AN14599

Implementing wired communication with the NPM8

Rev. 1.0 — 27 March 2025 Application note

Document information

Information	Content
Keywords	NPM8, wired communication, SPI, I^2C
Abstract	This application note explains how to implement wired communication with the NPM8.

1 Introduction

The NPM8 device can be used in a variety of applications. For example, it may be connected on the printed-circuit board (PCB) to another MCU, to external memory, to external sensors, or to tag devices. In such implementations, the NPM8 must communicate with other devices using wired communication.

This application note explains how to use the NPM8 as a 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 NPM8 to act as an I^2C target for the following reasons:

- In most applications, the NPM8 needs to be a host for other downstream devices such as memory or other sensors.
- The NPM8 already supports SPI Client mode if the NPM8 needs to be a peripheral to an upstream host.

In this document, there are 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 NPM8 family.

2 NPM8 as SPI client

2.1 SPI configuration

The NPM8 device includes a hardware SPI block that can be used in Client mode only. The block is described in detail in the NPM8 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 NPM8. 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 NPM8 user manual[1]. Their durations are specified in the NPM8 data sheet[2].

2.2 Enabling and disabling the SPI block

The NPM8 SPI block is enabled by one of the following two methods:

- By the NPM8 application setting the SPIEN bit in the SIMOPT1 register.
- At power-on reset (POR), by the host holding the PTA0 pin at Low state for a duration greater than t_{SPI_EN} specified in the data sheet[2].

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

- By the NPM8 application clearing the SPIEN bit in SIMOPT1 register.
- By the NPM8 entering a Stop mode (STOP4 or STOP1).
- By the NPM8 MCU resetting.

Therefore, care must be taken that the NPM8 SPI block is not disabled while SPI transfers are ongoing by inadvertently clearing SPIEN bit or by the NPM8 MCU entering a Stop mode or resetting.

2.3 SPI commands and error handling

When the SPI is enabled in the NPM8, the SPI host sends commands to perform read and write operations on the NPM8 RAM and flash memory. In the NPM8, 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 NPM8 user manual[1] are supported by the NPM8 SPI block. Custom SPI frame formats are not supported. An excel file allowing users to encode and decode SPI commands is available as an associated file to this application note.

Figure 1 shows that the host transfers a command during transfer number N, and the NPM8 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, the R2 frame contains the response to the command included in the T2 frame.

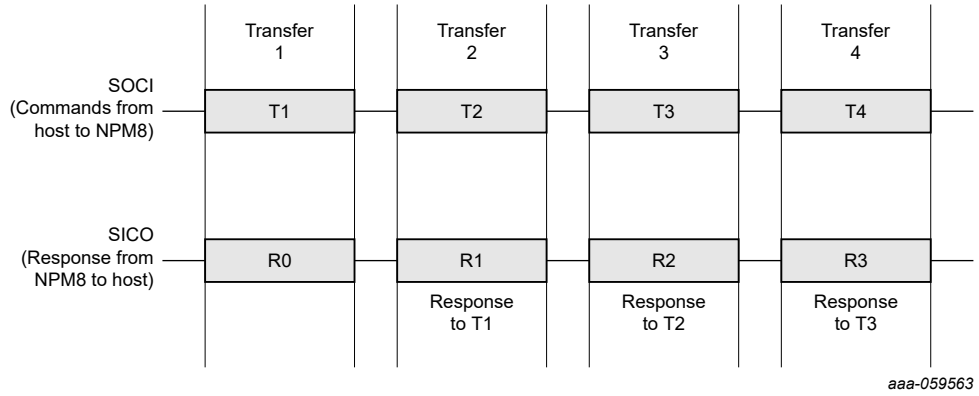

Figure 1. SPI commands and responses

A read operation is performed in two transfers, as shown in Figure 2.

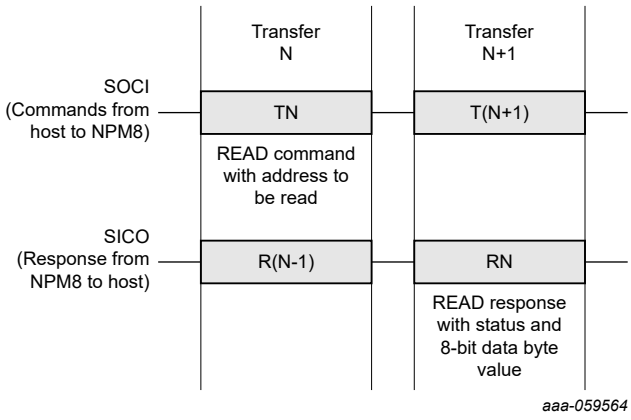

Figure 2. Transfers for a read operation

A write operation is performed in three transfers, as shown in Figure 3.

