AN14182

Implementing Wired Communication with the NTM88

Rev. 1 — 21 March 2024 Application note

Document information

Information	Content
Keywords	NTM88, wired communication, spi, I^2C
Abstract	This application note explains how to implement wired communication with the NTM88.

1 Introduction

The NTM88 device can be used in a variety of applications. For example, it may be connected on the PCB to another MCU, to external memory, to external sensors or to tag devices. In such implementations, the NTM88 must communicate with other devices using wired communication.

This application note explains how to use the NTM88 as SPI Client, SPI Server (Host) and I^2C Controller. An associated zip file is available in support of this application note. The zip file contains an excel spreadsheet allowing users to encode and decode hardware SPI commands. The zip file also contains a demo project implementing SPI and I^2C bit banging.

NXP does not anticipate the need for the NTM88 to act as an I^2C target for the following reasons:

- In most applications, the NTM88 needs to be a Host for other downstream devices such as memory, or other sensors.
- The NTM88 already supports SPI Client mode if the NTM88 needs to be a peripheral to an upstream Host.

In this document, there a few new acronyms. A brief explanation of each acronym follows:

- SPI “SOCI” means Server Out Client In, and is to be interpreted as being the same as the MOSI signal name.
- SPI “SICO” means Server In Client Out, and is to be interpreted as being the same as the MISO signal name.
- SPI “CS” means Chip Select, and is to be interpreted as being the same as the SS signal name.

The information presented in this application note is applicable to all part numbers of the NTM88 family.

2 NTM88 as SPI Client

2.1 SPI configuration

The NTM88 device includes a hardware SPI block that can be used in Client mode only. The block is described in details in the NTM88 User Manual[1], and the main points are highlighted in this document.

The SPI block supports the following configuration:

- Mode: Client only
- Maximum baud rate: 10 MHz
- Number of bits per frame: 16
- Clock polarity: active high (CPOL = 0)
- Clock phase: first edge (CPHA = 0)
- Data direction: MSB first
- CS polarity: active low

Care must also be taken by the SPI Host to comply with the SPI timings supported by the NTM88. A few examples of timings are:

- t_{LEAD}: The delay between CS asserted and first clock edge.
- t_{LAG}: The delay between last clock edge and CS idle.
- t_{SSN}: The minimum duration for CS not asserted.
- t_{SCLKR} and t_{SCLKF}: The maximum rise and fall times of SCLK

All timings are described in the NTM88 manual[1] and their durations are specified in the NTM88 data sheets[2],[3],[4].

2.2 Enabling and disabling the SPI block

The NTM88 SPI block is enabled by one of the following two methods:

- By the NTM88 application setting the SPIEN bit in the SIMOPT1 register.
- At Power On Reset, by the Host holding the PTA0 pin at low state for a duration greater than t_{SPI_EN} specified in the data sheet[2],[3],[4].

The SPI block is disabled by one of the following methods:

- By the NTM88 application clearing SPIEN bit in SIMOPT1 register.
- By the NTM88 entering a STOP mode (STOP4 or STOP1).
- By the NTM88 MCU resetting.

Therefore, care must be taken that the NTM88 SPI block is not disabled while SPI transfers are ongoing by inadvertently clearing SPIEN bit or by the NTM88 MCU entering a STOP mode or resetting.

2.3 SPI commands and error handling

When SPI is enabled in the NTM88, the SPI Host sends commands to perform READ and WRITE operations on the NTM88 RAM and FLASH memory. In the NTM88, the SPI commands are handled by the SPI block directly, not by the application. As a result, only the READ and WRITE commands with the format described in the NTM88 manual[1] are supported by the NTM88 SPI block. Custom SPI frame formats are not supported. An excel file allowing users to encode and decode SPI commands is available as associated file to this application note.

Figure 1 shows that the Host transfers a command during transfer number N, and the NTM88 transfers the response to that command during transfer number N+1. In Figure 1, the R1 frame contains the response to the command included in frame T1. Similarly, R2 frame contains the response to the command included in frame T2.

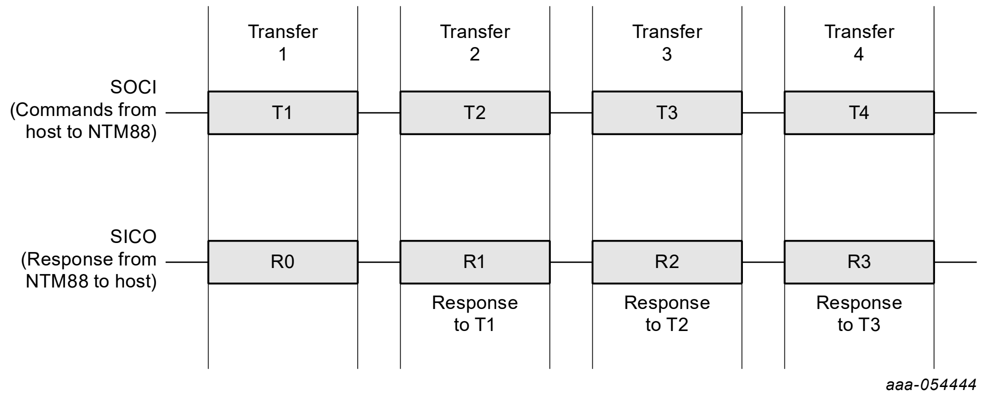


Figure 1. SPI commands and responses

A READ operation is performed in two transfers, as illustrated in Figure 2.

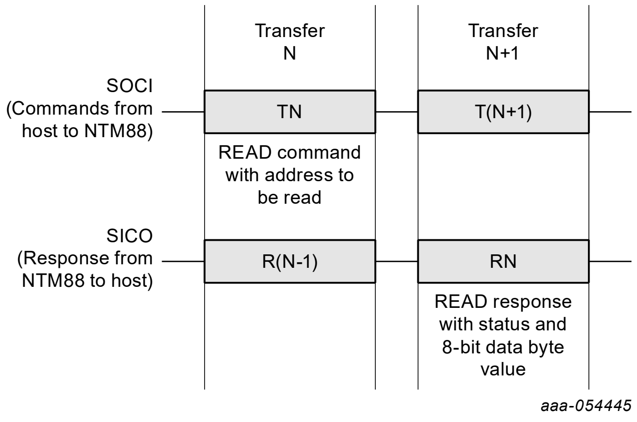

Transfer N
Transfer N+1
SOCI (Commands from host to NTM88)
TN
T(N+1)
READ command with address to be read
SICO (Response from NTM88 to host)
R(N-1)
RN
READ response with status and 8-bit data byte value
aaa-054445

Figure 2. Transfers for a READ operation

A WRITE operation is performed in three transfers as illustrated in Figure 3.

