AN14170

SPI/DMA Implementations Using i.MX RT500

Rev. 1 — 25 January 2024 Application note

Document information

Information	Content
Keywords	i.MX RT500, i.MX RT600, SPI
Abstract	This application note provides details on how to replicate and solve the SPI limitation that can occur in case of high SPI DMA traffic use case.

1 RT500 introduction

The i.MX RT500 is a family of dual-core microcontrollers for embedded applications featuring an Arm Cortex-M33 CPU combined with a Cadence Xtensa Fusion F1 Audio Digital Signal Processor CPU. The Cortex-M33 includes two hardware coprocessors providing enhanced performance for an array of complex algorithms. The family offers a rich set of peripherals and very low power consumption. The device has up to 5 MB SRAM, two FlexSPIs (Octal/Quad SPI Interfaces) each with 32 KB cache, one with dynamic decryption, high-speed USB device/ host + PHY, 12-bit 1 MS/s ADC, Analog Comparator, Audio subsystems supporting up to 8 DMIC channels, 2.5D Vector GPU and LCD Controller with MIPI DSI PHY, 2 SDIO/eMMC; FlexIO; AES/SHA/Crypto M33 coprocessor and PUF key generation.

The i.MX RT500 provides as well 12 Flexcomm modules that can be configured for one of these protocols: USART, SPI, I2C, I2S.

Coupled with the DMA, the i.MX RT500 offers an efficient way to communicate with peripherals by offloading the CPU that improves latency and performance.

This Application Note discusses performance limitations with high-bandwidth scenarios and how to overcome these limitations.

2 SPI + DMA performance limitation

In a standard use of SPI + DMA, both the SPI Tx and Rx traffic are handled by the DMA to transfer data from/to the memory to/from peripherals. In some scenarios, many peripherals can be connected to the SPI bus creating high DMA traffic. In such cases, a bandwidth limitation can be reached, resulting in SPI data stalls. We noticed that sometimes the SPI data can reach the RT500 SPI interface but would not end up in the SRAM, resulting in missing bytes. It is explained by a DMA heavily loaded, conducting to an Rx FIFO overflow.

This test demonstrates this limitation, where Flexcomm 5 is used as SPI (SPI5) in mode 0 (CPOL=0, CPHAL=0). SPI5 MISO and MOSI are interconnected. A GPIO ERROR is triggered when an Rx FIFO overflow has occurred caused by the DMA not emptying the FIFO fast enough.

The DMA is configured to service two requests coming from SPI5 – the SPI Tx DMA channel (11) copies test pattern bytes from 0x01 to 0x3F located in the SRAM to the SPI 5 FIFOWR register 28 times; the SPI5 Rx DMA channel (10) transfers data from the SPI5 FIFORD to the GPIOs D[3:0]. D[3:0] represents the data copied by the DMA to GPIOs instead of the SRAM, it is defined by the macro *TEST_DESTINATION = DESTINATION_PORT*. To reproduce the limitation, the macro *TEST_TO_RUN* must be defined to *ISSUE*.

