I2C + NI2C

Emulated I²C over GPIO and Native I²C

Version 3.3
(2014-05-02)

Windows CE

About This Document

I²C is a serial protocol using two open drain signal lines (data SDA and clock SCL) to transfer data up to 100 kBit/s (normal speed) or up to 400 kBit/s (high speed) on a bus topology. F&S provides two different device drivers to enable the use of I²C:

1. the I²C over GPIO device driver, called I2C, and
2. the Native I²C device driver using the I2C controller of the SoC, called NI2C.

This document describes how to install these drivers and how to use them in own applications. The drivers are available for all boards from F&S under Windows Embedded CE, Windows Embedded Compact 7 and Windows Embedded Compact 2013.

The I2C device driver on one hand uses any pair of general purpose input/outputs (GPIOs) of the board and emulates an I²C bus in software. This allows to implement more than one I²C bus on a board, but is usually restricted to rather slow transfer speeds. The NI2C driver on the other hand uses the dedicated hardware lines of the underlying micro controller. This results in higher transfer speeds, but usually only one such native bus can be implemented. However both drivers can be installed in parallel if required.

On versions V1.x and V2.x of the two drivers, there used to be a completely different software interface (API) for I2C and NI2C. Starting with version V3.0, the I2C driver was completely rewritten and now uses the same interface as the NI2C driver.

This document describes all features of this NI2C interface. Functions, that were not available in V1.x and V2.x of the NI2C driver are marked. However please note that the V1.x interface of the I2C driver is *not* covered in this document.

The latest version of this document can always be found at http://www.fs-net.de in the documents downloads section.

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

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

History

Date	V	Platform	A,M,R	Chapter	Description	Au
2009-09-07	3.0	All	A, M	-	Ported from MS Word Din A5 to new OpenOffice Writer Din A4 format. Overall rework. Added information for I2C over GPIO driver and V3.0 API. Added more examples.	HK
2011-01-18	3.1	All	M	3	IO-Pin configuration is platform dependent	MK
2012-01-12	3.2	PC*	A, M	2,3	PicoCOM1,2,3,4 and 5 added.	MK
2014-05-02	3.3	All	A,M	-	armStoneA5, PicoCOMA5, EfusA9 and QBlissA9 added	HF

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

Table of Contents

1 Introduction

After a short description of the I²C protocol, we'll introduce the two device drivers available for the F&S Windows Embedded board families. We show how they are installed on the board and how they are used in own applications by grouping messages in transmission requests. The main part of the document is the application programming interface (API) reference that discusses all functions provided by the drivers, including examples.

1.1 The I²C protocol

I²C is a serial protocol over a two-wire bus. The two wires are called SDA for the data line and SCL for the clock line. The signals are open drain, i.e. only the low level is actively driven. If a device wants to send a high level, it simply puts the line to high impedance. The signal is then automatically pulled high by an external pull-up resistor that must be present somewhere on each wire. Standard devices can operate up to 100 kBit/s, high speed devices support transfer speeds up to 400 kBit/s.

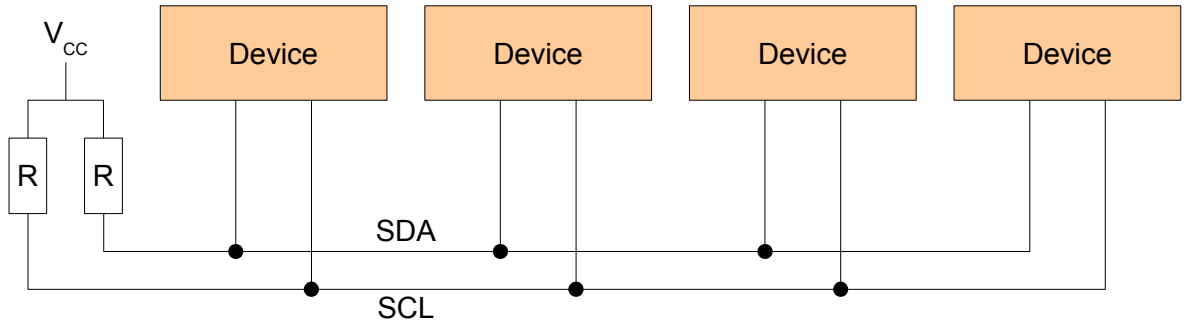

Figure 1: I²C bus topology

All devices connected to an I²C bus are identified by a 7-bit device address. Each transmission must transfer the target device address in the first byte and whether the remaining transmission is receiving or sending. This is coded as the eighth bit of the address byte: A high bit denotes receiving (Read, R), a low bit denotes sending (Write, $\overline{W}$).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
A6	A5	A4	A3	A2	A1	A0	R/$\overline{W}$

Figure 2: 7-Bit address and read/write as eighth bit

Because of this, sometimes the address is also given as an 8-bit address pair, an even address for sending and the next higher odd address for receiving. In this case, the 7-bit address is already shifted one bit to the left.

If both wires are high, the bus is in idle state. If a device wants to initiate a transmission, it must generate a so-called START condition on the bus by first pulling SDA low and then SCL.

From now on this device is the *master* during this transmission. That means it must provide the target address of the device it wants to talk to and determine the transmission direction. Only the addressed device (the *slave*) is allowed to answer, all other devices must stay silent during this transmission. The master is also responsible for toggling the SCL clock line during the whole transmission.

The end of the transmission is indicated by a STOP condition, which is first releasing the SCL line and then the SDA line. After that the I²C bus is again in idle state and a new transmission can be initiated by any device on the bus.

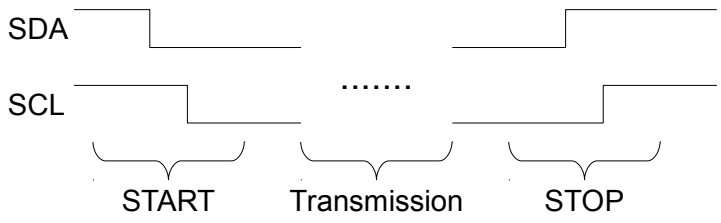

Figure 3: Transmission with START and STOP condition

The bits of a byte are transferred MSB first, i.e. the most significant bit 7 is transferred first and the least significant bit 0 is transferred last. Switching the SDA line is only allowed while SCL is held low, otherwise it would be interpreted as a START or STOP condition.

Each byte that is transferred must be acknowledged by the receiving side of the transmission, which is the target device when sending and the master when receiving. Acknowledging is done by holding the SDA line low during a ninth clock pulse that is issued after the eight data clocks for the eight bits of the byte. A missing acknowledge bit usually indicates the end of the transmission.

Here are two typical transmission sequences for sending and receiving some bytes. Bits sent by the slave are highlighted in grey. First there is a sequence to send two bytes 0x12 and 0x34 to a device with address 0x40. Please note how each byte is acknowledged by the device (the low bits marked with “A”).

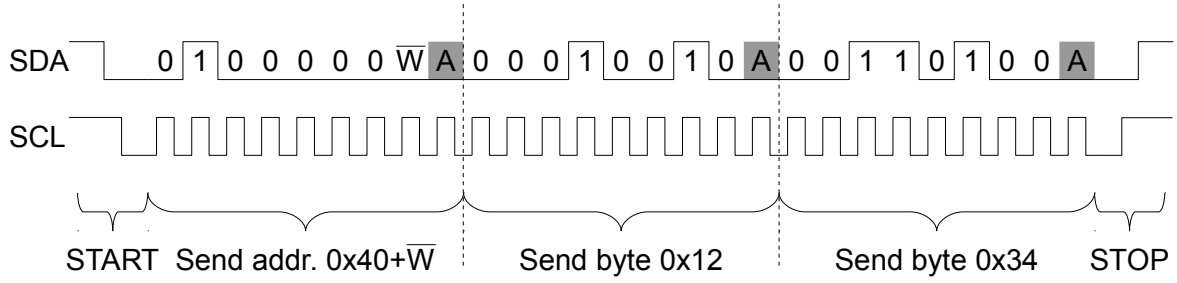

Figure 4: Sequence sending two data bytes to device 0x40

Whether the last data byte is acknowledged or not depends on the type of device. If the device expects exactly two bytes, for example a value for some internal 16-bit register, then the last byte would *not* be acknowledged. But if the device expects any number of data bytes, for example a kind of memory device that can store more than two bytes, then the last

byte is acknowledged (like in the figure) and the master simply stops the sequence at any time by issuing the STOP condition.

