Software Documentation NetDCUx

Native I2C – Software Interface for .NET

Version 1.01
2007-10-04

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	1.01	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 and PicoMOD boards support the so-called Native I²C, or NI2C for short. This is an I²C 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 I²C bus can be used in applications written in a Microsoft `.NET` programming language like `C#` or `Visual Basic`.
The main device driver provides a Win32 interface. To use this driver from `.NET`, an additional library called `NativeI2C.dll` is required. This library provides some useful data types and classes to access the NI2C driver interface in a comfortable way from the `.NET` environment. For example we introduce the wrapper class `NI2CFile` for access and a special exception class `NI2CException`, allowing easy error handling.
In the following chapters, the programming concept of NI2C, all functions and all data types provided by `NativeI2C.dll` are explained. We also have included some sample programs, showing the usage of the `NI2CFile` class.
Please note that this documentation is only valid for the Native I2C. There is a second I2C driver available on some boards emulating the I²C bus with GPIOs. This other driver will not be discussed here.

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 boards, even if they are not mentioned especially.

2 Installing the NI2C Software Driver

The NI2C driver is usually installed as `I2C1:`. 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 `ni2c.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 `I2C1:`. 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 ni2c.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 I²C driver takes place in the registry under `[HKLM\Drivers\BuiltIn\I2C1]`

Entry	Type	Value	Description
`Dll`	String	`ni2c.dll`	Driver DLL
`FriendlyName`	String	`Native I2C driver`	Description
`Prefix`	String	`I2C`	For `I2C1:`
`Index`	DWORD	`1`	For `I2C1:`
`Order`	DWORD	`101`	Load sequence
`ClockFreq`	DWORD	`200000`	in Hz
`Priority256`	DWORD	`103`	Thread priority
`Debug`	DWORD	`0`	Debug verbosity

Most of the values will get meaningful defaults if omitted, only those values highlighted in grey above really have to be given. The library `ni2c.dll` has to be stored in flash memory into the `\FFSDISK` directory, if it is not already pre-loaded in the kernel.
There is a file `ni2c-reg.txt` provided on the CD that allows doing these settings in `NDCUCFG`. Just edit the file to set your specific values, then send the text file to `NDCUCFG`.
Please refer to the document “NetDCU: NI2C – Native I²C Support” for further installation details of the driver.
If you also plan to use the 5-wire touch panel adapter `NetDCU-ADP-TP5`, then `ni2c.dll` *must* be installed as `I2C1:`, or the touch panel won’t work. Please refer to the separate document “NetDCU: NetDCU-ADP-TP5 – 5-Wire Touch Panel” (`NetDCU_ADP-TP5_eng.pdf`) for how to set up the touch panel.

2.3 Installing the .NET library NativeI2C.dll

To use the `NativeI2C.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:

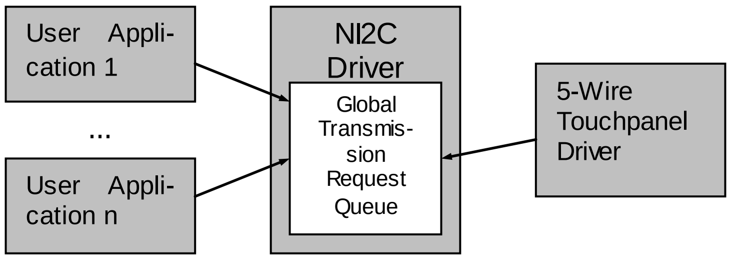
1. In the solution explorer, right click on the “References” entry and select “Add Reference...”

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 `NI2CFile.dll` in your project directory. After clicking OK, entry “NativeI2C” will appear in the References section of the Solution Explorer.
If the `NI2CFile` 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 NI2C Driver in Applications

When using the NI2C driver in own applications, please keep in mind that you always have to co-operate with other applications using devices on the same I²C bus. For example it is a rather common case, that the 5-wire touch panel driver also accesses the same bus, so don't block the bus longer than required or else the touch panel operation will suffer.
On the other hand you have to be aware that the touch panel driver communicates with its I²C hardware about 50 times per second (on standard settings) and thus may issue transfers between any of your own transmission requests, probably delaying your communication slightly. However the NI2C driver keeps transmission requests strictly separated, and serves them in a first come first serve manner, as fast as the I²C bus allows. So the data of different programs will not mix up, each request is finished before the next request is scheduled.