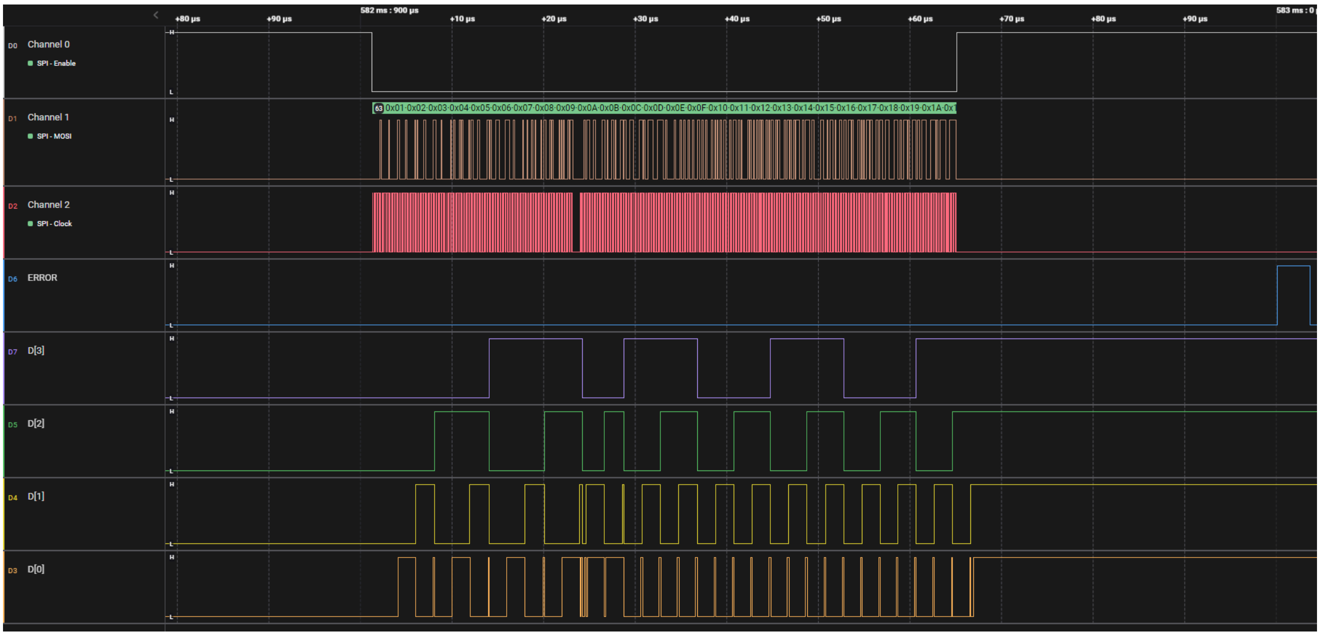

Figure 1. Capture of the full SPI transfers containing the error

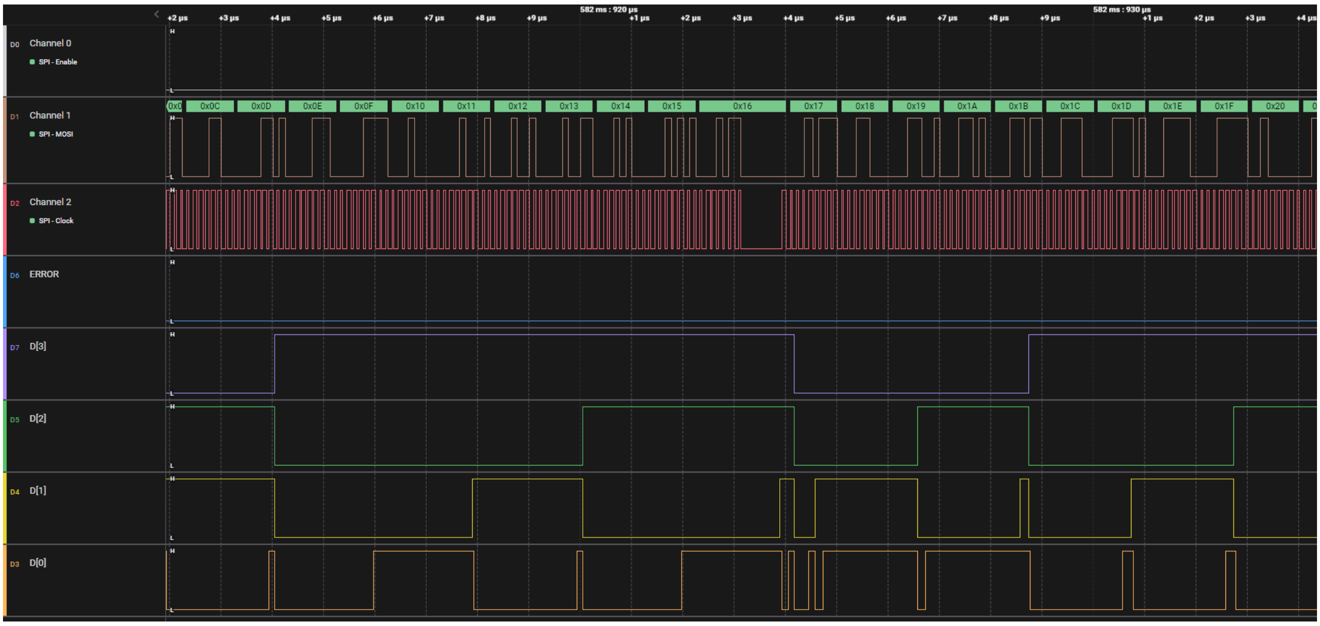

Figure 2. Capture of SPIs transfers and the values transmitted

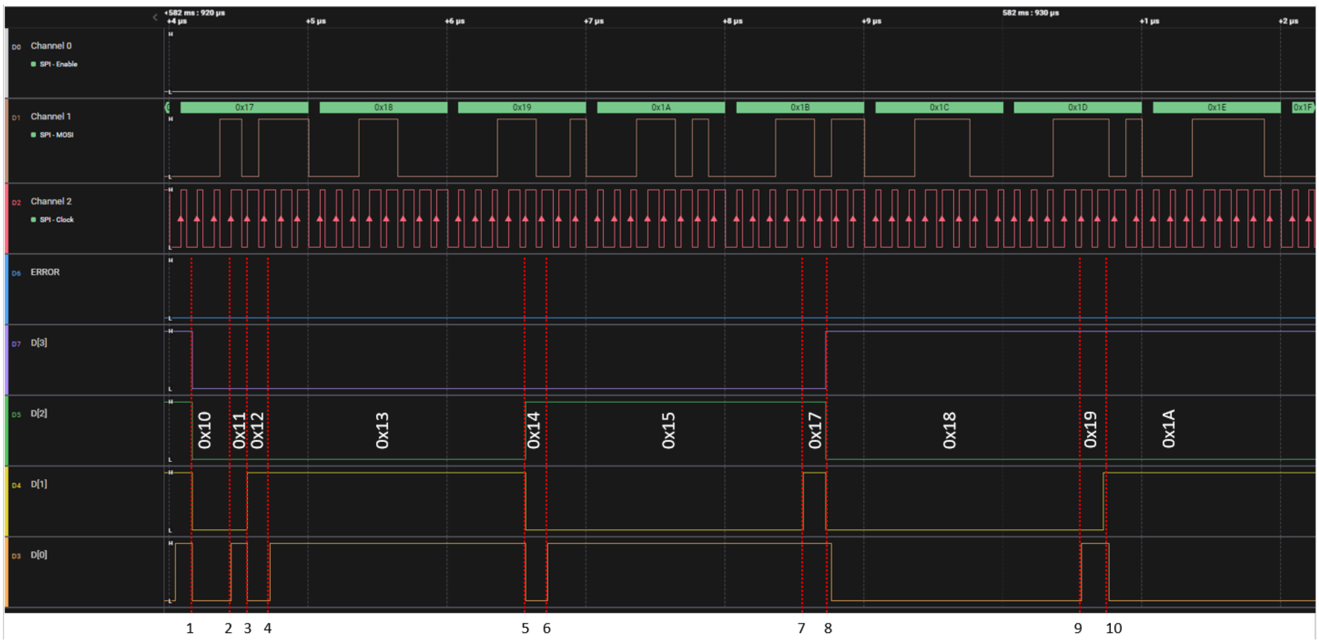

Figure 3. Capture of the SPIs values transmitted and the missing value

Table 1. Data received on the SPI bus

Transition	D[3]	D[2]	D[1]	D[0]	D[3.0]
1-2	0	0	0	0	0x10
2-3	0	0	0	1	0x11
3-4	0	0	1	0	0x12
4-5	0	0	1	1	0x13
5-6	0	1	0	0	0x14
6-7	0	1	0	1	0x15
7-8	0	1	1	1	0x17
8-9	1	0	0	0	0x18
9-10	1	0	0	1	0x19
10	1	0	1	0	0x1A

The data are received consecutively, until data 0x16 that is missing. However, the first capture shows that the data byte 0x16 is correctly received by the SPI Rx. ERROR GPIO is triggered at the end of the SPI communication.