The second sequence shows the reception of two bytes from this device (address 0x40 + R results in byte value 0x41). After the address byte from the master that is acknowledged by the device, now the device is taking control of the data bits by sending the two data bytes 0x56 and 0x78 and the master is acknowledging the reception in the ninth clock cycle. Here in this example, the master does not acknowledge the last byte to show the slave that it should stop sending bytes (the high bit marked with “N”).

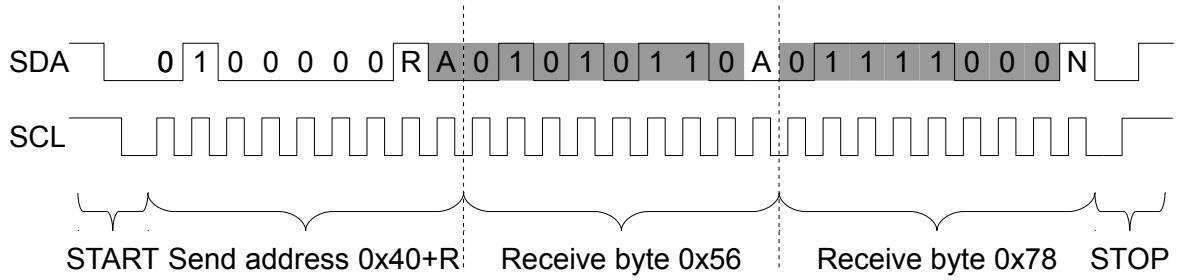

Figure 5: Sequence receiving two data bytes from device 0x40

Sometimes it is required to send *and* receive bytes in one transmission. For example if the I²C device is some kind of memory, then it is usually necessary to send a base memory address first to the device before reading data from the device starting at this memory address. This is actually done by combining a send and a receive transmission. The trick is here that the master does not release the bus between these two transfers by issuing a special REPEATED START condition instead of a normal STOP and START sequence. This is done by releasing first SDA and then SCL. This looks like a normal data bit at first but then a normal START condition is appended, i.e. SDA changes to low and then SCL changes to low. This is different to a normal data bit where SDA must keep its state while SCL toggles to high and back to low. Therefore the bus is actually never released during this REPEATED START condition and therefore no other device can interrupt and start its own transmission. It is guaranteed that the sending and receiving part is completed as a whole.

The following figure shows the interesting part of the sequence that would result if the sending and receiving transmissions from above would be combined with REPEATED START to one single transmission.

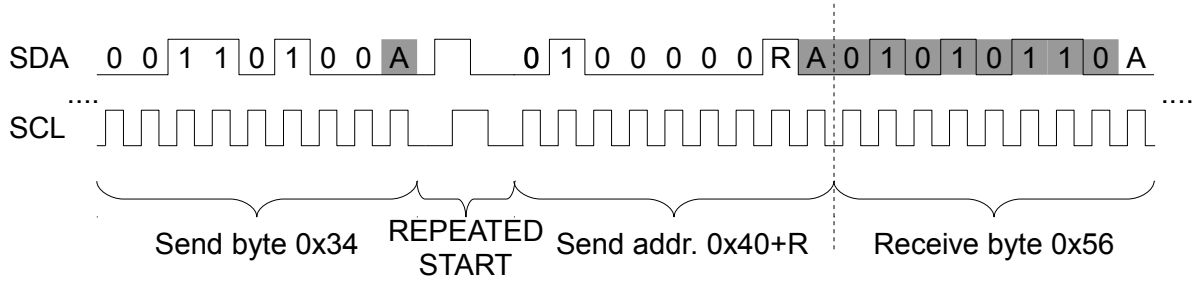

Figure 6: Combining transmissions with REPEATED START

1.2 The Device Drivers I2C and NI2C

F&S Elektronik Systeme provides several families of Single Board Computers that differ in size and performance: the armStone, efus, NetDCU, PicoMOD, PicoCOM and QBliss series of boards. F&S tries to keep the usage of the device drivers consistent across all these boards. For I²C, there exist two different device drivers under Windows Embedded:

1. the I²C over GPIO device driver, called I2C, and
2. the Native I²C device driver using the I2C controller of the SoC, called NI2C.

The two drivers serve different purposes. I2C allows to establish many different I²C buses on one board at the expense of rather high CPU usage, while NI2C uses the existing dedicated I²C hardware to get the best speed with as little CPU resources as possible.

Let's look at these two drivers one after the other.

1.2.1 Using GPIOs for I²C

The I2C device driver was the first I²C driver available for our boards. The idea was rather simple: use two general purpose input/outputs (GPIOs) of the board to emulate an I²C bus. A low signal is done by switching the GPIO pin to output with logical 0. This actively drives the low signal on the line. A high signal is done by switching the GPIO pin to input, which means high impedance. Then the pull-up resistor on the wire can pull the signal to logical 1.

This special handling of GPIOs is not implemented in hardware on the boards and must be done in software. Especially the bit timing is rather critical and can only be implemented with busy-wait loops. Neither sleeping functions nor interrupts can be used for these short delays to free the CPU. As a result, this driver is rather CPU intensive. While transmitting, it will use most of the CPU time. And even then, the final transfer speed is still limited. For example on a NetDCU8, the maximum speed possible with this driver is around 40 to 50 kBit/s. On other boards it might be slightly higher, for example 100 kBit/s, but it is illusory to talk about the maximum I²C transfer speed of 400 kBit/s in connection with this driver.

On the other hand it is possible to build more than one I²C bus using this I2C driver. Each instance can use a different pair of GPIOs and thus the number of I²C buses is only limited by the number of available GPIO pins. But be careful: if the buses want to transmit at the same time, there will be additional delays by the competing drivers. It is just software that can only be executed quasi-parallel, i.e. one after the other in time slices.

1.2.2 Using Dedicated Hardware for I²C

As already mentioned, the I2C driver does not allow very high transfer speeds over the I²C bus because of the pure software implementation. To obtain higher speeds, support from the underlying hardware is required. This was the idea behind the introduction of the second I²C device driver, the Native I2C, or NI2C for short. This driver provides access to the dedicated

I²C hardware that is available on almost all of the currently available micro controllers. Now even the maximum I²C speed of 400 kBit/s is no problem and the CPU is nearly completely relieved from the bit banging procedure. In fact the CPU has a rather low load when using this driver and can be used to perform other stuff in parallel to the I²C transmission.

Therefore this driver is the one that is best to use if only one I²C bus is necessary. Because the only downside of this driver is the fact, that all the micro controllers that we have used so far only support at most one single I²C bus. And you have to use exactly one pair of I/O pins of the board, as the hardware I²C bus is only available on exactly these pins.

The good news is, that both drivers, NI2C and I2C can be used in parallel if required. Thus you can build one fast I²C bus with NI2C and several slow I²C buses with I2C all on one board.

1.2.3 Different Driver Versions

The first versions V1.x of the I2C driver used a rather simple application programming interface (API) that had the disadvantage, as we later discovered, that it could not be ported easily to the .NET framework. Therefore when we introduced the NI2C driver, the API was completely reworked. It proved to be a very powerful interface that worked very well also on all our new boards. This was the V1.x series of the NI2C drivers. The API was slightly improved when we ported the NI2C driver to the PicoMOD3, which was the only driver with the V2.x interface.

In early summer of 2009, also new ports of the I2C driver were requested for our new boards. This was the point when we decided to bring the interface of the I2C driver in line with the NI2C driver. Again the API was slightly improved, the I2C driver completely rewritten and now both drivers, NI2C and I2C use the exact same API, which we now call the V3.x interface. Therefore all new ports of NI2C and I2C will sail under the V3.x version.

This document describes all features of this NI2C interface. Functions, that were not available in V1.x and V2.x of the NI2C driver are marked appropriately. However please note that the now obsolete V1.x interface of the I2C driver is *not* covered in this document.

