AN5543

Application note

Enhanced methods to handle SPI communication on STM32 devices

Introduction

The serial peripheral interface (SPI) permits transparent and easy handling of data transfer between peripheral and microcontroller. The SPI has a wide range of possible configurations, which increase the potential for trouble related to specific handling or settings applied to the configuration.

This application note provides a basic overview of STM32 devices SPI capabilities, while highlighting potential issues when handling SPI communications. This document provides tips on how to prevent, and how to manage, the most frequent difficulties encountered when handling SPI communications. An effective peripheral handling decreases the overall system load.

This document targets an audience already familiar with the basic SPI principles and peripheral configuration options. Additional information about the available SPI features and applied instances for a specific product is available in the product reference manuals and datasheets available at www.st.com.

1 General information

This document applies to Arm®-based devices.

Note: *Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.*

2 SPI released versions

The SPI peripheral for STMicroelectronics microcontrollers has evolved over time and different versions with different features had been released over time. The table below summarizes the main differences between active versions of the SPI on STM32 devices families.

Table 1. Main SPI versions differences

Feature/Version	1.2.x	1.3.x	2.x.x[1]	3.x.x[1]
Data size	8 or 16 bit	4-16 bit	4-16/32 bit	4-16/32 bit
Tx & Rx FIFOs	No	2x 4 bytes	2x 4-32 bytes	2x 4-32 bytes
Data packing by data-register access	No	Yes	Yes	Yes
Data packing by packets	No	No	Yes	Yes
Dual clock domain (APB and kernel)	No	No	Yes	Yes
Programmable transaction counters	No	No	Yes	Yes
Underrun detection/ configuration	No	No	Yes[2]	Yes
Autonomous operation at low-power modes	No	No	Yes[2]	Yes
Master transaction suspension	No	No	Yes	Yes
Master automatic suspension	No	No	Yes	Yes
Flushing content of FIFOs	No	No	Yes	Yes
Master data/SS interleaving	No	No	Yes	Yes
GPIOs alternate function control when SPI is disabled	No	No	Yes	Yes
Swap of MOSI/MISO, reversed SS logic	No	No	Yes	Yes
RDY signal option with suspension	No	No	No	Yes
Applied at STM32 products	STM32F1, STM32F2, STM32F4, STM32L0, STM32L1	STM32F0, STM32F3, STM32F7, STM32L4, STM32L5, STM32WB, STM32WL	STM32H7, STM32MP1	Most of STM32 devices launched in 2021 or later

1. *Derivative instances exist where data size configuration or some other features can be limited. Size of the FIFOs and I2S audio specific protocol support always depend on the instance implementation.*
2. *Limited capability; not fully supported.*

2.1 Differences between SPI versions

The different SPI versions have some similarities such as the fact that all of them feature DMS and the capability to wake up from low-power mode. Some featuers are supported as an option between the different SPI versions such as the inter-IC sound (I2S) support and the enhanced slave-select modes. The main differences between the diverse SPI design versions are mainly related to data size, data buffering, dual-clock domain and programmable transaction counters as described below.

Data size

The data size feature differ between the diverse SPI versions in a way that it can be set to be fixed and multiply by 8-bits or it can be set to be adjustable by bits.

Data buffering of Tx and Rx streams

Transmit and receive data buffering streams management has evolved over SPI versions evolution. The earlier versions support only single register pair to handle both the streams while the more recent versions use a set of registers collected at dedicated FIFOs.

Dual-clock domain

The older SPI versions use a single-peripheral clock source which feeds both the peripheral interface and the kernel.

More recent SPI versions feature the capability of an autonomous run at low-power mode under kernel or also under external clock in the cases where the system peripheral interface clock is stopped (refer to Figure 1). This capability is enabled by the fact that in recent SPI versions the kernel clock is separated from the clock feeding the system peripheral bus (APB) interface.

Programmable transaction counters

The specific control related to "end of transaction" actions such as slave-select management, CRC (cyclic redundancy check) append, update of the FIFO data threshold or the termination of data streams can be performed by a proper software action, but ideally it should be performed automatically by hardware using predefined transaction counters.

Earlier versions of SPI do not feature the programmable counters and DMA overtakes this hardware feaure based on its data-counters settings.

The latest SPI versions feature embedded counters, hence SPI takes over control of programable counters actions via the SPI configuration. In these cases, the DMA role is limited to manage the data transfers only.

2.2 SPI frequency constraints

When considering theoretical limits of the SPI bus bandwidth, there is basic dependence on frequency(ies) applied at the associated clock domain(s) supposing that there is sufficient rest of the system performance margin to handle all the fast data flow in time (see Section 4.1 System performance and data-flow problems).

At system featuring single clock domain, the theoretical limit is up to half of the APB clock applied for the peripheral at both master and slave configurations.

On dual clock domain system, there is no specific constrain concerning the ratio between clock feeding APB and the kernel domains. Only in case of their significant difference, the user has to respect the necessary number of clock-periods required to synchronize and propagate the signals shared between the domains (such as changes of flags or registers or data transfers). For the same reason, some specific functionalities can be limited when the data size is set too short, for example the underrun protection logic at slave or handling an optional RDY status signal between master and slave.

The most overkilling factor of the theoretical limits of the SPI bus bandwidth is distortion of the fast SPI signals due to their capacitance versus the throughput of the associated internal GPIO alternate function logic and IO design especially. Besides the IO design and the bus load, the most negative associated factors are extreme temperature and low-supply voltage. These factors may degrade the highest theoretical accessible communication speed for some instances (or their mapping options) for some specific SPI modes, for some power supply ranges or for some temperature ranges. The real limits verified by characterization or guaranteed by design are listed on the dedicated product datasheets.

3 Data-flow handling principle

The SPI hardware interface is based on a pair of physically separated internal shift registers which handle the input and output serial bit streams on MOSI and MISO pins synchronously clocked by the SCK signal.