Here is an example of a working communication:

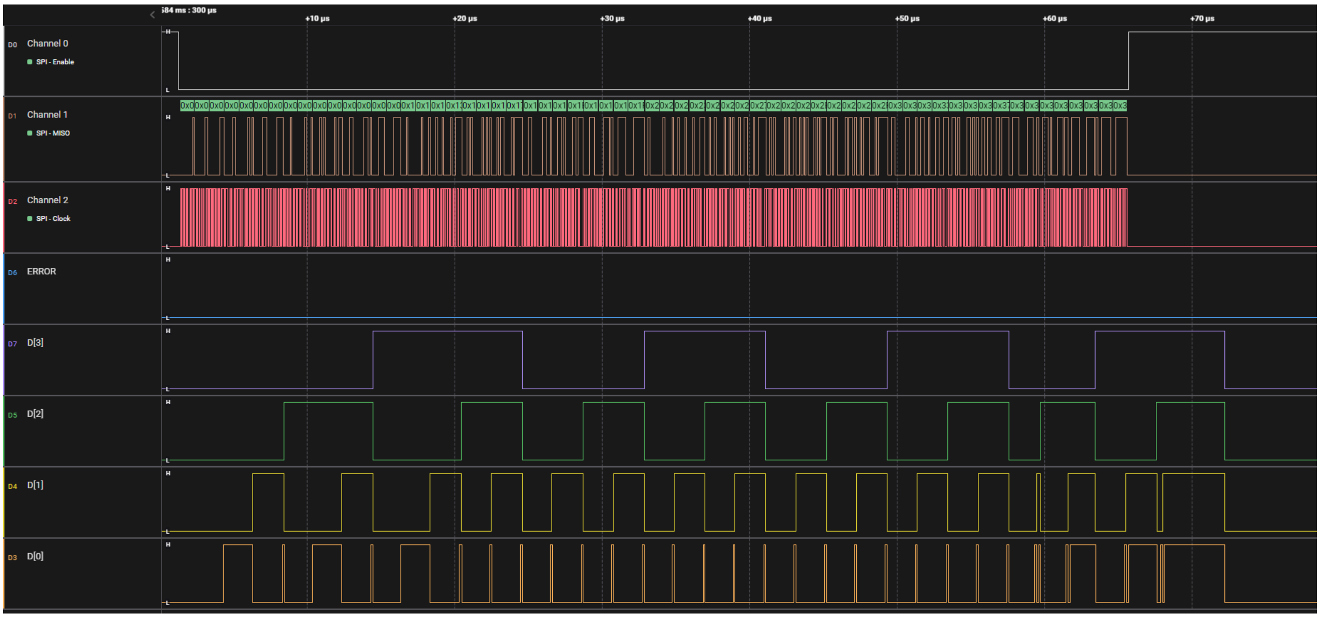

Figure 4. Capture of the full SPI transfers without errors

3 Workarounds

To overcome this limitation, different mechanisms can be developed. We will explore two possible SPI + DMA configurations, with their pros and cons and how they work.

3.1 Implementation 1

The idea of this implementation is to get a "back to back" transfer pattern and use 100 % of the SPI bus with minimal additional resources.

The first implementation is a fully hardware-based approach using the DMA trigger output mechanism to drive additional DMA channels, called channel chaining. Channel chaining is a feature that allows completion of a DMA transfer on channel x to trigger a DMA transfer on channel y.

Here one trigger and 2 DMA channels (Rx DMA and Tx DMA) are used, where the falling edge triggers both channels.

The SPI5 Rx DMA channel (10) has peripheral requests enabled, uses the falling edge trigger and burst transfer with burst size of 4 word to perform a single transfer from the SPI RxFIFO to the buffer array in SRAM. The first transfer is triggered by software, then is hardware triggered for the remaining transfers. This DMA channel input and output triggers are both routed to the DMAC0_TRIGOUT_A. SPI5 Rx DMA channel has a priority higher than the SPI5 DMA Tx channel. SPI5 Rx DMA may uses two linked descriptors, in the case where the DMA transfers length is m (1/2/3) + 4 x n, otherwise only one descriptor is necessary.

The SPI5 Tx DMA channel (11) performs memory-to-memory transfers from the Tx buffer array in the SRAM to the SPI5 TxFIFO. It has peripheral requests disabled, and uses the falling edge trigger and burst transfer with burst size of 4 words to perform a single transfer. The input trigger of the channel comes from the SPI5 DMA Rx channel trigger output, and uses two linked descriptors. The first descriptor performs several 8-bit transfers to the TxFIFO, always a multiple of 4, and the second (the tail) performs no less than a single and no more than 4 32-bit write to the TxFIFO generating the last SPI exchange, depending on the transfer length. For instance, if the total number of SPI bytes that must be sent is divisible by 4 then the tail descriptor sends 4 bytes + SPI control content, otherwise this descriptor sends the number of bytes + control/words that equals the overall number of bytes modulo 4.

Having the parameter m of the transfer length other than 0 helps an artificial "misalignment" to be implemented between the SPI Rx and Tx DMA channels in a way that when the Rx DMA channel has officially completed a burst and is about to generate a new one, the FIFOWR still has entries in it ready to get into the SPI serial shift register (written there by the matching Tx DMA burst); these residual entries in the FIFOWR help bridge the gap that causes the SPI bus to be idle until the SPI Rx DMA channel generates a new trigger and causes the next Tx DMA burst to write a new set of 4 data.

For instance, let us consider a SPI transfer of 16 bytes, and parameter m = 1.

For Tx: 16 / 4 = 0, so the Tx main descriptor transfers 12 bytes, and the tail descriptor transfers 4 bytes.

For Rx: (16 - 1) mod 4 = 3, so Rx main descriptor transfers 13 bytes, and the tail descriptor transfers 3 bytes.