Remark:

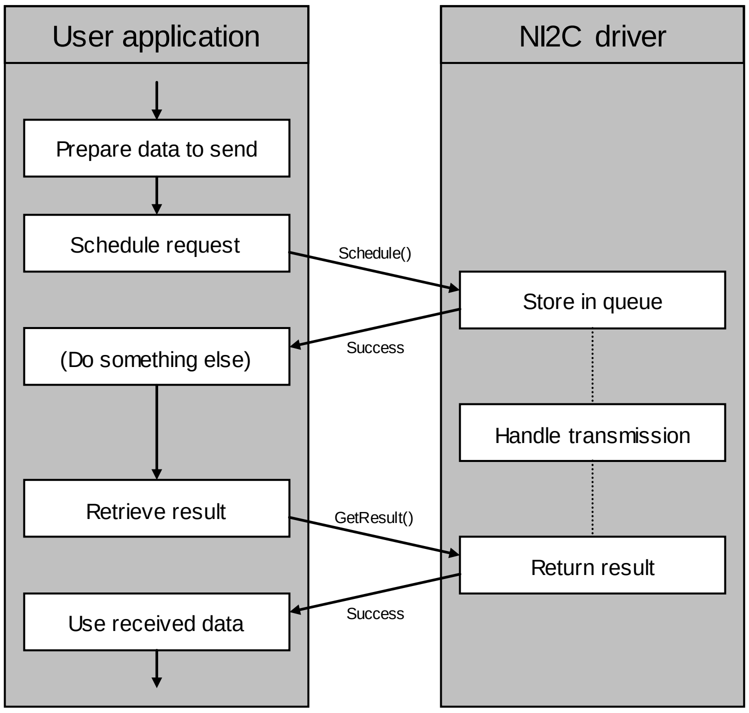
The NI2C driver is designed to work as a sole master, all other devices must be slaves. If another device on the I²C bus takes the role of a master and starts sending, the driver will fail.

3.1 Messages and Transmission Requests

A message is the basic element of communicating with a device. A message may either send some bytes to, or receive some bytes from a specific I²C device.
A transmission request is a group of arbitrary messages, executed in one go. Therefore a transmission request can switch forth and back between sending and receiving at will, depending on the contained messages. It is also not restricted to communicate with one single device, each message can talk to a different device.

Example
This is a transmission request with four messages.
1. Send 1 byte to device A
2. Receive 2 bytes from device A
3. Send 3 bytes to device B
4. Receive another 2 bytes from device A

Such a transmission request is handled as a whole, even if other requests are in the queue. It is essential that every program participating in the I²C bus communication behaves fair and only groups those messages in a single transmission request that really must belong together and can not be split.
The NI2C driver handles transmission requests in a non-blocking way. So you first have to prepare the request, including all the data bytes to be sent, and then call a `Schedule()` function. This function puts the request in a global request queue and then returns immediately. The driver now handles the transmission in the background and later, when the transmission is complete, you can call another function `GetResult()` to retrieve the result, i.e. the data bytes that were received and the success status of each message.

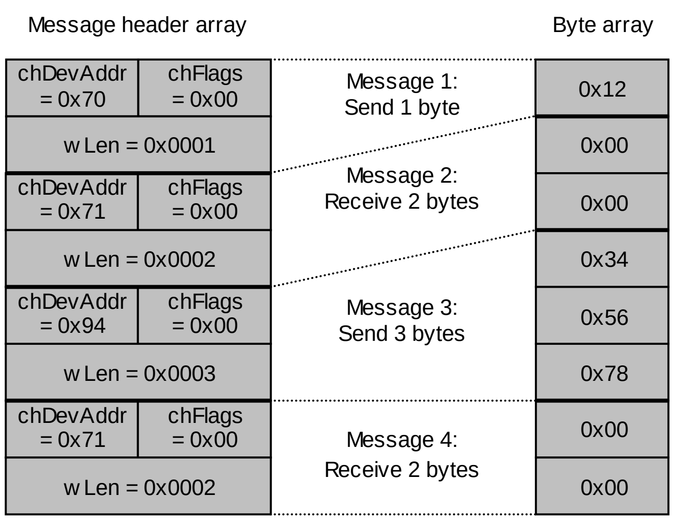


The data structures are rather simple. You have to prepare two arrays. One array with message headers defining the message parameters, and a second array with all the bytes to transfer. This second array comprises all bytes of all messages is one array. You have to give dummy bytes in those places, where data will be received, as the NI2C driver will simply fill in the received data into these spaces. This allows giving the same data structures and pointers to the scheduling function as well as the result retrieving function.

