PIC32MX SPI Library Documentation

Revision 1.0

Luka Gacnik

November 2022

Table of contents

1 Library description **1**

1.1 About the SPI 1

1.2 Library features 4

2 Dependencies **5**

3 Notes and warnings **7**

4 Future ideas **8**

5 Custom types **9**

5.1 SpiStandardConfig_t 9

5.2 SpiPin_t 10

5.3 SpiSsState_t 11

5.4 SpiSfr_t 11

6 Function description **12**

6.1 SPI_ConfigStandardModeSfr() 13

6.2 SPI_EnableSsState() 15

6.3 SPI_DisableSsState() 17

6.4 SPI_SetSsState() 19

6.5 SPI_GetSsState() 21

6.6 SPI_IsSpiBusy() 22

6.7 SPI_IsRxDone() 23

6.8 SPI_MasterReadWrite() 24

6.9 SPI_MasterWrite() 26

6.10 SPI_MasterWrite2() 28

6.11 SPI_DummyRead() 30

6.12 SPI_SlaveWrite() 31

6.13 SPI_SlaveWrite() 33

7 Examples **35**

7.1 Multi-packet 8-bit Master transmission 35

7.2 Multi-packet 32-bit Master transmission 36

7.3 Multi-packet 32-bit Slave reception 39

A Time diagrams **41**

A.1 Master write with polling 41

A.2 Master write with interrupts 42

B Revision history **43**

Table of figures

1.1 Independent Slave configuration 2

1.2 SPI module block diagram of the PIC32MX series 3

A.1 Polling-based Master mode operation 41

A.2 Interrupt-based Master mode operation 42

Acronyms

SPI	Serial Peripheral Interface
TX FIFO	Transmit First In First Out buffer
RX FIFO	Receive First In First Out buffer
ISR	Interrupt Service Routine
PPS	Peripheral Pin Select
MCU	Microcontroller
GPIO	General Purpose Input Output
PIO	Programmable Input Output
SFR	Special Function Register

1 Library description

The library consists of basic functions that control the Serial Peripheral Interface module of an PIC32MX series of the Microchip microcontrollers.

1.1 About the SPI

The Serial Peripheral Interface (SPI) module is a synchronous serial interface useful for communicating with external peripherals and other microcontroller devices. In case of a microcontroller it is embedded into its hardware as one of its peripheral modules.

SPI communication is a full duplex mode operation where one or multiple Master devices control one or multiple Slave devices. The standard SPI bus uses four-wire setup that specifies four logic signals:

- **SCK** - Serial Clock (controlled by Master device)
- **MOSI** - Master Out Slave In (data output from Master)
- **MISO** - Master In Slave Out (data output from Slave)
- **SS** - Slave Select (controlled by Master device, often follows active low logic)

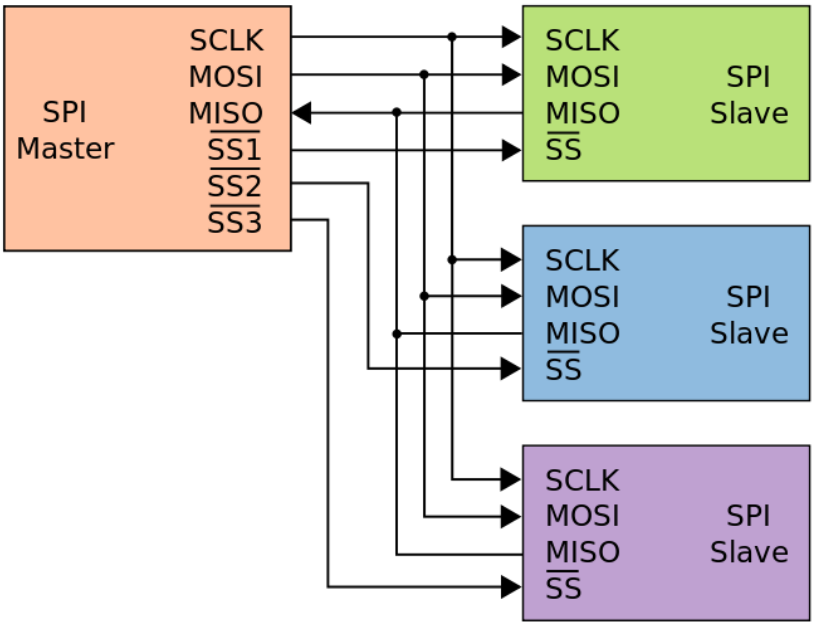

Figure 1.1: Independent Slave configuration

The full duplex mode enables simultaneous data exchange between a Master and a Slave device - as Master completely shifts out TX data, a Slave feeds the RX data into Master device shift register. Such operation makes serial communication fast and efficient. The following facts apply to a generalized SPI device:

- The SS signal is used for a Master-Slave synchronisation purpose and is generally required by any kind of a Slave device
- SPI can be configured into four different modes that correspond to altered clock polarity and phase. This is useful for accommodating the communication requirements of many SPI devices
- SPI signal nodes are based on push-pull drivers
- Different possible data width (usually either one or more of 8/16/24/32-bit width)
- No Slave addressing (like I2C), no acknowledgement (like UART), no error-checking protocol

The library is based on the PIC32MX MCU series. Some of the key features of its SPI module are:

- Master and Slave modes supported
- Four different modes
- Framed SPI protocol support
- Standard and Enhanced Buffering modes
- User configurable 8/16/32-bit data width
- RX/TX FIFO buffers are 4/8/16 deep in Enhanced Buffering mode
- Programmable interrupts
- Audio Protocol Interface mode