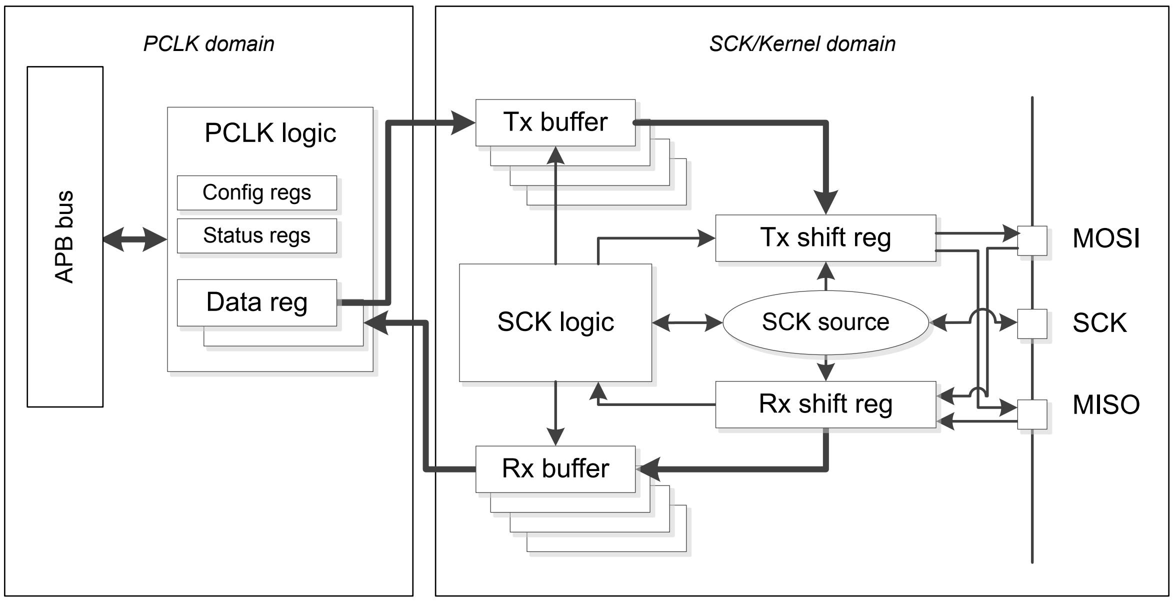
The configuration of the SPI (if it operates as a slave or a master) determines which SPI pin works as input or as output and wether if the SCK clock source is external or internal. As shown in the figure below, data are clocked in and out through the MISO and MOSI pins via shift registers associated with the RAM storage area providing separated Tx and Rx buffers to keep data for reception and for transmission. In early SPI versions, each of this buffers are made by a single-data register. More recent SPI versions feature an extended set of registers which work under the FIFO principle. The content of the buffers is accessible via *read* or *write* access of the SPI data register allocated at APB (PCLK) clock domain. All the other peripheral configurations and status registers are allocated at this domain too.

At transmission, once the content of the Tx shift register dedicated for data output is shifted out completely, the oldest data is on to be moved into this register. This oldest data is moved out from the associated Tx buffer; the space formerly occupied by this data is now released and the buffer is ready to accept new data.

At reception, once the Rx shift register dedicated for data input is clocked in completely, the latest receive-data is moved from this register into the associated Rx buffer.

When the peripheral is configured as slave, the SCK clock is always provided by an external source; when configured as master, the SCK clock is sourced internally from kernel (providing either the peripheral interface APB bus clock or a specific separated kernel clock). This clock signal divided by embedded clock baud rate generator, then feeds the outer serial interface of the SCK signal mastering communication with the slave nodes.

Figure 1. **Simplified data flow scheme**



At full-duplex communication, the data handshake for transmission and for reception is done in parallel. New data write access must always precede its corresponding reading because the transmitted information has to be available at the Tx buffer before its transcation begins. When no new data is available for transmission, SPI master suspends the communication, but when the slave is forced by its master to continue at operation at this case, it faces data underrun and starts to provide invalid data. The next data can be written into the buffer once there is enough space to store this new data.

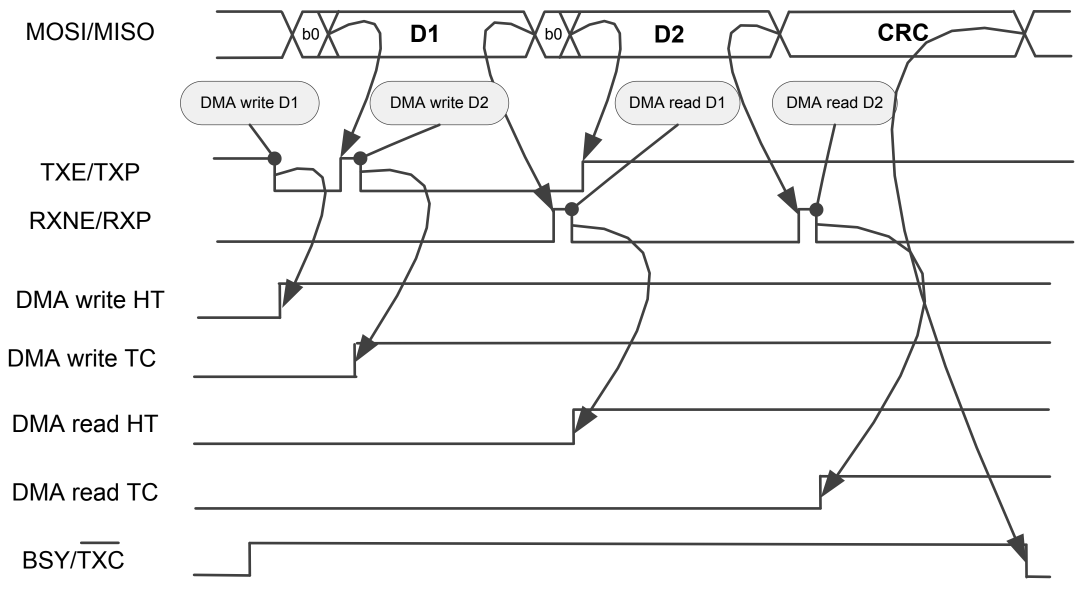
The first bit of data is always provided out from the Tx buffer and the data buffered for transmission are moved into to internal Tx shift register after its first bit is fully transacted out. That is why the value of the first transacted bit is always dependent on the actual buffer content, while rest of the data-frame provided already from the associated shift register is fixed. As a consequence, data output signal can toggle independently on SCK signal prior and during the first bit transaction period at dependency on the Tx buffer access and its content and clock configuration as discussed later at this document.

The transfer between the buffer and the shift register can happen almost immediately after the initial data is written into the buffer at *transaction start* because the buffer becomes empty as soon as the first bit of the data is transacted. When buffer features FIFO structure with sufficient capacity, it can accept an initial sequence of data practically immediately. Oppositely, the received data is copied into the receive buffer considerably later after the ongoing data frame is fully completed.

The following figure which provides a simplified example of a continuous SPI communication handled by DMA where the size of the data corresponds to a single DMA access of the SPI data register. The DMA write-stream is completed earlier than the read stream, while the transaction on the bus can continue after both read and write DMA streams are completed (if CRC is applied).

Despite it is not always necessary, it is wise to monitor the BSY or EOT/TXC flags at the end of the transfer to prevent either a premature SPI disable or the kernel clock-source removal; if any of these two situations happens, the ongoing flow can be corrupted or wrongly terminated. The BSY flag becomes low between data frames if the flow is not continuous and it always drops at slave for a single SCK clock-period between data frames (even if the SCK flow is continuous). The latest SPI versions replace the BSY signal by the EOT and TXC flags. Single-data handling events signalized by TXE and RXNE flags are replaced by TXP and RXP or DXP flags which handle the data packets events; the timing principles stay the same among versions. Note that the BSY flag monitoring is not suggested to handle the data flow.