Which version you have can be determined by looking at the debug output of the board. The first line starting with `NI2C:` or `I2C:` shows the appropriate driver version. Somewhere during V1.x there was also an IOCTL command code added called `IOCTL_DRIVER_GETINFO`. If this command succeeds, it returns the appropriate driver version in a data structure, and if it fails, it's definitely an early V1.x driver.

Remark

In this document, we'll use the generic term "NetDCU" for all our boards. It should also mean PicoMOD, PicoCOM or any other future board type or family, where the I²C drivers are available. We will also refer to the driver file generally as `ni2c.dll` or `i2c.dll` respectively, even if the real name may have a board specific prefix added, e.g. `pm3_ni2c.dll` on the PicoMOD3. We hope that this does not cause any inconvenience.

2 Pin Assignment

With the Native I²C driver NI2C, you are not free to choose the pins to use like with the GPIO I2C driver. Instead they are given by the I²C hardware on the board. The following table shows the dedicated I²C lines on the different boards. On the PicoMOD and PicoCOM modules, we give the pin number of the module connector itself and of the connector on the starter interface board.

Board	Connector	SDA	SCL
armStoneA5	J12 (feature con.) J8 (TTL RGB con.) J16 (I2C con.)	Pin 17 Pin 32 Pin 2	Pin 16 Pin 33 Pin 3
efusA9	MXM2 I2C_A MXM2 I2C_B MSM2 I2C_C	Pin 151 Pin 82 Pin 130	Pin 155 Pin 84 Pin 132
NetDCU5.2	J5	Pin 10	Pin 11
NetDCU8,9,9.2,10,11,11.2,14	J5	Pin 10	Pin 11
NetDCUA5	J5	Pin 10	Pin 11
PicoMOD3,4,6 Module PicoMOD3,4,6 Startinterface	J1 J5	Pin 31 Pin 10	Pin 34 Pin 11
PicoCOM1 Module PicoCOM1 Startinterface	J1 J11	Pin 32 Pin 21	Pin 33 Pin 22
PicoCOM2,3,4,5 Module PicoCOM2,3,4,5 Startinterface	J1 J10	Pin 32 Pin 9	Pin 33 Pin 10
PicoCOMA5 Module PicoCOMA5 Startinterface	J1 J1 J10	Pin 32 Pin 63 (shared) Pin 9	Pin 33 Pin 64 (shared) Pin 10
QBlissA9	MXM2 I2C MXM2 HDMI CTRL	Pin 68 Pin 150	Pin 66 Pin 152

Table 1: Pin assignment of I²C signals for the NI2C driver

You can use this driver in combination with the GPIO I2C 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 I2C and/or NI2C Driver

On many F&S boards, the **NI2C driver is already pre-installed in the WindowsCE kernel image** as `I2C1:`. As a consequence, we recommend to install the I2C driver starting with number 2, i.e. `I2C2:`, `I2C3:`, and so on. We usually provide special Windows Cabinet Files ("CAB-Files") for an automatic installation, but you can also do the installation manually.

Remark

If you also plan to use the 5-wire touch panel adapter `NetDCU-ADP-TP5` from F&S, 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.

3.1 Installation with the CAB file

The easiest way to install one of the drivers is to use the provided Windows Cabinet File `ni2c.cab` or `i2c.cab` respectively. 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:` (NI2C) or `I2C2:` (I2C). When asked for a destination directory, just click `OK`. This will install the driver in directory `\FFSDISK`. 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
```

or

```
wceload /noui i2c.cab
```

respectively.

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

3.2 Manual installation and configuration

You can also do the installation by hand. This requires setting some registry values. Installation of the NI2C and I2C drivers takes place in the registry under

`[HKLM\Drivers\BuiltIn\I2C`*`n`*`]`

where *`n`* is the number of the device (usually 1 for NI2C and 2, 3, 4, and so on for I2C).

Entry	Type	Default Value	Description
Dll	*String*	*ni2c.dll* *i2c.dll*	*Driver DLL for NI2C* *Driver DLL for I2C*
FriendlyName	*String*	*Native I2C driver* *I2C over GPIO driver*	*Description for NI2C* *Description for I2C*
Prefix	*String*	*I2C*	*For I2Cn:*
Index	*DWORD*	*1* *2*	*I2C1: for NI2C* *I2C2: for I2C*
Order	*DWORD*	*101*	*Load sequence*
ClockFreq	DWORD	200000 20000	Default transfer speed NI2C Default transfer speed I2C
PinSDA	DWORD	<platform dependent>	Pin configuration for SDA pin (Only required for I2C driver)
PinSCL	DWORD	<platform dependent>	Pin configuration for SCL pin (Only required for I2C driver)
Priority256	DWORD	103	Thread priority
Debug	DWORD	0	Debug verbosity

Table 2: I2C and NI2C registry values

Most of the registry values will get meaningful defaults if omitted, only those values highlighted in blue/yellow and italics in the first few rows of the table really have to be given. The library `ni2c.dll` (or `i2c.dll` respectively) has to be stored into the `\FFSDISK` directory in flash memory, if it is not already pre-loaded in the kernel.

3.3 Description of the Available Registry Values

3.3.1 ClockFreq

The transfer speed (in Bit/s) to be used for I²C transmissions. The maximum value defined by the I²C specifications is 400000. Setting a higher value does not make sense.

Please note that not all values can actually be implemented by the driver. In the I2C driver, the speed is limited by the CPU power and the time required to access the GPIO pins. In the NI2C driver, the speed is usually divided from some base clock frequency and the set of

possible dividers decides which speeds are actually available. Nevertheless the driver always tries to use a setting as close as possible to the value given in `ClockFreq`.

3.3.2 PinSDA, PinSCL

These two settings are only required for the I2C driver. They define the pin numbers to be used for the SDA and SCL signals. As the pin-numbering is not common on all platforms these values are board specific. See documentation “DeviceDriver_xxx.pdf” chapter “Digital I/O” for more information.

On NetDCU-family

The value is the same that you would set for this pin in the `UseAsIO` entry of the DIO driver for the board (please refer to the separate “NetDCU Device Drivers” document).

On PicoMOD- / PicoCOM-family

This value is the IO-Pin number of the pin the signal should be available on. Please refer to the separate “PicoMOD/PicoCOM Device Drivers” document for further information on IO-Pin numbering.

Attention!

The NI2C driver has dedicated pins that can not be changed and thus don't need any configuration (see chapter 2 on page 6).

3.3.3 Priority256

The actual transfer will take place with the Windows CE priority given in `Priority256`. Changing this value is only required if the I2C or NI2C 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. This may result in sporadic malfunctions if these drivers are interrupted for too long.

Note 1: Wrong priority settings may result in malfunctions

3.3.4 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 The I²C Drivers in Applications

Both I²C drivers are designed to work as the sole master on the bus. Therefore no other device is allowed to take the role of a master and start an I²C transmission. They all have to act as slaves.

When using an I²C driver in own applications, please keep in mind that you might have to co-operate with other applications using devices on the same I²C bus. For example if the 5-wire touch panel driver is also installed (like on the PicoMOD1), it uses the NI2C driver and accesses the same internal message queue as your own devices connected to this bus. So don't block the bus longer than required or else the touch panel operation will suffer.

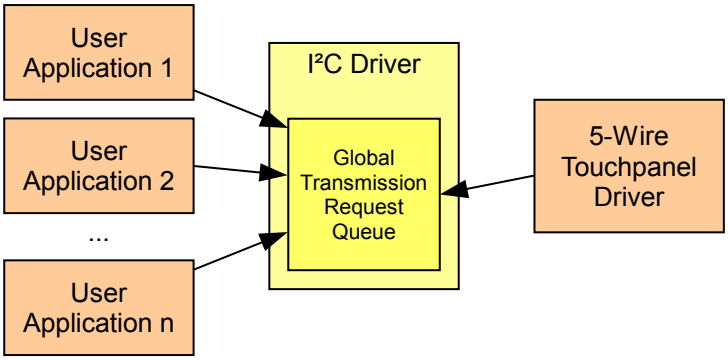

Figure 7: All applications accessing an I²C bus share a global queue

In fact in this case 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 I²C drivers keep transmission requests strictly separated, and serve 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.

4.1 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

Let's assume two devices A and B on an I²C bus. Device A has different internal 16-bit wide registers that can be read out. By sending a byte, the register number to be read can be selected. Device B accepts three bytes of data to manipulate some I/O pins.

Then the following four messages could be combined to one transmission request.

1. Send 1 byte to device A to select a register number
2. Receive 2 bytes from device A to read this register
3. Send 3 bytes to device B to set the I/Os
4. Receive another 2 bytes from device A to read the same register again

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 I²C driver handles transmission requests in a non-blocking way. So you first have to prepare the request, including all data bytes to be sent, and then call a `DeviceIoControl()` function. This function schedules 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 `DeviceIoControl()` function to retrieve the result, i.e. the data bytes that were received and the success status of each message.

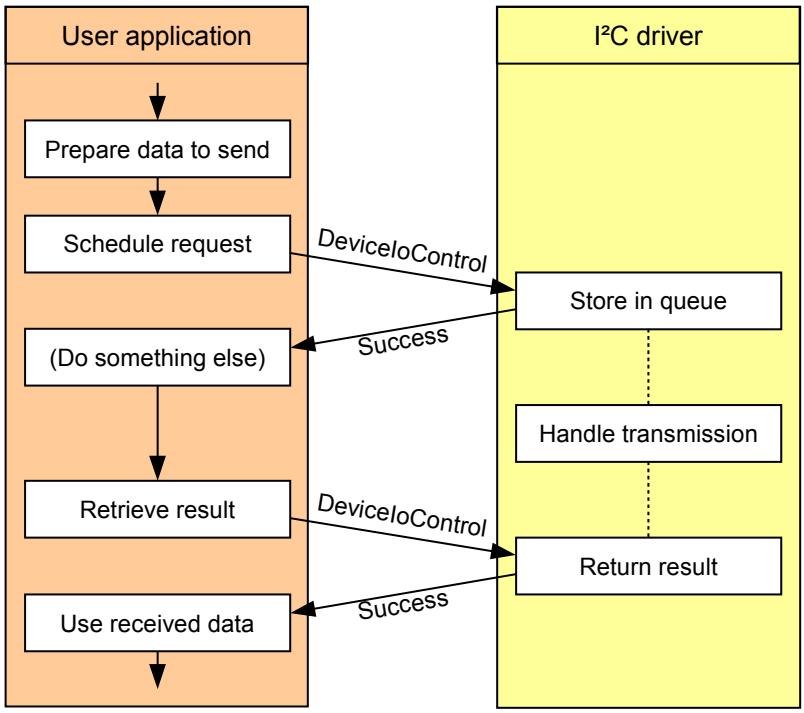

Figure 8: Scheduling requests and retrieving the results

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. You have to give dummy bytes in those places, where data will be received, as the I²C 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 (although this is not always advisable).

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

