SPECIFIC PURPOSE, NON-INFRINGEMENT OR THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION, EXCEPT THE WARRANTY EXPRESSLY STATED HEREIN. THE REMEDIES SET FORTH HEREIN SHALL BE THE SOLE AND EXCLUSIVE REMEDIES OF ANY PURCHASER WITH RESPECT TO ANY DEFECTIVE PRODUCT.

Limitation on Liability

UNDER NO CIRCUMSTANCES SHALL F&S BE LIABLE FOR ANY LOSS, DAMAGE OR EXPENSE SUFFERED OR INCURRED WITH RESPECT TO ANY DEFECTIVE PRODUCT. IN NO EVENT SHALL F&S BE LIABLE FOR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES THAT YOU MAY SUFFER DIRECTLY OR INDIRECTLY FROM USE OF ANY PRODUCT. BY ORDERING THE PRODUCT, THE CUSTOMER APPROVES THAT THE F&S PRODUCT, HARDWARE AND SOFTWARE, WAS THOROUGHLY TESTED AND HAS MET THE CUSTOMER'S REQUIREMETS AND SPECIFICATIONS