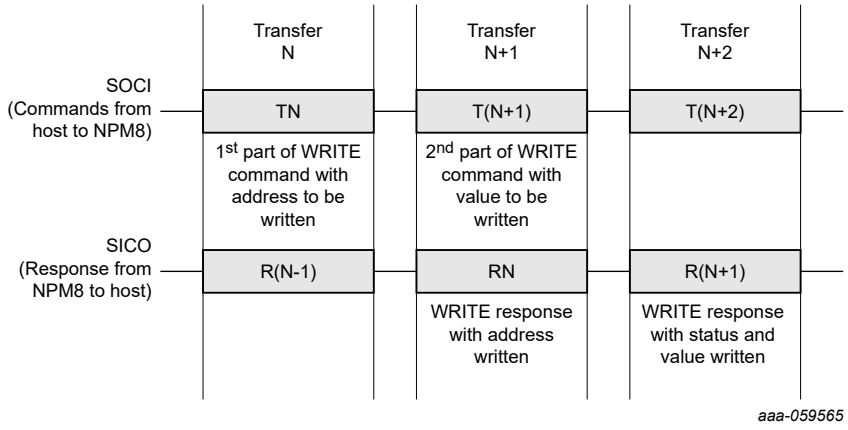

Figure 3. Transfers for a write operation

When reading a response frame from the NPM8, the host MCU should check the following two points:

- Verify that the parity is correct. If the parity is incorrect, the frame should be considered invalid and then discarded by the host.
- If the 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 NPM8 manual[1].

If a SPI error occurs during transfer number N, the command included during that transfer is not executed by the NPM8, and the following transfer (number N+1) is used by the NPM8 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 NPM8. Figure 4 shows an example of the sequence in case a SPI error occurs during Transfer 3. In such situations, both T3 and T4 commands are ignored by the NPM8.

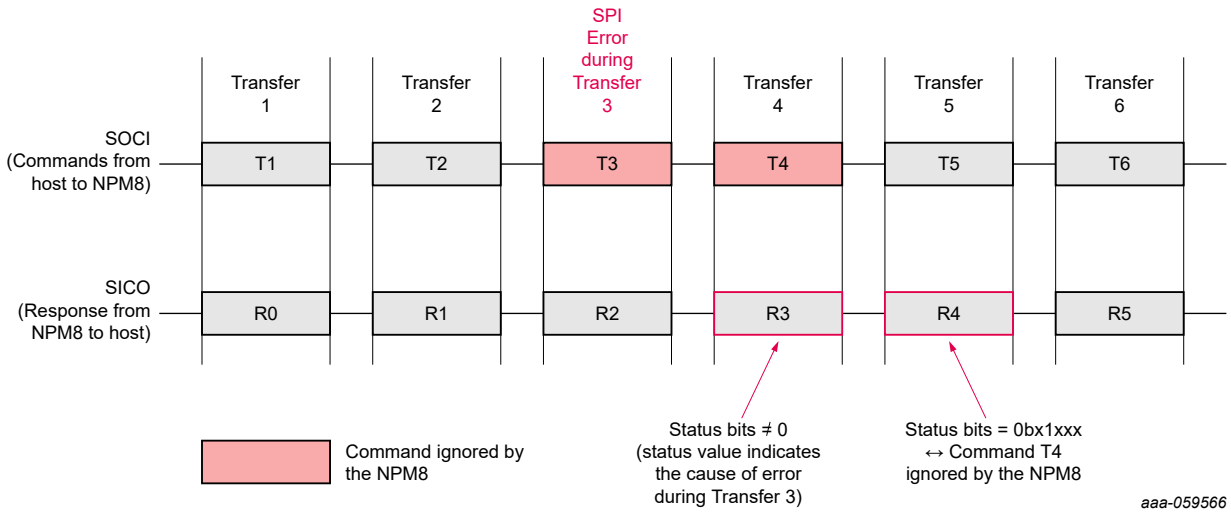

Figure 4. Example of a sequence with a SPI error occurring during transfer number 3

When a command is ignored by the NPM8, it is not executed. In these situations, the host should retry and send the command again.

Note: *Response R0 frame includes status bits equal to 0bx1xxx in case the T1 command is the first SPI command transferred after an NPM8 reset (including STOP1 exit). An 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 NPM8, pins PTA0 to PTA3 can be configured as KBI pins allowing a host to wake up the NPM8 from Stop mode. When PTA0 is used to wake up the NPM8 in order to perform SPI transfers, the specific wake-up sequence described in Figure 5 generates a SPI clock fault error on the NPM8 side.