```
/* Status flags used in NI2C_MSG_HEADER */
enum NI2C_FLAGS
{
    NI2C_FLAGS_LASTBYTE_ACK = 0x01,
    NI2C_FLAGS_DATA_NAK = 0x02,
    NI2C_FLAGS_DEVICE_NAK = 0x04,
    NI2C_FLAGS_ARBITRATION_LOST = 0x08
};

/* Message header */
typedef struct NI2C_MSG_HEADER
{
    unsigned char chDevAddr;  /* Bit 7..1: Device address
                                 Bit 0: 0: Send, 1: Receive */
    unsigned char chFlags;    /* See NI2C_FLAGS */
    unsigned short wLen;      /* Message length */
} NI2C_MSG_HEADER, *PNI2C_MSG_HEADER;
```

Listing 1: Driver API data structures

As you can see, the transfer direction (send or receive) is set 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.

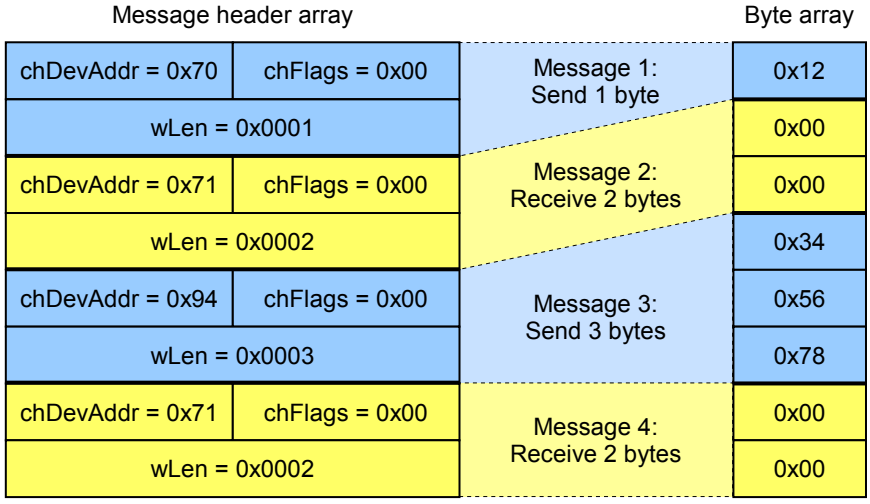

Figure 9: Data arrays required for a transmission request with four messages

This might result in the following C code.

```
/*************************************************************/
/*** File:        ni2c-example.c                           ***/
/*** Author:      Hartmut Keller, (C) F&S 2006             ***/
/***                                                       ***/
/*** Description: Example for NI2C transmission request ***/
/*************************************************************/

#include <windows.h>          /* BYTE, HANDLE, NULL, ... */
#include "ni2cio.h"           /* NI2C_MSG_HEADER, ... */

/* Message headers for transmission request */
static NI2C_MSG_HEADER msg[] =
{
    { 0x70, 0x00, 0x0001},    /* Send 1 byte to 0x70 */
    { 0x71, 0x00, 0x0002},    /* Receive 2 bytes from 0x70 */
    { 0x94, 0x00, 0x0003},    /* Send 3 bytes to 0x94 */
    { 0x71, 0x00, 0x0002},    /* Receive 2 bytes from 0x70 */
};

/* Data bytes for transmission request */
static BYTE data[] =
{
    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 */
};

/* Main program */
int main(int argc, char **argv)
{
    HANDLE hI2C;

    /* Open I2C device file */
    hI2C = CreateFile(TEXT("I2C1:"), GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING,
                      FILE_FLAG_WRITE_THROUGH, 0);

    /* Schedule transmission request */
    DeviceIoControl(hI2C, IOCTL_NI2C_SCHEDULE, (LPBYTE)msg,
                    sizeof(msg), data, sizeof(data), NULL, NULL);

    /* ... Do something else here ... */

    /* Retrieve and print result */
    DeviceIoControl(hI2C, IOCTL_NI2C_GET_RESULT, (LPBYTE)msg,
                    sizeof(msg), data, sizeof(data), NULL, NULL);
    printf(TEXT("Message 2: 0x%02x 0x%02x\r\n"), data[1], data[2]);
    printf(TEXT("Message 4: 0x%02x 0x%02x\r\n"), data[6], data[7]);

