nRF52832 Errata Attachment

Anomaly 109 Addendum

DMA access transfers might be corrupted

January 11, 2017

Contact details

For your nearest dealer, please see www.nordicsemi.com

Main office:

Otto Nielsens veg 12
7004 Trondheim
Phone: +47 72 89 89 00
Fax: +47 72 89 89 89
www.nordicsemi.com

Revision history

Date	Version	Description
January 2017	1.1	Added SPIS workaround: Trigger GPIOTE on the CSN signal
December 2016	1.0	First version

Contents

Introduction

This document is an attachment to the nRF52832 Errata, ID 109. It explains the product anomaly in more detail and the suggested workarounds.

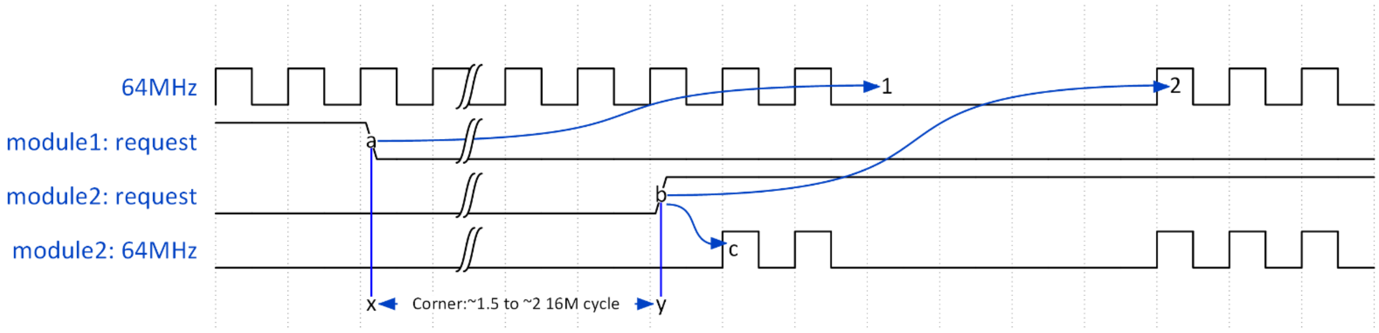
1.1 Anomaly description

System enters IDLE and stops the 64 MHz clock at the same time as the peripheral that is using DMA is started. This results in the wrong data being sent to the external device.

This anomaly affects the following peripherals:

- PWM
- SPIS TX
- SPIM TX
- TWIS
- UARTE
- TWIM TX

2 Root cause



1. Module 1 requests the 64 MHz clock. This can be the CPU or any peripheral with an active DMA channel.
2. At point a in the figure above the module is finished with its task and the clock request is dropped.
   Due to pipelines in the system the clock source is not stopped immediately (point 1 in the figure above).
3. Module 2 (for example, PWM) starts requesting the clock (point b in figure above).
   Due to the pipeline the clock source logic does not immediately see the request (point 2 in figure).
   At this point a clock gate releasing 64 MHz to the module is closed but, the module will see the clock if it is already running.
4. If the delay between point a and point b is 1.5 to 4 x tCK16M then Module 2 will see two clock cycles (the remainders of the request from module 1) before the clock is stopped (point 1) and then later started again (point 2).
5. Module 2 uses these two clock pulses to start accessing RAM, but the clock stops before the transaction can finish so the transaction never reaches RAM. The result is the read of a previous bus value, most likely zero.

If PPI is used to start any of the affected master peripherals or any of the slave peripherals and sleep modes are used -- WFE, WFI, sd_app_evt_wait() -- then a workaround should be used.

3 Workarounds

3.1 All communication peripherals workaround: Mask by protocol

Communication protocols can be written to mask out this issue. For example, always ignore first byte on SPIS. Another example is that the serial protocol can be written to disregard wrong packets due to incorrect CRC.

Pros: Minimum impact solution.