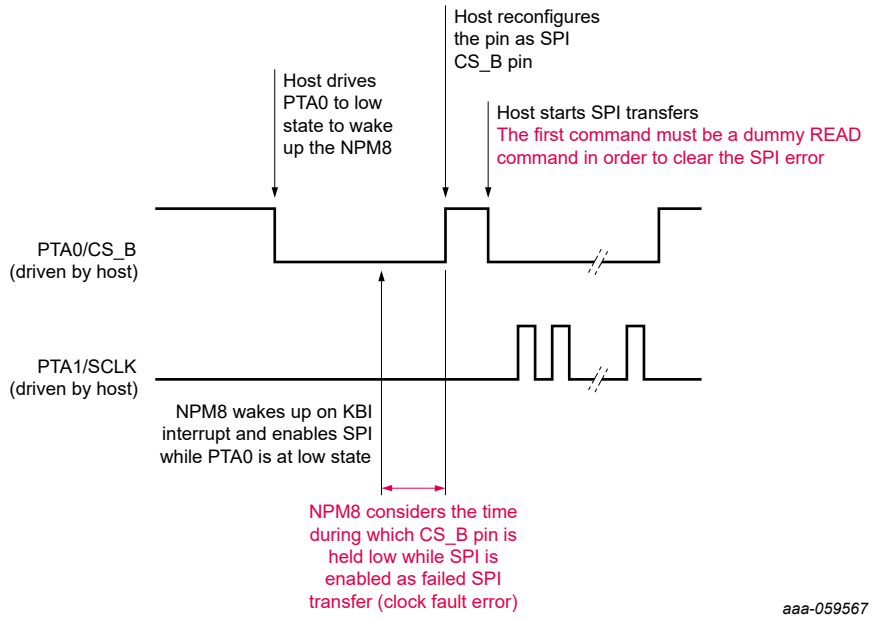

Figure 5. Sequence leading to a SPI clock fault error

A SPI clock fault error is generated when the NPM8 enables the SPI block while PTA0 is still at low state, following the KBI wake-up. In this situation, the NPM8 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 NPM8 SPI block sees CS_B pin driven at low state with no clock cycle generated. In order to clear the SPI error, the NPM8 uses the following SPI transfer to clear the error. As a result, the first SPI command sent by the host is discarded by the NPM8. 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 NPM8 to clear the SPI error. The SPI status transferred by the NPM8 during the first transfer indicates a clock fault error. The SPI status transferred by the NPM8 during the second transfer indicates that the command of the first transfer was ignored.

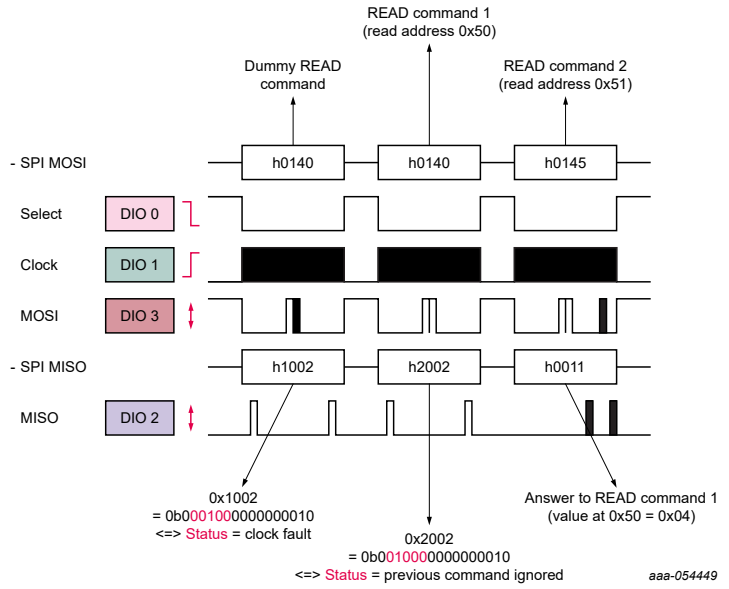

Figure 6. Example of SPI sequence following a SPI clock fault error

In order to avoid generating a clock fault error on the NPM8 side after a KBI wake-up, the following options are available:

- Do not use PTA0 as the KBI pin to wake up the NPM8. Use PTA1, PTA2 or PTA3 instead.
- If using PTA0 as the KBI pin, the NPM8 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 NPM8 application enables the SPI block before the host starts the SPI transfers.

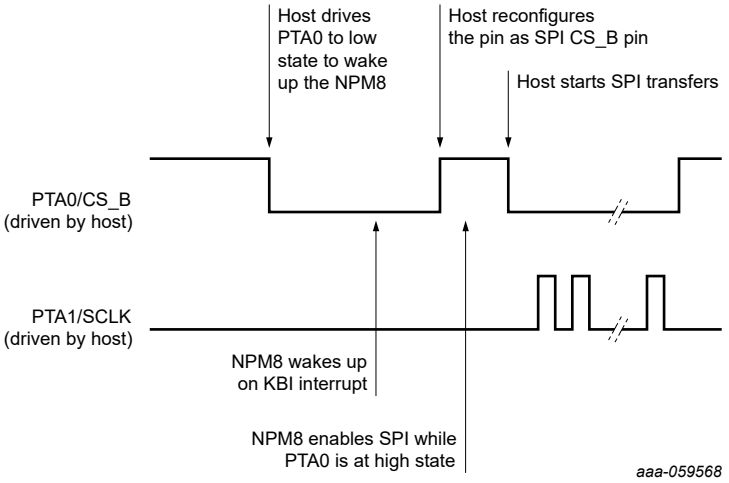

Figure 7. Sequence not generating a SPI error

2.5 Bus resources

In the NPM8, 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 NPM8 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 NPM8 does not have access to the bus, the NPM8 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 NPM8 uses the following transfer to clear the error. The command included in the following transfer is discarded as well.