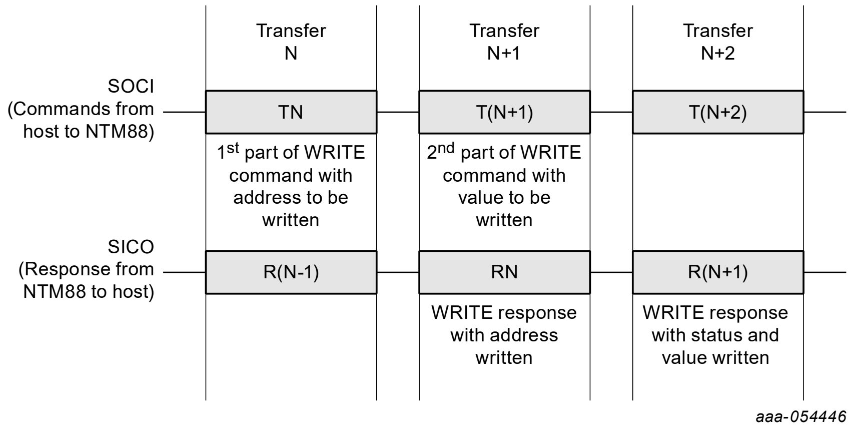

Transfer N
Transfer N+1
Transfer N+2
SOCI (Commands from host to NTM88)
TN
T(N+1)
T(N+2)
1st part of WRITE command with address to be written
2nd part of WRITE command with value to be written
SICO (Response from NTM88 to host)
R(N-1)
RN
R(N+1)
WRITE response with address written
WRITE response with status and value written
aaa-054446

Figure 3. Transfers for a WRITE operation

When reading a response frame from the NTM88, the Host MCU should check the following two points:

- Verify that the parity is correct. If parity is incorrect, the frame should be considered invalid and then discarded by the Host.
- If parity is correct, check the status bits that are included in the frame. Status bits equal to 0 indicate that the frame contains a valid response. Status bits different from 0 indicate that a SPI error occurred and the command was not executed. The list of SPI errors and their corresponding status values is given in the NTM88 manual[1].

If a SPI error occurs during transfer number N, the command included during that transfer is not executed by the NTM88, and the following transfer (number N+1) is used by the NTM88 to clear the error in the SPI block. As a result, the command included in transfer number N+1 is also not executed by the NTM88. Figure 4 illustrates an example of the sequence in case a SPI error occurs during Transfer 3. In such situations, both commands T3 and T4 are ignored by the NTM88.

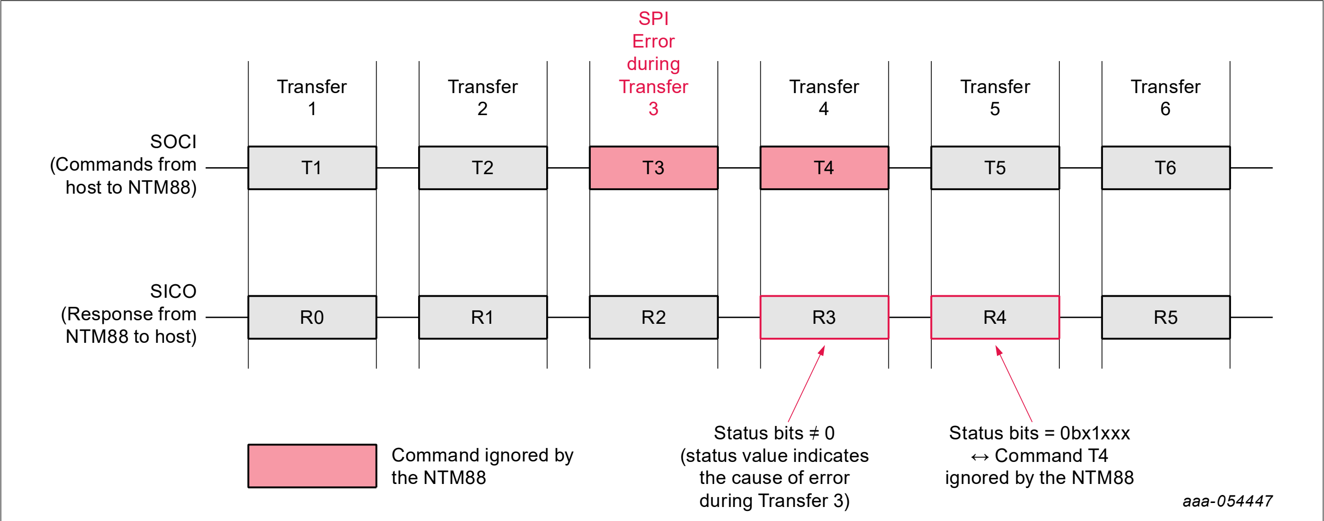

SPI Error during Transfer 3
Transfer 1
Transfer 2
Transfer 4
Transfer 5
Transfer 6
SOCI (Commands from host to NTM88)
T1
T2
T3
T4
T5
T6
SICO (Response from NTM88 to host)
R0
R1
R2
R3
R4
R5
Command ignored by the NTM88
Status bits ≠ 0 (status value indicates the cause of error during Transfer 3)
Status bits = 0bx1xxx ↔ Command T4 ignored by the NTM88
aaa-054447

Figure 4. Example of sequence with a SPI error occurring during transfer number 3

When a command is ignored by the NTM88, it is not executed. In these situations, the Host should retry and send the command again.

Note: *Response frame R0 includes status bits equal to 0bx1xxx in case the T1 command is the first SPI command transferred after a NTM88 reset (including STOP1 exit). A R0 frame with a status equal to 0b01000 is the only situation where non-zero status bits do not correspond to a SPI error.*

2.4 SPI clock error when using PTA0 as KBI pin

In the NTM88, pins PTA0 to PTA3 can be configured as KBI pins allowing a Host to wake up the NTM88 from STOP mode. When PTA0 is used to wake up the NTM88 in order to perform SPI transfers, the specific wake up sequence described in Figure 5 generates a SPI clock fault error on the NTM88 side.