The overall mechanism is as follows:

- SPI5 Rx DMA channel gets setup first so that SPI Rx stream can be handled. By using the SW trigger this channel gets the initial trigger and awaits for a request to come from the peripheral.
- SPI5 Tx DMA channel is set next and as it is configured to perform memory-to-memory transfers, the moment the SW trigger is set it performs a burst of 4 transfers into the TxFIFO. When completed, the SPI5 Tx DMA channel trigger is cleared as bursts are used and is awaiting a new trigger.
- When the SPI5 receives bytes it generates DMA requests; as the DMA Rx channel is already triggered by SW, the DMA Rx channel generates a burst of 4 transfers and copies received data from the SPI RxFIFO into the Rx array in the SRAM. A DMA channel burst clears the trigger and this falling edge is routed via the output trigger signal both to the DMA Rx channel input trigger and the DMA Tx channel input trigger.
- As both DMA Rx and Tx channels are configured to be trigger falling edge sensitive both get triggered again, this time by the HW mechanism.
- This trigger causes the DMA Tx channel to send the second piece of data over the SPI5 while the same edge primes the DMA Rx channel that is now ready to pick-up data the moment it becomes available in the SPI5 RxFIFO.
- This mechanism is repeated until the end of the transfer, where the second Tx descriptor is loaded and the second Rx descriptor may be loaded depending on the transfer size. The Tx descriptor performs one to four 32-bits writes to the TxFIFO to complete the SPI transfers and deselect the SSEL line. The Rx descriptor, if loaded, performs the necessary 8-bit reads to complete the SPI transfers.

3.1.1 Code implementation

```
#define DESC_ADDON              2
#define RT500_DMA_CH_PERI_REQ   37
#define DEMO_DMA_CH_PERI_REQ    (RT500_DMA_CH_PERI_REQ + DESC_ADDON)
enum demo_dma_descriptor_enum
{
    dma_desc_spi_rx_main    = 10,
    dma_desc_spi_tx_main    = 11,

    dma_desc_spi_tx_add1    = RT500_DMA_CH_PERI_REQ,
    dma_desc_spi_rx_add1    = dma_desc_spi_tx_add1+1
};
```

Figure 5. Definition of the SPI channels used in DMA requests

The following code:

- The DMA channel dma_desc_spi_rx_main request is routed via the OTRIG_SEL[1] to the DMAC0_TRIGOUT_A.
- The DMA channel dma_desc_spi_rx_main input trigger is routed via ITRIG_SEL[12] to the DMAC0_TRIGOUT_A.

The intent of this configuration is to trigger the DMA transfer of the channel based on a falling edge on DMAC0_TRIGOUT_A, and for this DMA channel to trigger on a falling-edge DMA channel dma_desc_spi_tx_main when the SPI5 Rx DMA request is gone.

```
// dma_desc_spi_rx_main: SPI Rx request, drive DMA out A, input trigger = DMA out A
INPUTMUX->DMAC0_REQ_ENA0_SET = 1<<dma_desc_spi_rx_main;
INPUTMUX->DMAC0_OTRIG_SEL[0] = dma_desc_spi_rx_main;    // drive trigger out A
INPUTMUX->DMAC0_ITRIG_ENA0_SET = 1<<dma_desc_spi_rx_main;
INPUTMUX->DMAC0_ITRIG_SEL[dma_desc_spi_rx_main] = 14;   // input trigger = trigger out A
```

Figure 6. SPI Rx DMA channel routing and trigger configuration

The following code:

- The DMA channel dma_desc_spi_tx_main request is routed via the OTRIG_SEL[1] to the DMAC0_TRIGOUT_A.
- The DMA channel dma_desc_spi_tx_main input trigger is routed via ITRIG_SEL[12] to the DMAC0_TRIGOUT_A.

```
// dma_desc_spi_tx_main: SPI tx request, input trigger = DMA out A
INPUTMUX->DMAC0_REQ_ENA0_SET = 1<<dma_desc_spi_tx_main;
INPUTMUX->DMAC0_ITRIG_ENA0_SET = 1<<dma_desc_spi_tx_main;
INPUTMUX->DMAC0_ITRIG_SEL[dma_desc_spi_tx_main] = 14;   // input trigger = trigger out A
```

Figure 7. SPI Tx DMA channel routing and trigger configuration

The following code configures the Rx tail transfer created depending on the transfer size. Perform a total of rx_tail_len 8-bit transfers with the destination address incremented by 1 width of the transfer. The trigger is cleared when this descriptor is exhausted.

rx_tail_len size depends on the transfer size and the 'm' parameters creating the Rx and Tx artificial "misalignment".

dma_desc_spi_rx_add1 descriptor copies data from the SPI RxFIFO to the holding variable in the SRAM "spi_rx_array_8bit".

```
//------------------------
if (rx_tail_len != 0)
{
    // dma_desc_spi_rx_add1
    demo_dma_descriptor[dma_desc_spi_rx_add1].xfercfg   =
        1<<0    |   // valid configuration
        0<<1    |   // link/reload disabled
        0<<2    |   // no sw trigger
        1<<3    |   // clear trigger at the end
        0<<4    |   // no int A at the end
        0<<5    |   // no int B at the end
        0<<8    |   // 8-bit transfers
        0<<12   |   // src: +0
        1<<14   |   // dst: +1
        (rx_tail_len-1)<<16;    // transfer count...;
    demo_dma_descriptor[dma_desc_spi_rx_add1].src_addr  = (uint32_t)&TEST_SPI->FIFORD;
    demo_dma_descriptor[dma_desc_spi_rx_add1].des_addr  = (uint32_t)&spi_rx_array_8bit[dma_txrx_data_len-1];
    demo_dma_descriptor[dma_desc_spi_rx_add1].link      = 0;

    dma_desc_spi_rx_main_xfercfg_temp |= 1<<1;  //prime the main rx descriptor link/reload feature
}
```

Figure 8. Rx tail transfer configuration

The following code defines the dma_desc_spi_rx_main descriptor that copies data from the SPIS RxFIFO to the holding variable in the SRAM spi_rx_array_8bit. Depending on the transfer size, reload the tail descriptor dma_desc_spi_rx_add1.

```
// dma_desc_spi_rx_main
demo_dma_descriptor[dma_desc_spi_rx_main].xfercfg   = 0;
demo_dma_descriptor[dma_desc_spi_rx_main].src_addr  = (uint32_t)&TEST_SPI->FIFORD;
demo_dma_descriptor[dma_desc_spi_rx_main].des_addr  = (uint32_t)&spi_rx_array_8bit[rx_body_len-1];
if (rx_tail_len == 0)
{
    demo_dma_descriptor[dma_desc_spi_rx_main].link = 0;
}
else
{
    demo_dma_descriptor[dma_desc_spi_rx_main].link = (uint32_t)&demo_dma_descriptor[dma_desc_spi_rx_add1];
}
```

Figure 9. dma_desc_spi_rx_main descriptor definition

The following code configures the dma_desc_spi_rx_main DMA that is HW triggered on the falling edge, with the burst transfer enabled of size 4.