In order to prevent bus contention errors, it is possible to halt the NPM8 CPU while SPI transfers are ongoing. The CPU is halted by setting the CORE_TR_HOLD bit of SPIOPS register. When the CORE_TR_HOLD bit is set, the CPU is halted, so program instructions are not executed anymore. The CORE_TR_HOLD bit can be set by the NPM8 application executing the instruction SPIOPS_CORE_TR_HOLD = 1, or by the host via a SPI write command. The CORE_TR_HOLD bit must be cleared by the host via a SPI WRITE command to allow the NPM8 CPU to resume at the end of the SPI sequence.

Halting the CPU does not halt the NPM8 internal clocks. The functions relying on the clocks, such as the watchdog, continue to work when the CPU is halted. A watchdog reset automatically restarts the CPU. Use the watchdog to ensure that the NPM8 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 bits long, resulting in a SPI address range from $0000 to $1FFF. The address range is enough to address the entire NPM8 RAM memory ($0000 to $028F), but additional bits are necessary to address the full range of the flash. These additional bits are in the SPIOPS register in the 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 NPM8 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 NPM8 CPU. Clearing the CORE_TR_HOLD bit allows the CPU to resume. As described in Section 2.5 "Bus resources", the NPM8 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 NPM8 flash

It is important to understand that writing or erasing the NPM8 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 NPM8 user manual[1].

Flash is written byte by byte, and is erased page by page. In the NPM8, 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 NPM8 user manual[1] states that a command to perform a mass erase on the NPM8 flash is available. A mass erase allows the entire flash memory of the NPM8 to be erased at once. It is important to understand that a mass erase also erases the trim section of the NPM8. The trim section includes coefficients calibrating the sensors that are unique to each device. If the NPM8 trim section is erased, the sensors are not functional anymore and the device cannot be recovered.

2.8 Example of implementation

On the NPM8 side, because transfers are handled at block-level and not application level, no application or software driver is needed. The NPM8 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 NPM8 must not reset nor enter a Stop mode.
- In order to prevent memory contention issues, the NPM8 CPU should be halted for the duration of the transfers. At the end of the transfers, the host lets the NPM8 CPU resume by clearing the SPIOPS register.

Figure 8 summarizes the flow of instructions on the NPM8 and host sides for a typical implementation.

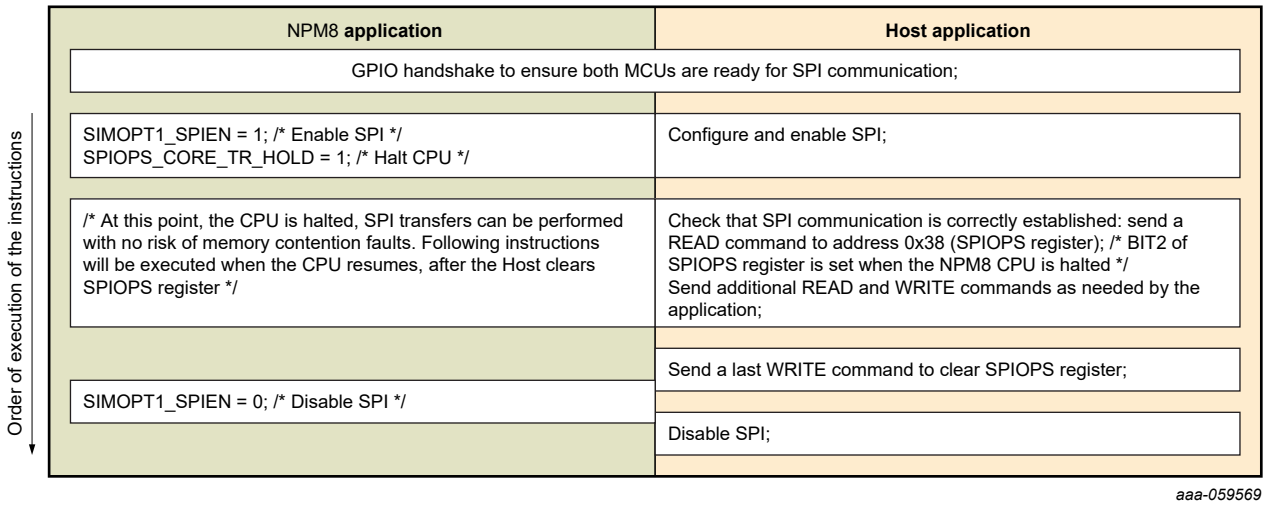

Figure 8. Flow of instructions in a typical implementation

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 NPM8 answers 0x2002 in the R0 frame, corresponding to a status value of 0b01000. As mentioned in SPI commands and error handling, a status value of 0b01000 included in the R0 frame does not correspond to a SPI error but indicates that R0 is the first response transferred by the NPM8 after reset.
- The response to command 0x00E1 is given by the NPM8 during the second transfer. The NPM8 answers 0x0011 in the R1 frame, corresponding to a status clear and data byte equal to 0x04. Such a response indicates that BIT2 (0x04) of SPIOPS register is set, meaning that the NPM8 CPU is halted, and SPI communication has been successfully established.

