AN14170

使用i.MX RT500实现SPI/DMA

第1版—2024年1月25日 **应用笔记**

文档信息

信息	内容
关键词	i.MX RT500、i.MX RT600、SPI
摘要	本应用笔记详细介绍了如何复现和解决高SPI DMA流量情况下可能出现的SPI限制。

1 RT500介绍

i.MX RT500是一个面向嵌入式应用的双核微控制器系列，其特色是Arm Cortex-M33 CPU与Cadence Xtensa Fusion F1音频数字信号处理器CPU的组合。此Cortex-M33包含两个硬件协处理器，为一系列的复杂算法提供增强的性能。该系列产品提供了一套丰富的外设和极低的功耗。该芯片具有高达5MB的SRAM，两个FlexSPI（八线/四线SPI接口），每个接口都有32KB的缓存，其中一个带有动态解密功能，高速USB设备/主机+PHY，12位1 MS/s的ADC，模拟比较器，支持多达8个DMIC通道的音频子系统，2.5D矢量GPU和带有MIPI DSI PHY的LCD控制器，两个SDIO/eMC，FlexIO，AES/SHA/Crypto M33协处理器和PUF密钥生成功能。

i.MX RT500还提供了12个Flexcomm模块，可针对以下协议之一进行配置：USART、SPI、I2C、I2S。

与DMA相结合，i.MX RT500通过减轻CPU的负载，提供了一种高效地与外设通信的方式，，从而改善了延迟并提高性能。

本应用笔记介绍了高带宽场景下的性能限制，以及如何克服这些限制。

2 SPI+DMA的性能限制

在SPI+DMA的标准使用场景中，SPI的Tx和Rx流量均由DMA处理，在存储器和外设之间传输数据。在某些情况下，多个外设可以连接到SPI总线，从而产生高DMA流量。在这种情况下，可能会达到带宽的限制，从而导致SPI数据的停滞。我们注意到，有时SPI数据可以到达RT500的SPI接口，但最终不会进入到SRAM中，从而导致字节丢失。这是因为DMA的负载过重，导致Rx FIFO溢出。

此测试演示了这个限制，其中Flexcomm 5以模式0（CPOL=0，CPHAL=0）用作SPI接口（SPI5）。SPI5的MISO和MOSI是相互连接的。当由于DMA未能足够快地清空FIFO而导致Rx FIFO溢出时，会触发GPIO错误（GPIO ERROR）。

DMA配置为用于服务来自SPI5的两个请求——SPI Tx DMA通道（11）将SRAM中从0x01到0x3F的测试样式字节复制到SPI 5的FIFOWR寄存器28次；SPI5-Rx DMA通道（10）将数据从SPI5-FIFORD传输到GPIO D[3:0]。D[3:0]表示的是DMA复制到GPIO的而不是SRAM的数据，它由宏*TEST_DESTINATION = DESTINATION_PORT*定义。要重现此限制，必须将宏*TEST_TO_RUN*定义为*ISSUE*。