    /* Close I2C device file */
    CloseHandle(hI2C);

    return 0;
}
```

Listing 2: Example for an I²C transmission

4.2 Acknowledgement Mode and Status Flags

The `chFlags` value in the message header serves two different purposes: setting the acknowledgement mode for receiving messages and returning the status of all transmissions.

4.2.1 Setting the Acknowledgement Mode

For messages that send data, the value of `chFlags` going into the driver is unimportant. But for messages that receive data, `chFlags` must be initialised before scheduling the transmission request, because this value determines how the I²C driver handles the acknowledgements of the received bytes. Only one bit has a meaning here, other bits have no effect.

chFlags	Explanation
0	The last byte of the received message will *not* be acknowledged by the driver
NI2C_FLAGS_LASTBYTE_ACK	The last byte of the received message *will* be acknowledged by the driver.

Table 3: Possible values for entry `chFlags` when receiving data

4.2.2 Status Flags

Whether a message could be successfully transmitted or not is reported in the `chFlags` entry. It can be zero or any combination of the following bit values.

chFlags	Explanation
NI2C_FLAGS_LASTBYTE_ACK	The last byte of the message was acknowledged.
NI2C_FLAGS_DATA_NAK	There was a missing acknowledgement before the last byte, indicating that the receiving device could not take more data. Transmission was aborted at this point.
NI2C_FLAGS_DEVICE_NAK	The device address was not acknowledged, i.e. no device responded. The message could not be transmitted.
NI2C_FLAGS_ARBITRATION_LOST	The I²C bus was already busy when the message transfer was about to start. Transmission failed.

Table 4: Possible values for entry `chFlags` after transmission

After having retrieved the result of a transmission request, you should check the `chFlags` values of all individual messages to get a status report.

Here, `NI2C_FLAGS_LASTBYTE_ACK` is just a result value, telling whether the receiving device did acknowledge the last byte or not. This might be an indication for some error as the receiving device expected more data, but for some devices that can accept any number of bytes, this behaviour is completely OK. For this reason it must be decided by the application whether this is an error or not.

All other flags indicate an aborted transfer. That means at some point in the message the transmission stopped because of an error and the remaining bytes of the message could not be transferred. To mark the abortion point within the message, the I²C driver inverts the bit patterns of all bytes of the remaining message that were not transmitted. In case of a receiving message, the dummy bytes that were in the data array before the request was scheduled, are inverted. Therefore by comparing the data array that was scheduled to the data array returned in the result you can tell for each message exactly where the error occurred.

By the way this is the main reason why it is sometimes no good idea to use the same arrays in the calls to scheduling a request and getting the result. As the data would be overwritten, It would not be possible to compare the “before” and “after” state.

Example

The following message bytes should be sent to some device:

```
0x11 0x22 0x33 0x44 0x55
```

After the transmission, the `chFlags` value shows that the `NI2C_FLAGS_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.

4.3 Scanning the I²C Bus for Devices

If the hardware configuration on the I²C bus is variable and not some fix arrangement, it is usually one of the first tasks to determine which devices are actually present on the bus. This is called *scanning the bus*.

The idea is rather simple. Each transmission starts with the address of the target device. If the device is present, it will acknowledge the address. If not, `NI2C_FLAGS_DEVICE_NAK` will be set. It is not necessary to actually transfer some data with the message, just sending the address (usually together with $\overline{W}$) is enough. Then a bus scan is nothing more than sending all the addresses of the devices that are expected on the bus one after the other and looking at the `NI2C_FLAGS_DEVICE_NAK` flag on return.

Attention!

Due to hardware restrictions, **NI2C drivers of PicoCOM1 and PicoCOM2** are not able to send or receive messages with a size of 0 bytes. For this reason it is not able to execute the addressing phase only and check the slave acknowledgement. To be able to scan the devices connected to the bus anyway it is possible to read 1 dummy-byte. But in doing so it must be assured that reading one byte does not manipulate the internal state of any slave connected.

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 code determines the actual address of the device on the I²C bus.

```
/*****************************************************/
/*** File:        ni2c-scan.c                       ***/
/*** Author:      Hartmut Keller, (C) F&S 2006      ***/
/***                                                ***/
/*** Description: NI2C bus scan for ADS7828 device  ***/
/*****************************************************/

#include <windows.h>        /* BYTE, HANDLE, NULL, ... */
#include "ni2cio.h"         /* NI2C_MSG_HEADER, ... */

/* Message headers for scan transmission request */
static NI2C_MSG_HEADER scanmsg[] =
{
    { 0x90, 0x00, 0x0000},  /* Send 0 bytes to 0x90 */
    { 0x92, 0x00, 0x0000},  /* Send 0 bytes to 0x92 */
    { 0x94, 0x00, 0x0000},  /* Send 0 bytes to 0x94 */
    { 0x96, 0x00, 0x0000},  /* Send 0 bytes to 0x96 */
};

/* Main program */
int main(int argc, char **argv)
{
    HANDLE hI2C;
    int i;
    BYTE myaddr = 0;

    /* Open I2C device file */
    hI2C = CreateFile(TEXT("I2C1:"), GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING,
                      FILE_FLAG_WRITE_THROUGH, 0);

    /* Send scanning transmission request */
    DeviceIoControl(hI2C, IOCTL_NI2C_SCHEDULE, (LPBYTE)scanmsg,
                    sizeof(scanmsg), NULL, 0, NULL);
    DeviceIoControl(hI2C, IOCTL_NI2C_GET_RESULT, (LPBYTE)scanmsg,
                    sizeof(scanmsg), NULL, 0, NULL);
```

```
    /* Search for address and print result */
    for (i=0; i<sizeof(scanmsg)/sizeof(NI2C_MSG_HEADER); i++)
    {
        if (scanmsg[i].chFlags == 0)
        {
            myaddr = scanmsg[i].chDevAddr;
            break;
        }
    }

    if (myaddr)
        printf(TEXT("ADS7828 on address 0x%02x\r\n"), myaddr);
    else
        printf(TEXT("No ADS7828 found\r\n"));

    /* Close I2C device file */
    CloseHandle(hI2C);

    return 0;
}
```

Listing 3: Scanning the I²C bus for devices

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

5 NI2C Reference

5.1 CreateFile()

Signature

```
HANDLE CreateFile(LPCTSTR lpFileName, DWORD dwAccess, DWORD dwShare,
                  LPSECURITY_ATTRIBUTES lpSecurity, DWORD dwCreate,
                  DWORD dwFlags, HANDLE hTemplate);
```

Parameters

`lpFileName`.....Device file name, usually "`I2C1:`" or "`I2C2:`"
`dwAccess`.....Ignored, set to `GENERIC_READ | GENERIC_WRITE`
`dwShare`.....Ignored, set to `0`
`lpSecurity`.....Ignored, set to `NULL`
`dwCreate`.....Set to `OPEN_EXISTING`
`dwFlags`.....Set to `FILE_FLAG_WRITE_THROUGH`
`hTemplate`.....Ignored, set to `0`

Return

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

Description

Opens the `I2Cx:` device file for access. This is required for all other functions using this I²C bus. If the file handle is not required anymore, you have to call function `CloseHandle()`.

Example

```
int main(int argc, char **argv)
{
    HANDLE hI2C;

    /* Open I2C device file */
    hI2C = CreateFile(TEXT("I2C1:"), GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING,
                      FILE_FLAG_WRITE_THROUGH, 0);

    /* ... */
}
```

Listing 4: `CreateFile()`

5.2 CloseHandle()

Signature

`BOOL CloseHandle(HANDLE hDevice);`

Parameters

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

Return

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

`!=0`.....Success

Description

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

Example

```
int main(int argc, char **argv)
{
    HANDLE hI2C;

    /* Open I2C device file */
    hI2C = CreateFile(TEXT("I2C1:"), GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING,
                      FILE_FLAG_WRITE_THROUGH, 0);

    /* ... */

    /* Close the file again */
    CloseHandle(hI2C);
}
```

Listing 5: `CloseHandle()`

5.3 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 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, given by a control code in `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 NI2C driver V1.x.

Control Code	Function
`IOCTL_NI2C_SCHEDULE`	Schedules a transmission request for execution
`IOCTL_NI2C_GET_RESULT`	Retrieves the result of a transmission request
`IOCTL_NI2C_SKIP_RESULT`	Discards the result of a transmission request
`IOCTL_NI2C_CHECK_RESULT`	Checks if a transmission request is completed and the result available
`IOCTL_NI2C_GET_CLOCKFREQ`	Retrieves the current I²C bus speed
`IOCTL_DRIVER_GETINFO`	Retrieves the driver version (not available in all versions)

Table 5: IOCTL command codes for V1.x

Starting with version V2.0, the set of control codes was extended. The following table lists all control codes added in V2.0.

Control Code	Function
IOCTL_DRIVER_GETINFO	Retrieves the driver version
IOCTL_NI2C_TRANSFER	Schedule transmission request and get result in one go

Table 6: IOCTL command codes added in V2.x

And again starting with V3.0, another control code was added.

Control Code	Function
IOCTL_NI2C_SET_CLKFREQ	Set a new transfer speed at runtime

Table 7: IOCTL command codes added in V3.x

5.4 IOCTL_NI2C_SCHEDULE

Parameters

`hDevice`	Handle to already open device file
`dwIoControlCode`	`IOCTL_NI2C_SCHEDULE`
`lpInBuffer`	Pointer to message header array
`dwInBufferSize`	Size of the array in bytes; this value determines the number of messages contained in the request and must therefore be a multiple of `sizeof(NI2C_MSG_HEADER)`
`lpOutBuffer`	Pointer to the data byte array; if `dwOutBufferSize` is `0`, you can use `NULL` here
`dwOutBufferSize`	Number of bytes in the data byte array; this value must match the sum of the message lengths
`lpReturned`	Unused, set to `NULL`
`lpOverlapped`	Unused, set to `NULL`

Return

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

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 `IOCTL_NI2C_GET_RESULT` or discarded with `IOCTL_NI2C_SKIP_RESULT`. You can use `IOCTL_NI2C_CHECK_RESULT` to check whether the transmission is complete.

The splitting between scheduling of requests and fetching the result allows for more parallelism by having more than one request in the queue (see example below). If you don't need this flexibility, there is the new command `IOCTL_NI2C_TRANSFER` that combines scheduling of a request and fetching the result ins one go.

See chapter 4.1 on page 11 for how to set up the data arrays.

Remark

This function needs two arrays going in: the message headers 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.

Example

Schedule three similar requests and then fetch the results. As the structure is the same for all messages, the header is reused in all requests, but the data differs.