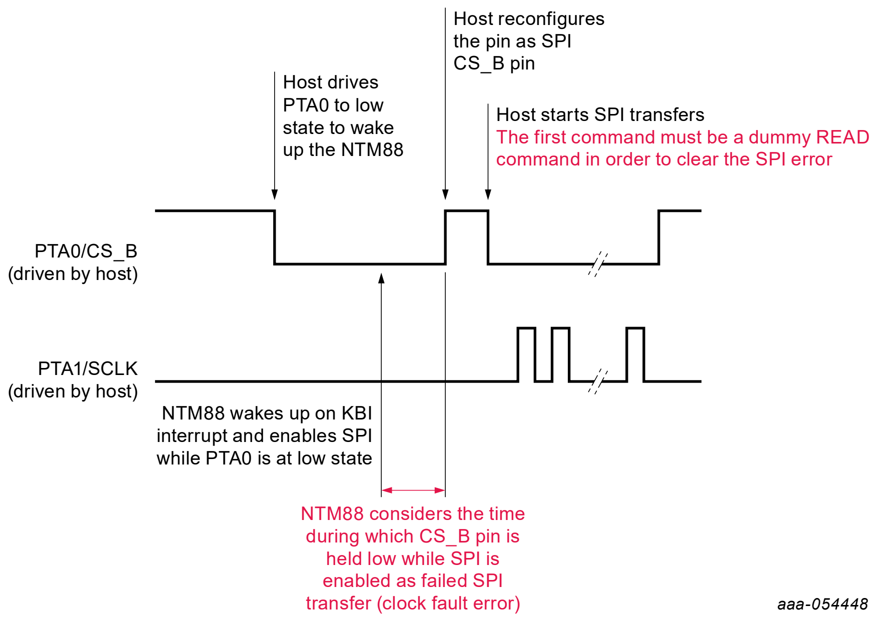

Host reconfigures the pin as SPI CS_B pin
Host drives PTA0 to low state to wake up the NTM88
Host starts SPI transfers
The first command must be a dummy READ command in order to clear the SPI error
PTA0/CS_B (driven by host)
PTA1/SCLK (driven by host)
NTM88 wakes up on KBI interrupt and enables SPI while PTA0 is at low state
NTM88 considers the time during which CS_B pin is held low while SPI is enabled as failed SPI transfer (clock fault error)
aaa-054448

Figure 5. Sequence leading to a SPI clock fault error

A SPI clock fault error is generated when the NTM88 enables the SPI block while PTA0 is still at low state, following the KBI wake up. In this situation, the NTM88 considers the time during which the CS_B pin is held low while the SPI is enabled as a failed SPI transfer due to clock fault error. This scenario results because the NTM88 SPI block sees CS_B pin driven at low state with no clock cycle generated. In order to clear the SPI error, the NTM88 uses the following SPI transfer to clear the error. As a result, the first SPI command sent by the Host is discarded by the NTM88. The first SPI command transferred by the Host must therefore be a dummy READ command.

Figure 6 shows an example of SPI sequence following the clock fault error described in the previous paragraph. The first command transferred by the Host is a dummy READ command used by the NTM88 to clear the SPI error. The SPI status transferred by the NTM88 during the first transfer indicates a clock fault error. The SPI status transferred by the NTM88 during the second transfer indicates that the command of the first transfer was ignored.

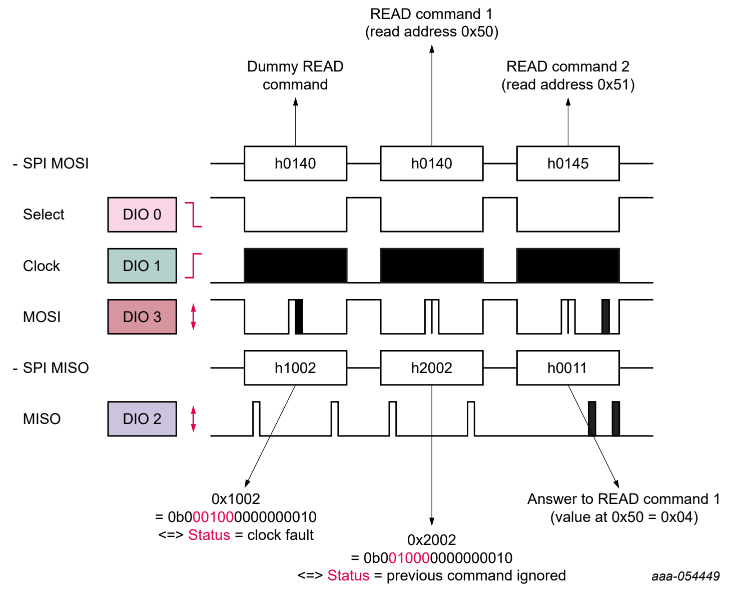


Figure 6. Example of SPI sequence following a SPI clock fault error

In order to avoid generating a clock fault error on the NTM88 side following a KBI wake up, the following options are available:

- Do not use PTA0 as the KBI pin to wake up the NTM88. Use PTA1, PTA2 or PTA3 instead.
- If using PTA0 as the KBI pin, the NTM88 application should wait for PTA0 to be driven back to high state before enabling the SPI block. Figure 7 shows a sequence that does not generate a SPI error. If such a sequence is executed, care must be taken that the NTM88 application enables the SPI block before the Host starts the SPI transfers.

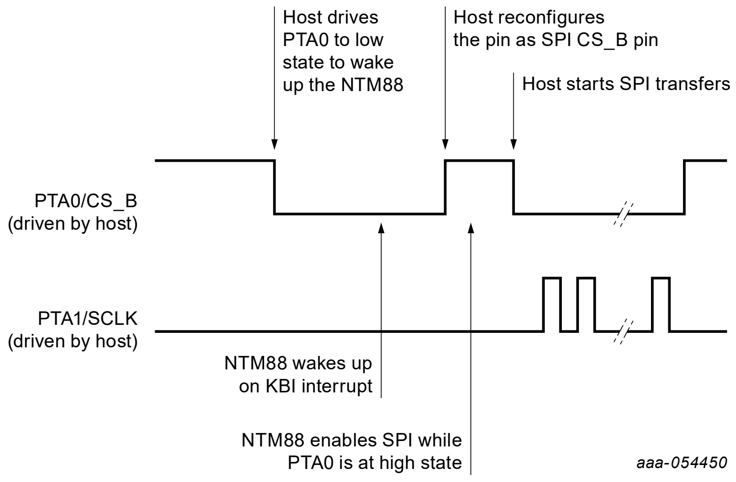

Host drives PTA0 to low state to wake up the NTM88
Host reconfigures the pin as SPI CS_B pin
Host starts SPI transfers
PTA0/CS_B (driven by host)
PTA1/SCLK (driven by host)
NTM88 wakes up on KBI interrupt
NTM88 enables SPI while PTA0 is at high state
aaa-054450

Figure 7. Sequence not generating a SPI error

2.5 Bus resources

In the NTM88, the SPI block and the CPU share the internal address, data and control bus. The SPI and CPU cannot use the internal bus at the same time. In other words, when the SPI block is accessing the bus, CPU instructions are not executed. In order to avoid unwanted inhibition of CPU instructions, enable the NTM88 SPI block only when needed.

Similarly, when the CPU is accessing the bus, access to the bus is denied to the SPI block. If the Host sends SPI commands while the NTM88 does not have access the to bus, the NTM88 is not able to process the commands. When this occurs, the status bits in the SPI response frame indicate that an *internal bus contention fault* has occurred. Similar to any other SPI error, the NTM88 uses the following transfer to clear the error. So, the command included in the following transfer is discarded as well.

In order to prevent bus contention errors, it is possible to halt the NTM88 CPU while SPI transfers are ongoing. The CPU is halted by setting CORE_TR_HOLD bit of SPIOPS register. When CORE_TR_HOLD bit is set, the CPU is halted, so program instructions are not executed anymore. CORE_TR_HOLD bit can be set by the NTM88 application executing the instruction **SPIOPS_CORE_TR_HOLD = 1**, or by the Host via a SPI WRITE command. CORE_TR_HOLD bit must be cleared by the Host via a SPI WRITE command to allow the NTM88 CPU to resume at the end of the SPI sequence.