Cons: Only applicable to certain peripherals with higher layer protocols.

3.2 Master peripherals and PWM workaround: Do not use PPI

Use interrupts to wake up the CPU.

Ensure the CPU is active in the window from 1 µs before serial engine is started and for 16 µs after that, or until transfer is complete.

Pros: Simple workaround.

Cons: Higher current consumption. Does not work for Slave peripherals.

3.3 Master peripherals and PWM workaround: Protect with a Dummy Peripheral

Use the DMA of another peripheral immediately before the target peripheral. This ensures that the clock is already active for the target peripheral's DMA.

To protect SPIM, TWIM or UARTE with an unused (dummy) peripheral, a delay between triggering the two tasks must be achieved.

This is done by:

1. Routing the triggering event directly to the unused peripheral's START task via PPI.
2. Delaying the START task for the active peripheral by routing the PPI fork to an EGU task.
3. Routing the corresponding EGU event to trigger the active peripheral's START task via a second PPI channel. This gives a 2 x 16 MHz clock cycle delay.

To protect PWM with a different module the EGU delay is not required, but if protecting one PWM instance with another PWM instance the EGU delay is required.

Note: If protecting PWM with PWM, only TASKS_SEQSTART[0] can be protected, so only SEQ[0] should be used.

Note: TASKS_SEQSTART[0] and TASKS_SEQSTART[1] must be protected when using both SEQ[0] and SEQ[1].

It is required that the unused peripheral is configured and enabled, to ensure low current consumption set the TXD.MAXCNT=0. The output signals can be left unconnected to the pins (i.e. PSEL can be left disconnected).

This is verified for the following combinations:

- Unused SPIM protecting active SPIM connected through EGU
- Unused SPIM protecting active TWIM connected through EGU
- Unused SPIM protecting active UARTE connected through EGU
- Unused SPIM protecting active PWM
- Unused PWM protecting active SPIM connected through EGU
- Unused UARTE protecting active PWM
- Unused PWM protecting active PWM (SEQ[0] only) connected through EGU

It is recommended to use SPIM as the protecting peripheral, if available. Otherwise UARTE can be used on all affected peripherals. PWM can also be used on all affected peripherals apart from on another PWM (unless only SEQ[0] is used).

Note: It is not recommended to use TWIM as a protecting peripheral.

Pros: Limited additional current consumption. Works for all master peripherals and PWM. Simpler than using a Timer.

Cons: Requires an unused (dummy) peripheral and an extra PPI channel and EGU Task/Event. Shortcuts to START tasks cannot be used. It is also fairly complex and does not work for slave peripherals.

3.4 Master peripherals workaround: Two compares on a counter/timer

Use a timer interrupt to trigger the CPU at least 1 µs in advance and keep it awake until the actual peripheral is triggered by the second timer compare.

Pros: May not require additional resources.

Cons: Does not work for Slave peripherals. Will increase current consumption.

3.5 PWM workarounds

3.5.1 ONLY USE 3 CHANNELS.

Use only CH1-3 in Individual Loading (DECODER_LOAD).

Use only CH2 and 3 in Grouped Loading.

Do not use Common Loading.

Pros: Simple workaround, no current consumption hit.

Cons: Fewer channels available.

3.5.2 USE TIMER TO WAKE CPU/OTHER PERIPHERAL DMA

PWM is very periodic, and the exact period is set in software, which makes using a timer to trigger another peripheral's DMA or cause the core to wake up first very achievable.

Note: PWM can be used as before. For example, with SHORTS used to trigger DMA. To maintain timing, it is recommended to keep the Timer and PWM in sync by using a PPI channel from PWM->TASKS_SEQSTART[0] to TIMER->TASKS_CLEAR.

Pros: Will not increase current consumption significantly.

Cons: Exact timing of PWM sequences must be calculated and the timer exactly matched to them, requires dedicated Timer and PPI channel.