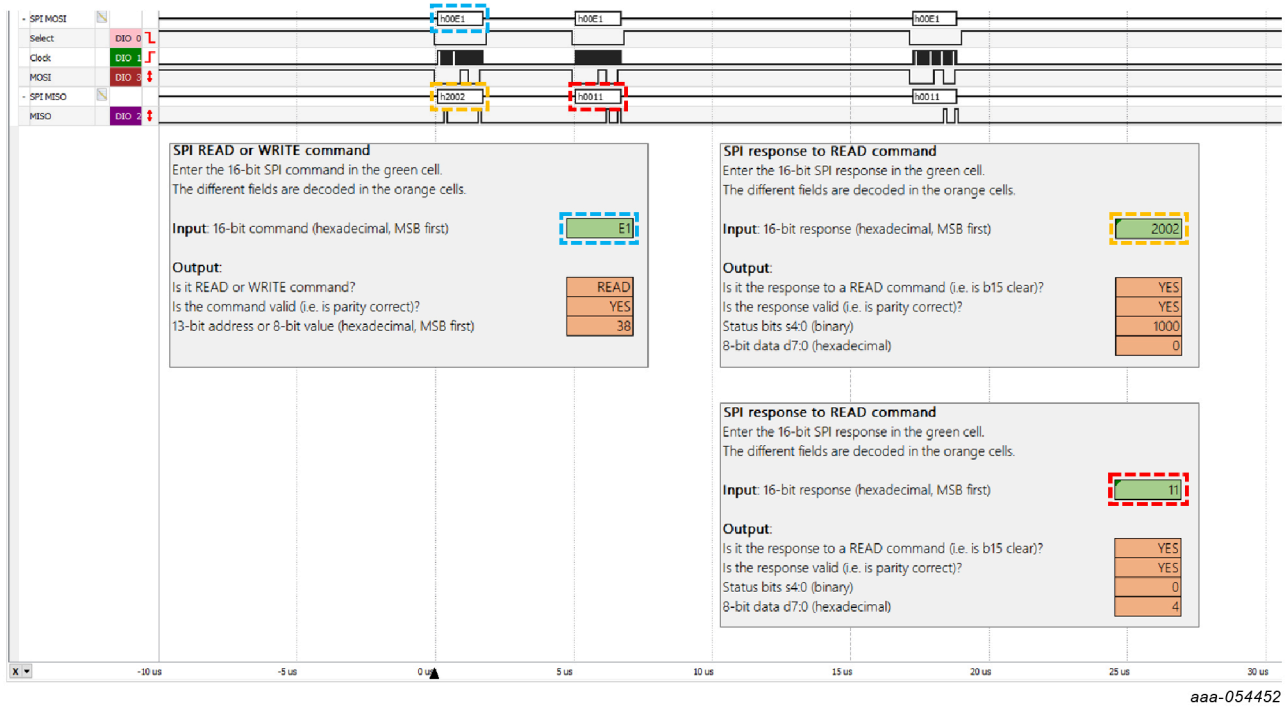

Figure 9. Start of the SPI sequence: Host reading SPIOPS register

Figure 10 shows the SPI signals during the first two transfers.