```
/* Same message header for all transmission requests */
static NI2C_MSG_HEADER msg[] =
{
    { 0x70, 0x00, 0x0001},     /* Send 1 byte to 0x70 */
    { 0x94, 0x00, 0x0003},     /* Send 3 bytes to 0x94 */
};

/* Different data bytes for transmission requests */
static BYTE data1[] =
{
    0x11,                      /* Message 1: send 1 byte */
    0x22, 0x33, 0x44,          /* Message 2: send 3 bytes */
};

static BYTE data2[] =
{
    0x55,                      /* Message 1: send 1 byte */
    0x66, 0x77, 0x88,          /* Message 2: send 3 bytes */
};

static BYTE data3[] =
{
    0x99,                      /* Message 1: send 1 byte */
    0xAA, 0xBB, 0xCC,          /* Message 2: send 3 bytes */
};

/* Main program */
int main(int argc, char **argv)
{
    /* ... */

    /* Schedule three transmission requests */
    DeviceIoControl(hI2C, IOCTL_NI2C_SCHEDULE, (LPBYTE)msg,
                    sizeof(msg), data1, sizeof(data1), NULL, NULL);
    DeviceIoControl(hI2C, IOCTL_NI2C_SCHEDULE, (LPBYTE)msg,
                     sizeof(msg), data2, sizeof(data2), NULL, NULL);
    DeviceIoControl(hI2C, IOCTL_NI2C_SCHEDULE, (LPBYTE)msg,
                     sizeof(msg), data3, sizeof(data3), NULL, NULL);

    /* Retrieve the results */
    DeviceIoControl(hI2C, IOCTL_NI2C_GET_RESULT, (LPBYTE)msg,
                     sizeof(msg), data1, sizeof(data1), NULL, NULL);
    DeviceIoControl(hI2C, IOCTL_NI2C_GET_RESULT, (LPBYTE)msg,
                     sizeof(msg), data2, sizeof(data2), NULL, NULL);
    DeviceIoControl(hI2C, IOCTL_NI2C_GET_RESULT, (LPBYTE)msg,
                     sizeof(msg), data3, sizeof(data3), NULL, NULL);

    /* ... */
}
```

Listing 6: Scheduling transmission requests

5.5 IOCTL_NI2C_GET_RESULT

Parameters

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

`dwIoControlCode`.....`IOCTL_NI2C_GET_RESULT`

`lpInBuffer`.....Pointer to message header array

`dwInBufferSize`.....Size of the array in bytes; this value must be a multiple of `sizeof(NI2C_MSG_HEADER)`

`lpOutBuffer`.....Pointer to the data byte array; if `dwOutBufferSize` is `0`, you can use `NULL` here

`dwOutBufferSize`.....Number of bytes in the data byte array; this value must match the sum of the message lengths

`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

Waits until at least one transmission request is complete. Then returns the result by copying the message headers from the global transmission request queue of the driver to the buffer given in `lpInBuffer` and the data bytes from the global queue of the driver to the buffer given in `lpOutBuffer`.

Please note that this call will block if no result is yet available. If you don't want this, you can use `IOCTL_NI2C_CHECK_RESULT` in advance to check whether some transmission is already complete.

After return, the data bytes in `lpOutBuffer` also contain the received bytes. And you can check the `chFlags` value of the returned message headers in `lpInBuffer` to get information about the success or failure of each individual message of the transmission request. In case of a transmission error, all remaining bytes of a message that could not be transmitted have an inverted bit pattern (see chapter 4.2.2 on page 15).

For the command to succeed, the message headers given in `lpInBuffer` and the size of the buffer in `lpOutBuffer` must match exactly the structure of the completed transmission request in the driver. Otherwise the command will fail. See chapter 4.1 on page 11 for how to set up the data arrays. If more than one transmission request is pending, their results must be retrieved in the same order as the requests were issued with `IOCTL_NI2C_SCHEDULE`.

Remark

This function needs two arrays going out: the message headers to report the `chFlags` and the data bytes for the received bytes. Therefore this call uses both data pointers of the `DeviceIoControl()` as `OUT` pointers, `lpInBuffer` and `lpOutBuffer`. Both arrays must point to alterable memory!

Attention!

Both arrays are completely rewritten and the original content is lost! Therefore it is recommended to use different arrays for scheduling the request and retrieving the result, especially if the data should be reused in subsequent transmissions or the result should be compared to the original data.

Note 2: Message headers and data bytes are overwritten

Example

Schedule a transmission request, get the result and check for errors. There are different arrays used for scheduling the request and fetching the result to be able to compare the data before and after. But note that the sizes must match and that the message headers have to be set up in exactly the same way in the result as in the request.

```
/* Message header for transmission request and result */
static NI2C_MSG_HEADER msg[3] =
{
    { 0x70, 0x00, 0x0004},       /* Send 4 bytes to 0x70 */
    { 0x94, 0x00, 0x0004},       /* Send 4 bytes to 0x94 */
    { 0x70, 0x00, 0x0004},       /* Send another 4 bytes to 0x70 */
};
static NI2C_MSG_HEADER rmsg[3];  /* Result */

/* Data bytes for transmission request and result */
static BYTE data[12] =
{
    0x11, 0x22, 0x33, 0x44,     /* Message 1: send 4 byte */
    0x55, 0x66, 0x77, 0x88      /* Message 2: send 4 bytes */
    0x99, 0xAA, 0xBB, 0xCC      /* Message 3: send 4 bytes */
};
static BYTE rdata[12];           /* Result */

/* Main program */
int main(int argc, char **argv)
{
    /* ... */

    /* Schedule the transmission request */
    DeviceIoControl(hI2C, IOCTL_NI2C_SCHEDULE, (LPBYTE)msg,
                    sizeof(msg), data, sizeof(data), NULL, NULL);

    /* Copy the message headers to the result array */
    memcpy(rmsg, msg, sizeof(msg));
```

```
    /* Retrieve the result in the result arrays */
    DeviceIoControl(hI2C, IOCTL_NI2C_GET_RESULT, (LPBYTE)rmsg,
                    sizeof(rmsg), rdata, sizeof(rdata), NULL, NULL);

    /* Loop over all messages */
    for (m=0, m<3; m++)
    {
        /* Check for error in this message */
        if (rmsg[m].chFlags != NI2C_FLAGS_LASTBYTE_ACK)
        {
            /* Check how many bytes were successfully transmitted */
            for (i=0; i<4; i++)
                if (data[m*4+i] != rdata[m*4+i])
                    break;

            /* Print error message */
            printf("Error in msg[%d]: %d bytes could not be sent\n",
                   m, 4-i);
        }
    }