3.6 TWIS workaround

Do not set PREPARETX/RX tasks until after address matching.

Enable and configure the TWIS peripheral, but do not set TASKS_PREPARETX or TASKS_PREPARERX.

Then either:

1. Enable Read and Write interrupts in TWIM peripheral during initialization.

```
void peripheral_init(void)
{
...
    TWIS->INTENSET = TWIS_INTENSET_READ_Msk;
    TWIS->INTENSET = TWIS_INTENSET_WRITE_Msk;
...
}
```

2. Wake CPU from an interrupt on EVENTS_READ or EVENTS_WRITE. In the interrupt handler, trigger TASKS _PREPARETX or TASKS _PREPARERX.

```
void TWIS_IRQ_HANDLER(void)
{
    if (TWIS->EVENTS_READ)
    {
        TWIS->EVENTS_READ     = 0;
        TWIS->TASKS_PREPARETX = 1;
    }

    if (TWIS->EVENTS_WRITE)
    {
        TWIS->EVENTS_WRITE    = 0;
        TWIS->TASKS_PREPARERX = 1;
    }
}
```

Or

1. Use the method described in 3.3 to:
   a. Trigger an unused peripheral (SPIS/UARTE/PWM) via PPI channels from EVENTS_WRITE and EVENTS_READ
   b. Connect EVENTS_WRITE to TASKS _PREPARERX and EVENTS_READ to TASKS _PREPARETX via EGUs to add a 2 x 16 MHz delay, to ensure that the unused peripheral starts the clock first.

Pros: Minimal additional current consumption.

Cons: Slave device must support clock stretching. Either CPU wake up or four PPI channels and two EGU's are required.

3.7 Workaround for all peripherals: Turn on the 64 MHz clock domain

To turn clock domain on:

```
*(volatile uint32_t *)0x4006EC00 = 0x00009375;
*(volatile uint32_t *)0x4006ED08 = 0x00000003;
*(volatile uint32_t *)0x4006EC00 = 0x00009375;
```

To turn clock domain off:

```
*(volatile uint32_t *)0x4006EC00 = 0x00009375;
*(volatile uint32_t *)0x4006ED08 = 0x00000000;
*(volatile uint32_t *)0x4006EC00 = 0x00009375;
```

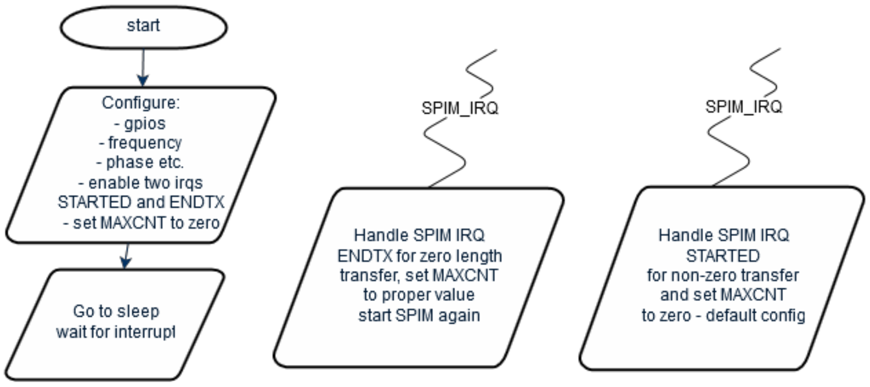
Pros: Simple workaround that fixes all peripherals.

Cons: This increases CPU idle current by around 500 µA. Thus, this is in most cases not a valid workaround.