Figure 2. Timing of SPI data events and DMA transaction complete flags during a transfer



4 Data-flow potential problems

The most frequent problems when handlig the data-flow in applications are listed below grouped in three categories:

- System performance and data-flow
  - Ratio between APB and kernel clock
  - Ratio between master and slave performances
  - Frequency of the SPI and frequency of the DMA events
- Specific SPI modes handling
  - Data corruption by a premature SPI termination
  - Handling exact number of data
  - SPI reconfiguration and handling of associated GPIOs
  - Handling half-duplex operations via a single bidirectional data-line
- Specific SPI signals handling
  - Internal interface signals
  - Optional external interface signals

The following sections provide more details on the mentioned data-flow potential problems.

4.1 System performance and data-flow problems

The SPI bus in an application is potentially fast and the system must be high-performance in order to handle all the data-flow on time. Depending on the SPI mode applied, system has to prevent data overrun at reception and data underrrun at transmission. The ability of the system to handle all the data-flows efficiently becomes more critical when the flow (from SCK clock signal) is continuous, the data frame is short and the SCK clock is comparable or faster than the system core clock.

A practical example is a situation when the baud rate = PCLK/2 and data frame is shortened to 4 bits; the system faces two SPI events within each interval of 8 PCLK periods at full Duplex-mode. In this case, if PCLK is similar (or equal to) the system clock, there is not enough performance margin at system level to handle such continuous full-duplex SPI communication even when all the available CPU performance is spent at an optimized program loop polling directly the SPI flags.

The system lacks of performance margin because it must test the associated occupancy flags at the SPI status register and also has to handle the transfer between the SPI data register and the memory if the corresponding flag is active.

See below a simplified piece of such a C-code loop followed by its assembly compilation which assumes that R1 and R2 CPU registers keep the Tx and Rx data pointers towards the memory while R3 register keeps the base address of the SPI registers):

```
while (1) {
  if ((SPI->SR & SPI_SR_TXE) == 0) {
    SPI->DR = *tx_data++;
  }
  if ((SPI->SR & SPI_SR_RXNE) != 0) {
    *rx_data++ = SPI->DR;
  }
}
```

The assembled result of the compilation of the C-code presented above:

```
??main_1:
   LDR      R0, [R3, #+08]    ;test TXE flag
   LSLS     R0,R0,#+30
   BMI.N    ??main_2
   LDRB     R0, [R1], #+1    ;read and send next data from a memory
   STRB     R0, [R3, #+12]
??main_2:
   LDR      R0, [R3, #+08]    ;test RXNE flag
   LSLS     R0,R0,#+31
   BPL.N    ??main_1
   LDRB     R0, [R3, #+12] ;store next data received to a volatile memory
   STRB     R0, [R2], #+1
   B.N         ??main_1
```

Even a simplified loop of code like the one presented above is impossible to be performed within 8 CPU cycles despite the C-compiler is configured for speed optimization of the code on this Cortex® M3 example. The execution of this loop would be also critical even within 16 cycles with a standard 8-bit data-frame configuration.

If the SPI events are handled by interrupts, system needs to perform the same number of the cycles at detection of the pending event and required memory transfer inside the interrupt service execution as in polling loop, but it consumes additional cycles at the service entry and return, necessary to safe and restore the interrupted main execution context at stack. That is why a solution based on interrupts is not useful in this case.

A possible solution to this problem is to implement DMA but the user must not overestimate the DMA bandwidth in order to provide a real solution to the problem. DMA still need a few system cycles to perform the required data transfer while execution of the related DMA service could be postponed due to another ongoing system operation. A DMA transaction between peripheral data register and SRAM memory can consume up to seven system-clock cycles by including the bus matrix arbitration.

Note that due to a round-robin scheme principle applied to share the bus matrix bandwidth, the transfer request addressed to DMA can have significant latency after it is raised by the peripheral. The latency can be caused by a simultaneous request from another DMA channel or by the execution of a non-interruptible data-transfer sequence being performed by the CPU. Some usual non-interruptible data-transfer sequences are the stack context handling (either at entry to or at return from an interrupt service which normally takes eight cycles) or a simultaneous data-batch transfer such as a LDMIA instruction execution which can take up to 14 consecutive cycles.

In some cases, a delay to serve a pending DMA request can exceed few tens system-clock cycles. If system-clock cycles are extended by optional memory-wait states or by APB/AHB clock-ratio compensation cycles, at the moment when two DMA channels are pending and while the CPU performs an execution of two subsequent LDMIA instructions or context handling due to raised interrupt service in parallel, an extreme delay can appear. If such factors become accidentally cumulated, multiple 4-bit data frames can be missed on the SPI bus.

Software or DMA unable to handle data on time

There are three main indicators signalizing that data management is not done on time, either by software or by DMA, despite all the associated configurations of the peripheral and the system are applied at correct way formally. These are data corruption, data overrun and data underrun.

- **Data corruption (data flow is not as expected)**
  - Some data patterns are missing, wrongly repeated or in any way partially corrupted.
  - Note that when correct order of bits is shifted by one or more bits in the frames, this situation reflects a synchronization issue between master and slave rather than a late handling data by system.
  - Some actions that help to identify data-corruption problems are:
    - The implementation of a HW CRC checksum at the end of each transaction batch.
    - Any data redundancy check such as repetition of the messages, plausibility check (comparison of the message with the expected content) or any other similar predefined protocol.
- **Data overrun (RxFIFO space overflow)**
  - A system performance issue during reception is put in evidence when an OVR error flag is rised.
  - Data overrun is mainly a slave-device problem.
  - Data overrun can also appear on a master device configured at Receive-only mode because the SCK clock signal is provided continuously and independently on the data-buffering process on this mode, except for devices featuring master automatic suspension of the communication mode when the RxFIFO is full (follow MASRX bit control at RM).
- **Data underrun (TxFIFO space underflow)**
  - The primary cause for data underrun is a low system performance during transmission.
  - Data underrun occurs only on slave devices because master always suspend the transmission when its Tx data buffer becomes empty and there is no more data to transmit.
  - The most recent devices feature configurable behaviour and specific indication (via UDR flag) when underrun condition happens. The device can either repeat a constant pattern defined by the user or apply the latest received pattern, behaving as a simple shift register which is applicable at a multi-slave daisy-chain circular topology. Repetition of the lastly transacted data-frame is useful at I2S mode to achieve a smooth audio-output in case of slave accidental data dropout.
  - For oldest devices, not featuring UDR flag, the underrun event stays hidden. Some side effects can be visible such as a flow corruption by repetition of a data-patterns previously transacted which value depends on the SPI design. For example, when FIFO is implemented, the oldest FIFO pattern is applied(1).
  - Data underrun appearance depends on exact timing between the moments when data just to be transmitted are written to the data register by slave and start of its transaction by master. If slave manages to write this data still within its first-bit transaction period despite of the underrun condition, the new data can still be accepted for transmission but there is a potential risk of corruption of the first transacted bit. Note that it depends on the LSBFIRST bit setting if the most or the least significant bit of the data frame is considered. If the write comes later, the underrun pattern is transacted completely while new data is buffered for a next transfer.