    /* ... */
}
```

Listing 7: Retrieving the result and checking for errors

5.6 IOCTL_NI2C_TRANSFER

Parameters

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

`dwIoControlCode`.....`IOCTL_NI2C_TRANSFER`

`lpInBuffer`.....Pointer to message header array

`dwInBufferSize`.....Size of the array in bytes; this value determines the number of messages contained in the request and must therefore be a multiple of `sizeof(NI2C_MSG_HEADER)`

`lpOutBuffer`.....Pointer to the data byte array; if `dwOutBufferSize` is `0`, you can use `NULL` here

`dwOutBufferSize`.....Number of bytes in the data byte array; this value must match the sum of the message lengths

`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

Quite often `IOCTL_NI2C_SCHEDULE` and `IOCTL_NI2C_GET_RESULT` are grouped together as a command pair, using the same data arrays. Then it is easier to use this new command that avoids having to call `DeviceIoControl()` twice.

However this command `IOCTL_NI2C_TANSFER` has two disadvantages.

1. The data array used for scheduling the request is also used for retrieving the result. This overwrites the data which is not always wanted.
2. The command will only succeed if no other results are currently pending in the queue. Because otherwise the next result would be the one from the pending request, not the one of the request it scheduled itself.

See chapter 4.1 on page 11 for how to set up the data arrays and chapter 4.2.2 on page 15 on how to interpret the returned `chFlags` values.

Remark

This function uses both data pointers of the `DeviceIoControl()` as `IN` *and* `OUT` pointers. *This command was added in V2.0 of the driver; it is not available in V1.x.*

Example

This is the example from page 14 but now using `IOCTL_NI2C_TRANSFER` instead of the command pair `IOCTL_NI2C_SCHEDULE` and `IOCTL_NI2C_GET_RESULT`, which results in slightly shorter code.

```
/* Message headers for transmission request */
static NI2C_MSG_HEADER msg[] =
{
    { 0x70, 0x00, 0x0001},    /* Send 1 byte to 0x70 */
    { 0x71, 0x00, 0x0002},    /* Receive 2 bytes from 0x70 */
    { 0x94, 0x00, 0x0003},    /* Send 3 bytes to 0x94 */
    { 0x71, 0x00, 0x0002},    /* Receive 2 bytes from 0x70 */
};

/* Data bytes for transmission request */
static BYTE data[] =
{
    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 */
};

/* Main program */
int main(int argc, char **argv)
{
    HANDLE hI2C;

    /* Open I2C device file */
    hI2C = CreateFile(TEXT("I2C1:"), GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING,
                      FILE_FLAG_WRITE_THROUGH, 0);

    /* Schedule transmission request and fetch result in one go */
    DeviceIoControl(hI2C, IOCTL_NI2C_TRANSFER, (LPBYTE)msg,
                    sizeof(msg), data, sizeof(data), NULL, NULL);

    /* Print the result */
    printf(TEXT("Message 2: 0x%02x 0x%02x\r\n"), data[1], data[2]);
    printf(TEXT("Message 4: 0x%02x 0x%02x\r\n"), data[6], data[7]);

    /* Close I2C device file */
    CloseHandle(hI2C);

    return 0;
}
```

Listing 8: Using `IOCTL_NI2C_TRANSFER`

5.7 IOCTL_NI2C_SKIP_RESULT

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NI2C_SKIP_RESULT`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

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

Description

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. If you don't want this, you can use `IOCTL_NI2C_CHECK_RESULT` in advance to check whether some transmission is already complete.

Example

Schedule a transmission request but discard the result.

```
/* Message headers for transmission request */
static NI2C_MSG_HEADER msg[] = { ... };

/* Data bytes for transmission request */
static BYTE data[] = { ... };

/* Schedule transmission request */
DeviceIoControl(hI2C, IOCTL_NI2C_SCHEDULE, (LPBYTE)msg,
                sizeof(msg), data, sizeof(data), NULL, NULL);

/* Skip the result */
DeviceIoControl(hI2C, IOCTL_NI2C_SKIP_RESULT, NULL, 0, NULL, 0,
                NULL, NULL);
```

Listing 9: Discarding the result of a transmission request

5.8 IOCTL_NI2C_CHECK_RESULT

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NI2C_CHECK_RESULT`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....No result available
`1`.....At least one result is available

Description

Checks if a completed transmission request exists. If yes, the result can be retrieved with `IOCTL_NI2C_GET_RESULT` or discarded with `IOCTL_NI2C_SKIP_RESULT` without blocking.

Example

Schedule some requests, then do something else until the results are available one by one.

```
/* Message headers and data bytes for transmission request */
static NI2C_MSG_HEADER msg[] = { ... };
static BYTE data[] = { ... };

/* Schedule transmission request */
DeviceIoControl(hI2C, IOCTL_NI2C_SCHEDULE, (LPBYTE)msg,
                sizeof(msg), data, sizeof(data), NULL, NULL);

/* Wait until the result is available */
while (!DeviceIoControl(hI2C, IOCTL_NI2C_CHECK_RESULT, NULL, 0,
                        NULL, 0, NULL, NULL))
{
    /* ... Do something else ... */
}

/* Get the result */
DeviceIoControl(hI2C, IOCTL_NI2C_GET_RESULT, (LPBYTE)msg,
                sizeof(msg), data, sizeof(data), NULL, NULL);
```

Listing 10: Avoid blocking of `IOCTL_NI2C_GET_RESULT`

5.9 IOCTL_NI2C_GET CLKFREQ

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NI2C_GET_CLKFREQ`
`lpInBuffer`.....Unused, set to `NULL`
`dwInBufferSize`.....Unused, set to `0`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`clkfreq`.....Transfer speed of the I²C bus (in Bit/s)

Description

Returns the current speed of the I²C bus.

In V1.x and V2.x of the I²C device drivers, the speed could only be set in the registry with entry `ClockFreq`. Starting with V3.0, a new command `IOCTL_NI2C_SET_CLKFREQ` was added that allows setting the transfer speed also at runtime.

Example

Print the current transfer speed.

```
int main(int argc, char **argv)
{
    HANDLE hI2C;
    DWORD dwClockFreq;

    /* Open I2C device file */
    hI2C = CreateFile(TEXT("I2C1:"), GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING,
                      FILE_FLAG_WRITE_THROUGH, 0);

    /* Get and print the transfer speed */
    printf("Current transfer speed: %d Bit/s\n",
           DeviceIoControl(hI2C, IOCTL_NI2C_GET_CLKFREQ, NULL, 0,
                           NULL, 0, NULL, NULL));

    /* Close I2C device file */
    CloseHandle(hI2C);
}
```

Listing 11: Print the current transfer speed

5.10 IOCTL_NI2C_SET CLKFREQ

Parameters

`hDevice`.....Handle to already open device file
`dwIoControlCode`.....`IOCTL_NI2C_SET_CLKFREQ`
`lpInBuffer`.....Pointer to `DWORD` value with new transfer speed
`dwInBufferSize`.....`sizeof(DWORD)`
`lpOutBuffer`.....Unused, set to `NULL`
`dwOutBufferSize`.....Unused, set to `0`
`lpReturned`.....Unused, set to `NULL`
`lpOverlapped`.....Unused, set to `NULL`

Return

`0`.....Success

Description

Sets a new transfer speed for the I²C bus. The current speed can be determined with `IOCTL_NI2C_GET_CLKFREQ`.

This command was added in V3.0 of the driver; it is not available in V1.x or V2.x.

Example

Wrapper function for setting a new transfer speed.

```
void SetClockFreq(HANDLE hI2C, DWORD dwNewClockFreq)
{
    DeviceIoControl(hI2C, IOCTL_NI2C_SET_CLKFREQ, &dwNewClockFreq,
                    sizeof(DWORD), NULL, 0, NULL, NULL);
}
```

Listing 12: Set new I²C transfer speed

5.11 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 I2C or NI2C driver.

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

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

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

This command was already added in some V1.x versions of the NI2C driver, but not all. In V2.0, this command was officially added and is guaranteed to exist from then on. If the call fails, then the I²C driver version is definitely an early V1.x. Otherwise the correct version information is returned in the `DRIVER_INFO` structure.

Example

Get the driver version and print it to stdout.

```
#include <fs_driverinfo.h>

...