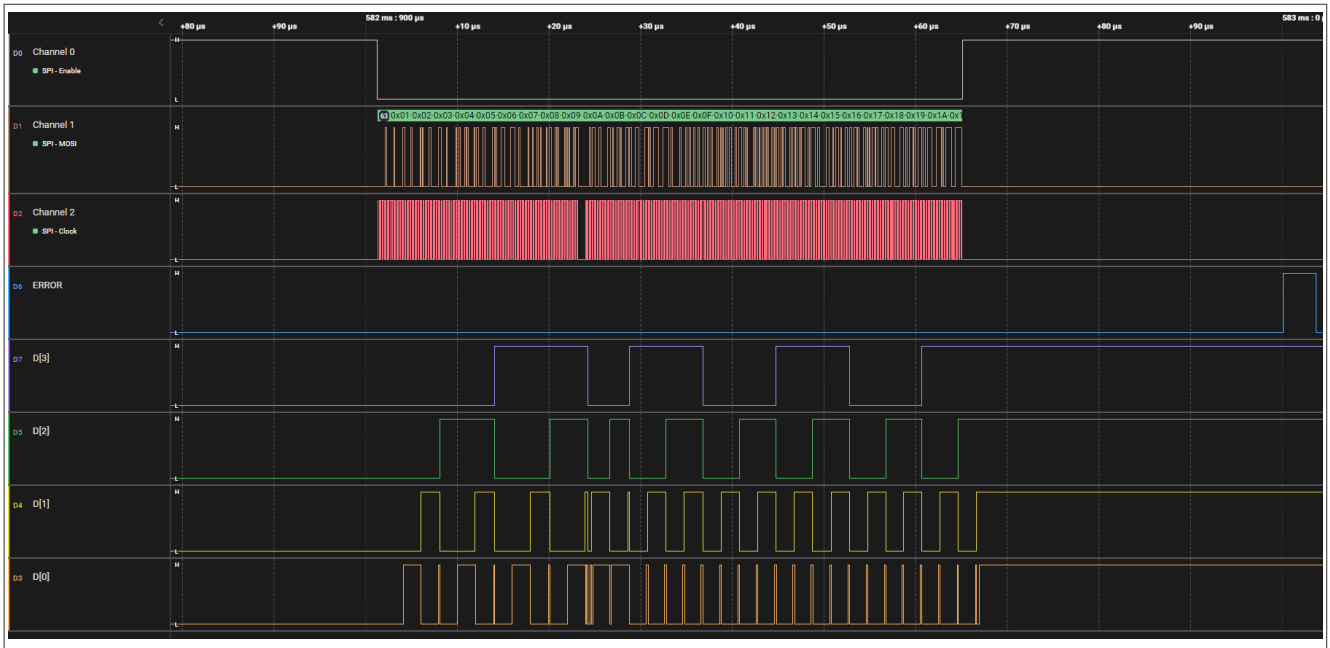

图1. 包含错误的完整SPI传输的截图

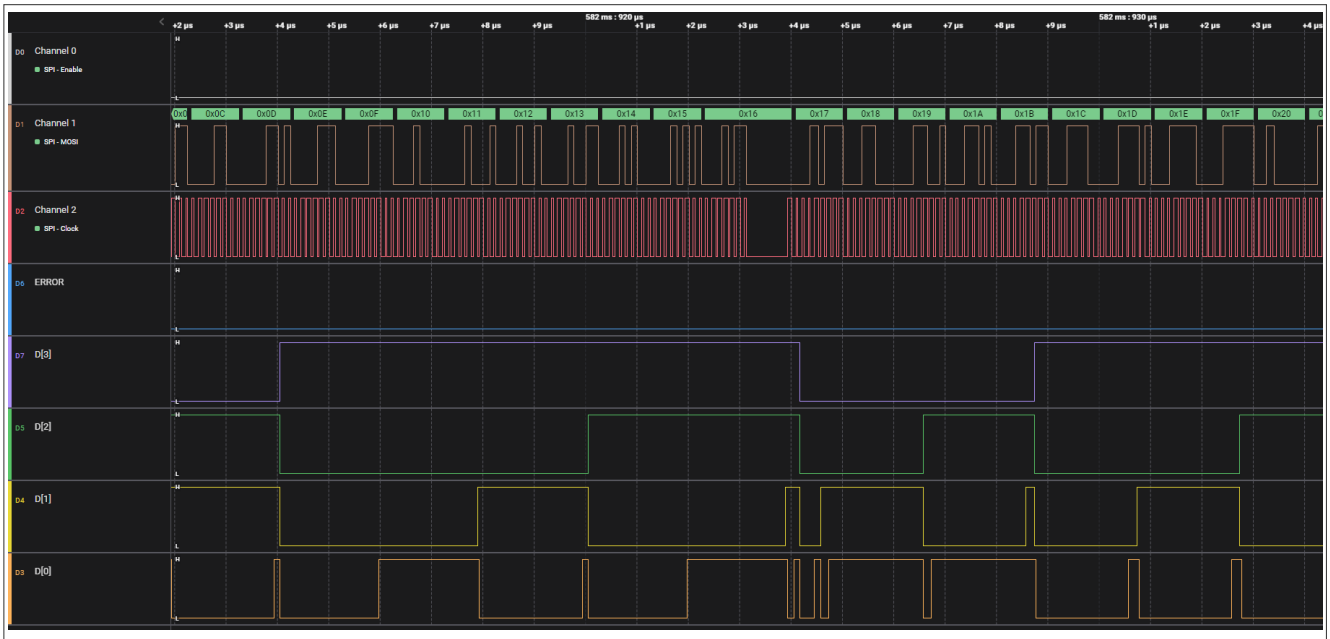

图2. SPI传输及所传输值的截图

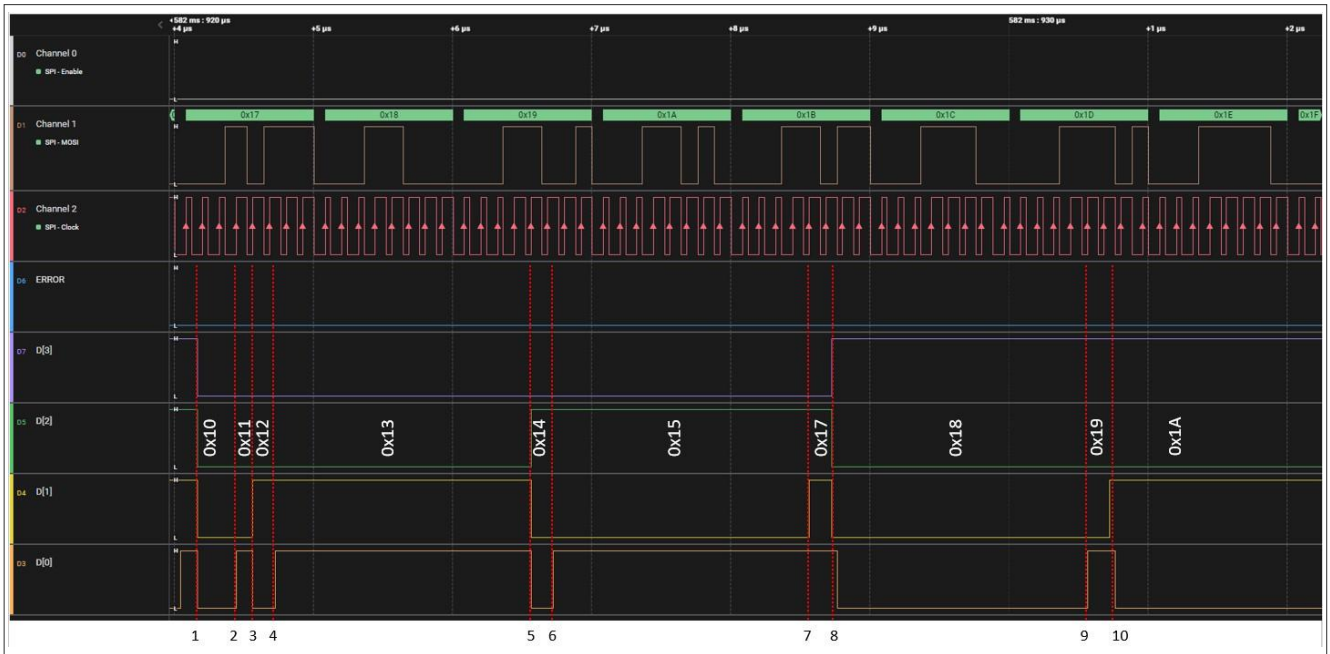

图3. 传输的SPI值和丢失值的截图

表1. SPI总线上接收的数据

传输	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

连续接收数据，直至丢失的数据0x16为止。然而，第一个截图显示，SPI Rx正确地接收到了数据字节0x16。ERROR GPIO（错误GPIO）在SPI通信结束时被触发。