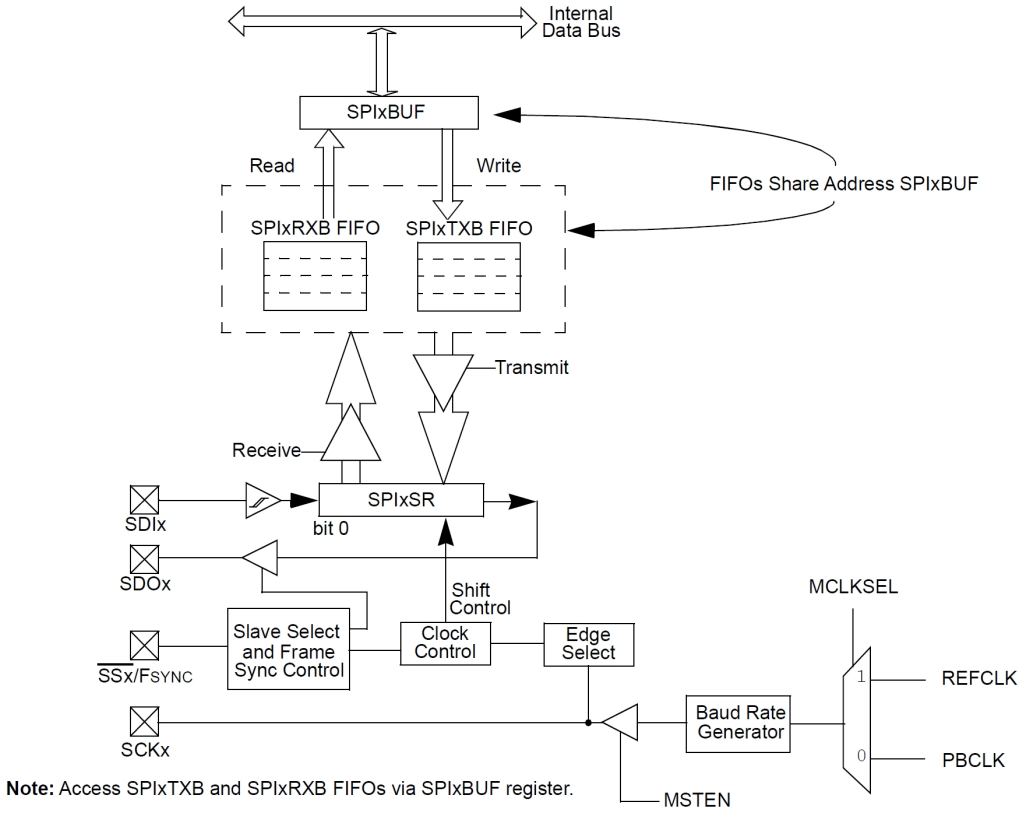

Figure 1.2: SPI module block diagram of the PIC32MX series

1.2 Library features

Currently, the SPI driver is capable of performing the following operations:

- Configuring the selected SPI module for the standard (non-audio mode) SPI Master operation
- Performing polling-based SPI write and read operation which offers the fastest execution
- Performing interrupt-based SPI write or read operation which results in the highest program execution efficiency
- Performing interrupt-based special SPI write or read operation where a user-defined function is executed at the end of an SPI operation

2 Dependencies

The library consists of the following source and header files:

- `Spi.c`
- `Spi.h`
- `Spi_sfr.h`

All the external user-accessible function prototypes are provided in the `Spi.h` file. Their definitions and other local functions like ISR handlers are contained in the `Spi.c` file. The file `Spi_sfr.h` contains all the macro definitions for register bit-fields and SFR memory layout type.

The library depends on the following other libraries:

- `Pio.h`
- `Ic.h`
- `Osc.h`

The library `Pio.h` provides the functions for modifying each signal's pin settings, and for Slave Select control. Interrupts are configured using the `Ic.h` library. Lastly, the library `Osc.h` is used only for a purpose of the proper baud rate generator configuration. As a side note, for the PIC32MX MCUs the Peripheral Pin Select (PPS) module must be configured for a proper SPI operation within the scope of the `Pio.h` library. Such an action requires (un)locking a specific set of registers which is provided by the `Cfg.h` library and is already contained within the `Pio.h` library.

Additionally, there needs to be one of the standard Microchip libraries included (since the compiler XC32 is the made by the Microchip), that is the `sys\attrib.h`

library. It is used for its ISR macro definition that is needed for ISR function attribute definition.

Lastly, its important to provide the fact that this library was meant to be used with a specific MCU - the PIC32MX170Bxxx series made by the Microchip. As previously mentioned the XC32 compiler was used to compile the library files, meaning there are some specifics in the library which are unique to the XC32 compiler (such as ISR function attribute). In case any other MCU of the PIC32 family would be used, some peripheral libraries (used by this driver) might need modifications (like different register sets, etc.).

3 Notes and warnings

There are few constrains which limit the capability of the nRF24L01 library:

a) **No support for the SPI multi-module simultaneous operation**

The library doesn't support the action of multiple SPI modules to be controlled by the same library at the same time. Doing so would require much more complex operation which is in fact not needed in most cases (from author's perspective). One operation for a certain SPI module must be concluded before the next one starts for any of the modules. Defying this limitation causes a software data collision and results in a faulty SPI operation. However, multiple modules can simultaneously stay configured.

One could bypass the limitation explained above by copying the same library under a different name and rename all the external functions – some other code would need to be renamed (such as the macros for interrupt priority control) or deleted to prevent re-definition.

b) **Independent Slave configuration only**

The only Slave configuration setup that is currently supported by this library is the Independent Slave configuration shown by the figure (1.1). There is no support for well-known Daisy chain setup. However, it is possible to simultaneously transmit data to multiple devices over the SPI bus by using one SS pin for all devices. In such a case it would only need to be ensured that only up to one Slave device responds, otherwise data collision will occur.

In addition, some code related facts must be known prior to using the SPI library:

a) **Interrupt priority of an SPI module**

Prior to any of the interrupt-based functions is used, the (sub)priority level of the corresponding SPI module's interrupt vectors must be defined. This is done via providing a constant value using macros `SPIx_ISR_IPL`, `SPIx_ICX_IPL`, and `SPIx_ICX_ISL` (where lowercase `x` can be in the range of `1` to `2`).

4 Future ideas

Here are some possible development ideas for the future development:

- Re-format the library in a way that it would support simultaneous operation of multiple SPI modules (possible use of dynamic memory allocation or creation of run-time variable names in C++)
- Implement configuration and operation functions for the Audio Protocol Interface mode

5 Custom types

The library comes with a set of custom made structure or enumeration types. For the sake of convenience these types are explained throughout this chapter. Note that individual enumeration types are not covered here because they (and their set of values) follow self-explanatory naming conventions.