File `ni2cio.h` contains the required types and structures.

```
/* Status flags used in NI2C_MSG_HEADER */
[Flags]
public enum NI2C_FLAGS : uint
{
      LASTBYTE_ACK = 0x01,
      DATA_NAK = 0x02,
      DEVICE_NAK = 0x04,
      ARBITRATION_LOST = 0x08,
      TIMEOUT = 0x80
}
/* Message header */
[StructLayout(LayoutKind.Sequential)]
public struct NI2C_MSG_HEADER
{
      public byte chDevAddr;
      public byte chFlags;
      public ushort wLen;
      public NI2C_MSG_HEADER(byte chDevAddr, byte chFlags,
                                             ushort wLen);
}
```

As you can see, the transfer direction (send or receive) is indicated in bit 0 of the device address byte. Therefore the address is in bits 1 to 7 (shifted 1 bit to the left). This is identical to the way how address and direction are actually transmitted on the I²C bus. We always use this representation for device addresses in this document. For example instead of the unshifted address 0x38 we use the shifted address 0x70 (which gets 0x71 when receiving data).
Let's continue the transmission request example from above. Assume address A is 0x70, address B is 0x94, the byte to send in step 1 is 0x12, and the three bytes to send in step 3 are 0x34, 0x56, and 0x78. We'll use 0x00 for the receive dummy bytes. Then the data structure for this transmission request is as follows.


Message header array
Byte array
chDevAddr = 0x70
chFlags = 0x00
w Len = 0x0001
chDevAddr = 0x71
chFlags = 0x00
w Len = 0x0002
chDevAddr = 0x94
chFlags = 0x00
w Len = 0x0003
chDevAddr = 0x71
chFlags = 0x00
w Len = 0x0002
Message 1: Send 1 byte
Message 2: Receive 2 bytes
Message 3: Send 3 bytes
Message 4: Receive 2 bytes
0x12
0x00
0x00
0x34
0x56
0x78
0x00
0x00

This might result in the following C# code.

```
/***********************************************************/
/*** File:        ni2c-example.cs                        ***/
/*** Author:      Hartmut Keller, (C) F&S 2007           ***/
/***                                                     ***/
/*** Description: Example for NI2C transmission request ***/
/***********************************************************/
using System;
using System.Collections.Generic;
using System.Text;
using FS.NetDCU;
namespace FS.NetDCU
{
      class NI2C_Example
      {
          static void Main(string[] args)
          {
               /* Message headers for transmission request */
               NI2CFile.NI2C_MSG_HEADER[] mymsg =
                         new NI2CFile.NI2C_MSG_HEADER[]
               {
                    new NI2CFile.NI2C_MSG_HEADER(0x70, 0x00, 0x0001),
                    new NI2CFile.NI2C_MSG_HEADER(0x71, 0x00, 0x0002),
                    new NI2CFile.NI2C_MSG_HEADER(0x94, 0x00, 0x0003),
                    new NI2CFile.NI2C_MSG_HEADER(0x71, 0x00, 0x0002)
               };
               /* Data bytes for transmission request */
               byte[] mydata =
               {
                    0x12,                        /* Message 1: send 1
byte */
                    0x00, 0x00,                  /* Message 2: receive
2 bytes */
                    0x34, 0x56, 0x78,       /* Message 3: send 3 bytes
*/
                    0x00, 0x00,                  /* Message 4: receive
2 bytes */
               };
               /* Create NI2CFile object */
               NI2CFile ni2c = new NI2CFile("I2C1:",

      NI2CFile.NI2CAccess.READ_WRITE);
               /* Schedule transmission request */
               ni2c.Schedule(mymsg, mydata);
               /* ... Do something else here ... */
               /* Retrieve and print result */
               ni2c.GetResult(mymsg, mydata);
               Console.WriteLine(
                    "Received in message 2: 0x{0:X2} 0x{1:X2}",
                    mydata[1], mydata[2]);
               Console.WriteLine(
                    "Received in message 4: 0x{0:X2} 0x{1:X2}",
                    mydata[6], mydata[7]);
          }
     }
}
```