3.8 SPIM – TX, UART, and TWIM workarounds: Use master's peripheral IRQs to wake up CPU

3.8.1 SPIM - TX WORKAROUND

Configure the peripheral the normal way but, set MAXCNT to zero and enable STARTED interrupt. When something triggers transfer, SPIM generates Events for zero-length transfer. Handle it, set proper value in MAXCNT and start the peripheral again. At the end of a good transfer, set MAXCNT to default – zero value.


start
Configure:
- gpios
- frequency
- phase etc.
- enable two irqs STARTED and ENDTX
- set MAXCNT to zero
Go to sleep
wait for interrupt
SPIM_IRQ
Handle SPIM IRQ ENDTX for zero length transfer, set MAXCNT to proper value start SPIM again
SPIM_IRQ
Handle SPIM IRQ STARTED for non-zero transfer and set MAXCNT to zero - default config

Code snippet:

```
static void peripheral_init(void)
{
    static uint8_t spi_data[8] = { 0x00, 0x00, 0x00, 0x00,
                                   0x01, 0xFF, 0xFF, 0x00 };
    SPIM->PSEL.SCK  = PIN_0;
    SPIM->PSEL.MOSI = PIN_1;
    SPIM->PSEL.MISO = PIN_2;
    SPIM->ENABLE    = (SPIM_ENABLE_ENABLE_Enabled << SPIM_ENABLE_ENABLE_Pos);
    SPIM->FREQUENCY  = SPIM_FREQUENCY_FREQUENCY_M1;
    SPIM->TXD.PTR    = (uint32_t)spi_data;

    SPIM->TXD.MAXCNT = 0;
    SPIM->INTENSET = SPIM_INTENSET_STARTED_Msk;
    SPIM->INTENSET = SPIM_INTENSET_ENDTX_Msk;

    NVIC_EnableIRQ(SPIM_IRQ);

}

void SPIM_IRQ_HANDLER(void)
{

    // First at all catch event start of the first, zero-length transmission,
    // set proper values and start new transmission
    if (SPIM->EVENTS_STARTED)
    {
        SPIM->EVENTS_STARTED = 0;
        If (SPIM->TXD.MAXCNT == 0)
        {
            SPIM->TXD.MAXCNT = 8;
            SPIM->TASKS_START = 1;
        }
    } else
    {
        // Second step catch event end of the last good transmission
        // and set default values
        if (SPIM->EVENTS_ENDTX)
        {
            SPIM->EVENTS_ENDTX  =  0;
            if (SPIM->TXD.MAXCNT != 0)
            {
                SPIM->TXD.MAXCNT = 0;
            }
        }
    }
}
```

3.8.2 UART WORKAROUND

This uses the same workaround as SPIM. But, use TXSTARTED interrupt instead of STARTED.

Code snippet:

```
static void peripheral_init(void)
{
    static uint8_t data[8] = { 0x00, 0x00, 0x00, 0x00,
                               0xFF, 0x00, 0x00, 0x00 };

    UARTE->PSEL.TXD = PER_PIN_0;
    UARTE->PSEL.RXD = PER_PIN_1;
    UARTE->ENABLE   = (UARTE_ENABLE_ENABLE_Enabled << UARTE_ENABLE_ENABLE_Pos);

    UARTE->BAUDRATE   = UARTE_BAUDRATE_BAUDRATE_Baud115200;
    UARTE->TXD.PTR    = (uint32_t)data;


    UARTE->TXD.MAXCNT = 0;                          //FTPAN109
    UARTE->INTENSET = UARTE_INTENSET_TXSTARTED_Msk; //FTPAN109
    UARTE->INTENSET = UARTE_INTENSET_ENDTX_Msk;     //FTPAN109

    NVIC_EnableIRQ(UARTE_IRQ);

}
void UARTE_IRQ_HANDLER(void)
{
    // First at all catch event start of the first, zero-length transmission,
    // set proper values and start new transmission
    if (UARTE->EVENTS_TXSTARTED)
    {
        UARTE->EVENTS_TXSTARTED = 0;
        if (UARTE->TXD.MAXCNT == 0)
        {
            UARTE->TXD.MAXCNT = 8;
            UARTE->TASKS_STARTTX = 1;
        }
    } else
    {
        // Second step catch event end of the last good transmission
        //and set default values
        if (UARTE->EVENTS_ENDTX)
        {
            UARTE->EVENTS_ENDTX  =  0;
            if (UARTE->TXD.MAXCNT != 0)
            {
                UARTE->TXD.MAXCNT = 0;
            }
        }
    }
}
```