Note: *The CS_B signal goes back to high state after each 16-bit transfer.*

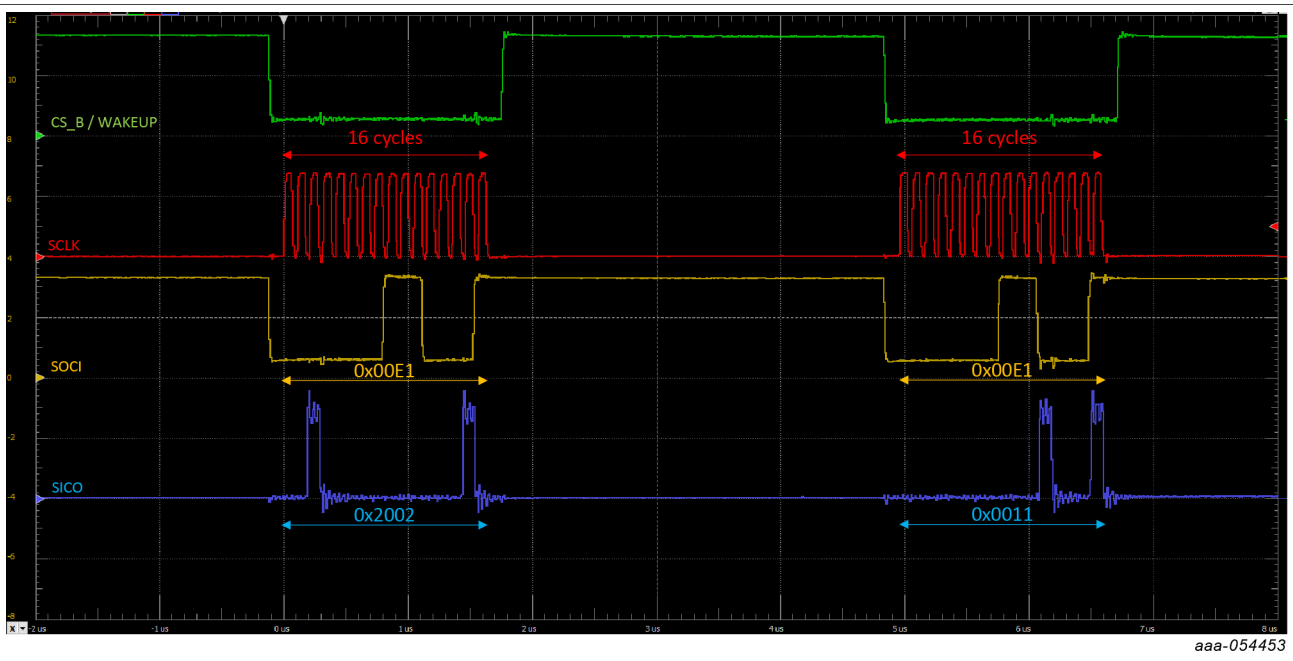

Figure 10. View of the SPI signals during the first two transfers

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 the SPIOPS register and let the NPM8 CPU resume.

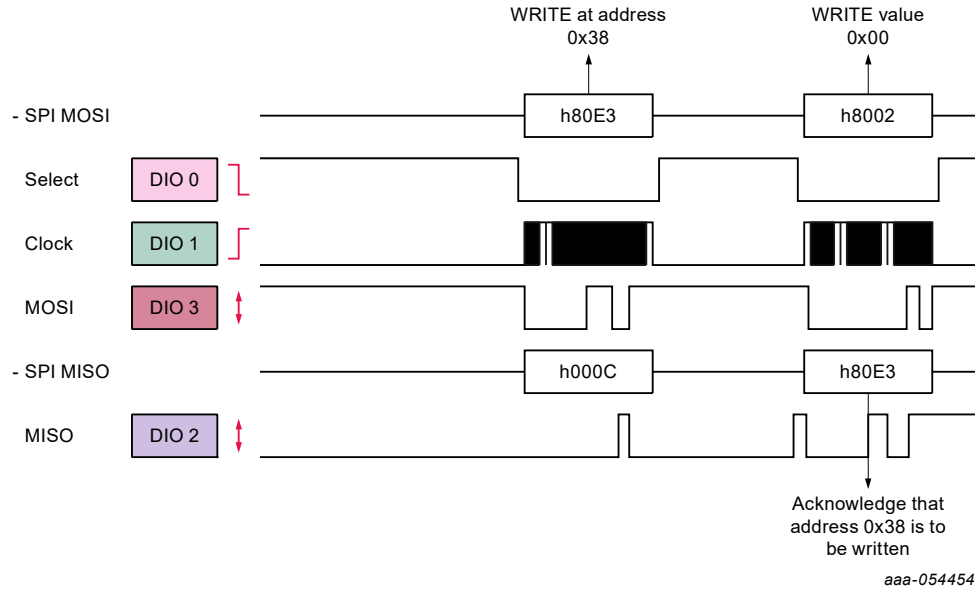

Figure 11. End of the SPI sequence: Host clearing SPIOPS register

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 NPM8 has executed the command clearing the SPIOPS register, the NPM8 CPU restarts and the NPM8 application then typically disables the SPI block. Any additional SPI commands transferred after the second transfer of the write command are not answered because the SPI block is disabled.

3 NPM8 as SPI server (host)

3.1 Implementation

The NPM8 device does not include a hardware SPI block that can be configured in Host mode. In order for the NPM8 to act as SPI host, the NPM8 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 an 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, the receive buffer, and the number of bytes of the transfer. The function returns when the transfer has been performed.
- The transfers are performed in Run mode. 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. The PTA4 pin can be configured as a GPIO output only by clearing the 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 a GPIO input.

Note: *By default after reset (including STOP1 exit), the PTA4 pin is configured as a BKGD pin with an internal pullup enabled. As a result, if the PTA4 pin is used as a SPI signal, care must be taken not to connect a pulldown resistor to the pin. Background Debug mode is entered when the PTA4 pin is driven to Low state during reset. Care must also be taken not to drive the 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 kbit/s. 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.