5.1 SpiStandardConfig_t

This structure-based type contains the properties for configuration of the selected SPI module. The properties define the key parameters which the SPI operation depends on. The type consists of the following properties:

- **isMasterEnabled** (*Bool_t*) - defines whether an SPI module operates in either Master or Slave mode
- **pinSelect** (*SpiPin_t*) - a structure that stores information about the selection of SPI related pins which control the SPI bus
- **frameWidth** (*SpiFrameWidth_t*) - sets the desired data width
- **clkMode** (*SpiClkMode_t*) - sets the desired SPI mode which reflects to one of the four standard clock polarity and phase setups
- **sckFreq** (*uint32_t*) - the clock frequency of the SPI bus (used only in Master mode)

Not each and every `sckFreq` value can be configured. If uncertain whether a frequency will be set to the user-defined value, actual frequency to be configured can be verified through formula $f_{sck} = f_{pbclk}/(2 \cdot (BAUD + 1))$, where $BAUD$ can be a natural number in the range 0 to 511. The highest frequency $f_{max} = f_{pbclk}/2$ and the lowest frequency $f_{min} = f_{pbclk}/1024$ limit the user-defined frequency f_{sck}. If user-defined value results in non-natural $BAUD$ value, it's fractional part is truncated.

5.2 SpiPin_t

This structure-based type contains the properties for configuration SPI related pins which control an SPI bus. It is intended for the pin codes from the `Pio` library to be used. These pins can be configured:

- **`sdiPin`** (*`uint32_t`*) - the Serial Data In (or MISO) pin
- **`sdoPin`** (*`uint32_t`*) - the Serial Data Out (or MOSI) pin
- **`ssxPin`** (*`uint32_t`*) - the Slave Select (or CS) pin (where **`x`** can be in a range of **`1`** to **`5`**)

Note that if more Slave devices are desired to be used, the user can manually add more SS pin members to the existing structure *`SpiPin_t`* type. Additionally, note that whenever an MCU is selected to operate in the Slave mode, only the `ss1Pin` should be configured. The SCK pin doesn't need manual selection because it is always set at a fixed pin location for the PC32MX170xxx MCU series.

Additionally, there are two specifics about the possible `ssxPin` values:

- Master mode - possible parameters are the GPIO pin codes which are identified through the `GPIO_RPyn` notation (where `y` can be either `A` or `B` and `n` can be any number in the range of `0` to `15`)
- Slave mode - possible parameters are the SS pin codes which are identified through the `SSx_RPyn` notation (where `x` can be either `1` or `2`, `y` can be either `A` or `B`, and `n` can any number in the range of `0` to `15`)

The `x` identifier refers to either of SPI modules, the `y` identifier refers to either of PIO modules (ports), and the `n` refers to any of pin positions, which are available for the PIC32MX170xxx MCU series.

The same naming conventions are used for SDI and SDO pin codes that can be easily found in the `Pio_sfr.h` file.

5.3 SpiSsState_t

This structure-based type contains the properties for configuration Slave devices which will be active during the SPI operation. The following properties can be set:

- **pioA** (*uint32_t*) - 32-bit state of the output LATA register which will be active during the SPI transmission
- **pioB** (*uint32_t*) - 32-bit state of the output LATB register which will be active during the SPI transmission

Note that this type is used with functions SPI_GetSsState() or SPI_SetSsState() where multiple SS pins are desired to be enabled at once. Primarily, the functions SPI_EnableSsState() and SPI_DisableSsState() are intended for use where individual Slave devices are enabled/disabled per function call.

5.4 SpiSfr_t

This structure-based type contains the properties for modification of SPI related SFRs. Throughout this library it is mostly used as a pointer to a base address of any SPI module. MCU memory contents starting at such a specific base address are modified when changing the value of any member that is contained by this structured type. The way of each member is positioned in the structure is strictly defined and shall not be modified by the user. Member layout defines the exact position of SPI SFRs, relative to any of SPI base address.

6 Function description

The library includes the following functions:

- `SPI_ConfigStandardModeSfr()`
- `SPI_EnableSsState()`
- `SPI_DisableSsState()`
- `SPI_GetSsState()`
- `SPI_SetSsState()`
- `SPI_IsSpiBusy()`
- `SPI_IsRxDone()`
- `SPI_MasterReadWrite()`
- `SPI_MasterWrite()`
- `SPI_MasterWrite2()`
- `SPI_DummyRead()`
- `SPI_SlaveWrite()`
- `SPI_SlaveRead()`

The following sub-sections describe the use and operationality of these functions.

6.1 SPI_ConfigStandardModeSfr()

PROTOTYPE

```
SPI_ConfigStandardModeSfr(spiSfr, spiConfig)
```

DESCRIPTION

Configures SPI pins and SPI related SFRs for standard (non-audio) operation.

PARAMETERS

- **spiSfr** (*SpiSfr_t* ∗) - a pointer to the base address of either of the available SPI modules
- **spiConfig** (*SpiStandardConfig_t*) - contains the settings which determine operation parameters of the desired SPI operation

RETURNS

Returns `false` when either of the following is true:

- SPI base module address is incorrect
- SPI library is handling some other operation at the time of this function call
- Slave mode is selected but the `ss1Pin` is not configured
- SDO and SDI pins aren't configured in any mode

Otherwise it returns `true`.

OPERATION

Initially, two basic safety checks are done before the function continues. Then all SPI registers for the selected module are reset. This is followed by enabling/disabling any of SDI/SDO/SS pins if necessary. Afterwards, the SPI related SFRs are configured. Then the Programmable Input/Output (PIO) SFRs are configured. This process disables interrupts for a brief period. In the Master mode the baud rate generator register is written to. In case the currently modified SPI module is being re-configured, the RX FIFO is flushed. Lastly, interrupt SFRs are configured and interrupts enabled.