下面是一个正常通信的示例：

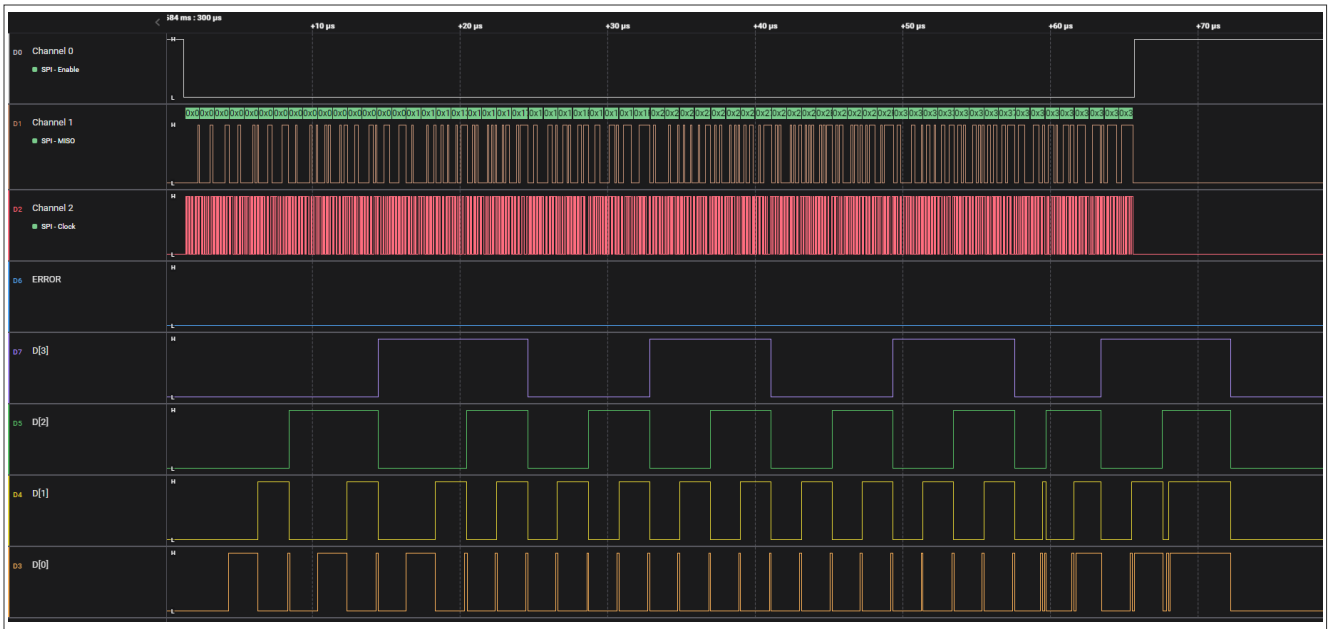

图4. 无错误的完整SPI传输的截图

3 解决方法

为了克服这个限制，可以开发不同的机制。我们将探讨两种可能的SPI+DMA配置，及它们各自的优缺点及工作原理。

3.1 实施方案1

此实施方案的理念是利用一种“背靠背”的传输模式，使用全部的SPI总线，同时使用尽可能少的其他资源。

第一种实施方案是一种完全基于硬件的方法，使用DMA触发输出机制来驱动额外的DMA通道，称为通道链接。通道链接功能允许通道x上的DMA传输完成后触发通道y上的DMA传输。

这里使用一个触发信号和两个DMA通道（Rx DMA和Tx DMA），其中两个通道均是下降沿触发。

SPI5 Rx DMA通道（10）已启用外设请求，使用下降沿触发和突发大小为4个字的突发传输来执行从SPI RxFIFO到SRAM中的缓冲区数组的单次传输。第一次传输由软件触发，其余的传输由硬件触发。该DMA通道输入和输出触发都被传输到DMAC0_TRIGOUT_A。SPI5 Rx DMA通道的优先级高于SPI5 DMA Tx通道。在DMA传输长度为m（1/2/3）+ 4 x n的情况下，SPI5-Rx DMA可以使用两个连接的描述符，在其他情况下只需要一个描述符。

SPI5 Tx DMA通道（11）从SRAM中的Tx缓冲区数组到SPI5 TxFIFO进行存储器到存储器的传输。它禁用了外设请求，并使用下降沿触发和突发大小为4个字的突发传输来执行单次传输。通道的输入触发来自SPI5 DMA Rx通道触发输出，并使用两个连接的描述符。第一个描述符进行多次到TxFIFO的8位传输，始终为4的倍数，第二个描述符（尾）根据传输长度对TxFIFO进行不少于1次且不超过4次的32位写入，以产生最终的SPI交换。例如，如果必须发送的SPI字节的总数可被4整除，则尾描述符会发送4字节+SPI控制内容，否则该描述符发送的字节数+控制/字等于总字节数除以4的商。