Halting the CPU does not halt the NTM88 internal clocks. The functions relying on the clocks, like the watchdog, continue to work when the CPU is halted. A watchdog reset automatically restarts the CPU. Use the watchdog to ensure that the NTM88 is not trapped in RUN mode with the CPU halted when the SPI Host fails to clear CORE_TR_HOLD bit.

2.6 Selecting the FLASH address range

In the SPI command format, the address to be read or written is 13-bit long, resulting in a SPI address range from $0000 to $1FFF. The address range is enough to address the entire NTM88 RAM memory ($0000 to $028F), but additional bits are necessary to address the full range of the FLASH. These additional bits are located in the SPIOPS register located in RAM at address $0038.

The last two bits of the SPIOPS register, FLS_ADDR[1:0], allow the Host to select the FLASH range to address, as described in Table 1.

Table 1. Mapping of SPI address to FLASH address

SPI Address Range		FSL_ADDR[1:0]			
		00	01	10	11
Start	0x0800	0xC000	0xD000	0xE000	0xF000
End	0x17FF	0xCFFF	0xDFFF	0xEFFF	0xFFFF

Note: *The SPI Address Range 0x0000-0x07FF corresponds to the NTM88 RAM memory (valid range from 0x0000 to 0x028F).*

When the External MCU wants to access an address in FLASH, the SPIOPS_FSL_ADDR[1:0] field must be configured first with the appropriate value, and then the READ or WRITE command can be transferred along with the 13-bit SPI address within the range 0x0800 to 0x17FF.

Examples:

- To access address $C000 in FLASH, FSL_ADDR[1:0] must be set to ‘00’ and the 13-bit address transferred via SPI must be equal to 0x0800.
- To access address $C010 in FLASH, FLS_ADDR[1:0] must be set to ‘00’ and the 13-bit address transferred via SPI must be equal to 0x0810.
- To access address $E010 in FLASH, FSL_ADDR[1:0] must be set to ‘10’ and the 13-bit address transferred via SPI must be equal to 0x0810.
- To access address $FFFF in FLASH, FSL_ADDR[1:0] must be set to ‘11’ and the 13-bit address transferred via SPI must be equal to 0x17FF.

Important: *The SPIOPS register includes the following fields described in Table 2.*

Table 2. SPIOPS register fields (address $0038)

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
—	—	—	—	—	CORE_TR_HOLD	FSL_ADDR1	FSL_ADDR0

Setting the CORE_TR_HOLD bit halts the NTM88 CPU. Clearing the CORE_TR_HOLD bit allows the CPU to resume. As described in Section 2.5 "Bus resources", the NTM88 CPU can be kept halted for the duration of the SPI transfers in order to avoid bus contention faults. Care must be taken by the Host MCU to keep the CORE_TR_HOLD bit set while writing the SPIOPS register to update FSL_ADDR[1:0] bits.

2.7 Writing and erasing the NTM88 FLASH

It is important to understand that writing or erasing the NTM88 FLASH cannot be done by simply sending a SPI WRITE command to an address in FLASH. In order to write or erase FLASH bytes, the Host MCU must follow the program and erase command flows described in section “Flash Memory Controller (FMC) module” of the NTM88 user manual[1].

FLASH is written byte by byte, and is erased page by page. In the NTM88, one page is 512-byte long. It is not possible to erase a single byte in FLASH. The minimum amount that can be erased is 512 bytes.

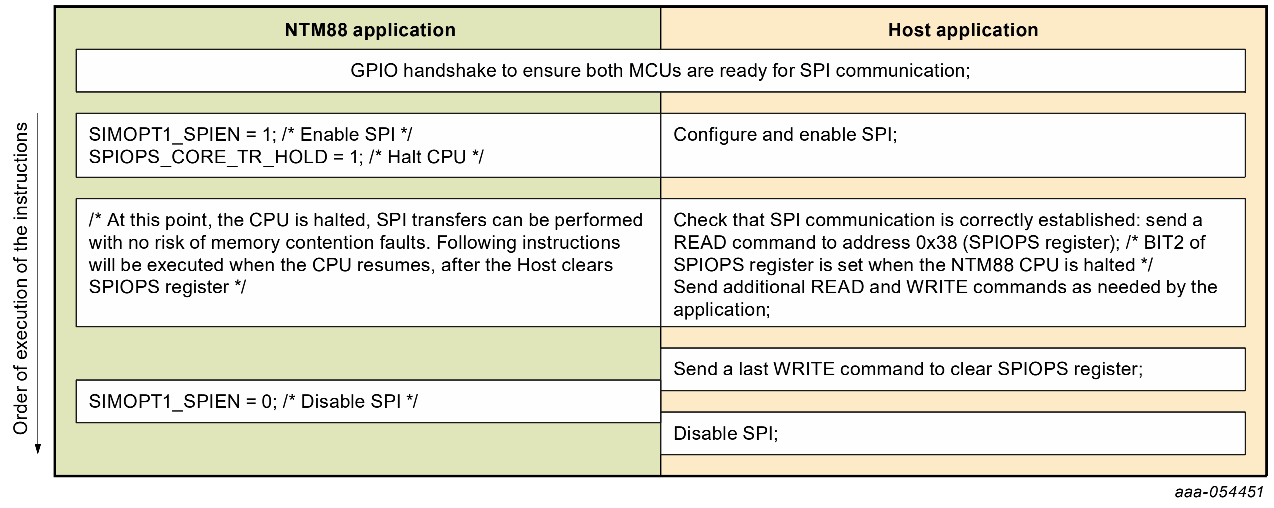
The “Flash Memory Controller (FMC) module” section of the NTM88 user manual[1] states that a command to perform a mass erase on the NTM88 FLASH is available. A mass erase allows the entire FLASH memory of the NTM88 to be erased at once. It is important to understand that a mass erase also erases the trim section of the NTM88. The trim section includes coefficients calibrating the sensors that are unique to each device. If the NTM88 trim section is erased, the sensors are not functional anymore and the device cannot be recovered.

2.8 Example of implementation

On the NTM88 side, since transfers are handled at block-level and not application level, no application or software driver are needed. The NTM88 application should ensure the following points:

- The SPI block must be enabled before the Host starts the transfers.
- The SPI block must remain enabled as long as transfers are ongoing: SIMOPT1_SPIEN bit must not be cleared and the NTM88 must not reset nor enter a STOP mode.
- In order to prevent memory contention issues, the NTM88 CPU should be halted for the duration of the transfers. At the end of the transfers, the Host lets the NTM88 CPU resume by clearing SPIOPS register.

Figure 8 summarizes the flow of instructions on the NTM88 and Host sides for a typical implementation.