NOTES

- The interrupt priority related macros (located in the `Spi.h` file) must be set to the desired values prior to calling this function. If a certain module isn't used then modifying the interrupt priority macros for that specific module isn't required
- When configuring an SPI module for the Slave mode, the `ss1Pin` needs to be set. Note that if any other Slave Select member is set it will be ignored during configuration of the PIO SFRs. Or, if the `ss1Pin` isn't set then this function returns `false`
- Refer to the section (5.2) for entering an appropriate value for the `pinCode` parameter

LIMITATIONS

Not each and every SCK signal frequency setting can be configured. If uncertain whether a frequency will be set to the user-defined value, actual frequency to be configured can be verified through formula $f_{sck} = f_{pbclk}/(2 \cdot (BAUD + 1))$, where $BAUD$ can be a natural number in the range 0 to 511. The highest frequency $f_{max} = f_{pbclk}/2$ and the lowest frequency $f_{min} = f_{pbclk}/1024$ limit the user-defined frequency f_{sck}. If user-defined value results in non-natural $BAUD$ value, it's fractional part is truncated.

6.2 SPI_EnableSsState()

PROTOTYPE

`SPI_EnableSsState(pinCode)`

DESCRIPTION

Enables a specific Slave device on the SPI bus which is activated during any operation on the SPI bus. Slave Select pins that are not enabled by this function won't be activated during the SPI transmission.

PARAMETERS

- **`pinCode`** (*`uint32_t`*) - pre-defined pin code that notifies the PIO related functions about the current pin information. In case of non-GPIO pins information such as pin module, input/output, pin position, PPS code, and PPS register offset are encoded into the macro definitions which are provided by the `Pio_sfr.h` file

RETURNS

Returns `false` if false pin code was passed into the function, otherwise it returns `true`.

OPERATION

Pin code, whose value holds information about multiple pin related parameters, is passed to the `PIO_ReadPinCode()` function from the PIO library. This function decodes all the user-accessible information about the specific pin. Then, a specific bit is set for a specific PIO module related member in the structure `ssStatus`. This variable, private to the `Spi.c` source file, is copied into the PIO related SFRs which sets or clears enabled outputs during any operation on the SPI bus.

NOTES

- This function is intended for enabling one Slave device (which will be active during SPI operation) at a time. If enabling multiple Slave devices per function call is preferred, use the `SPI_SetSsState()` function

- Refer to the section (5.2) for entering an appropriate value for the `pinCode` parameter
- Do not modify the state of enabled/disabled Slave devices during any activity on the SPI bus

LIMITATIONS

None.

6.3 SPI_DisableSsState()

PROTOTYPE

`SPI_DisableSsState(pinCode)`

DESCRIPTION

Disables a specific Slave device on the SPI bus which is activated during any operation on the SPI bus. Existing Slave Select pins that are not disabled by this function will still be activated during the SPI transmission.

PARAMETERS

- **`pinCode`** (*`uint32_t`*) - pre-defined pin code that notifies the PIO related functions about the current pin information. In case of non-GPIO pins information such as pin module, input/output, pin position, PPS code, and PPS register offset are encoded into the macro definitions which are provided by the `Pio_sfr.h` file

RETURNS

Returns `false` if false pin code was passed into the function, otherwise it returns `true`.

OPERATION

Pin code, whose value holds information about multiple pin related parameters, is passed to the `PIO_ReadPinCode()` function from the PIO library. This function decodes all the user-accessible information about the specific pin. Then, a specific bit is set for a specific PIO module related member in the structure `ssStatus`. This variable, private to the `Spi.c` source file, is copied into the PIO related SFRs which sets or clears enabled outputs during any operation on the SPI bus.

NOTES

- This function is intended for disabling one Slave device (which will be active during SPI operation) at a time. If disabling multiple Slave devices per function call is preferred, use the `SPI_SetSsState()` function

- Refer to the section (5.2) for entering an appropriate value for the `pinCode` parameter
- Do not modify the state of enabled/disabled Slave devices during any activity on the SPI bus

LIMITATIONS

None.

6.4 SPI_SetSsState()

PROTOTYPE

`SPI_SetSsState(input)`

DESCRIPTION

Directly modifies the private variable `ssState` of the `Spi.c` file which is intended for Slave device enable/disable mechanism. In contrast to the `SPI_EnableSsState()` function, the current function allows multiple Slave devices to be enabled/disabled per function call.

PARAMETERS

- **`input`** (*`SpiSsState_t`*) - structure type variable where individual member corresponds to each of available PIO ports. Value of individual member's bits correspond to which outputs will be enabled during SPI operation

RETURNS

None.

OPERATION

Direct modification of the private variable `ssState` of the `Spi.c` file via externally accessible function. A mask `ssMask` is used to ignore any non-SS pins (the variable `ssMask` is reset and modified during the `SPI_ConfigStandardModeSfr()` function).

NOTES

- Setting any non-SS pins for either member of the `input` structure will not have any effect on the resulting `ssState` structure
- Do not modify the state of enabled/disabled Slave devices during any activity on the SPI bus

LIMITATIONS

None.

6.5 SPI_GetSsState()

PROTOTYPE

`SPI_GetSsState()`

DESCRIPTION

Copies the private variable `ssState` of the `Spi.c` file which is intended for Slave device enable/disable mechanism.

PARAMETERS

None.

RETURNS

Returns a structure of the type *`SpiSsState_t`*.

OPERATION

Returning a copy of the private variable `ssState` of the `Spi.c` file via externally accessible function.

NOTES

None.

LIMITATIONS

None.

6.6 SPI_IsSpiBusy()

PROTOTYPE

`SPI_IsSpiBusy(spiSfr)`

DESCRIPTION

Checks whether any SPI operation is in progress in the moment of function call.

PARAMETERS

- **`spiSfr`** (*`SpiSfr_t *`*) - a pointer to the base address of either of the available SPI modules

RETURNS

Returns `false` if any SPI operation is currently in progress (or a faulty SPI base address is passed), otherwise it returns `true`.

OPERATION

A status register of an SPI module is read to determine a status of an SPI bus. Additionally, all of the SPI interrupt source enable and flag bits are checked. This provides a safety mechanism which is triggered if the `SPI_IsSpiBusy()` is called after a certain interrupt flag is set but the ISR haven't been started yet due to some latency.