具有传输长度非0的参数m有助于在SPI Rx和Tx DMA通道之间实现人为的“错位”，这样，当Rx DMA通道正式完成一个突发并即将生成一个新的突发时，FIFOWR中仍有项准备进入SPI串行移位寄存器（由匹配的Tx DMA突发写入）；FIFOWR中的这些剩余项有助于弥补导致SPI总线空闲的间隙，直到SPI Rx DMA通道生成一个新的触发并导致下一个Tx DMA突发写入新的一组4字节数据。

例如，考虑一种16字节的SPI传输，参数m=1。

对于Tx:16/4=0，Tx主描述符传输12个字节，尾描述符传输4个字节。

对于Rx:（16-1）mod4=3，Rx主描述符传输13个字节，尾描述符传输3个字节。

其总体机制如下：

- 首先设置SPI5 Rx DMA通道，以便处理SPI Rx数据流。通过使用SW触发，该通道会获得初始触发并等待来自外设的请求。
- 接下来设置SPI5 Tx DMA通道，当它被配置为执行存储器到存储器的传输时，一旦设置SW触发，它就会立即执行一个突发传输，向TxFIFO传输4次。当完成传输后，SPI5 Tx DMA通道的触发信号会在使用完突发后被清除，然后等待一个新的触发信号。
- 当SPI5接收到字节时，它会生成DMA请求；由于DMA Rx通道已经由SW触发，DMA Rx通道会产生一个4次传输的突发，并将从SPI RxFIFO接收到的数据复制到SRAM中的Rx数组中。一个DMA通道的突发会将触发信号清除，该下降沿会通过输出触发信号路由到DMA Rx通道的输入触发和DMA Tx通道的输入触发。
- 由于DMA Rx和Tx通道都配置为下降沿触发，两者都被再次触发，这一次是由硬件机制触发的。
- 该触发导致DMA Tx通道通过SPI5发送第二条数据，而同一个边沿会使DMA Rx通道做好准备，以使其在SPI5 RxFIFO中有数据的时刻立即进行读取。
- 该机制会重复进行，直到传输结束，其中第二个Tx描述符会被加载，而第二个Rx描述符是否被加载具体取决于传输的大小。Tx描述符对TxFIFO执行1次到4次的32位写入，以完成SPI传输并取消选择SSEL行。如果加载了Rx描述符，则执行所需的8位读取以完成SPI传输。

3.1.1 代码实现

```
#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
};
```

图5. 在DMA请求中使用的SPI通道的定义

以下代码：

- DMA通道dma_desc_spi_rx_main的请求通过OTRIG_SEL[1]连通至DMAC0_TRIGOUT_A。
- DMA通道dma_desc_spi_rx_main的输入触发通过ITRIG_SEL[12]连通至DMAC0_TRIGOUT_A。

该配置的目的是基于DMAC0_TRIGOUT_A上的下降沿触发通道的DMA传输，并且当SPI5 Rx的DMA请求发出时，该DMA通道在DMA通道dma_desc_spi_tx_main的下降沿触发。

```
// 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
```

图6. SPI Rx DMA通道的连通和触发配置

以下代码：

- 通过OTRIG_SEL[1]连通至DMAC0_TRIGOUT_A。
- DMA通道dma_desc_spi_tx_main的输入触发通过ITRIG_SEL[12]连通至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
```

图7. SPI Tx DMA通道的连通及触发配置

以下代码根据传输的大小配置所创建的Rx的尾部传输。共执行rx_tail_len次8位传输，同时目标地址增加1个传输宽度。当该描述符耗尽时，触发信号将被清除。

rx_tail_len的大小取决于传输的大小和造成Rx和Tx人为“错位”的参数“m”。

dma_desc_spi_rx_add1描述符将数据从SPI RxFIFO复制到SRAM“spi_rx_array_8bit”中的保持变量（holding variable）。

```
//------------------------
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
}
```

图8. Rx的尾部传输配置

以下代码定义了dma_desc_spi_rx_main描述符，该描述符将数据从SPIS的RxFIFO复制到SRAM spi_rx_array_8bit中的保持变量（holding variable）。根据传输的大小，重新加载尾部描述符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];
}
```