NTM88 application	Host application
GPIO handshake to ensure both MCUs are ready for SPI communication;	

SIMOPT1_SPIEN = 1; /* Enable SPI */
SPIOPS_CORE_TR_HOLD = 1; /* Halt CPU */
	Configure and enable SPI;

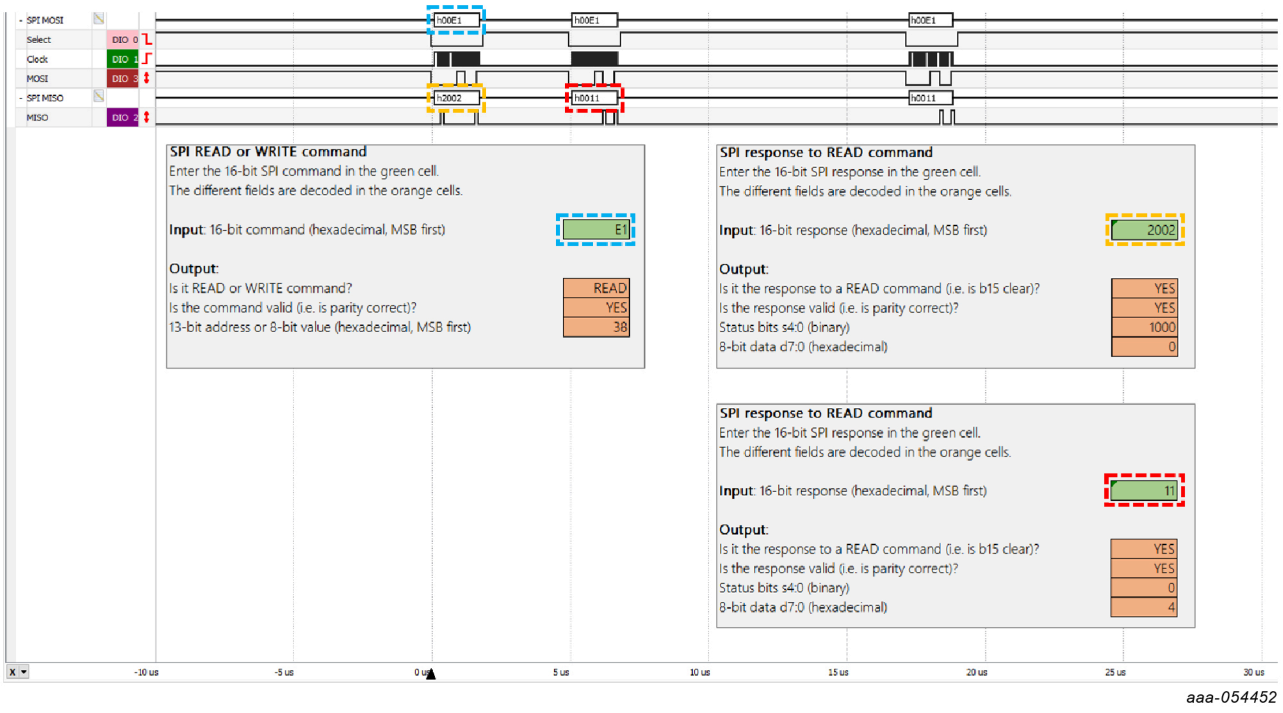
/* At this point, the CPU is halted, SPI transfers can be performed with no risk of memory contention faults. Following instructions will be executed when the CPU resumes, after the Host clears SPIOPS register */	Check that SPI communication is correctly established: send a READ command to address 0x38 (SPIOPS register); /* BIT2 of SPIOPS register is set when the NTM88 CPU is halted */ Send additional READ and WRITE commands as needed by the application;
	Send a last WRITE command to clear SPIOPS register;
SIMOPT1_SPIEN = 0; /* Disable SPI */	
	Disable SPI;

Figure 8. Flow of instructions in a typical implementation

2.9 Examples of transfers

Figure 9 shows SPI transfers allowing the Host to check the status of SPI communication at the start of the transfer sequence. Extracts of the excel file allowing encoding and decoding of SPI commands and responses are also provided.

- During the first transfer, the Host sends command 0x00E1 in T1 frame, corresponding to a READ command to address $0038 (SPIOPS register).
- During the first transfer, the NTM88 answers 0x2002 in R0 frame, corresponding to a status value of 0b01000. As mentioned in section SPI commands and error handling, a status value of 0b01000 included in frame R0 does not correspond to a SPI error but indicates that R0 is the first response transferred by the NTM88 after reset.
- The response to command 0x00E1 is given by the NTM88 during the second transfer. The NTM88 answers 0x0011 in R1 frame, corresponding to a status clear and data byte equal to 0x04. Such response indicates that BIT2 (0x04) of SPIOPS register is set, meaning that the NTM88 CPU is halted, and SPI communication has been successfully established.



SPI READ or WRITE command
Enter the 16-bit SPI command in the green cell.
The different fields are decoded in the orange cells.

Input: 16-bit command (hexadecimal, MSB first)	E1
Output:	
Is it READ or WRITE command?	READ
Is the command valid (i.e. is parity correct)?	YES
13-bit address or 8-bit value (hexadecimal, MSB first)	38

SPI response to READ command
Enter the 16-bit SPI response in the green cell.
The different fields are decoded in the orange cells.

Input: 16-bit response (hexadecimal, MSB first)	2002
Output:	
Is it the response to a READ command (i.e. is b15 clear)?	YES
Is the response valid (i.e. is parity correct)?	YES
Status bits s4:0 (binary)	1000
8-bit data d7:0 (hexadecimal)	0

SPI response to READ command
Enter the 16-bit SPI response in the green cell.
The different fields are decoded in the orange cells.

Input: 16-bit response (hexadecimal, MSB first)	11
Output:	
Is it the response to a READ command (i.e. is b15 clear)?	YES
Is the response valid (i.e. is parity correct)?	YES
Status bits s4:0 (binary)	0
8-bit data d7:0 (hexadecimal)	4

Figure 9. Start of the SPI sequence: Host reading SPIOPS register

Figure 10 shows the SPI signals during the first two transfers. Note that the CS_B signal goes back to high state after each 16-bit transfer.