DRIVER_INFO cInfo;

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

printf("I2C driver V%d.%d", cInfo.wVerMajor, cInfo.wVerMinor);
```

Listing 13: Get the driver version with `IOCTL_DRIVER_GETINFO`

6 Header File ni2cio.h

The following listing shows the contents of the header file `ni2cio.h`. This file must be included in all applications that want to use the I2C or NI2C driver. It contains all IOCTL values, constants and data structures required to call the driver.

```
/***************************************************************************/
/***                                                                     ***/
/***                 _  _       _  _____  ____  _   _                   ***/
/***                | \  |_|     |_| |  __ \ / ___/ |  | |              ***/
/***                |  \ | | ___ _| |_| |  \ | |   | |  | |              ***/
/***                |   \| |/ _ \_   _| |  | | |   | |  | |              ***/
/***                | |\   | |/_/ | | | |  | | |   | |  | |              ***/
/***                | | \  | |__  | | | |__/ | |___| |__| |              ***/
/***                |_|  \_|\___\ \_\ |_____/ \____\_____/               ***/
/***                                                                     ***/
/***************************************************************************/
/***                                                                     ***/
/***                                                                     ***/
/***           N a t i v e   I 2 C   D e v i c e   D r i v e r           ***/
/***                                                                     ***/
/***                                                                     ***/
/***************************************************************************/
/*** File:     ni2cio.h                                                ***/
/*** Authors:  Hartmut Keller                                          ***/
/*** Created:  02.12.2005                                              ***/
/*** Modified: 04.09.2009 17:10:25 (HK)                                ***/
/***                                                                     ***/
/*** Description:                                                      ***/
/*** Include file for I2C IOCTLs. You have to include this file to use the ***/
/*** NI2C driver in your own applications.                             ***/
/***                                                                     ***/
/*** Modification History:                                             ***/
/*** 16.03.07 HK: NI2C_FLAGS_TIMEOUT added.                            ***/
/*** 20.05.09 HK: IOCTL_NI2C_TRANSFER added (V2.0 interface)           ***/
/*** 21.05.09 HK: IOCTL_NI2C_SET_CLKFREQ added (V3.0 interface)        ***/
/***************************************************************************/

/*---------------------------------------------------------------------------
  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) 2009 F&S Elektronik Systeme GmbH
---------------------------------------------------------------------------*/

#ifndef __NI2CIO_H__
#define __NI2CIO_H__

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

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

/* New IOControlCode values */
#define FILE_DEVICE_NI2C     0x00008037

/* Schedule a transmission request */
#define IOCTL_NI2C_SCHEDULE \
    CTL_CODE(FILE_DEVICE_NI2C, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Wait until a transmission result is available and return it */
#define IOCTL_NI2C_GET_RESULT \
    CTL_CODE(FILE_DEVICE_NI2C, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

```
/* Wait until a transmission result is available and discard it */
#define IOCTL_NI2C_SKIP_RESULT \
    CTL_CODE(FILE_DEVICE_NI2C, 0x802, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Check if the result of a transmission request is available */
#define IOCTL_NI2C_CHECK_RESULT \
    CTL_CODE(FILE_DEVICE_NI2C, 0x803, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Get the current speed of the I2C bus */
#define IOCTL_NI2C_GET_CLKFREQ \
    CTL_CODE(FILE_DEVICE_NI2C, 0x804, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Set the current speed of the I2C bus */
#define IOCTL_NI2C_SET_CLKFREQ \
    CTL_CODE(FILE_DEVICE_NI2C, 0x805, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Schedule a transmission request and wait for result */
#define IOCTL_NI2C_TRANSFER \
    CTL_CODE(FILE_DEVICE_NI2C, 0x806, METHOD_BUFFERED, FILE_ANY_ACCESS)

/* Status flags used in NI2C_MSG_HEADER */
enum NI2C_FLAGS
{
    NI2C_FLAGS_LASTBYTE_ACK = 0x01,       /* Receive: Send ACK on last byte;
                                             Transmit: Got ACK on last byte */
    NI2C_FLAGS_DATA_NAK = 0x02,           /* No ACK when sending data */
    NI2C_FLAGS_DEVICE_NAK = 0x04,         /* No ACK when talking to device */
    NI2C_FLAGS_ARBITRATION_LOST = 0x08,   /* Lost arbitration */
    NI2C_FLAGS_TIMEOUT = 0x80,            /* Timeout on I2C bus */
};

/* -------------------- Exported Types ------------------------------------ */

/* A transmission request defines a group of messages to be sent and/or
   received on the I2C bus. Each message can individually send or receive
   to/from any device address with any length. The driver will use RESTART
   between the messages and therefore handles all messages of the transmission
   request in one go, without letting other tasks interrupt the transfer. This
   allows time critical transfers on one hand, but on the other hand can also
   block the bus for quite some time. So try to be fair and split transfers in
   different requests whenever possible.

   A transmission request consists of two parts:

     1. An array of message headers, defining the parameters of the messages.
        Each message header describes the 7-bit address of the device to
        communicate with, the transfer direction (send/receive as the eighth
        bit of the address), and the message length. On receiving messages you
        can determine by setting a flag whether the last received byte should
        be acknowledged or not.

     2. A byte array containing the concatenated bytes of all messages. For
        receiving messages you have to provide as many dummy bytes with
        arbitrary content.

   Example:
   --------
   msg1: Send three bytes 0x01, 0x02, 0x03 to device 0x40
   msg2: Receive two bytes from device 0x40, don't send ACK on last byte
   msg3: Send two bytes 0x04, 0x05 to device 0x78
   msg4: Receive three bytes from device 0x78, send ACK on last byte

   Message Array[]:  chDevAddr chFlags wLen
   ---------------------------------------------------
        0            0x40      0x00    0x0003 (msg1, send)
        1            0x41      0x00    0x0002 (msg2, receive, no ACK)
        2            0x78      0x00    0x0002 (msg3, send)
        3            0x79      0x01    0x0003 (msg4, receive, ACK)
```

```
   Byte Array[]:     Content
   ------------------------------------------------------------
        0            0x01 (msg1, 1st byte, send)
        1            0x02 (msg1, 2nd byte, send)
        2            0x03 (msg1, 3rd byte, send)
        3            0x00 (msg2, 1st dummy byte, receive)
        4            0x00 (msg2, 2nd dummy byte, receive)
        5            0x04 (msg3, 1st byte, send)
        6            0x05 (msg3, 2nd byte, send)
        7            0x00 (msg4, 1st dummy byte, receive)
        8            0x00 (msg4, 2nd dummy byte, receive)
        9            0x00 (msg4, 3rd dummy byte, receive)

   A transmission request can be scheduled by IOCTL_NI2C_SCHEDULE. This means
   the request is stored in the driver and transferred asynchronously. The
   call returns immediately. By using IOCTL_NI2C_GET_RESULT and the same
   parameters, the result can be obtained later when the transfer is finished.
   After return, the special flags entry in the message headers is valid and
   reports the transfer status of each message individually. And also the byte
   array now contains the received bytes.

   You can schedule several requests in a row with IOCTL_NI2C_GET_RESULT before
   obtaining the results with IOCTL_NI2C_SCHEDULE. But please note that you
   will get the results in the same sequence as you had scheduled the requests
   before and it is important to provide the same parameters with the result
   call as with the corresponding schedule call, or IOCTL_NI2C_GET_RESULT will
   fail.

   In many cases, the asynchronous scheduling of requests and getting the
   results is not required. Therefore starting from V3.0, the driver also
   supports a combined command IOCTL_NI2C_TRANSFER, that schedules a request
   and waits for the result in one go. However this is simply the same as
   issuing IOCTL_NI2C_SCHEDULE and IOCTL_NI2C_GET_RESULT in direct sequence
   and using the same data arrays. */
typedef struct NI2C_MSG_HEADER
{
    unsigned char chDevAddr;              /* Bit 7..1: Device address
                                             Bit 0: 0: Send, 1: Receive */
    unsigned char chFlags;                /* See NI2C_FLAGS above */
    unsigned short wLen;                  /* Message length */
} NI2C_MSG_HEADER, *PNI2C_MSG_HEADER;

#endif /*!__NI2CIO_H__*/
```

Listing 14: Header File `ni2cio.h`

7 Appendix

Listings

List of Figures

List of Tables

Important Notice

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

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

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

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