1. *When a 32-bit TxFIFO of a slave is fully filled by 0xAA, 0xBB, 0xCC, 0xDD and a transfer of five 8-bit patterns is applied by master without any other update of the slave's TxFIFO, the slave output is 0xAA, 0xBB, 0xCC, 0xDD, 0xAA.*

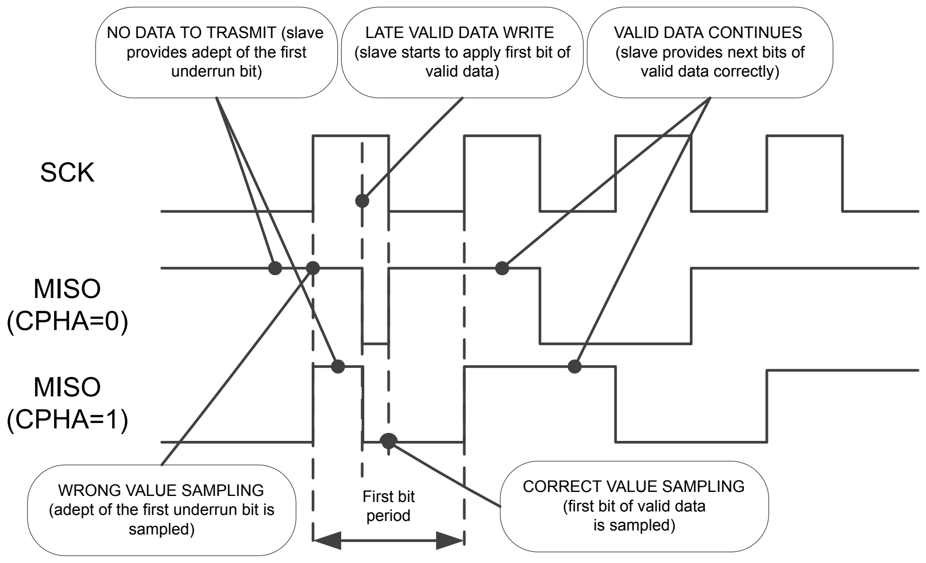
In most SPI designs, when a last-valid data bit is processed out and the data register of the TxFIFO is empty (node does not have next-data to transfer), an adept to first underrun bit is taken into account from a previously transacted pattern. The data correctness is unsure at this case as the behaviour of the MISO slave output depends on many factors: on the clock-phase setting, on the timing between an ocassional late-data write versus the beginning of the next transaction handled by the SCK signal or on the value of the underrun candidate bit versus the first new valid one to be transmitted.

In systems where the CPHA bit is set, the MOSI maintains the level of the last valid bit transacted and the first underrun bit starts to be propagated our with the leading SCK edge of the next transaction if no new data is applied at that time. If the CPHA bit is cleared, the adept underrun bit appears immediately on the bus right after the last-data transaction is completed even if the underrun condition is not yet filled as valid data can still be written in time before the next data-frame transaction starts. At this case, the MISO line starts to propagate (with possible toggle) the correct level immediately once the valid data is written into the data register and propagated into the empty SPI Tx buffer while node becomes ready for the data transaction at this configuration.

A typical observation of a late data-write correlated with the next data-frame transaction start is the slave output peak toggle. This peak toggle output is present prior or during the transaction of the first bit of a frame still provided by the buffer. This situation supposes that the value of the adept underrun bit and the corresponding bit at the new data are different, else there is no toggle and the switch between the underrun and the valid data source is not visible.

The following figure shows a typical deformation of the first bit when the software or the DMA writes late the new data to be transmitted, while the transaction of the first bit is already ongoing (or close to the moment when the data transaction starts). For illustration purposes, the valid bit of the next-data frame has opposite value than the underrun candidate in order to highlight the adequate change of the MISO signal. The valid bit-value is sampled correctly only if the write of the valid data preceeds the bit sampling-edge, else, the underrun adept bit-value is wrongly sampled. Wrong sampling is performed at configuration of CPHA=0 when the data-write comes too late after the sampling is performed at the beginning of the period.

Figure 3. **A late write (underrun) of data while first-bit transaction is ongoing**


NO DATA TO TRASMIT (slave provides adept of the first underrun bit)
LATE VALID DATA WRITE (slave starts to apply first bit of valid data)
VALID DATA CONTINUES (slave provides next bits of valid data correctly)
SCK
MISO (CPHA=0)
MISO (CPHA=1)
WRONG VALUE SAMPLING (adept of the first underrun bit is sampled)
First bit period
CORRECT VALUE SAMPLING (first bit of valid data is sampled)

A correct management of value sampling is observed at configuration CPHA=1 despite the data being written with significant delay; this is possible due to the fact that sampling is provided later at the middle of the bit period. Correct data transaction continues for the rest of the data frame at both configuration cases because data is written at the first-bit transaction period, so the transfer between Tx buffer and Tx shift register is still accepted.

4.1.1 Recommended methods to handle frequent SPI and DMA events

There are different possible ways to handle frequent SPI and DMA events, but their applicability depends on the device's SPI design. Older devices do not necessarily feature data FIFOs or widely configurable data size and they have limited configuration capabilities compared to recent devices.

As previously stated in this document, handling all SPI events by interrupts is not the most convenient method when fast baud rate is applied as repetition of data-handling requests is very frequent.

An optimum performance is achieved when DMA takes care of all the data flow and when at the same time the events for specific transactions are either polled by software or monitored by interrupts enabled by selective flags. The user must always check the status of error flags in order to detect problems with the bus throughput or with the system's performance. In some cases even an optimal distribution of the system's performance is not enough to make a fast SPI flow and the user must apply additional methods to secure a sufficient bandwidth.

Some effective actions to improve the management of frequent SPI and DMA events are:

- **Decrease the bus rate.** By decreasing the bus rate, the ratio between the system and the kernel clock is increased.
- **Increase the data size.**
- **Collect data into packets** when the software fills in and reads out the data.
- **Balance** the data register access with the data size and the threshold level of the FIFOs or of the data configuration setting. More information is detailed in the following section.
- **Slow down the data rate** leading to discontinuous clock by using fixed or enhanced adaptable methods of the temporal communication suspension between provided data frames or sessions.

The actions listed above decrease the frequency and the number of data-handling events; by these means, they permit an improved system performance based on a better event detection management.