NOTES

None.

LIMITATIONS

None.

6.7 SPI_IsRxDone()

PROTOTYPE

`SPI_IsRxDone(spiSfr, rxbufSize)`

DESCRIPTION

Checks if the size of the RX FIFO buffer is equal or larger than the value of `rxbufSize`. This function is intended for status checking when operating in the Slave mode.

PARAMETERS

- **`spiSfr`** (*`SpiSfr_t *`*) - a pointer to the base address of either of the available SPI modules
- **`rxbufSize`** (*`uint32_t`*) - size of the RX FIFO buffer

RETURNS

Returns `false` if size of the RX FIFO buffer is less than the value of `rxbufSize` (or SPI module base address is false), otherwise returns `true`.

OPERATION

Initial SPI module base address check which is followed by checking of a specific bit-field in an SPI status register.

NOTES

None.

LIMITATIONS

None.

6.8 SPI_MasterReadWrite()

PROTOTYPE

`SPI_MasterReadWrite(spiSfr, rxPtr, txPtr, txSize)`

DESCRIPTION

Performs a polling-based SPI TX data write and RX data read. All data is handled within the function itself. RX data available after function completes.

PARAMETERS

- **spiSfr** (*SpiSfr_t* ∗) - a pointer to the base address of either of the available SPI modules
- **rxPtr** (*void* ∗) - a pointer to the memory address where the input RX data will be stored at
- **txPtr** (*void* ∗) - a pointer to the memory address where the output TX data is stored at
- **txSize** (*uint32_t*) - the size of a packet to be transmitted

RETURNS

Returns `false` when either of the following is true:

- SPI base module address is incorrect
- SPI library is handling some other operation at the time of this function call
- SPI module not configured for Master mode
- No Slave device is enabled
- Both of the passed pointers point to `NULL` or the `txSize` value is zero

Otherwise it returns `true`.

OPERATION

Initially, input safety checks are performed (factors listed above). Then the RX FIFO buffer is flushed and interrupts are temporarily disabled. Afterwards, the previously enabled Slave devices are now activated and the SPI write and read operation starts. Depending on the data width (8/16/32-bit width) a packet of either 16/8/4 bytes is loaded into TX FIFO buffer. Actual transmission begins after first FIFO element is loaded. An MCU then polls for busy status flag and reads RX data afterwards. If more data than the maximum size of the TX FIFO buffer needs to be transmitted of the SPI bus, new packet is loaded into the TX FIFO buffer and the process repeats. Lastly, SS pins are deactivated and interrupt state restored.

NOTES

- The amount of valid data words available at the `rxPtr` address is determined by the `txSize` value
- The amount of reserved space at the `rxPtr` address should be no less than the size of `txSize` words
- Interrupts are temporarily disabled during the TX FIFO buffer loading. When operation is completed, interrupt status is restored to the state prior to this function call
- If the `NULL` pointer is passed instead of `txPtr`, the SPI module transmits dummy TX data (value `0x00`)
- If the `NULL` pointer is passed instead of `rxPtr`, the SPI module dumps all RX data

LIMITATIONS

None.

6.9 SPI_MasterWrite()

PROTOTYPE

`SPI_MasterWrite(spiSfr, rxPtr, txPtr, txSize)`

DESCRIPTION

Performs an interrupt-based SPI TX data write. TX data is loaded by the function and if further TX data needs to be transmitted the ISR handler is triggered. All RX data is available for use after the `SPI_IsSpiBusy()` returns `false`.

PARAMETERS

- **`spiSfr`** (*`SpiSfr_t *`*) - a pointer to the base address of either of the available SPI modules
- **`rxPtr`** (*`void *`*) - a pointer to the memory address where the input RX data will be stored at
- **`txPtr`** (*`void *`*) - a pointer to the memory address where the output TX data is stored at
- **`txSize`** (*`uint32_t`*) - the size of a packet to be transmitted

RETURNS

Returns `false` when either of the following is true:

- SPI base module address is incorrect
- SPI library is handling some other operation at the time of this function call
- SPI module not configured for Master mode
- No Slave device is enabled
- Both of the passed pointers point to `NULL` or the `txSize` value is zero

Otherwise it returns `true`.

OPERATION

Initially, input safety checks are performed (factors listed above). Then the RX FIFO buffer is flushed and interrupts are temporarily disabled. Afterwards, the previously enabled Slave devices are now activated and the TX FIFO buffer is loaded. Actual transmission begins after first FIFO buffer element is loaded. Before the function is terminated, interrupts are re-enabled.

After last element is shifted out of the SPI shift register an ISR is triggered and the TX handler `ISR_SpiTxHandler_MasterWrite()` executes. First, if all TX data was transmitted, the currently activated Slave devices are disabled. Then RX data is read from the RX FIFO buffer. If more TX data needs to be send over the SPI bus, the TX FIFO buffer loading is initiated, interrupts are enabled, and the TX handler is terminated. After last element is shifted out of the SPI shift register, a new ISR is triggered and the process repeats.

NOTES

- The amount of valid data words available at the `rxPtr` address is determined by the `txSize` value
- The amount of reserved space at the `rxPtr` address should be no less than the size of `txSize` words
- Interrupts are temporarily disabled during the TX FIFO buffer loading. When operation is completed, interrupt status is restored to the state prior to this function call
- If the `NULL` pointer is passed instead of `txPtr`, the SPI module transmits dummy TX data (value `0x00`)
- If the `NULL` pointer is passed instead of `rxPtr`, the SPI module dumps all RX data

LIMITATIONS

None.

6.10 SPI_MasterWrite2()

PROTOTYPE

`SPI_MasterWrite2(spiSfr, rxPtr, txPtr, txSize, isrHandler)`

DESCRIPTION

Performs an interrupt-based SPI TX data write. TX data is loaded by the function and if further TX data needs to be transmitted the ISR handler is triggered. All RX data is available for use after the `SPI_IsSpiBusy()` returns `false`. An additional user-defined function is executed immediately after TX handler is terminated.