3.2 Status Flags and Scanning the Bus

We haven't talked about the `chFlags` value contained in the message header yet. Here the NI2C driver registers the success or failure of each message transmission.
The I²C protocol defines that after each transferred byte the receiving side of the communication has to issue an acknowledgement. A missing acknowledgement right after sending the device address is a special case and indicates that no device with this address exists. Otherwise a missing acknowledgement after a data byte indicates the end of the transmission.
The NI2C driver reports in the `chFlags` value whether the message transmission succeeded or not. So after having retrieved the result of the transmission request, you can check the `chFlags` values of all individual messages to get a status report. It can be any combination of the following values.

Flag	Explanation
`LASTBYTE_ACK`	The last byte sent was acknowledged. This might be an indication for some error as the receiving device expected more data.
`DATA_NAK`	There was a missing acknowledgement somewhere in the middle of the message before the last byte. The receiving device could not take more data. The transmission was aborted at this point.
`DEVICE_NAK`	When sending the device address, it was not acknowledged, i.e. no device responded. The message could not be transmitted.
`ARBITRATION_LOST`	The I²C bus was already busy when the message transfer was about to start. Therefore the message could not be transmitted.
`TIMEOUT`	The NI2C driver did not get a hardware response within a timeout period of about 1 second

All flags but `LASTBYTE_ACK` indicate an aborted transfer. To mark the abortion point, the NI2C driver inverts the bit patterns of all bytes of the message that were not transmitted.

Example
The following message bytes should be sent:
`0x11 0x22 0x33 0x44 0x55`
After the transmission, the `chFlags` value shows that the `DATA_NAK` flag is set. The data bytes array now shows the corresponding bytes as
`0x11 0x22 0x33 0xBB 0xAA`
This tells us that the first three bytes were successfully transmitted, but the acknowledgement was missing on the third byte, and therefore the last two bytes were not transmitted anymore.

By looking at the `DEVICE_NAK` flag, we can tell if a device responded to the message. This can be used to scan the bus for the specific address or the mere existence of some device. Simply send the device address with no data bytes. If the device sends an acknowledgement, we know that it exists.

Example
The ADS7828 is an 8 channel ADC from Texas Instruments. The device has two configurable address lines, the five most significant bits are fixed as 10010. This means it can be configured to listen on address 0x90, 0x92, 0x94, or 0x96.
The following C# code determines the actual address of the device on the I²C bus.

```
/*******************************************************/
/*** File:         ni2c-scan.cs                     ***/
/*** Author:       Hartmut Keller, (C) F&S 2007     ***/
/***                                                ***/
/*** Description: NI2C bus scan for ADS7828 device   ***/
/*******************************************************/
using System;
using System.Collections.Generic;
using System.Text;
using FS.NetDCU;
namespace FS.NetDCU
{
	class NI2C_Scan
	{
		static void Main(string[] args)
		{
			/* Message headers for scan transmission request */
			NI2CFile.NI2C_MSG_HEADER[] scanmsg =
								new
NI2CFile.NI2C_MSG_HEADER[]
			{
				new NI2CFile.NI2C_MSG_HEADER(0x90, 0x00, 0x0000),
				new NI2CFile.NI2C_MSG_HEADER(0x92, 0x00, 0x0000),
				new NI2CFile.NI2C_MSG_HEADER(0x94, 0x00, 0x0000),
				new NI2CFile.NI2C_MSG_HEADER(0x96, 0x00, 0x0000)
			};
			/* No data bytes needed for the above request */
			byte[] scandata = {};
			/* Create NI2CFile object */
			NI2CFile ni2c = new NI2CFile("I2C1:",

	NI2CFile.NI2CAccess.READ_WRITE);
			/* Send scanning transmission request */
			ni2c.Schedule(scanmsg, scandata);
			ni2c.GetResult(scanmsg, scandata);
			/* Check if any of the scanned addresses was
			   acknowledged */
			byte myAddr = 0;
			foreach (NI2CFile.NI2C_MSG_HEADER msg in scanmsg)
			{
				if (msg.chFlags == 0)
				{
					myAddr = msg.chDevAddr;
					break;
				}
			}
			/* Print result */
			if (myAddr != 0)
				Console.WriteLine(
						"ADS7828 on address 0x{0:X2}",
myAddr);
			else
				Console.WriteLine("No ADS7828 found");
		}
	}
}
```