Some actions that can be taken in order to decrease the latency when serving the raised events are:

- Temporarily disable other interrupts, DMA channels or both.
- Setup and order priorities of the DMA channels.

Suppressing the system's interrupts minimize the CPU's contribution at the round-robin bus matrix distribution scheme. The user must then avoid automatic register's refresh or memory dump windows during debug to avoid decreasing the BUS matrix throughput. In cases where the DMA interrupts are enabled, the half DMA transfer-completion interrupt must be suppressed if not used. Servicing this event becomes critical when a small number of continuous data is transacted by the DMA session or when frequent DMA data-handling services face a bad system performance (refer to the DMA HT and TC events on Figure 2).

At master side, the data-rate flow can also be slowed down by opting out from a continuous SCK flow (if applicable). The latest SPI versions feature optional programmable fixed interleaving gaps between frames, or between the SS signal which becomes active and first data transaction at session begin, and they can also implement an automatic suspension of the master. This temporary suspension of the master's transfers prevents data underflow and overflow and it can be achieved by monitoring the RxFIFO occupancy (MASRX feature) or by monitoring additional specific status-ready signal (RDY) which can be optionally provided by the slave.

Other features present in latest SPI designs and which simplify the data-handling processes are:

- Capability to setup data counters
- Capability to use an optional coupled dual-flag event to control both transmissions and receptions of a common event with a single software service.

A dual flag puts in evidence that both transmission and reception flags are active. When a double event like this one is handled at master level, there is a risk that the flow becomes discontinuous. Discontinuity may occur when new data is written and its corresponding transaction is started after the previous transaction is finished, hence the bus becomes idle.

The dual-flag handling must be avoided at slave mode when the master is continuously providing a clock signal and the packet size is set close to half of FIFO space. There is a high risk of data underrun due to a postponed service of the pending TXP event (which always precedes a RXP event).

The main sense of the DXP appears when targeting low-power applications, especially when the data-handling services are cumulated and their number is decreased to minimum, which prevents system from its frequent wake ups. The counterpart for that is slowing down the data rate.

4.1.2 Balanced handling of communication events

A correct and optimized data handling is achieved by balancing the data access with the configured data size and packing. The user configures the threshold levels of the data buffer which determine the frequency of data handling events.

The data register can be read or write and can be single or multiple access (the amount of access correspond to the number of data at the packet). The data register access is always equal or higher than the selected data size.

A wider access of the SPI data register can be provided by software or by DMA; the access corresponds to a multiple of the data size. In this case, data packing is automatically applied during the read or write of its content. For example, four data with size up to 8 bits can be read or written in parallel by a single 32-bit access of the data register. The type of access is defined by casting the address of the data register at code level.

In recent devices, the user can additionally define longer data packets and cumulate series of data-register accesses on a single event. The number of accesses must correspond to defined threshold of the FIFO. The events then signalize that just the packet service is available and a next sequence of repeated data accesses is expected. For example, if a 32-bit access is applied to data register, a sequence of three of them has to be applied upon a single packet event when data packet is defined as 12x 8-bit data. This feature helps to limit the number of data handling events and decreases the system's load in a significant way as data batch is always serviced at a single run. Internal data exchange between data buffer and shift register is automatically handled based on *one by one* data access sequences.

A TXE respective to a TXP event is raised once that the buffer for transmission is capable to accept the next data respective to the complete data packet. The transmission buffer is ready once the first bit of the data which is releasing the necessary space is transacted out.

The Tx buffer may become empty and the SCK clock signal can stop and become non continuous if the master faces a significant latency when servicing a raised event that releases some place at the transmission buffer. When the same situation happens at slave side while the master continues the transaction, the slave faces a data underrun condition.

There is a risk of first-bit corruption whenever the service latency is close to time interval necessary for a single packed data transaction where next data is written to the empty buffer of the slave when its first bit is already sampled. Refer to Figure 3. A late write (underrun) of data while first-bit transaction is ongoing for a related example. This case can also be observed at MISO toggle.

Ideally the packet size should not exceed half of the FIFO space. This configuration allows that a subsequent additional packet to be applied early after the current packet transaction starts. This allows a big potential margin for servicing next packet close to a whole single-packet transaction duration.

If the data packet size configuration is equal or almost equal to the overall FIFO space and if there is a continuous flow on the bus, the available time to store next data or to read the received data becomes similat to a single-data transaction duration. In this situation, the user must wait until the space corresponding to a complete packet is either released at the TxFIFO during a transaction or that the space corresponding to a complete packet is fully completed at receive data side at RxFIFO.

Under the above configuration, even a small service latency may generate a transaction problem and data packing becomes useless. That type of configuration makes sense for interleaved SPI batches with a number of data to be completely accommodated at FIFO space. In that case, the application has enough room to handle data once a batch is completed before the next batch starts.

When the Full-duplex mode is applied, the number of the overall data transacted (transmitted and received) is fully balanced. The transacted data goes in parallel upon common clock signal and the handling of both data directions is shifted each time. Transmission handling must precede the reception handling, otherwise the master does not start or continue any transaction, and the slave faces underrun whenever there is not available data ready at the Tx buffer while a transaction is ongoing.

The most recent devices, a specific dual flag DXP is applied to enable common services to handle both transmission and reception flows together. This functionality is especially useful at low-power modes.

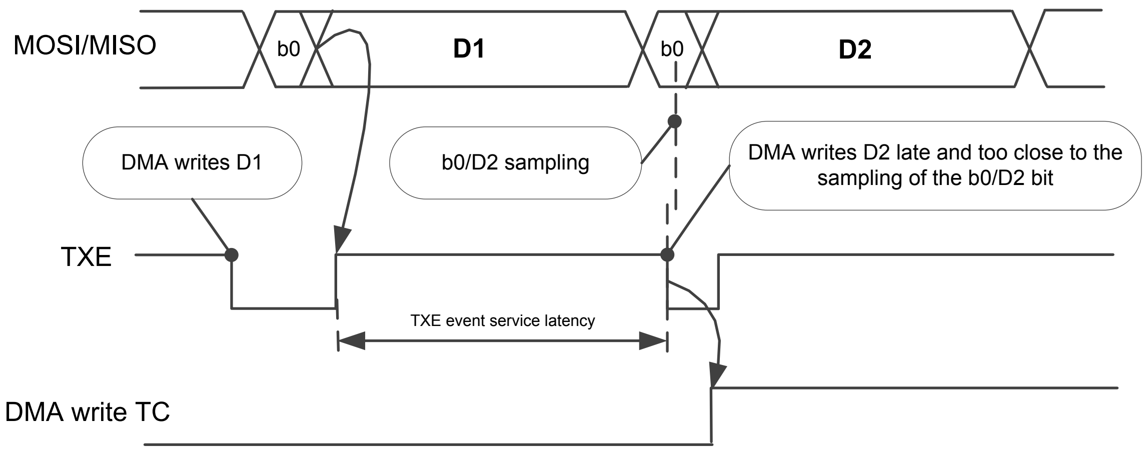
Some actions to cope with different timing of the both data flows are:

- Store some data for transmission in advance
- Handle the middle of the transaction by common DXP events
- Finish the session by handling the remaining received data at the end of the session (for example by an EOT event).