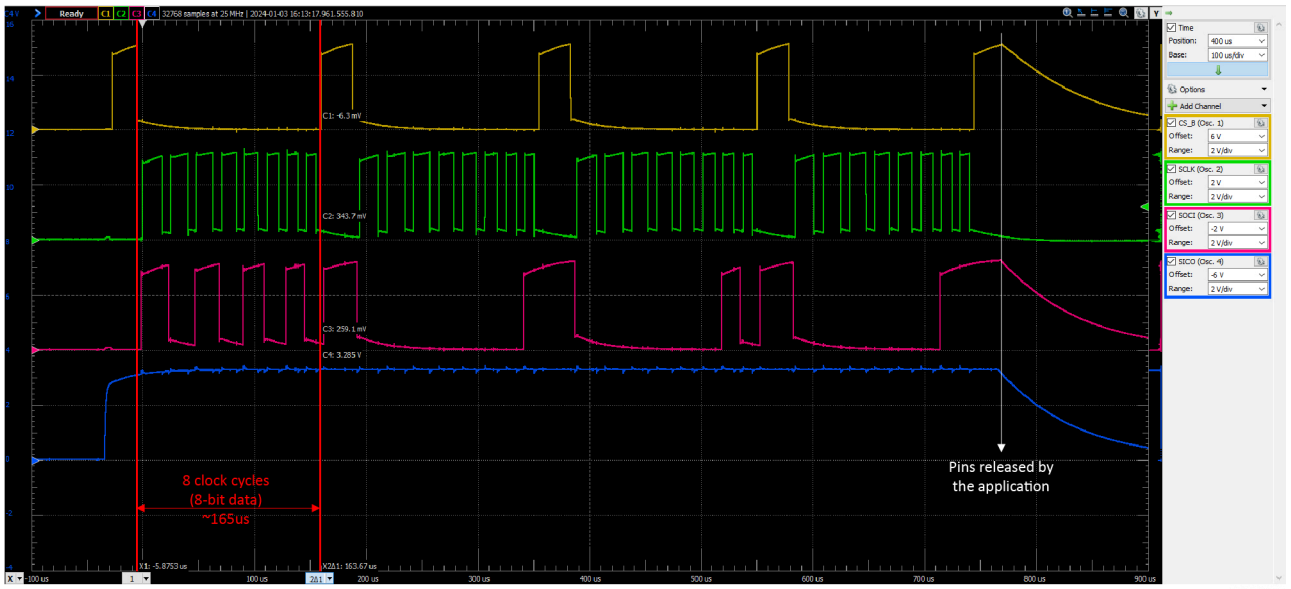

Figure 12. SPI signals during the transfer implemented in the demo project

4 NPM8 as I²C controller

4.1 Implementation

The NPM8 device does not include an I²C block. In order for the NPM8 to act as an I²C controller, the NPM8 application performs I²C bit-banging, using software drivers controlling the GPIOs in order to emulate an I²C block working in Controller mode.

The software drivers and an example of implementation are provided in the demo project provided as an 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 I²C. 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 I²C 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 I²C address, the transmit buffer, and the number of data bytes to transmit. The number of data bytes does not include the length of the I²C address.
  In the function, after transmission of the I²C 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 a value of 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 I²CAdd, UINT8 *pTxB, UINT8 DataLen) transfers data bytes on the SDA line. The function takes as parameter the I²C address, transmit buffer, and number of data bytes to transmit. The number of data bytes does not include the length of the I²C address.
  In the function, after transmission of the I²C 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 I²C protocol[3], 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 I²C address, receive buffer and number of data bytes to read.
  In the function, after transmission of the I²C 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 I²C address was correctly acknowledged and data bytes were read and stored. The function returns value 1 when the I²C 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 I²C 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. The PTA4 pin can be configured as a GPIO output only by clearing SIMOPT1_BKGDPE bit. As a result, the PTA4 can be used for SCL signal (with clock stretching disabled), but not for an 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 a BKGD pin with an internal pullup enabled. Background Debug mode is entered when the PTA4 pin is driven to Low state during reset. Care must be taken not to drive the 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 kbit/s. 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 NPM8. The acknowledges and data bytes received on the SDA line come from the sensor.