As you can see it is very easy to expand this example to scan for more device addresses. Simply add more messages to the `scanmsg` array.

One last thing. When receiving data, the `chFlags` value serves yet another purpose. Most I²C devices send data bytes until they receive no acknowledgement anymore. So the normal behaviour of the NI2C driver is to not acknowledge the last byte received. For example in a message to receive five bytes, the driver acknowledges the first four bytes, but not the fifth byte.
However in special cases you might like the driver to acknowledge all bytes, including the last byte. This can be achieved by setting the `chFlags` value of the specific message(s) to `LASTBYTE_ACK` before scheduling the request. Then the driver will also acknowledge the last byte.
Please note that setting this value makes only sense in receiving messages. Setting other bits in `chFlags` has no effect.

Remark

The examples `ni2c-example.cs` and `ni2c-scan.cs` are also available as Visual Studio 2005 projects on the CD.

4 The NI2CFile class

The `NI2CFile` class defines all functions needed for accessing the I²C bus, including some data types, constants and enumerations. The class is embedded in the `FS.NetDCU` name-space, so the fully qualified name is `FS.NetDCU.NI2CFile`.
First we will describe the member functions, then follow the data types used together with them in the second part of this chapter.

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 `NI2CFile` class, we let you choose what behaviour you want. By default any error in a `NI2CFile` function will throw a `NI2CException`. However you can change this behaviour by calling `HandleErrorsViaReturn()` immediately after constructing the `NI2CFile` 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 NI2CFile() (Construction)

Signature:
`NI2CFile(string FileName, NI2CAccess access)`

Parameters:

`FileName`	Name of the device (`I2C1:`, `I2C2:`)
`access`	Access type: Device query access, read access, write access, or read-write access.

Description:
Open the device file. Throw a `NI2CPortException` if it fails. The device file is automatically closed by the destructor when the object is destroyed.
The NI2C bus usually has the device name `I2C<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 `NI2Access` see page 23.