Note: *A DXP event interrupt is suppressed by hardware once all the data for transmission is submitted.*

The following figure depicts an example of a TXE/TXP event service latency limit available at slave for DMA. The example shows a continuous transaction of two consequent data configured with respect to available buffer size what involves to write D2 in time after D1 data is released from the buffer. The data is released from the buffer once its first bit is transacted out on the bus. If D1 is a data packet, TXP signal (replacing the TXE one then) can raise later during D1 transaction once a space large enough to accommodate a full D2 packet is released and available at the buffer to accept new data.

Figure 4. **DMA late service of TXE/TXP event**


MOSI/MISO
b0
D1
b0
D2
DMA writes D1
b0/D2 sampling
DMA writes D2 late and too close to the sampling of the b0/D2 bit
TXE
TXE event service latency
DMA write TC

4.2 Handling specific SPI modes

SPI communication transactions are usually based on a predefined exact number of data. Some enhanced SPI modes are sensitive to deterministic data-counting and require specific control-handling related actions close to the end of the data transaction.

4.2.1 End of the bus activity detection

Specific situations which require proper detection of the bus activity-end to prevent any corruption of ongoing data flow by some premature fatal terminating access are:

- SS signal control
- Necessity to disable the peripheral after the session is completed due to system entry into low-power mode, the peripheral clock removal or change the peripheral configuration for whatever reason (for example change of the data direction at Half-duplex mode or setup another configuration of the SCK signal or baud rate, or simply reset the internal state machine prior next session starts).

Usually the end of the bus activity is acted when the RXNE of the respective RXP event is raised from the last data. The only exception is when the CRC frames are upended at the end of a transaction to validate the data (refer to Figure 1 or when SPI is exclusively used at transmission mode. In such cases, the only suitable signals to monitor the end of the bus activity are BSY, EOT or TXC flag. Any monitoring of DMA complete events is not applicable as SPI can still transmit data (refer to Figure 2).

At BSY monitoring, additional specific measurements has to be applied (such as timeout monitoring in parallel) due to this flag not being totally reliable on older device releases. BSY should be normally cleared by hardware between data frames when communication is not continuous or when it is fully completed; otherwise it highlights the synchronization problem between master and slave (suppose it does not stays hanged by its accidental malfunction).

4.2.2 Specific aspects when SPI is disabled

Once the end of the bus activity is detected correctly, SPI can be disabled but user should be aware of some specific aspects when doing so.

The FIFO content handling while SPI is disabled depends on the SPI design.

- On 1.3.x SPI versions, the content of the Rx FIFO is preserved and accessible. SPI disable is standard procedure to terminate the simplex-receive modes.
- On 2.x.x SPI versions and higher, *suspension* is used. Tx and Rx FIFO contents are flushed, lost and impossible to access when SPI is disabled.

Another problem related to peripheral disable is the *control of the associated GPIOs* The possible risks differ depending on the SPI version:

- SPI versions 1.x.x: the peripheral takes no control of the associated GPIOs when it is disabled. The SPI signals float if they are not supported by external resistor and if they are not reconfigured and they are kept at alternate function configuration.
- SPI versions 2.x.x and higher: the peripheral can keep the GPIO control even when it is disabled (depending on IO locking configuration) to prevent unexpected changes of the associated master outputs especially. In the contrary, when user apply some SPI configuration affecting the locked IOs (such as change of the SCK signal polarity), new default levels become propagated out immediately despite SPI still being disabled. User has to be careful then on desynchronization between master and slave and perform such a change of the SPI configuration when no slave is selected exclusively.

SPI must be disabled before the system entries into Halt mode and between the transactions when SPI has to be reconfigured or to restart the internal state machine properly. Despite the SPI disable between sessions is not mandatory when configuration is not changed, it is strongly suggested anyway to assure correct functionality overall and especially at case of the SPI reconfiguration or when complex functionalities are applied such as:

- CRC calculation
- Counting data which is configured to a non standard data size
- Counting data which number is not aligned with a configured data packet
- Recovery from transaction suspension done by software.

At principle, the SPI must not be disabled before the communication is fully completed and it should be as short as possible at slave, especially between sessions, to avoid missing any communication.

The most recent SPI designs are edge sensitive for hardware SS signal detection; if the master provides an active edge of the SS while the SPI is still temporarily disabled, the slave might miss the start of the slave select-signal and the following session. This is not the case on older SPI design where the NSS detection is not edge but level sensitive.

4.2.3 Time critical control at the end of transaction

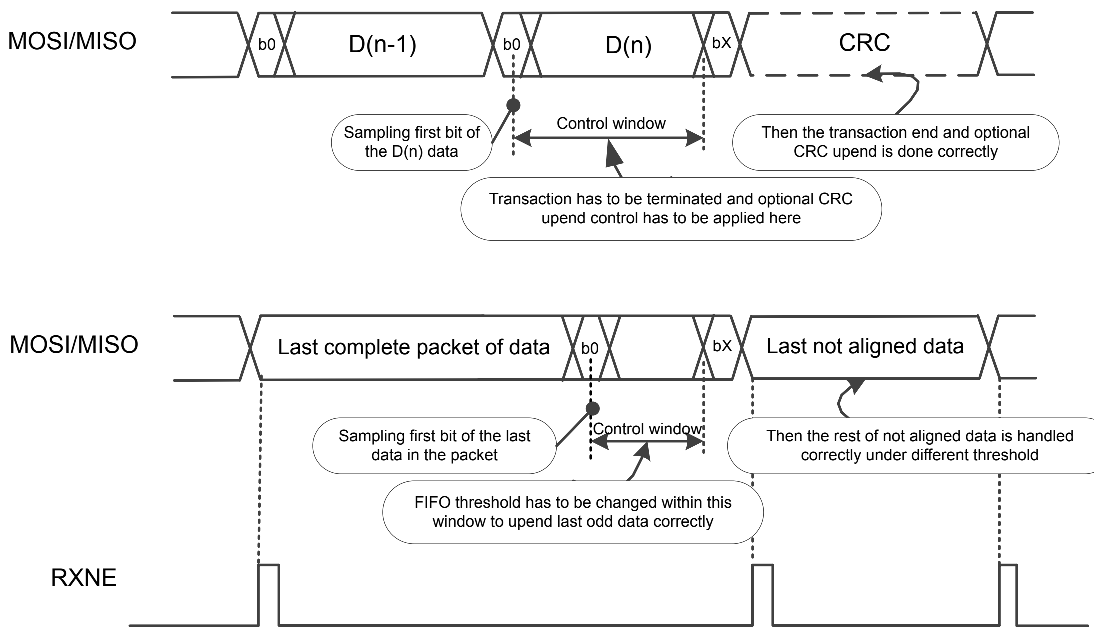
There are specific situations requiring a time-critical access even earlier before the transaction ends within specific control window, just during the last-data handling. Those are:

- Change of the FIFO threshold configuration to handle number of data not aligned with data or packet size correctly
- Upend the CRC pattern control to proper data validation
- Proper terminating of continuous data-flow like master configured at Receive-only mode to prevent any unexpected additional dummy-data transaction and assure just the required data to be transacted. To terminate this specific continuous mode requires nonstandard SPI disable access during ongoing traffic at former designs. This is the case only when SPI disable is required to be applied when communication is still ongoing on the bus.