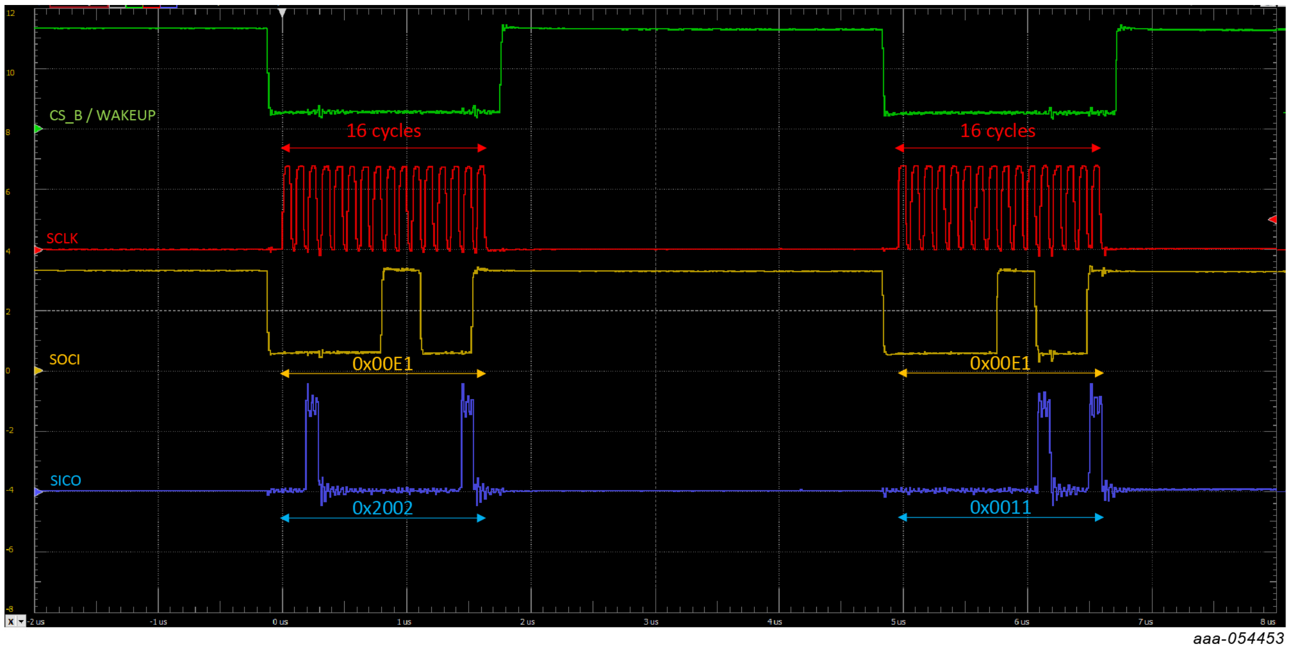


Figure 10. View of the SPI signals during the first two transfers

Figure 11 shows the SPI transfers performed at the end of the SPI sequence. The Host sends commands to write value 0x00 at address $0038, in order to clear SPIOPS register and let the NTM88 CPU resume.

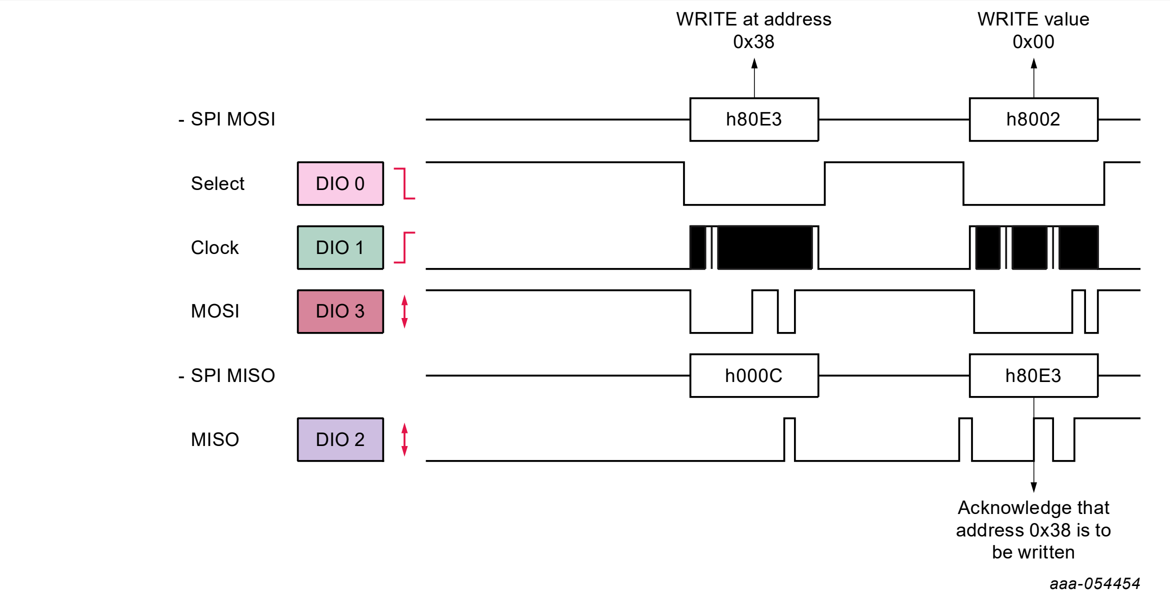


Figure 11. End of the SPI sequence: Host clearing SPIOPS register

Generally speaking, after the two transfers of a WRITE command have been performed, the Host can perform one additional transfer in order to read the status of the second transfer. However, this is not possible in case of the WRITE command clearing SPIOPS register. The reason is that once the NTM88 has executed the command clearing the SPIOPS register, the NTM88 CPU restarts and the NTM88 application then typically disables the SPI block. Any additional SPI commands transferred after the second transfer of the WRITE command are not answered due to the SPI block being disabled.

3 NTM88 as SPI Server (Host)

3.1 Implementation

The NTM88 device does not include a hardware SPI block that can be configured in Host mode. In order for the NTM88 to act as SPI Host, the NTM88 application performs SPI bit-banging using software drivers controlling the GPIOs in order to emulate a SPI block working in Host mode.

The software drivers and example of implementation are provided in the demo project provided as associated file to this application note. In the demo project, the files *spi.c* and *spi.h* include the code of the software drivers. The file *app_spi.c* gives an example of a function using the drivers to transfer four bytes via SPI. The function is called periodically in the main().

When using the drivers, the following points are to be noted:

- The macro SPI_INIT_PINS_MACRO is to be executed at the start of the SPI sequence.
- SPI transfers are performed by executing the function **SPI_XferPacket(UINT8 *pTxB, UINT8 *pRxB, UINT8 DataLen)**, which takes as parameters the transmit buffer, receive buffer and number of bytes of the transfer. The function returns when the transfer has been performed.
- The transfers are performed in RUN. While the transfers are ongoing, the watchdog keeps running, if enabled by the application.
- The macro SPI_INIT_PINS_MACRO is to be executed at the end of the SPI sequence, after all desired transfers have been performed.

The user selects which GPIO to map to each SPI signal in the file *spi.h*. Any of the PTA[3:0] and PTB[1:0] GPIO can be used for SS_B, SCLK, SOCI and SICO signals. PTA4 pin can be configured as GPIO output only by clearing SIMOPT1_BKGDPE bit. As a result, PTA4 can be used for SS_B, SCLK or SOCI signal, but not for SICO signal which requires the pin to be configured as GPIO input.