It also configures the dma_desc_spi_rx_main transfer that performs a total of rx_body_len 8-bit transfers, with the destination address incremented by 1 width of the transfer. The trigger is cleared when this descriptor is exhausted.

Finally, the channel control structure is reloaded when the current descriptor is exhausted (from dma_desc_spi_rx_main_xfercfg_temp).

```
DMA0->CHANNEL[dma_desc_spi_rx_main].CFG =
    1<<0    |   // peripheral req enable
    1<<1    |   // hw trigger enabled
    0<<4    |   // falling...
    0<<5    |   // ... edge
    1<<6    |   // burst transfer(s)
    2<<8    |   // burst size = 2^2 = 4 transfer
    DMACH_RX_PRIO<<16;      // priority =...

dma_desc_spi_rx_main_xfercfg_temp |=
    0<<0    |   // not valid configuration yet!!!
    0<<1    |   // link/reload disabled - adjusted by add1 if present
    0<<2    |   // no sw trigger
    1<<3    |   // clear trigger at the end
    0<<4    |   // no int A at the end
    0<<5    |   // no int B at the end
    0<<8    |   // 8-bit transfers
    0<<12   |   // src: +0
    1<<14   |   // dst: +1
    (rx_body_len-1)<<16;    // transfer count...;
DMA0->CHANNEL[dma_desc_spi_rx_main].XFERCFG = dma_desc_spi_rx_main_xfercfg_temp;
// dma_desc_spi_rx_* end
//----------------------
```

Figure 10. dma_desc_spi_rx_main DMA and transfer configuration

The following code configures the Tx tail transfer. It performs a total of tx_tail_len 32-bit transfers with the source address incremented by 1 width of the transfer. The trigger is cleared when this descriptor is exhausted.

rx_tail_len size depends on the transfer size.

It also defines the dma_desc_spi_tx_add1 descriptor that copies data from the buffer array in the SRAM spi_tx_array_tail to the TxFIFO.

```
//
// dma_desc_spi_tx_* begin
//========================
// dma_desc_spi_tx_add1
demo_dma_descriptor[dma_desc_spi_tx_add1].xfercfg   =
    1<<0    |   // valid configuration
    0<<1    |   // link/reload disabled
    0<<2    |   // no sw trigger
    1<<3    |   // clear trigger at the end
    0<<4    |   // no int A at the end
    0<<5    |   // no int B at the end
    2<<8    |   // 32-bit transfers
    1<<12   |   // src: +1
    0<<14   |   // dst: +0
    (tx_tail_len-1)<<16;    // transfer count...;
demo_dma_descriptor[dma_desc_spi_tx_add1].src_addr  = (uint32_t)&spi_tx_array_tail[tx_tail_len-1];
demo_dma_descriptor[dma_desc_spi_tx_add1].des_addr  = (uint32_t)&TEST_SPI->FIFOWR;
demo_dma_descriptor[dma_desc_spi_tx_add1].link      = 0;
```

Figure 11. Tx tail transfer configuration and dma_desc_spi_tx_add1 descriptor definition

The following code defines the dma_desc_spi_tx_main descriptor that copies data from the buffer array in the SRAM spi_tx_array_8bit to the TxFIFO and reloads the tail descriptor dma_desc_spi_tx_add1.

It also defines the dma_desc_spi_tx_main DMA configuration that is HW-triggered on the falling edge with the burst transfer enabled of size 4.

Finally, it defines the Tx transfer configuration with the channel's control structure reloaded when the current descriptor is exhausted, performs a total of tx_body_len 8-bit transfers with the source address incremented by 1 width of the transfer. The trigger is cleared when this descriptor is exhausted.

```
//dma_desc_spi_tx_main
demo_dma_descriptor[dma_desc_spi_tx_main].xfercfg   = 0;
demo_dma_descriptor[dma_desc_spi_tx_main].src_addr  = (uint32_t)&spi_tx_array_8bit[tx_body_len-1];
demo_dma_descriptor[dma_desc_spi_tx_main].des_addr  = (uint32_t)&TEST_SPI->FIFOWR;
demo_dma_descriptor[dma_desc_spi_tx_main].link      = (uint32_t)&demo_dma_descriptor[dma_desc_spi_tx_add1];

DMA0->CHANNEL[dma_desc_spi_tx_main].CFG =
    0<<0    |   // peripheral req disable
    1<<1    |   // hw trigger enabled
    0<<4    |   // falling...
    0<<5    |   // ... edge
    1<<6    |   // burst transfer(s)
    2<<8    |   // burst size = 2^2 = 4 transfer
    DMACH_TX_PRIO<<16;  // priority =...

DMA0->CHANNEL[dma_desc_spi_tx_main].XFERCFG =
    0<<0    |   // not valid configuration yet
    1<<1    |   // link/reload enabled
    0<<2    |   // no sw trigger
    1<<3    |   // clear trigger at the end
    0<<4    |   // no int A at the end
    0<<5    |   // no int B at the end
    0<<8    |   // 8-bit transfers
    1<<12   |   // src: +1
    0<<14   |   // dst: +0
    (tx_body_len-1)<<16;    // transfer count...;
```

Figure 12. dma_desc_spi_tx_main descriptor and DMA transfer configuration definition

The following code prepares spi_tx_array_tail written by dma_desc_spi_tx_add1 to the TxFIFO. spi_tx_array_tail is the buffer array in the SRAM spi_tx_array_8bit combined with the SPI command spi_fifowr_ctrl to deselect SSEL via the SPI FIFOWR register.

```
// prepare the tail array, add SSEL de-select for the last entry
for (i_loc = 0; i_loc != tx_tail_len; i_loc++)
{
    spi_tx_array_tail[i_loc] = spi_fifowr_ctrl | ((uint32_t)spi_tx_array_8bit[tx_body_len+i_loc]);
}
spi_tx_array_tail[tx_tail_len-1] |= 1<<20;
// dma_desc_spi_tx_* end
//----------------------
```

Figure 13. spi_tx_array_tail preparation

The following code configures the dma_desc_spi_rx_main transfer, which is the same as previously configured, but with the software trigger enabled. Finally, it enables the DMA channel 10.