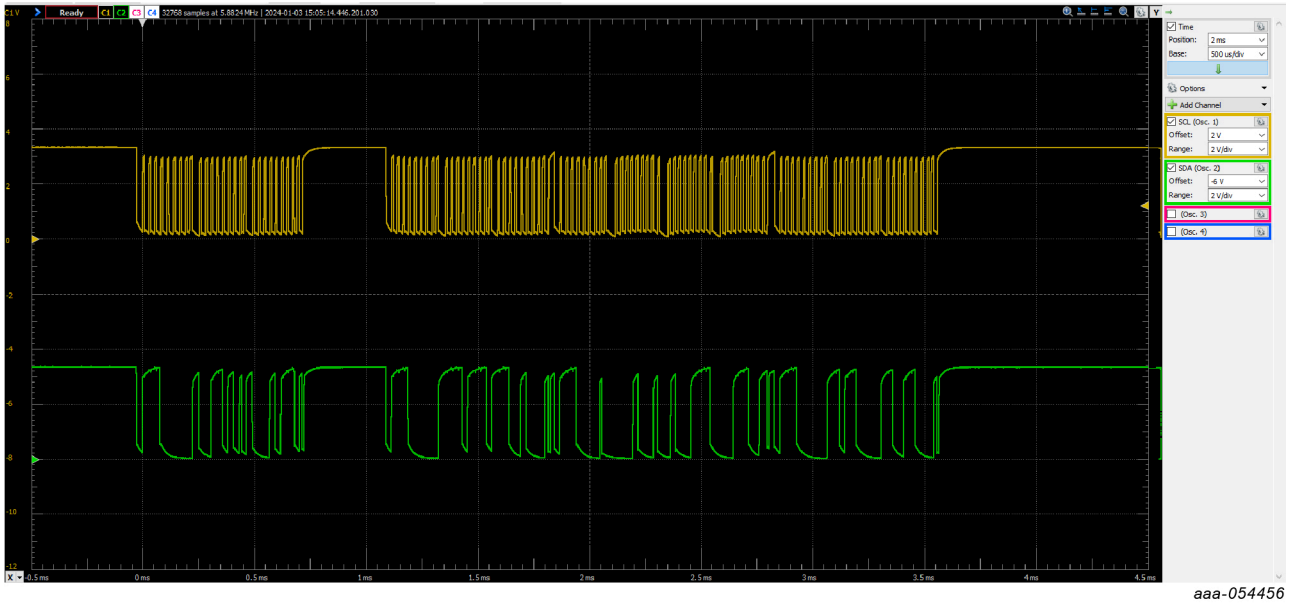

Figure 13. I^{2}C signals during the transfers implemented in the demo project

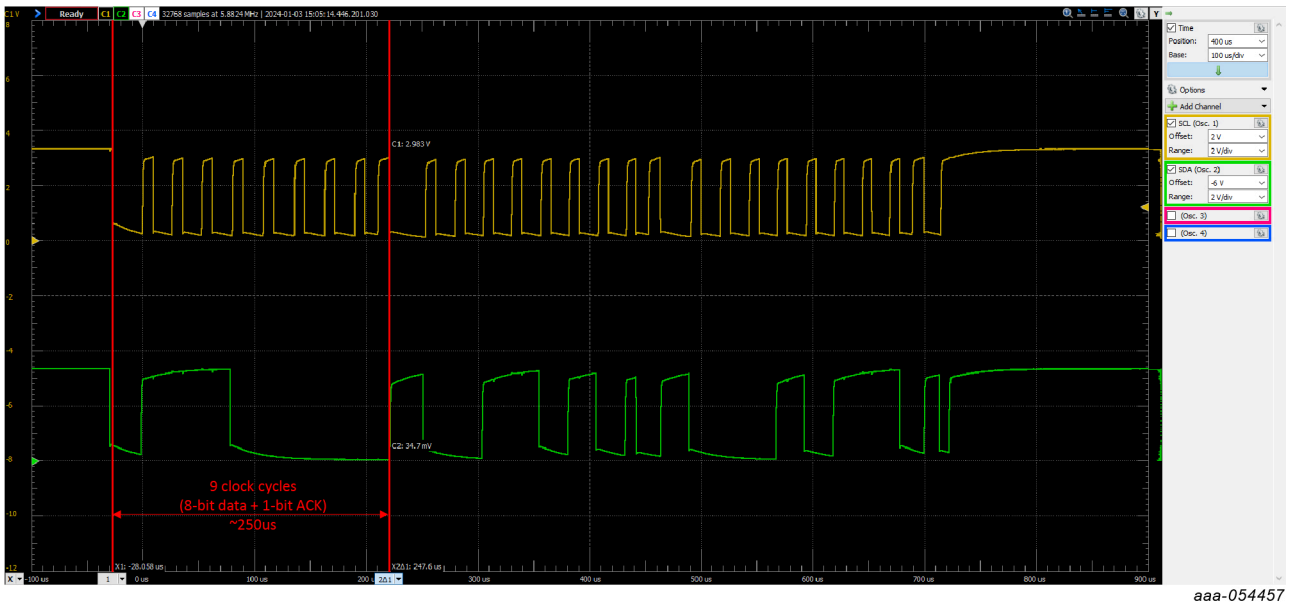

Figure 14. Zoom on the first I^{2}C transfer

5 References

[1] UM12295, NPM8 user manual

[2] NPM8KD4ST1, Industrial pressure monitor sensor

[3] UM10204, I^2C-bus specification and user manual

6 Revision history

Table 3. Revision history

Rev	Date	Description
AN14599 v.1	27 March 2025	Initial release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Tables

Figures

Contents