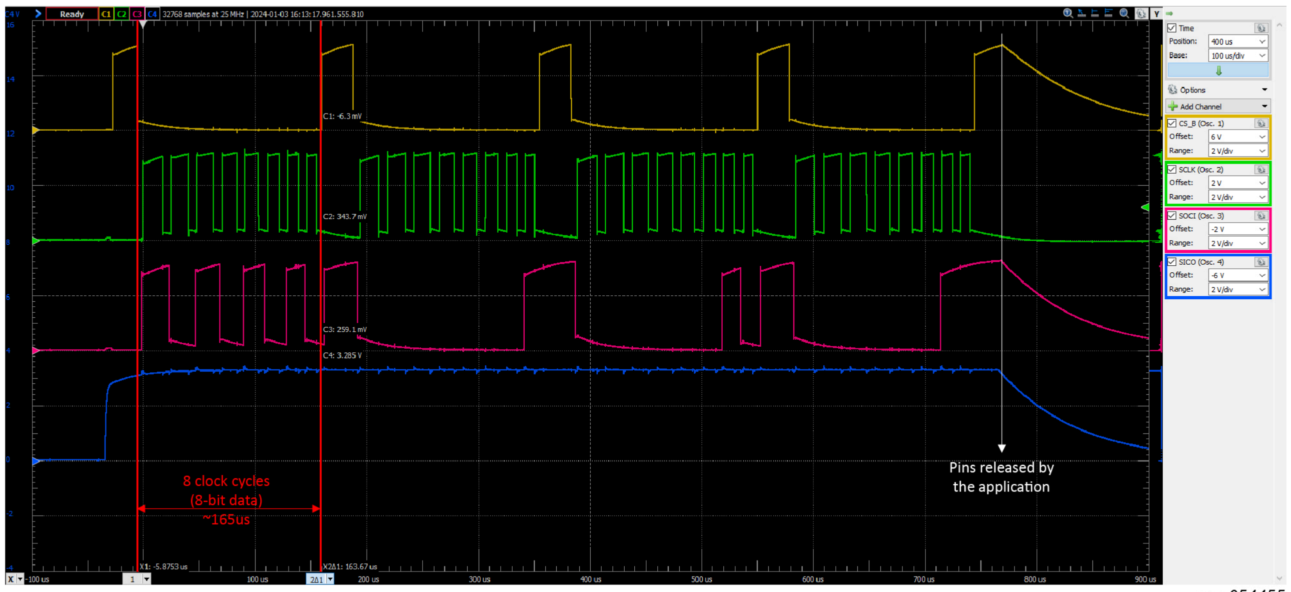
Note: *By default after reset (including STOP1 exit), PTA4 pin is configured as BKGD pin with an internal pull-up enabled. As a result, if PTA4 pin is used as SPI signal, care must be taken not to connect a pull-down resistor to the pin. Background debug mode is entered when PTA4 pin is driven to low state during reset. Care must also be taken not to drive PTA4 pin to low state during reset.*

3.2 Baud rate

The SPI clock signal is generated by the software drivers, and not by a timer. The resulting baud rate directly depends on the bus clock configuration. With the default bus clock speed of 4 MHz, the SPI baud rate is around 50 kbps. If the bus clock is configured to a lower speed, the baud rate will decrease accordingly.

3.3 Example of transfers

Figure 12 shows the SPI signals during the 4-byte transfer implemented in the demo project. When the screenshot was taken, no SPI client was connected to the Host, which explains why the SICO line remains at high state for the duration of the transfer.



32768 samples at 25 MHz | 2024-01-03 16:13:17.961.555.810

C1: -6.3 mV
C2: 343.7 mV
C3: 259.1 mV
C4: 3.285 V
X1: -5.8753 us
X2Δ1: 163.67 us

Time
Position: 400 us
Base: 100 us/div

CS_B (Osc. 1)
Offset: 6 V
Range: 2 V/div

SCLK (Osc. 2)
Offset: 2 V
Range: 2 V/div

SOCI (Osc. 3)
Offset: -2 V
Range: 2 V/div

SICO (Osc. 4)
Offset: -6 V
Range: 2 V/div

Figure 12. SPI signals during the transfer implemented in the demo project

4 NTM88 as I^{2}C Controller

4.1 Implementation

The NTM88 device does not include an I^{2}C block. In order for the NTM88 to act as I^{2}C Controller, the NTM88 application performs I^{2}C bit-banging, using software drivers controlling the GPIOs in order to emulate an I^{2}C block working in Controller mode.

The software drivers and example of implementation are provided in the demo project provided as associated file to this application note. In the demo project, the files *i2c.c* and *i2c.h* include the code of the software drivers. The file *app_i2c.c* provides an example of a function using the drivers to perform multiple transfers via I^{2}C. The function is called periodically in the main().

When using the drivers, note the following points:

- The macro I2C_INIT_PINS_MACRO is to be executed at the start of the I^{2}C sequence.
- The function **I2C_XferPacket_AckCheck(UINT8 I2CAdd, UINT8 *pTxB, UINT8 DataLen)** transfers data bytes on the SDA line. The function takes as parameter the I^{2}C address, transmit buffer and the number of data bytes to transmit. The number of data bytes does not include the length of the I^{2}C address.
  In the function, after transmission of the I^{2}C address, the ACK bit is checked. If acknowledge is received (ACK = 0), the function proceeds with the transmission of the data bytes. After the transmission of each data byte, the ACK bit is checked. The next data byte is transmitted only if the previous one has been acknowledged (ACK = 0). The function returns value 0 when all data bytes have been transferred or value 1 when one data byte has not been acknowledged and the transfer was aborted.
- The function **I2C_XferPacket_NoAckCheck(UINT8 I^{2}CAdd, UINT8 *pTxB, UINT8 DataLen)** transfers data bytes on the SDA line. The function takes as parameter the I^{2}C address, transmit buffer and number of data bytes to transmit. The number of data bytes does not include the length of the I^{2}C address.
  In the function, after transmission of the I^{2}C address, the ACK bit is checked. If acknowledge is received (ACK = 0), the function proceeds with the transmission of the data bytes. The ACK bit is then not checked anymore during the transmission of the data bytes. The function generates the ninth clock cycle (ACK cycle) after the transmission of the 8-bit data, as specified in the I^{2}C protocol[5], but the value of the ACK bit is not checked. The function returns when all data bytes have been transferred. No status is returned by the function to the application.
  Calling this function is suitable when the application is not expecting an acknowledge of the data bytes.
- The function **I2C_RcvPacket(UINT8 I2CAdd, UINT8 *pRxB, UINT8 DataLen)** reads data bytes on the SDA line. The function takes as parameter the I^{2}C address, receive buffer and number of data bytes to read.
  In the function, after transmission of the I^{2}C address, the ACK bit is checked. If acknowledge is received (ACK = 0), the function proceeds with reading the number of data bytes passed as parameter.
  The function returns value 0 when the I^{2}C address was correctly acknowledged and data bytes were read and stored. The function returns value 1 when the I^{2}C address was not acknowledged and data bytes were not read.
- The transfers are performed in RUN. While the transfers are ongoing, the watchdog keeps running, if enabled by the application.

The user selects which GPIO to map to each I^{2}C signal in the file *i2c.h*. Any of the PTA[3:0] and PTB[1:0] GPIO can be used for SCL and SDA signals. PTA4 pin can be configured as GPIO output only by clearing SIMOPT1_BKGDPE bit. As a result, PTA4 can be used for SCL signal (with clock stretching disabled), but not for SDA signal which requires the pin to be configured as GPIO input.