It configures the dma_desc_spi_tx_main transfer that is software-triggered with the channel's control structure reloaded when the current descriptor is exhausted, performs a total of tx_body_len 8-bit transfers with the source address incremented by 1 width of the transfer. The trigger is cleared when this descriptor is exhausted. Finally, it enables DMA channel 11.

```
DMA0->CHANNEL[dma_desc_spi_rx_main].XFERCFG = dma_desc_spi_rx_main_xfercfg_temp |
    1<<0    |   // valid configuration
    1<<2;       // sw trigger
DMA0->COMMON[0].ENABLESET = 1<<dma_desc_spi_rx_main;

DMA0->CHANNEL[dma_desc_spi_tx_main].XFERCFG =
    1<<0    |   // valid configuration
    1<<1    |   // link/reload enabled
    1<<2    |   // sw trigger
    1<<3    |   // clear trigger at the end
    0<<4    |   // no int A at the end
    0<<5    |   // no int B at the end
    0<<8    |   // 8-bit transfers
    1<<12   |   // src: +1
    0<<14   |   // dst: +0
    (tx_body_len-1)<<16;    // transfer count...;
DMA0->COMMON[0].ENABLESET = 1<<dma_desc_spi_tx_main;
```

Figure 14. dma_desc_spi_rx_main and dma_desc_spi_tx_main transfers configuration

Here is an example of the implementation with different SPI frequencies.

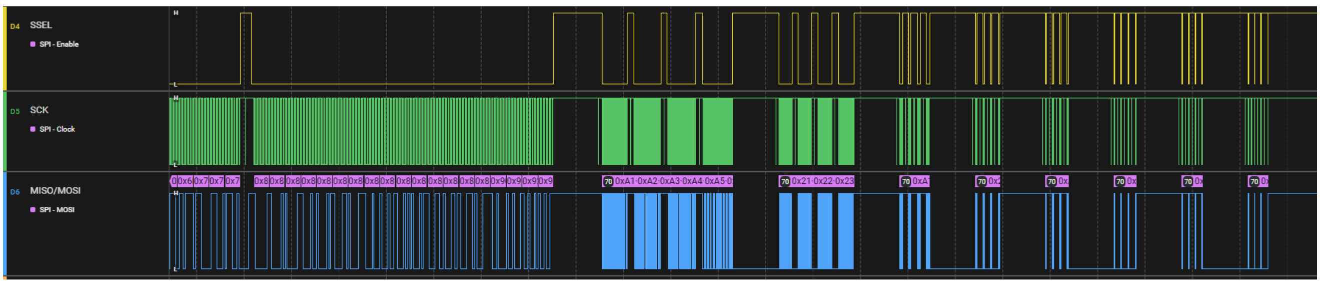

Figure 15. Capture of the workaround 1 tested with different SPI frequencies

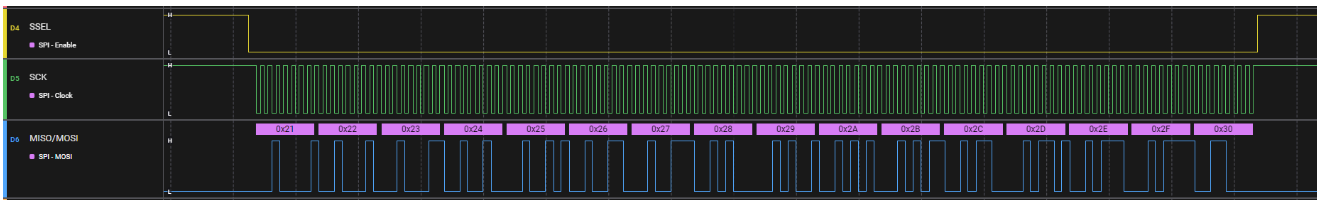

Figure 16. Capture of SPI transfers for a frequency

This implementation requires only 1 DMA output trigger as an additional resource. This workaround gives the best SPI bus performance with 100% of utilization by leveraging the back-to-back transfer pattern. Regarding the performance, there is no regression compared to the initial implementation. Therefore, the drawback of this implementation is in terms of resources, where 1 DMA output trigger is required.

3.2 Implementation 2

The idea is for the SPI TxFIFO empty to generate an interrupt and knowing the current level at the RxFIFO to write as many entries as possible/available to the TxFIFO. Only one DMA channel is used to transfer data from the RxFIFO to the SRAM. The SPI5 interrupt service routine is used to fill the SPI TxFIFO, when the TxFIFO is empty. This DMA channel input and output triggers are both routed to the DMAC0_TRIGOUT_A, as same as the SPI5 DMA Rx channel.

Here is an example showing the SPI Tx traffic handled by the SPI interrupts and the Rx traffic by the DMA.