Example:
```
try
{
      // Create a NI2CFile object
      NI2CFile pNI2C =
                        new NI2CFile("I2C1:", NI2CFile.NI2CAccess.READ);
}
catch (NI2CException 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 NI2CFile object.
Please note that the constructor of NI2CFile 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 NI2CFile object
NI2CFile pNI2C =
                new NI2CFile("I2C1:", NI2CFile.NI2CAccess.READ_WRITE);
// Set error handling by return value
pNI2C.HandleErrorsViaReturn(true);
// Schedule a request and check for error
int err = pNI2C.Schedule(...);
if (err != 0)
     Console.WriteLine("Error {0} in Schedule()", err);
```

```
Example 2:
// Create a NI2CFile object
NI2CFile pNI2C =
                new NI2CFile("I2C1:", NI2CFile.NI2CAccess.READ_WRITE);
// Set exception error handling
pNI2C.HandleErrorsViaReturn(false);
// Schedule a request and check for error
try
{
     pNI2C.Schedule(...);
}
catch (NI2CException e)
{
     Console.WriteLine("Error {0} in Schedule()", 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 Schedule()

Signature:
```
int Schedule(NI2C_MSG_HEADER[] msg,
             byte[] data)
```

Parameters:

`msg`	Array of message headers
`data`	Array of data bytes for all messages

Return:

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

Description:
This command copies the given data to the global transmission request queue and schedules the request for execution. Then it returns immediately. The execution of the request takes place in the background.
The result of the request must either be retrieved with `GetResult()` or discarded with `SkipResult()`. You can use `CheckResult()` to check whether the transmission is complete.
See section starting at page 7 for how to set up the data arrays.

4.4 GetResult()

Signature:
```
int GetResult(NI2C_MSG_HEADER[] msg,
              byte[] data)
```

Parameters:

`msg`	Array of message headers
`data`	Array of data bytes for all messages

Return:

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

Description:
Waits until at least one transmission request is complete.
If the structure of the given parameters matches the structure of the completed transmission request, the message headers of the request are copied from the global transmission request queue to `msg`, and the data bytes are copied from the global queue to `data`. Here the data bytes contain both, the sent bytes and the received bytes. And you can check the `chFlags` values of the returned message headers to get information about the success or failure of each individual message of the transmission request.
If more than one transmission request is pending, their results must be retrieved in the same order as the requests were issued with `Schedule()`. If the structure of the arguments does not match, `GetResult()` will fail.
Please note that this call will block if no result is available. But you can use `CheckResult()` in advance to check whether some transmission is already complete.
See section starting at page 7 for how to set up the data arrays and page 11 on how to interpret the returned `chFlags` values

Example:
```
// Create a NI2CFile object
NI2CFile pNI2C =
                new NI2CFile("I2C1:", NI2CFile.NI2CAccess.READ_WRITE)
// Set up arrays for scheduling
NI2CFile.NI2C_MESSAGE_HEADER [] msg = ...;
byte [] data = ...;
// Schedule a request
pNI2C.Schedule(msg, data);
// Prepare different result arrays of same size as in Schedule()
NI2CFile.NI2C_MESSAGE_HEADER [] resultmsg =
                           new NI2CFile.NI2C_MESSAGE_HEADER[msg.Length];
byte [] resultdata = new byte [data.Length];
// Wait for completion and get answer into new array
pNI2C.GetResult(resultmsg, resultdata);
```

4.5 SkipResult()

Signature:
`int SkipResult()`

Return:

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

Description:
Waits until at least one transmission request is complete and then discards the result.
Please note that this call will block if no result is available. But you can use `CheckResult()` in advance to check whether some transmission is already complete.

Example:
```
// Create a NI2CFile object
NI2CFile pNI2C = new NI2CFile("I2C1:", NI2CFile.NI2CAccess.READ_WRITE)
// Set up arrays for two requests
NI2CFile.NI2C_MESSAGE_HEADER [] msg1 = ...;
byte [] data1 = ...;
NI2CFile.NI2C_MESSAGE_HEADER [] msg2 = ...;
byte [] data2 = ...;
// Schedule the requests
pNI2C.Schedule(msg1, data1);
pNI2C.Schedule(msg2, data2);
// Wait for completion, discard the first and get the second result
pNI2C.SkipResult();
pNI2C.GetResult(msg2, data2);
```

4.6 CheckResult()

Signature:
`int CheckResult()`

Return:

`0`	All transmissions still in progress
`!=0`	At least one transmission is complete

Description:
Checks if a completed transmission request exists. If yes, the result can be retrieved with `GetResult()` or discarded with `SkipResult()` without blocking.

Example:
```
// Create a NI2CFile object
NI2CFile pNI2C =
                new NI2CFile("I2C1:", NI2CFile.NI2CAccess.READ_WRITE)
// Set up arrays
NI2CFile.NI2C_MESSAGE_HEADER [] msg = ...;
byte [] data = ...;
// Schedule the request
pNI2C.Schedule(msg, data);
// Wait until done
while (pNI2C.CheckResult() == 0)
{
     // Do something else...
}
// Get answer without further waiting
pNI2C.GetResult(msg, data);
```

4.7 GetClockFreq()

Signature:

```
int GetClockFreq()
```

Return:

`clkfreq`	Current speed of I²C bus

Description:
Returns the current speed of the I²C bus.
There is no way of changing the speed at runtime, but it can be set in the registry (see page 4) under