图9. dma_desc_spi_rx_main描述符的定义

以下代码配置在下降沿由HW触发的dma_desc_spi_rx_main DMA，并启用大小为4的突发传输。

它还配置了dma_desc_spi_rx_main传输，该传输共执行rx_body_len次8位传输，同时目标地址增加1个传输宽度。当该描述符耗尽时，触发将被清除。

最后，当当前的描述符耗尽后（来自dma_desc_spi_rx_main_xfercfg_temp），将重新加载通道控制结构。

```
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
//----------------------
```

图10. dma_desc_spi_rx_main DMA和传输配置

以下代码配置Tx尾部传输。它共执行tx_tail_len次32位传输，同时源地址增加1个传输宽度。当该描述符耗尽时，触发将被清除。

rx-tail_len的大小取决于传输的大小。

它还定义了将数据从SRAM中的缓冲区数组spi_tx_array_tail复制到TxFIFO的dma_desc_spi_tx_add1描述符。

```
//
// 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;
```

图11. Tx尾部传输配置和dma_desc_spi_tx_add1描述符的定义

以下代码定义了dma_desc_spi_tx_main描述符，该描述符将数据从SRAM中的缓冲区数组spi_tx_array_8bit复制到TxFIFO，并重新加载尾部描述符dma_desc_spi_tx_add1。

它还定义了dma_desc_spi_tx_main 的DMA配置，该配置在下降沿上由硬件触发，并启用了大小为4的突发传输。

最后，它定义了Tx传输配置，并在现有描述符耗尽时重新加载通道的控制结构，共执行tx_body_len次8位传输，同时源地址增加1个传输宽度。当该描述符耗尽时，触发被清除。

```
// 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...;
```

图12. dma_desc_spi_tx_main描述符和DMA传输配置的定义

以下代码准备由dma_desc_spi_tx_add1写入TxFIFO的spi_tx_array_tail。spi_tx_array_tail是SRAM中的缓冲区数组spi_tx_array_8bit与SPI命令spi_fifowr_ctrl的组合，通过SPI FIFOWR寄存器取消选择SSEL。

```
// 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
//----------------------
```