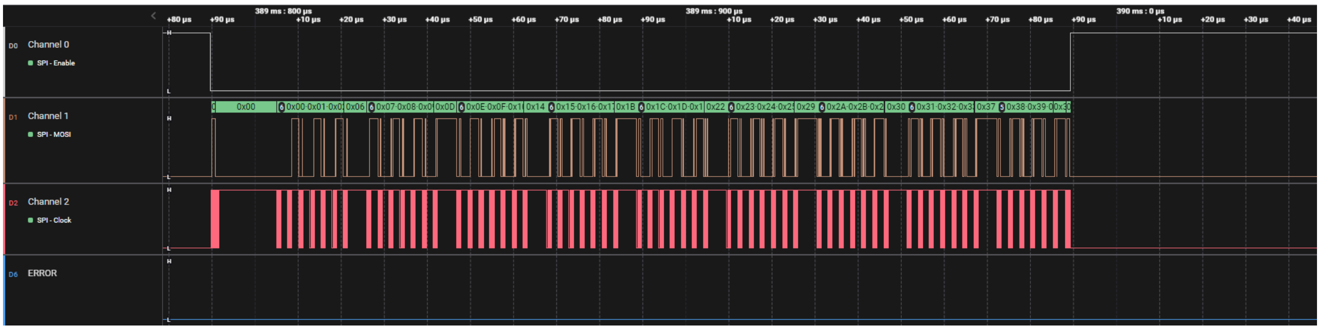

Figure 17. Capture of the workaround 2 implementation

3.2.1 Code implementation

The overall mechanism is as follows:

- SPI5 sends data directly by software writing to the SPI5 TxFIFO.
- DMA for SPI5 Rx (channel 10, that is dma_desc_spi_rx_main) is configured. When SPI5 Rx receives the data, it generates a DMA Rx request and dma_desc_fc5_rx_main copies this data from SPI5 RxFIFO into the buffer array in SRAM. The dma_desc_fc5_rx_main descriptor is programmed to perform several transfers corresponding to the number of free space in the RxFIFO. Therefore, it must increment the destination address of 1 width of the transfer. The dma_desc_spi_rx_main descriptor is SW-triggered.
- SPI5 is configured to trigger an interrupt when the SPI5 TxFIFO is empty. Each time the interrupt is triggered, the RxFIFO is checked to retrieve the remaining free space. If the RxFIFO is not full, there must be as many as possible entries written in the TxFIFO.

```
// the transmitter generates interrupts when the TxFIFO is empty
SPI5->FIFOTRIG = 0<<8 | 1<<0;   // enable TxFIFO empty trigger
SPI5->FIFOINTENSET = 1<<2;      // enable TxLVL interrupt
```

Figure 18. SPI configured to trigger an interrupt when the SPI TxFIFO is empty

```
uint32_t spi_rx_count_received_loc, spi_rx_count_left_loc;
uint32_t spi_rx_descriptor_index_loc, spi_rx_descriptor_count_loc;

GPIO->SET[TEST_PORT] = 1<<TEST_PIN;

// disable DMA ch, disable SPI Rx DMA request
DMA0->COMMON[0].ENABLECLR = 1<<dma_desc_spi_rx_main;
SPI5->FIFOCFG =
    SPI_FIFOCFG_ENABLETX(1) |   // enable TxFIFO
    SPI_FIFOCFG_ENABLERX(1) |   // enable RxFIFO
    SPI_FIFOCFG_DMARX(0)    |   // disable Rx DMA
    SPI_FIFOCFG_EMPTYTX(0)  |   // release Tx FIFO reset
    SPI_FIFOCFG_EMPTYRX(0);     // release Rx FIFO reset

spi_fifowr_preset((uint32_t)&SPI5->FIFOWR, tx_spi_first_control);
```

Figure 19. SPI FIFO configuration

The first two bytes are received (specifying the length/transfer count) and this information is collected with the SPI in the polling mode.

```
// send two byte command
SPI5->FIFOWR =
    (8-1)<<24 | // byte0
        0<<23 | // TXIGNORE
        0<<22 | // RXIGNORE
        1<<21 | // EOF
        0<<20 | // EOT
        1<<19 | // TXSSEL3
        1<<18 | // TXSSEL2
        1<<17 | // TXSSEL1
        0<<16 | // TXSSEL0
     ((tx_len>>0) & 0xFF)<<0;   // len, byte0

SPI5->FIFOWR =
    (8-1)<<24 | // byte0
        0<<23 | // TXIGNORE
        0<<22 | // RXIGNORE
        1<<21 | // EOF
        0<<20 | // EOT
        1<<19 | // TXSSEL3
        1<<18 | // TXSSEL2
        1<<17 | // TXSSEL1
        0<<16 | // TXSSEL0
     ((tx_len>>8) & 0xFF)<<0;   // len, byte1
```

Figure 20. Control bytes sent to SPI FIFO

The following code copies the SPI Rx DMA data from the SPI Rx FIFO and links itself to the next descriptor located in the descriptors array with the same properties.

```
for (i_loc = 0; i_loc != GP_SPI_DESCRIPTOR_COUNT; i_loc++)
{
    // read data from the FIFORD
    demo_gp_spi_descriptor[i_loc].src_addr = (uint32_t)&SPI5->FIFORD;

    // prepare links for the list
    if (i_loc != (GP_SPI_DESCRIPTOR_COUNT-1))
    {
        demo_gp_spi_descriptor[i_loc].link = (uint32_t)&demo_gp_spi_descriptor[i_loc+1];
    }
}
```

Figure 21. Data copied from the SPI Rx FIFO

```
// prepare DMA configuration/descriptors
spi_rx_count_received_loc = 0;
spi_rx_count_left_loc = rx_len;
spi_rx_descriptor_index_loc = 0;
```

Figure 22. DMA configuration and descriptors definitions

The following code stores the SPI Rx DMA data into the buffer array in the SRAM pnt_rx_array. The DMA transfer configuration enables the channel 's control structure to reload when the current descriptor is exhausted.