PARAMETERS

- **spiSfr** (*SpiSfr_t* ∗) - a pointer to the base address of either of the available SPI modules
- **rxPtr** (*void* ∗) - a pointer to the memory address where the input RX data will be stored at
- **txPtr** (*void* ∗) - a pointer to the memory address where the output TX data is stored at
- **txSize** (*uint32_t*) - the size of a packet to be transmitted
- **isrHandler** (*void (*f)()*) - a pointer to the user-defined function

RETURNS

Returns `false` when either of the following is true:

- SPI base module address is incorrect
- SPI library is handling some other operation at the time of this function call
- SPI module not configured for Master mode
- No Slave device is enabled
- Both of the passed pointers point to `NULL` or the `txSize` value is zero

Otherwise it returns `true`.

OPERATION

Operation is exactly the same as for the `SPI_MasterWrite()` function but with an additional step. A function pointer is configured to point to the passed user-defined function (user passes its name which identifies its memory address). It is executed after the TX handler `ISR_SpiTxHandler_MasterWrite()` is terminated (but before exiting an ISR function).

NOTES

- The amount of valid data words available at the `rxPtr` address is determined by the `txSize` value
- The amount of reserved space at the `rxPtr` address should be no less than the size of `txSize` words
- Interrupts are temporarily disabled during the TX FIFO buffer loading. When operation is completed, interrupt status is restored to the state prior to this function call
- If the `NULL` pointer is passed instead of `txPtr`, the SPI module transmits dummy TX data (value `0x00`)
- If the `NULL` pointer is passed instead of `rxPtr`, the SPI module dumps all RX data

LIMITATIONS

None.

6.11 SPI_DummyRead()

PROTOTYPE

`SPI_DummyRead(spiSfr)`

DESCRIPTION

Empties RX FIFO buffer of an SPI module. Intended for the Slave mode usage. It is redundant for the Master mode operations.

PARAMETERS

- **`spiSfr`** (*`SpiSfr_t *`*) - a pointer to the base address of either of the available SPI modules

RETURNS

Returns `false` if faulty base address is passed to this function via `spiSfr`, otherwise it returns `true`.

OPERATION

RX FIFO is completely emptied by reading its contents (if any) and data read is dumped.

NOTES

None.

LIMITATIONS

None.

6.12 SPI_SlaveWrite()

PROTOTYPE

```
SPI_SlaveWrite(spiSfr, txPtr, txSize)
```

DESCRIPTION

Loads TX FIFO buffer with data when SPI module operates in Slave mode.

PARAMETERS

- **spiSfr** (*SpiSfr_t* ∗) - a pointer to the base address of either of the available SPI modules
- **txPtr** (*void* ∗) - a pointer to the memory address where the output TX data is stored at
- **txSize** (*uint32_t*) - the size of a packet to be transmitted

RETURNS

Returns `false` when either of the following is true:

- SPI base module address is incorrect
- SPI library is handling some other operation at the time of this function call
- SPI module not configured for Slave mode
- The `txSize` value is zero

Otherwise it returns `true`.

OPERATION

Initially, input safety checks are performed (factors listed above). Then the TX FIFO buffer is loaded with data and the function is concluded.

NOTES

- Interrupts are temporarily disabled during the TX FIFO buffer loading
- If the `NULL` pointer is passed instead of `txPtr`, the SPI module transmits dummy TX data (value `0x00`)

LIMITATIONS

The maximum amount of elements stored per function call depend on the data width configured - the 8/16/32-bit data frame will have a limit of 16/8/4 bytes per TX FIFO buffer.

6.13 SPI_SlaveWrite()

PROTOTYPE

```
SPI_SlaveRead(spiSfr, rxPtr)
```

DESCRIPTION

Reads data from the RX FIFO buffer when SPI module operates in Slave mode.

PARAMETERS

- **spiSfr** (*SpiSfr_t* ∗) - a pointer to the base address of either of the available SPI modules
- **rxPtr** (*void* ∗) - a pointer to the memory address where the input RX data will be stored at

RETURNS

Returns `false` when either of the following is true:

- SPI base module address is incorrect
- SPI library is handling some other operation at the time of this function call
- SPI module not configured for Slave mode

Otherwise it returns `true`.

OPERATION

Initially, input safety checks are performed (factors listed above). Then the RX FIFO buffer is read and the function is concluded.

NOTES

The amount of reserved space at the `rxPtr` address should be the maximum RX FIFO buffer size (8/16/32-bit data frame results in 16/8/4 TX FIFO buffer size) if the size of the incoming packet is unknown

LIMITATIONS

None.

7 Examples

A set of examples is provided to demonstrate a simple operation of an SPI module. Examples provide only the main operation related code. Other processes such as the start-up of an MCU, programming of the device configuration register, setting up other peripheral modules, and similar is not provided here.

7.1 Multi-packet 8-bit Master transmission

This example provides the code for setting up an SPI module for the Master mode which carries out an 8-bit data frame transmission with a polling-based function.

An SPI structure with required user-defined settings is defined, SPI module base address is set, and SPI module configuration function is called. One slave pin is used and is enabled prior to SPI transmission. Lastly, transmission begins soon after the function `SPI_MasterReadWrite()` has been called. All the operations over SPI bus are executed within the scope of this function. This is why RX data is readily available for use immediately after this function is concluded.

```
/** Custom libs **/
#include "Spi.h"

int main(int argc, char** argv)
{
    /* Master device SPI settings */
    SpiStandardConfig_t spiMasterConfig = {
        .pinSelect = {
            .sdiPin = SDI2_RPB13,
            .sdoPin = SDO2_RPB2,
            .ss1Pin = GPIO_RPB10
        },
        .isMasterEnabled = true,
        .frameWidth = SPI_WIDTH_8BIT,
        .clkMode = SPI_CLK_MODE_0,
```

```
16          .sckFreq = 10000000
17      };
18
19      /* SPI module base address set */
20      SpiSfr_t *spiMasterSfr = &SPI2_MODULE;
21
22      /* Configure SPI module */
23      SPI_ConfigStandardModeSfr(spiMasterSfr, spiMasterConfig);
24
25      /* Test data variables */
26      uint8_t masterRxData[24] = {0};
27      uint8_t masterTxData[24] = {0xAB, 0x00, 0xCC, 0x00,
28                                  0xFF, 0xAB, 0x12, 0x21,
29                                  0xA0, 0x12, 0x00, 0xF1,
30                                  0xAB, 0x00, 0xCC, 0x00,
31                                  0xFF, 0xAB, 0x12, 0x21,
32                                  0xA0, 0x12, 0x00, 0xF1};
33
34      /* Configure which Slave will be addressed */
35      SPI_EnableSsState(spiMasterConfig.pinSelect.ss1Pin);
36
37      /* Polling-based SPI read and write */
38      SPI_MasterReadWrite(spiMasterSfr, masterRxData, masterTxData, 24);
39
40      while(1)
41      {
42          /* Main program execution */
43      }
44
45      return 0;
46  }
```