图13. spi_tx_array_tail的准备

以下代码配置dma_desc_spi_rx_main传输，与之前配置的相同，但启用了软件触发。最后，它启用DMA通道10。

它配置了软件触发的dma_desc_spi_tx_main传输，并在当前描述符耗尽时重新加载通道的控制结构，共执行tx_body_len次8位传输，同时源地址增加1个传输宽度。当该描述符耗尽时，触发被清除。最后，它会启用DMA通道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;
```

图14. dma_desc_spi_rx_main和dma_desc_spi_tx_main的传输配置

此处所示为一个使用不同SPI频率实现的示例。

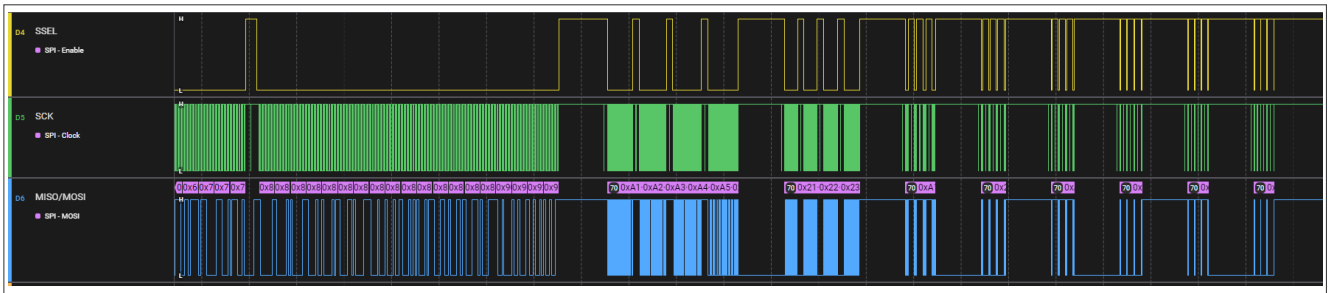

图15. 使用不同的SPI频率进行测试的解决方案1的截图

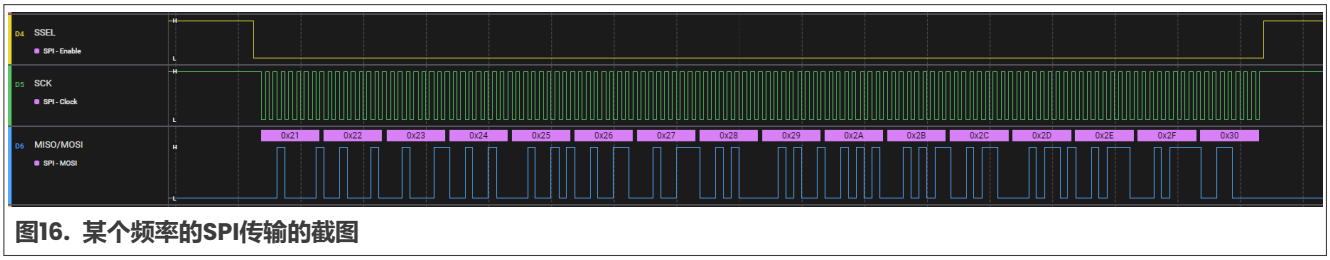

图16. 某个频率的SPI传输的截图

此实施方案仅需要1个DMA输出触发作为附加资源。该解决方案利用背靠背的传输模式，以100%的利用率提供最佳的SPI总线性能。在性能方面，与最初的实施方案相比，没有任何退步。因此，这种实施方案的缺点在于资源方面，因为它需要1个DMA输出触发。

3.2 实施方案2

该实施方案的理念是以SPI TxFIFO为空来生成中断，并知道RxFIFO的当前水平，以便向TxFIFO写入尽可能多的项。仅使用一个DMA通道将数据从RxFIFO传输到SRAM。当TxFIFO为空时，SPI5中断服务例程用于填充SPI TxFIFO。该DMA通道的输入和输出触发都被传输到DMAC0_TRIGOUT_A，与SPI5 DMA Rx通道相同。

此处所示为由SPI中断处理的SPI Tx流量和由DMA处理的Rx流量的示例。

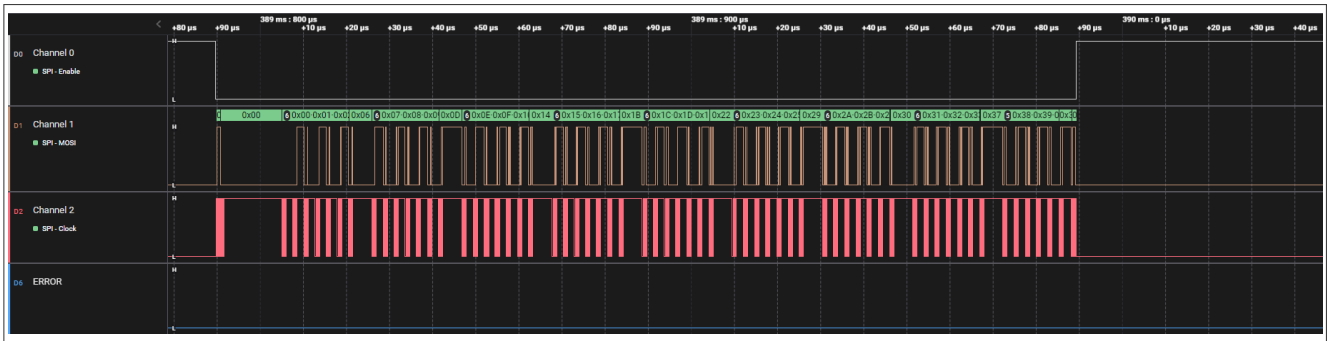

图17. 实施方案2的截图

3.2.1 代码实现

其总体机制如下：

- SPI5通过软件写入SPI5 TxFIFO以直接发送数据。
- SPI5 Rx（通道10，即dma_desc_spi_rx_main）的DMA已配置。当SPI5 Rx接收到数据时，它会生成一个DMA Rx请求，dma_desc_fc5_rx_main将该数据从SPI5 RxFIFO复制到SRAM中的缓冲区数组。dma_desc_fc5_rx_main描述符被编程为执行与RxFIFO中的空闲空间数量相对应次数的若干次传输。因此，它必须将目标地址增加1个传输宽度。dma_desc_spi_rx_main描述符是由软件触发的。
- SPI5被配置为当SPI5 TxFIFO为空时触发一个中断。每次触发中断时，都会检查RxFIFO以检索剩余的空闲可用空间。如果RxFIFO未满，则必须向TxFIFO写入尽可能多的项。