It is practically impossible to provide such a control when it is based on software polling or interrupt services upon the associated events. There is a very short time frame available during the ongoing transaction of the last data frame and there is a high risk of transaction corruption when this window is missed and the required action is serviced due to a latency outside this interval.

The available control window starts after sampling of the first bit of the frame and it ends before the begin of the last bit transfer time slot as it is shown at the following figure. This figure provides an example of an upend CRC frame pattern after the last data (or handling reception end) when data is not aligned with the FIFO threshold. In these cases, once the last complete data packet is processed, a final incomplete data has to be sent. The same window is applied when continuous clock flow at receive-only mode has to be terminated properly at master with target just to receive an expected number of data while prevent any next dummy-data reception.

Figure 5. **Examples of window available to perform specific control at the end of a transaction**


MOSI/MISO
b0
D(n-1)
b0
D(n)
bX
CRC
Sampling first bit of the D(n) data
Control window
Then the transaction end and optional CRC upend is done correctly
Transaction has to be terminated and optional CRC upend control has to be applied here
MOSI/MISO
Last complete packet of data
b0
bX
Last not aligned data
Sampling first bit of the last data in the packet
Control window
Then the rest of not aligned data is handled correctly under different threshold
FIFO threshold has to be changed within this window to upend last odd data correctly
RXNE

If the device features programmable data counters, a specific control is done autonomously at the end of each transaction. In these cases, the user can operate with the last not-complete packet or with a standard complete packet. Only consistent data is applied (redundant reads give zero data while redundant writes are ignored). Change of the FIFOs threshold during control window is required only if the programmable counter is not used (kept at zero), and such endless transaction setting is not useful if the number of data is known in advance. In the contrary, if the number of data is not known, it is better to use a single-data threshold (packet), assuming that the system has enough performance to handle on time all the data flow one by one. No packet event appears until the complete packet is received; so the user has to check the FIFO occupancy flags at the end of the transaction if a not aligned number of the data is received.

As older devices does not feature embedded data counters, it is favorable to apply DMA and use DMA data counters instead. DMA then performs the required control action autonomously at hardware level so the risk to miss the available control window is lower than when such an action is performed by software. User must correctly setup the data or DMA counters to assure handling of proper number of data on Tx and Rx side (especially when a CRC pattern has to be added). The size of the CRC pattern has to be excluded from the data counting number and the CRC pattern is received to RxFIFO from which it has to be always flushed out by software after the transaction upended by CRC is finished.

The following table provides an overview of the differences at the CRC settings and capabilities. More details can be found on the corresponding product's reference manual.

Table 2. **Differences at CRC control between active design versions of the SPI**

Feature/Version	1.2.x	1.3.x	2.x.x	3.x.x
Setting of DMA data counters	Number of data excluding the CRC	Number of data excluding the CRC[1]	NA	NA
Setting of TSIZE counter	NA	NA	Number of data	Number of data
Handling the end of a not aligned packed transaction	NA	By software (via LDMA_RX bit)	Automatically by hardware	Automatically by hardware
CRC size	Fixed CRC8 for 8-bit data and CRC16 for 16-bit data	CRC8 or CRC16 can be applied for data fixed to either 8-bit or 16-bit	4-16/32 bit[2]	4-16/32 bit[2]
Flushing of the CRC pattern from RxFIFO	Exclusively by software read access of RxFIFO	By software read access of RxFIFO[1]	Automatically by hardware	Automatically by hardware
Initialization CRC pattern	Fixed to all zeros	Fixed to all zeros	Selectable all zeros or all ones	Selectable all zeros or all ones
Reset CRC calculation	By disable both SPI and CRC	By disable both SPI and CRC	By SPI disable	By SPI disable

1. *At Full duplex mode, DMA Rx counter can be set to include the number of the CRC patterns too (1 or 2), as DMA Tx counter setting prevails to handle the pattern CRC transaction. The CRC pattern is read out from the RxFIFO by DMA. This method cannot be used at Rx-only mode, where the CRC pattern flushing is handled by software exclusively.*
2. *The size of the CRC pattern is defined independently from the data size one, but it must be either equal to the data size or be an integer multiple of the data size. CRC size cannot exceed the maximum data size of the instance. Polynomial size should be greater than the CRC pattern size to benefit from the CRC calculator capability, otherwise the pattern is upended by some invalid bits. TSIZE can never be set to its maximum value when CRC is enabled.*

4.2.4 Handling of Half-duplex mode

The most problematic SPI task is ***handling of Half-duplex mode***, which requires common single-data line shared between master and slave alternately for bidirectional transactions. This mode cumulates all the problematic handling listed previously at this chapter:

- **Handling exact number of data at particular simplex sessions**

  Terminating of master-receiver working at Receive-only mode is critical when the ongoing continuous flow must be stopped correctly. When handling CRC pattern upend or when the session is not aligned with the configured data threshold both data-flow directions require a specific control of the correct number of transacted data at the end of the flow.
- **Checking the end of the bus activity of a particular session**

  The bus has to be reconfigured between sessions in order to change direction of the data line, initialize CRC or configure the next session parameters. To do the configuration, SPI has to be temporary disabled. This disable might corrupt the end of the ongoing session, this is why the end of the bus activity has to be properly detected before disabling the SPI.
- **Reconfiguration of the bus between sessions**

  A change of the data-line direction must be synchronized on master and on slave sides, but these reconfigurations are independent, hence not fully synchronized. The reconfiguration timing is affected by how the involved nodes recognize the end of the bus activity and by the latencies of their SPI reconfiguration services. One node can initialize its GPIO output at common data-line while the opposite node is not yet able to manage the reconfiguration. Then it could happen that both GPIOs outputs propagate different output levels and compete for the line at the same time for a sure period (it depends on data being transacted and on the line clock phase setting). It is highly recommended to implement a serial resistor at this common data-line between the nodes. This action prevents a temporal short connection between the outputs and limits current blowing between them when just opposite levels are matching on the line.
- **Control of associated GPIOs to keep nodes synchronized**

  The necessity of the peripheral reconfiguration within the session requires an SPI disable. It brings additional effects like flushing the FIFO content or control loos of the associated GPIOs. Continuity of SPI signals during reconfiguration of the bus must be ensured to prevent any synchronization issue between the involved nodes. Master has to avoid any glitches on SCK and SS signal. These signals might have to be managed by software as a general purpose outputs during the interface reconfiguration phase. An interactive session often starts by a command (write) sequence sent by master followed by an adequate response (writer or read) sequence when slave receives or sends data matched with the command content (except for cases when the protocol between master and slave is fixed). The master must always give sufficient time to the slave to reconfigure the bus, to recognize the command and to prepare the adequate answer. It mostly requires to avoid a continuous SCK clock signal and insert an adequate pause between such a command and data phases.