```
[HKLM\Drivers\BuiltIn\I2C1]
```

4.8 enum NI2C_FLAGS

Values:

`LASTBYTE_ACK`	Receive: Send ACK on last byte Transmit: Got ACK on last byte
`DATA_NAK`	No ACK when sending data
`DEVICE_NAK`	No ACK when talking to device
`ARBITRATION_LOST`	Lost bus arbitration
`TIMEOUT`	Timeout on I2C bus

Description:
Values given in `chFlags` entry of `NI2C_MESSAGE_HEADER` when looking at the result after `GetResult()`. This can be used to determine transmission errors. See also page 11. There may be more than one flag set at a time, then they are combined by logical OR.
If the transmission was aborted due to such an error, all remaining bytes of a message that were not transferred are inverted after `GetResult()`. This allows to detect the exact position within the message where the transmission failed.
As a special case you can set `LASTBYTE_ACK` in `chFlags` of receive messages *before* scheduling the request to indicate that also the last byte received of these messages should be acknowledged by the NetDCU. Default is not to acknowledge.

4.9 enum NI2CAccess

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 `NI2CFile` object (see page 15). Usually you would like to use `READ_WRITE` for access.

4.10 enum APIError

Values:

`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_DEV_NOT_EXIST`	Device does not exist
`ERROR_INVALID_PARAMETER`	Bad parameters
`ERROR_INVALID_NAME`	Invalid device name

Description:
The most common values that are reported as errors when calling the `NI2CFile` functions. For additional values see the Win32 API.
Especially if the NI2C device driver is not installer, you'll get `ERROR_DEV_NOT_EXIST` when trying to create the `NI2CFile` object.
`ERROR_INVALID_PARAMETERS` is issued when the message header and data arrays are not set-up correctly, for example when the `wLen` entries do not sum up to the size of the data array or when the array sizes of `GetResult()` differ from the sizes given with `Schedule()`.

4.11 struct NI2C_MSG_HEADER

Entries:

`byte chDevAddr;`	Address of the target device, including send/receive flag
`byte chFlags;`	Before transmission: Ack mode After transmission: Status
`ushort wLen;`	Number of data bytes

Constructor:

```
NI2C_MSG_HEADER(byte chDevAddr, byte chFlags,
                                ushort wLen)
```

Description:
Each message is made of a message header and data bytes. This is the data structure for the message header.
The message header tells the driver to which target device to talk (`chDevAddr` bits 7..1), in which mode (send or receive, i.e. `chDevAddr` bit 0), and how many bytes to transfer (`wLen`).
After the transmission, `chFlags` tells about the success or failure of the message transfer.
Usually several message headers are combined in an array to form a more complex transmission request. All bytes to be transferred in the different messages of the request are collected in a second array. Both arrays have to be given to `Schedule()` and `GetResult()`. See also section starting at page 7 for more information of how to set-up the arrays.
There is a constructor included with this struct to make creation of entries easier.

5 The NI2CException class

The `NI2CException` class defines an exception used in combination with the `NI2CFile` class. When an error happens within a function of `NI2CFile`, it throws this kind of exception, so you can react to it in a `try-catch` statement.
The `NI2CException` 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
{
      NI2CFile pNI2C = new NI2CFile("I2C1:", ...);
      ...  // Use pNI2C
}
catch (NI2CException e)
{
      switch (e.Reason)
      {
      case NI2CFile.APIError.ERROR_DEV_NOT_EXIST:
            ...  // Handle error
      case NI2CFile.APIError.ERROR_ACCESS_DENIED:
            ...  // Handle error
      }
}
```

When examining the reason, `NI2CFile.APIError` (see page 24) may be of some help to check for possible error sources.

5.1 NI2CException() (Construction)

Signature 1:

```
NI2CException(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:

```
NI2CException(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:
`NI2CException(string text)`

Parameters:

`text`	Error text

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

Signature 4:
`NI2CException(string text, Exception inner)`

Parameters:

`text`	Error text
`inner`	Inner exception

Description:
Same as above, with inner exception.

Signature 5:
`NI2CException(int reason)`

Parameters:

`reason`	Error number

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

Signature 6:
`NI2CException(int reason, Exception inner)`

Parameters:

`reason`	Error number
`inner`	Inner exception

Description:
Same as above, but with inner exception.

Signature 7:
`NI2CException()`

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

Signature 8:
`NI2CException(Exception inner)`

Parameters:
`inner` Inner exception

Description:
Same as above, but with inner exception.

6 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