```
// 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
```

图18. SPI配置为当SPI TxFIFO为空时触发中断

```
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);
```

图19. SPI FIFO的配置

接收到前两个字节（指定长度/传输计数），并在轮询模式下使用SPI收集此信息。

```
// 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
```

图20. 发送到SPI FIFO的控制字节

以下代码会复制来自SPI Rx FIFO的SPI Rx DMA数据，并将其自身链接到位于描述符数组中的具有相同属性的下一个描述符。

```
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];
    }
}
```

图21. 从SPI Rx FIFO复制的数据

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

图22. DMA的配置和描述符定义

以下代码将SPI Rx DMA数据存储到SRAM中的缓冲区数组pnt_rx_array。DMA传输配置能够使当前描述符耗尽时重新加载通道的控制结构。

DMA软件被触发并配置为执行spi_rx_descriptor_count_loc次传输，同时目标地址增加1个传输宽度。

```
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);
```

图23. DMA传输配置的定义

以下代码更新了最后一个DMA传输配置，以在描述符耗尽时清除触发，禁用重新加载描述符，并禁用中断标志A。最后，它会启动SPI Tx和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);
```

图24. SPI Rx DMA传输配置更新

以下代码定义了SPI TxFIFO的空中断。它检查RxFIFO，以检索剩余的空闲空间。如果RxFIFO未满，则必须向TxFIFO中写入尽可能多的项。

```
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;
}
```

图25. SPI TxFIFO空中断处理程序的定义

这种实施方案可以避免RxFIFO的溢出和数据停滞，而不需要额外的DMA资源。相反，它利用SPI TxFIFO中断来发送新数据。然而，Tx的流量受到性能下降的影响，传输速度降低了28%。

3.3 性能比较

表2. SPI DMA限制和提出的解决方案的性能比较

	一次模式传输（单位：us）	传输间隔时间（单位：us）	总传输（模式x 28）（单位：ms）
问题	63.88	232	8.8
解决方案1	64	187	7.63
解决方案2	199	206	11.26

4 结论

本应用笔记演示了两种不同的方法来避免SPI+DMA的带宽限制，它们具有各自的优势和限制。然而，这些方法在性能或可用资源方面降低了SPI的能力。

主要的修改内容见本应用笔记。有关更多信息，请直接参阅提供的代码。

5 关于本文中源代码的说明

本文中所示的示例代码具有以下版权和BSD-3-Clause许可：

2024年恩智浦版权所有。在满足以下条件的情况下，允许以源代码和二进制文件的形式重新分发和使用本源代码（无论是否经过修改）：

1. 重新分发源代码必须保留上述版权声明、这些条件和以下免责声明。
2. 以二进制文件形式重新分发时，必须在文档和/或随分发提供的其他材料中必须复制上述版权声明、这些条件和以下免责声明。
3. 未经事先书面许可，不得使用版权所有者的姓名或参与者的姓名为本软件的衍生产品进行背书或推广。

本软件由版权所有者和参与者“按原样”提供，不承担任何明示或暗示的担保责任，包括但不限于对适销性和特定用途适用性的暗示保证。在任何情况下，无论因何种原因或根据何种法律条例，版权所有者或参与者均不对因使用本软件而导致的任何直接、间接、偶然、特殊、惩戒性或后果性损害（包括但不限于采购替代商品或服务；使用损失、数据损失或利润损失或业务中断）承担责任，无论是因合同、严格责任还是侵权行为（包括疏忽或其他原因）造成的，即使事先被告知有此类损害的可能性也不例外。

6 修订历史

表3. 修订历史

文档ID	发布日期	说明
AN14170 v.1	2024年1月25日	初始版本

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.cn/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.

目录

For more information, please visit: https://www.nxp.com.cn

Date of release: 25 January 2024
Document identifier: AN14170