The DMA software is triggered and is configured to perform a total of spi_rx_descriptor_count_loc transfers with the destination address incremented by 1 width of the transfer.

```
do
{
    if (spi_rx_count_left_loc > 1024)
    {
        spi_rx_descriptor_count_loc = 1024;
    }
    else
    {
        spi_rx_descriptor_count_loc = spi_rx_count_left_loc;
    }

    spi_rx_count_received_loc += spi_rx_descriptor_count_loc;
    spi_rx_count_left_loc     -= spi_rx_descriptor_count_loc;


    demo_gp_spi_descriptor[spi_rx_descriptor_index_loc].des_addr =
        (uint32_t)&pnt_rx_array[spi_rx_count_received_loc-1];

    demo_gp_spi_descriptor[spi_rx_descriptor_index_loc].xfercfg =
        1<<0    |   // valid configuration
        1<<1    |   // link/reload
        1<<2    |   // sw trigger
        0<<3    |   // do not clear trigger at the end
        0<<4    |   // no int A at the end
        0<<5    |   // no int B at the end
        0<<8    |   // 8-bit transfers
        0<<12   |   // src: +0
        1<<14   |   // dst: +1
        (spi_rx_descriptor_count_loc-1)<<16;    //transfer count...

    // if not done, add a descriptor
    if (spi_rx_count_left_loc != 0)
    {
        spi_rx_descriptor_index_loc++;
    }
}
while(spi_rx_count_left_loc != 0);
```

Figure 23. DMA transfer configuration definition

The following code updates the last DMA transfer configuration to clear the trigger when the descriptor is exhausted, disable the reload of the descriptor and disable the Interrupt flag A. Finally, it starts SPI Tx and the DMA Rx.

```
// make sure INTA not active, update the last descriptor:
// do not link, clear trigger at descriptor end, enable int A at the end,
// update main descriptor & XFERCFG
DMA0->COMMON[0].INTA = 1<<dma_desc_spi_rx_main;
demo_gp_spi_descriptor[spi_rx_descriptor_index_loc].xfercfg =
    (demo_gp_spi_descriptor[spi_rx_descriptor_index_loc].xfercfg & ~(1<<1)) |
    1<<3 | 1<<4;

demo_dma_main_descriptor[dma_desc_spi_rx_main].src_addr = demo_gp_spi_descriptor[0].src_addr;
demo_dma_main_descriptor[dma_desc_spi_rx_main].des_addr = demo_gp_spi_descriptor[0].des_addr;
demo_dma_main_descriptor[dma_desc_spi_rx_main].link     = demo_gp_spi_descriptor[0].link;
DMA0->CHANNEL[dma_desc_spi_rx_main].XFERCFG             = demo_gp_spi_descriptor[0].xfercfg;

// let the DMA ch run
DMA0->COMMON[0].ENABLESET = 1<<dma_desc_spi_rx_main;

// let SPI Tx run
spi_tx_transfer_count = 0;
NVIC_ClearPendingIRQ(FLEXCOMM5_IRQn);
NVIC_EnableIRQ(FLEXCOMM5_IRQn);

// wait for the DMA A int
while((DMA0->COMMON[0].ENABLESET & (1<<dma_desc_spi_rx_main)) != 0);
```

Figure 24. SPI Rx DMA transfer configuration update

The following code defines the SPI TxFIFO empty interrupt. It checks RxFIFO to retrieve the remaining free space. If the RxFIFO is not full, there must be as many as possible entries written in the TxFIFO.

```
void FLEXCOMM5_IRQHandler(void)
{
    uint32_t rx_fifo_vacancy_loc;

    GPIO->SET[AUX0_PORT] = 1<<AUX0_PIN;

    spi_isr_count++;

    if ((SPI5->FIFOSTAT & SPI_FIFOSTAT_TXEMPTY_MASK) != 0)
    {   // TXFIFO empty, proceed

        // step 1: find out is there any room in the RxFIFO for data to be received?
        // step 2: if room available, adjust by 1 for potentially an incoming entry
        // step 3: write as many entries as possible/available to the TxFIFO without causing RxFIFO overflow
        rx_fifo_vacancy_loc = 8 - ((SPI5->FIFOSTAT>>16) & 0x1F);

        if (rx_fifo_vacancy_loc != 0)
        {   // RX FIFO not full, proceed
            rx_fifo_vacancy_loc--;

            while((rx_fifo_vacancy_loc != 0) && (spi_tx_transfer_count != tx_len))
            {   // loop while room left in RX FIFO and not all data sent
                if (spi_tx_transfer_count != (tx_len-1))
                {   // not the last byte
                    SPI5->FIFOWR = (uint32_t)tx_data[spi_tx_transfer_count];
                }
                else
                {   // the last byte
                    SPI5->FIFOWR = tx_spi_last_control | ((uint32_t)tx_data[spi_tx_transfer_count]);
                }
                spi_tx_transfer_count++;
                rx_fifo_vacancy_loc--;
                if (spi_tx_transfer_count == tx_len)
                {
                    NVIC_DisableIRQ(FLEXCOMM5_IRQn);
                }// end of not all data sent
            } // end of RX FIFO not full AND not all data sent
        }// end of RX FIFO not full
    }// end of TX FIFO empty

    GPIO->CLR[AUX0_PORT] = 1<<AUX0_PIN;

    return;
}
```

Figure 25. SPI TxFIFO empty interrupt handler definition

This implementation avoids the RxFIFO overflow and data to be stalled without additional DMA resources required. It instead leverages the SPI TxFIFO interrupt to send new data. However, the Tx traffic is impacted with a performance reduction, transfers are 28 % slower.

3.3 Performance comparison

Table 2. Performance comparison of the SPI DMA limitation and proposed workarounds

	One pattern transfer (in us)	Time between transfer (in us)	Total transfer (pattern x 28) (in ms)
Issue	63.88	232	8.8
Workaround 1	64	187	7.63
Workaround 2	199	206	11.26

4 Conclusion

This application note demonstrated 2 different approaches to avoid the SPI + DMA bandwidth limitation with their own advantages and constraints. However, these approaches reduce SPI’s capabilities in terms of performance or available resources.

Major modifications are shown in this application note. For further information, refer directly to the available code.

5 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

6 Revision history

Table 3. Revision history

Document ID	Release date	Description
AN14170 v.1	25 January 2024	Initial version

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.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

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.

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.

AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μVision, Versatile — are trademarks and/or registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

Cadence — the Cadence logo, and the other Cadence marks found at www.cadence.com/go/trademarks are trademarks or registered trademarks of Cadence Design Systems, Inc. All rights reserved worldwide.

i.MX — is a trademark of NXP B.V.

Microsoft, Azure, and ThreadX — are trademarks of the Microsoft group of companies.

Contents

Date of release: 25 January 2024
Document identifier: AN14170