7.2 Multi-packet 32-bit Master transmission

This example provides the code for setting up an SPI module for the Master mode which carries out an 32-bit data frame transmission with a interrupt-based function.

Just like in previous case, an SPI structure and `SPI_2` module's base address is configured then passed into configuration function. In addition, the same steps are

taken for the `SPI_1` which is configured into Slave mode. Full TX FIFO buffer is loaded for the Slave module. Additional calls of the `SPI_SlaveWrite()` function aren't executed to re-load TX FIFO buffer of the Slave module, since two SPI operations cannot be simultaneously in progress due to nature of this library. Yet, the TX FIFO buffer of the Slave module generates random data afterwards, which is okay for this test example. Further code enables the Slave module and SPI write operation is initiated. Since this is an interrupt-based operation, a valid RX data is checked via the `SPI_IsSpiBusy()` function. RX data is available for use after the latter function returns `false`.

```
1  /** Custom libs **/
2  #include "Spi.h"
3
4  int main(int argc, char** argv)
5  {
6     /* Slave device SPI settings */
7     SpiStandardConfig_t spiSlaveConfig = {
8        .pinSelect = {
9           .sdiPin = SDI1_RPB8,
10          .sdoPin = SDO1_RPB11,
11          .ss1Pin = SS1_RPB7
12       },
13       .isMasterEnabled = false,
14       .frameWidth = SPI_WIDTH_8BIT,
15       .clkMode = SPI_CLK_MODE_0
16    };
17
18    /* Master device SPI settings */
19    SpiStandardConfig_t spiMasterConfig = {
20       .pinSelect = {
21          .sdiPin = SDI2_RPB13,
22          .sdoPin = SDO2_RPB2,
23          .ss1Pin = GPIO_RPB10
24       },
25       .isMasterEnabled = true,
26       .frameWidth = SPI_WIDTH_8BIT,
27       .clkMode = SPI_CLK_MODE_0,
28       .sckFreq = 10000000
29    };
30
```

```
31    /* SPI module base address set */
32    SpiSfr_t *spiSlaveSfr = &SPI1_MODULE;
33    SpiSfr_t *spiMasterSfr = &SPI2_MODULE;
34
35    /* Configure both SPI modules */
36    SPI_ConfigStandardModeSfr(spiSlaveSfr, spiSlaveConfig);
37    SPI_ConfigStandardModeSfr(spiMasterSfr, spiMasterConfig);
38
39    /* Test data variables */
40    uint32_t masterRxData[24] = {0};
41    uint32_t masterTxData[24] = {0xABACADAF, 0x00000000, 0xCCEEDDEE,
42                                 0x00110011, 0xFFF1FFF1, 0xABCDEF00,
43                                 0x12345678, 0x21314151, 0xA0A0A0A0,
44                                 0x12BB32CC, 0x00000000, 0xF1EBC43A,
45                                 0xABACADAF, 0x00000000, 0xCCEEDDEE,
46                                 0x00110011, 0xFFF1FFF1, 0xABCDEF00,
47                                 0x12345678, 0x21314151, 0xA0A0A0A0,
48                                 0x12BB32CC, 0x00000000, 0xF1EBC43A};
49    uint32_t slaveTxData[24] =  {0xF1DE5210, 0x12AB49CF, 0xA0015CD7,
50                                 0x12093472};
51
52    /* Load Slave with data */
53    SPI_SlaveWrite(spiSlaveSfr, slaveTxData, 4);
54
55    /* Configure which Slave will be addressed */
56    SPI_EnableSsState(spiMasterConfig.pinSelect.ss1Pin);
57
58    /* Interrupt-based SPI write */
59    SPI_MasterWrite(spiMasterSfr, masterRxData, masterTxData, 24);
60
61    while(1)
62    {
63       /* Main program execution */
64
65       if( SPI_IsSpiBusy(spiMasterSfr) )
66       {
67          /* RX data ready */
68       }
69    }
70
71    return 0;
72 }
```

7.3 Multi-packet 32-bit Slave reception

This example provides the code for setting up an SPI module for the Slave mode which handles out an 32-bit data frame reception.

Configuration of the Slave module proceeds in the same manner as in the previous example. However the code executed as part of the main program allows the Slave module to continuously re-load the TX data and therefore respond with valid user-defined data.