3.8.3 TWIM WORKAROUND

Solution shown in 3.8.2 and 3.8.1 will work with TWIM, but the peripheral will send an additional address. For some TWI devices like sensors it might be hard to handle:

Address (first, zero-length transfer) -> Address (second, good transaction) -> data

To avoid the first address being sent:

1. Set the peripheral frequency to zero. This configuration causes the TXSTARTED event but no data will be sent. However, the address still exists in the TWIM buffer.
2. Set TWIM->ENABLE = 0 to remove the address from the buffer and clear the peripheral .
3. Enable the peripheral again and set the proper frequency.
4. Start a new transmission.

Code snippet:

```
static void peripheral_init(void)
{
    static uint8_t twi_data[] = { 0xFF, 0xFF, 0xFF, 0xFF,
                                  0x00 };

    TWIM->PSEL.SCL = PIN_0;
    TWIM->PSEL.SDA  = PIN_1;
    TWIM->ENABLE    = (TWIM_ENABLE_ENABLE_Enabled << TWIM_ENABLE_ENABLE_Pos);

    TWIM->ADDRESS    = 0;
    TWIM->TXD.PTR    = (uint32_t)twi_data;
    TWIM->TXD.MAXCNT = 5;

    // Set frequency to zero not to transfer address or clock something
    TWIM->FREQUENCY = 0;
    TWIM->INTENSET = TWIM_INTENSET_TXSTARTED_Msk;
    TWIM->INTENSET = TWIM_INTENSET_STOPPED_Msk;

    TWIM->SHORTS = TWIM_SHORTS_LASTTX_STOP_Msk;

    NVIC_EnableIRQ(TWIM_IRQ);
}

void TWIM_IRQ_HANDLER(void)
{

     // First catch event start of the first, non-clocked address transmission
     if (TWIM->EVENTS_TXSTARTED)
    {
        TWIM->EVENTS_TXSTARTED = 0;
        if (TWIM->FREQUENCY == 0)
        {
            // Set enable to zero to clean all buffers and peripheral
            TWIM->ENABLE = 0;
            TWIM->ENABLE = (TWIM_ENABLE_ENABLE_Enabled << TWIM_ENABLE_ENABLE_Pos);
            // Set frequency
            TWIM->FREQUENCY = TWIM_FREQUENCY_FREQUENCY_K400;
            // Start good transmission
            TWIM->TASKS_STARTTX = 1;
        }
    }

    // Second step, catch event end of the last good transmission
    // and set default values
    if (TWIM->EVENTS_STOPPED)
    {
        TWIM->EVENTS_STOPPED = 0;
        // deconfigure TWIM
        if (TWIM->FREQUENCY != 0)
        {
            TWIM->FREQUENCY  = 0;
        }
    }
}
```

Pros: Workaround without use of extra resources. Simpler portability of SW code.

Cons: Some additional current consumption, CPU is active for several µs. Increased latency between the trigger event and STARTED task. Does not work for slave peripherals.

3.9 SPIS workaround: Trigger GPIOTE on the CSN signal

Use a GPIOTE event to wake the CPU when there is a falling edge on the SPIS CSN signal.

1. Set up a GPIOTE channel to generate events on falling edges of the CSN signal.
2. Enable interrupt generation for this event in the INTENSET register, and enable this interrupt in NVIC.
3. Handle the NVIC interrupt by clearing the generated event in the GPIOTE EVENT register and NVIC before putting CPU to sleep again.

Pros: Specific workaround for SPIS without very large increase of current consumption.

Cons: Workaround requires a GPIOTE, some increase of current consumption.