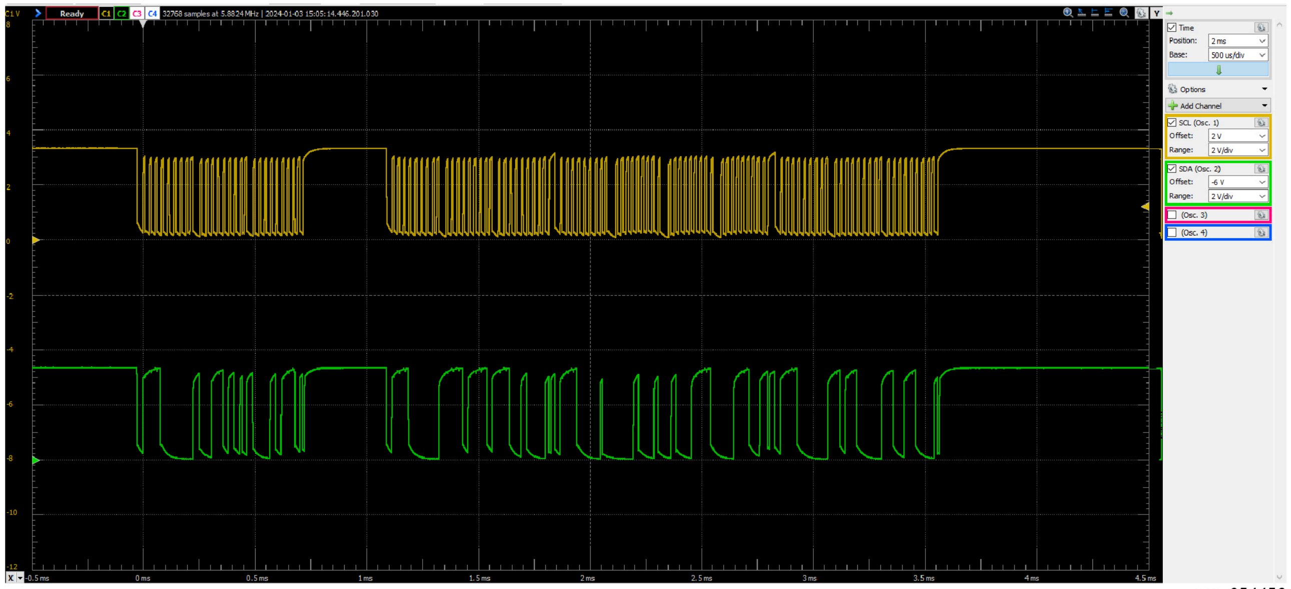
Note: *By default after reset (including STOP1 exit), the PTA4 pin is configured as BKGD pin with an internal pull-up enabled. Background debug mode is entered when the PTA4 pin is driven to low state during reset. Care must be taken not to drive PTA4 pin to low state during reset.*

4.2 Baud rate

The I^{2}C clock signal is generated by the software drivers, and not by a timer, so the resulting baud rate directly depends on the bus clock configuration. With the default bus clock speed of 4 MHz, the I^{2}C baud rate is around 35 kbps. If the bus clock is configured to a lower speed, the baud rate will decrease accordingly.

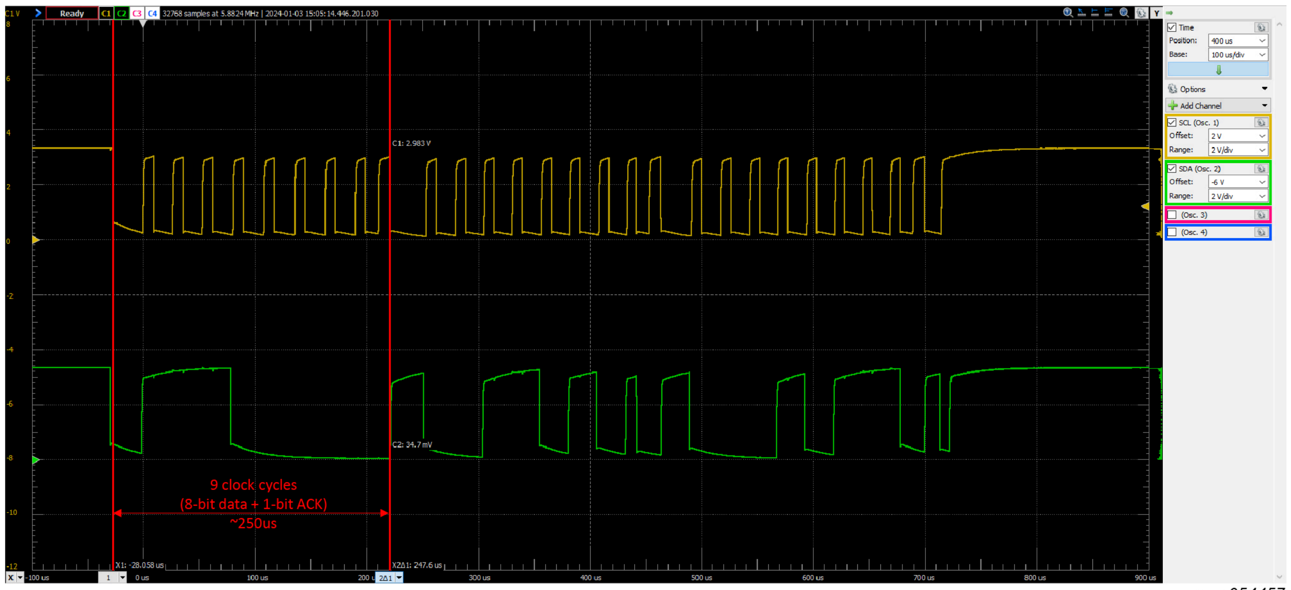
4.3 Example of transfers

Figure 13 and Figure 14 show the I^{2}C signals during the transfers implemented in the demo project. The screenshots were captured with an I^{2}C sensor connected to the NTM88. The acknowledges and data bytes received on the SDA line come from the sensor.



Time
Position: 2 ms
Base: 500 us/div
SCL (Osc. 1)
Offset: 2 V
Range: 2 V/div
SDA (Osc. 2)
Offset: -6 V
Range: 2 V/div

Figure 13. I^{2}C signals during the transfers implemented in the demo project



Time
Position: 400 us
Base: 100 us/div
SCL (Osc. 1)
Offset: 2 V
Range: 2 V/div
SDA (Osc. 2)
Offset: -6 V
Range: 2 V/div

Figure 14. Zoom on the first I^{2}C transfer

5 References

[1] UM11227, NTM88 User Manual

[2] NTM88Hxx5S, Tire pressure monitor sensor

[3] NTM88Jxx5S, Tire pressure monitor sensor

[4] NTM88Kxx5S, Tire pressure monitor sensor

[5] UM10204, I^{2}C-bus specification and user manual

6 Revision history

Table 3. Revision history

Rev	Date	Description
AN14182 v.1	21 March 2024	• Initial release

Legal information

Tables

Figures

Contents

Date of release: 21 March 2024
Document identifier: AN14182