A simple example is provided just to demonstrate a way the available Slave functions can be used. This program should either be executed on a separate MCU or via duplicate library.

```
1  /** Custom libs **/
2  #include "Spi.h"
3
4  int main(int argc, char** argv)
5  {
6      /* Slave device SPI settings */
7      SpiStandardConfig_t spiSlaveConfig = {
8          .pinSelect = {
9              .sdiPin = SDI1_RPB8,
10             .sdoPin = SDO1_RPB11,
11             .ss1Pin = SS1_RPB7
12         },
13         .isMasterEnabled = false,
14         .frameWidth = SPI_WIDTH_8BIT,
15         .clkMode = SPI_CLK_MODE_0
16     };
17
18     /* SPI module base address set */
19     SpiSfr_t *spiSlaveSfr = &SPI1_MODULE;
20
21     /* Configure SPI module */
22     SPI_ConfigStandardModeSfr(spiSlaveSfr, spiSlaveConfig);
23
24     /* Test data variables */
25     uint32_t slaveRxData[24] =  {0};
```

```
26    uint32_t slaveTxData[24] =  {0xF1DE5210, 0x12AB49CF, 0xA0015CD7,
27                                 0x12093472, 0x001100AB, 0xF344BDEF,
28                                 0xEDC1B33A, 0x93A3B8CE, 0xFFF1FFF1,
29                                 0xABCDEF00, 0x12345678, 0x21314151,
30                                 0xF1DE5210, 0x12AB49CF, 0xA0015CD7,
31                                 0x12093472, 0x001100AB, 0xF344BDEF,
32                                 0xEDC1B33A, 0x93A3B8CE, 0xFFF1FFF1,
33                                 0xABCDEF00, 0x12345678, 0x21314151};
34
35    /* Load Slave with data */
36    SPI_SlaveWrite(spiSlaveSfr, slaveTxData, 4);
37
38    uint8_t ptrIncr = 1;
39
40    /* Main program execution */
41    while(1)
42    {
43       IC_DisableInterrupts();
44
45       /* Re-load Slave with data and read data */
46       if( SPI_IsRxDone(spiSlaveSfr, 4) )
47       {
48          SPI_SlaveRead(spiSlaveSfr, slaveRxData + ptrIncr * 4);
49          SPI_SlaveWrite(spiSlaveSfr, slaveTxData + ptrIncr * 4, 4);
50
51          ptrIncr++;
52       }
53
54       IC_EnableInterrupts();
55    }
56
57    return 0;
58 }
```

A Time diagrams

This appendix provides additional information in form of time diagrams in regards to sample codes elaborated in the chapter (7).

A.1 Master write with polling

The captured time diagram shown by the figure (A.1) demonstrates the results provided by sample code shown in the section (7.1). Uppermost channel shows the SCK signal which is followed by the MOSI and MISO signal. Lowermost channel shows the SS (CS) signal. Major events are notated by colored arrows and text.

Note that the ISR execution time is closely comparable with the execution time of transmitting one packet. This is because the frequencies f_{pbclk} and f_{sck} are set in ratio 4:1. The next example demonstrates 40:1 ratio where ISR timing is much less than packet transmitting which is desirable.

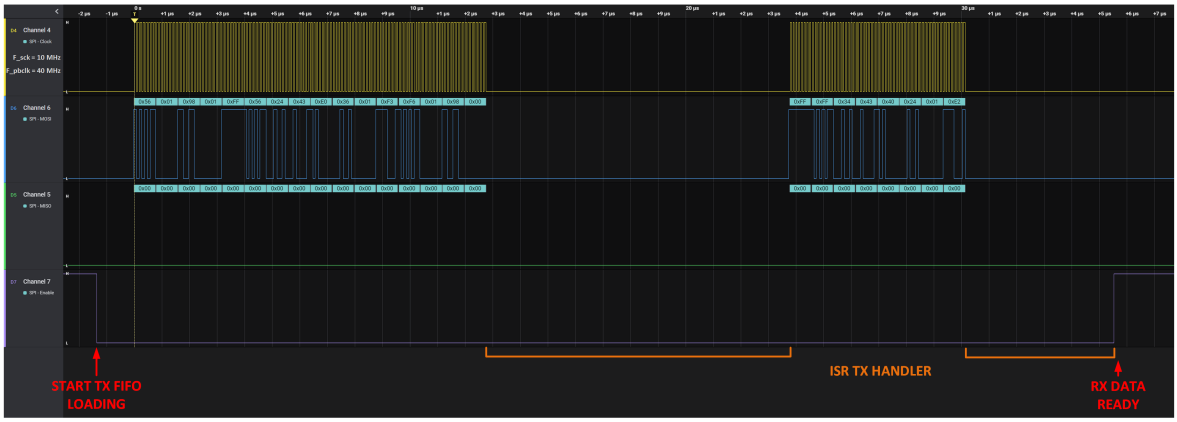

Figure A.1: Polling-based Master mode operation

A.2 Master write with interrupts

The captured time diagram shown by the figure (A.2) demonstrates the results provided by sample code shown in the section (7.2). Uppermost channel shows the SCK signal which is followed by the MOSI and MISO signal. Lowermost channel shows the main program execution (pin toggling) and interruptions of it. Major events are notated by colored arrows and text.

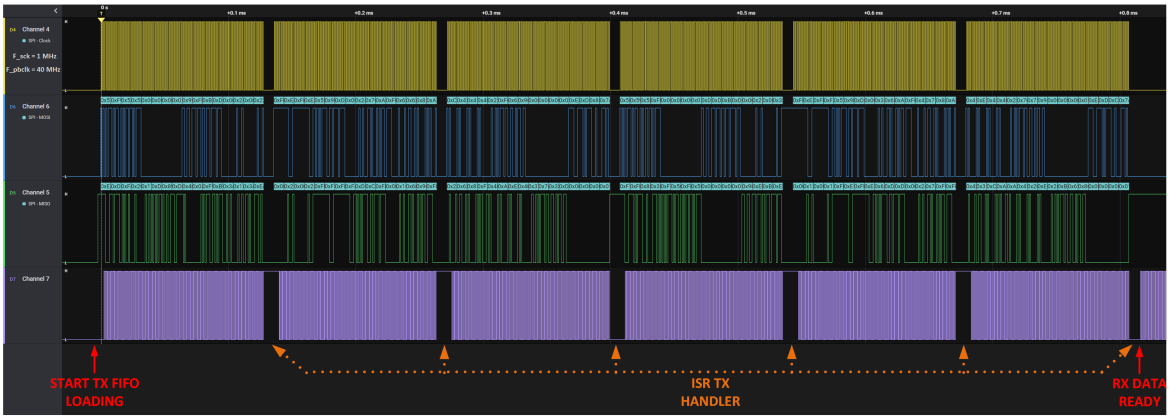

Figure A.2: Interrupt-based Master mode operation

B Revision history

Revision 1.0 (November 2022)

This is the initial released version of this document.