The Half-duplex mode handling is very complex. An analysis must be done to determine if the saved single data-line and the associated GPIO pin output are worth the effort, as Full-duplex communication mode is easier to handle. In Full-duplex communication mode, there is no need of any reconfiguration of the peripheral and the user can focus on handling the data content at the required direction while the data at opposite direction (not monitored) is either ignored or placed dummy.

4.3 Handling specific SPI signals

This section presents more details on handling specific internal-signals, interfaces and handling optional signals of the SPI interface

4.3.1 Handling specific internal signals and interfaces

In older SPI versions the DMA overtakes partial control over SPI functionalities due to the lack of transaction counters at SPI side, therefore there is a wide interface between DMA and SPI. The most recent SPI versions fully overtakes the control while the simplified DMA interface handles only the data transfers, hence the number of data involved at a session at SPI side must be correctly setup and well balanced on both the SPI and DMA side, especially when enhanced SPI modes discussed at previous chapter are involved. Note that all the DMA access is based on the TXP and RXP or EOT data threshold events. If an event is missing (for example when incomplete data packet is done), no DMA transfer is performed.

On recent SPI versions, the Simplex SPI modes are strictly separated. There is no longer needed to manage the unused RxFIFO flushing nor to handle its OVR errors. DMA channel enable must be prevented at the unused direction.

When SPI is temporarily disabled, the associated GPIOs configured at alternate function output-mode at master side have to stay at defined inactive levels. This need creates a problem on old SPI versions, as the peripheral does not handle them and leaves them undefined and floating.

Once the SPI is enabled, it may propagate active internal signal levels (such as empty *transmit buffer status*). This propagation may force a premature DMA stream request or may create series of initial interrupt services (if enabled at SPI configuration). Even a system without any FIFO could raise two consequent data requests when SPI is enabled and data is released from the buffer once the first bit is transacted out. The buffer raise the next data write request immediately after the transaction starts (refer to Figure 2).

The SPI enable sequence described at the corresponding product's reference manual must be followed to ensure a proper events processing and to avoid missing any event service or a dysfunctional initialization of DMA streams.

Some signals can be evaluated even with significant latency if dual-clock domain is supported. For example, data underrun at slave side is evaluated under peripheral domain SCK, in consequence some traffic on the bus is necessary to evaluate the status (few additional SCK clock signal cycles are needed). This additional traffic makes the event to appear considerably later. Undesirable effects like unexpected data insertion may then occur once the underrrun status is detected or cleared.

There is an internal synchronization between the APB bus clock and the kernel clock. If there is a big difference between both clocks, some system responses may be processed with significant delay, causing unavailability of some features (refer to Section 2.2 SPI frequency constraints).

Most of the SPI instances can wake up the system from low-power modes. If dual-clock domain is available, SPI can autonomously communicate at low-power mode while all the necessary internal clock requests are automatically provided at dependency on applied SPI configuration and actual FIFO status.

Recent SPI versions can autonomously initialize transactions by system triggers. This action enables a data transfer synchronization with other peripheral events without system wake up. Refer to the specific device's reference manual for more details.

4.3.2 Handling optional signals

Slave select (NSS/SS) is an optional SPI interface signal. This signal is useful at multi slave or multi master topologies especially when it assigns just a single slave node for communication at time. This signal is useful even at single master/slave systems as it permits to separate and synchronize transactions between both nodes.

Hardware support of the NSS master output at standard SPI Motorola mode is not ideal on old SPI versions. The signal logic copies the SPE status and causes problems to handle its GPIO alternate function. When the SPI is disabled, the signal is no longer propagated onto the associated GPIOs causing its floating once the SPE bit is set to zero.

On recent SPI versions, control over the alternate function is maintained even if the SPI is disabled. Also it is possible to select the polarity of the pin and to swap most of the MOSI and MISO signals capabilities. Recent SPI versions only support single NSS/SS pin. This alternate function has to be replaced by standard GPIO(s) control when handling a multi slave system or when a specific communication requires a reconfiguration of SPI within a single session (such at Half-duplex mode when the SPI has to be disabled temporarily while SS needs to stay active).

Recent SPI versions feature SS input edge selective when Hardware mode is applied (SMM=0). A session can be missed when SPI is enabled after the master has provided the active edge of the SS signal. An optional RDY signal is featured in some SPI versions, which performs the stave's FIFOs occupancy status. If the master disregards RDY and continues the transaction, the slave risks an underrun or overrun condition. On STM32 devices, the master communication is temporarily frozen when the slave signalize a not-ready status.

4.4 SPI instances with different features

STM32 devices implement the same peripherals but sometimes with different instances configurations. Some current differences are:

- Maximum supported data size and its configurability (by bit or by multiple of bytes)
- Maximum supported polynomial or frame CRC size
- CRC configurability (by bit or by multiple of bytes)
- FIFOs capacity

- I2S support availability
- Maximum number of data at single session
- Low-power mode operation capabilities

The detailed differences between devices are available at the SPI implementation table at the product’s reference manual.

4.5 SPI synchronization between nodes

SPI is a synchronous interface and it demands proper synchronization between the nodes to ensure a correct data handling. The only synchronization signals featured by SPI are NSS/SS, hence the user must be vigilant of any desynchronization symptoms.

Possible causes of desynchronization are a glitch on SCK line due to signal disturbance or a wrong handling of SCK alternate output during SPI reconfiguration.

Despite BSY flag can hang exceptionally, monitoring of its clearing can be helpful when communication becomes idle. If it stays set, it can just signalize possible synchronization issues between master and slave.

Correct data shifting by one or more bits and CRC errors are consequence of a synchronization problem. In these cases, the bus must be inspected to find the cause of the abnormality of the signals and to analyze the transacted data to find the root cause. If a desynchronization occurs, both the master and the slave nodes must be restarted.

As it is already noted, SPI disable is strongly suggested to be applied between the sessions anyway. This disable resets the internal state machine processes and corrects any desynchronization problems if present.

Revision history

Table 3. **Document revision history**

Date	Version	Changes
30-Nov-2020	1	Initial release.

Contents

List of tables

List of figures