SPI interface

Guide

Visibility and dissemination of the document:

Can be widely distributed

Language: **C**
Language version: **C99**
Endianness: **N/A**

Synchronous: **yes**
Asynchronous: **yes**
OS: **need adaptation**

MCU compatibility:
no limit

Tests: **N/A**

MISRA C: **to be determined**
CERT C: **to be determined**

PBIT: **yes**
CBIT: **possible**

SPI interface

Version: 1.0.0 date: 02 Oct 2021

This SPI interface definitions for all the https://github.com/Emandhal drivers and developments.

Established by		Reviewed		Approved	
Name:	**FMA**	Name:	FMA	Name:	FMA
Date:	**02 Oct 2021**	Date:	02 Oct 2021	Date:	02 Oct 2021

Change history

Issue	Date	Nature / comment	Paragraph	Writer
1.0.0	02 Oct 2021	**Initial release**	-	FMA

Content

Figure summary

Table summary

Aucune entrée de table d'illustration n'a été trouvée.

1. INTRODUCTION

1.1. Purpose

The purpose of this document is to explain how to use the definitions to create SPI interfaces for https://github.com/Emandhal drivers and developments. All drivers that use SPI will use theses interfaces.

The interface:

- Can be use with any MCU (little or big endian)
- Only take care of the SPI interface entries
- All functions are identical which guarantees that all drivers will communicates with the SPI driver the same way

1.2. Documents, References, and abbreviations

1.2.1. Applicable documents

IDENTIFICATION	TITLE	DATE
-	-	-

1.2.2. Reference documents

IDENTIFICATION	TITLE	DATE
-	-	-

1.2.3. Abbreviations and Acronyms

This is the list of all the abbreviations and acronyms used in this document and their definitions. They are arranged in alphabetical order.

CBIT	Continuous Built-In Test
CERT	Computer Emergency Response Team
CLK	Clock
DMA	Direct Memory Access
MCU	Micro-Controller Unit
MISRA	Motor Industry Software Reliability Association
OS	Operating System
PBIT	Power Up Built-In Test
PIO	Programmable Input/Output
RAM	Random Access Memory
SPI	Serial Peripheral Interface

2. FEATURES

This interface has been designed to:

- Use only one entry for both synchronous and asynchronous (by polling for example) mode
- The driver can ask for an endianness transformation
- The driver can use the result of the interface to correct/change the transfer
- Create virtual SPI ports and link a driver to it

3. PRESENTATION

This interface is the entry point of an SPI driver. The driver will point to the SPI Initialization's function of the SPI driver to configure it, and the driver will point to the SPI Transfer's function of the SPI driver to transfer data.

Example with driver which communicate directly through a hardware SPI peripheral:

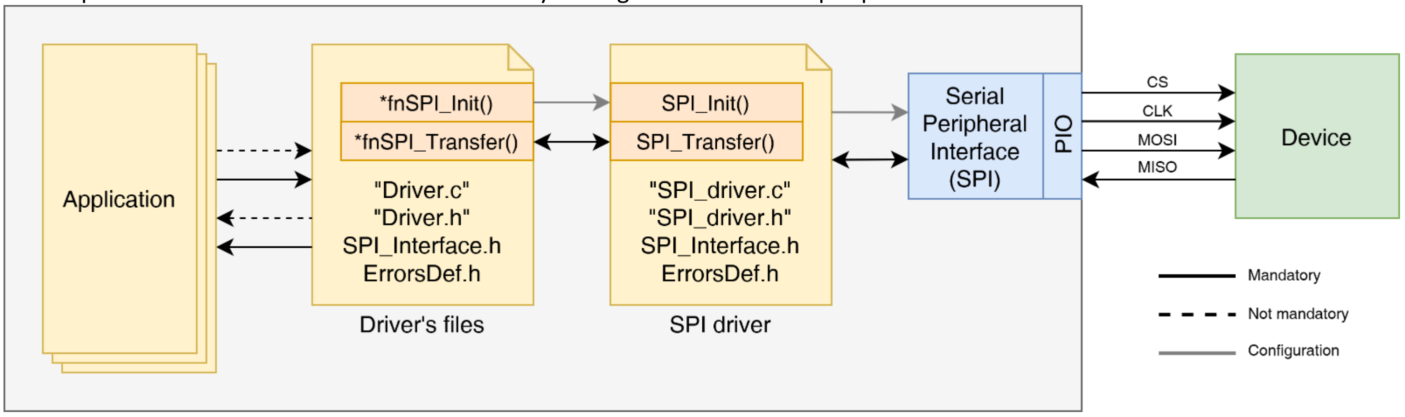


Figure 1 – Interface use with a hardware SPI peripheral

Example with driver which communicate through a software SPI interface:

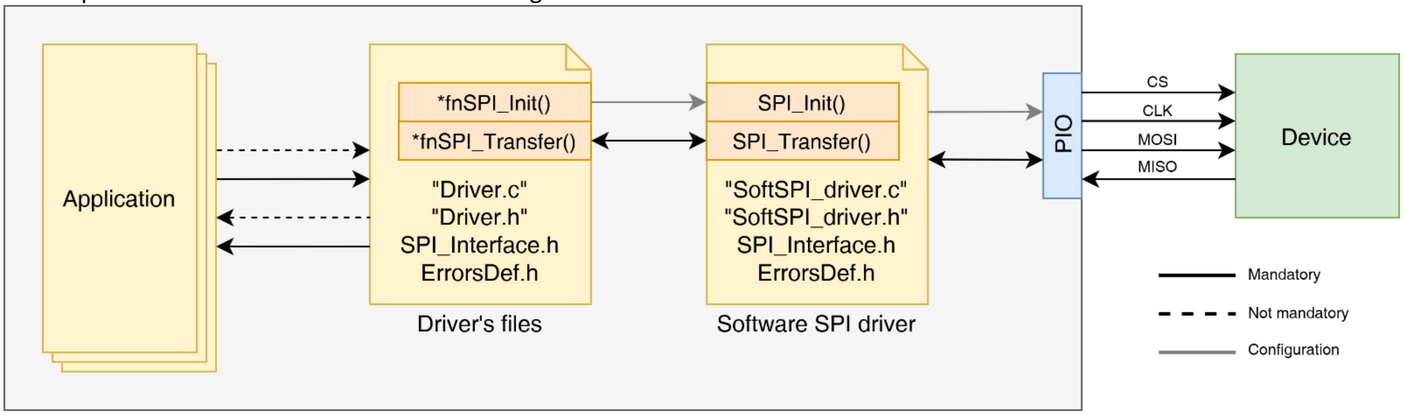


Figure 2 – Interface use with a software SPI interface

Example with a driver which communicate through a I2C to SPI device:

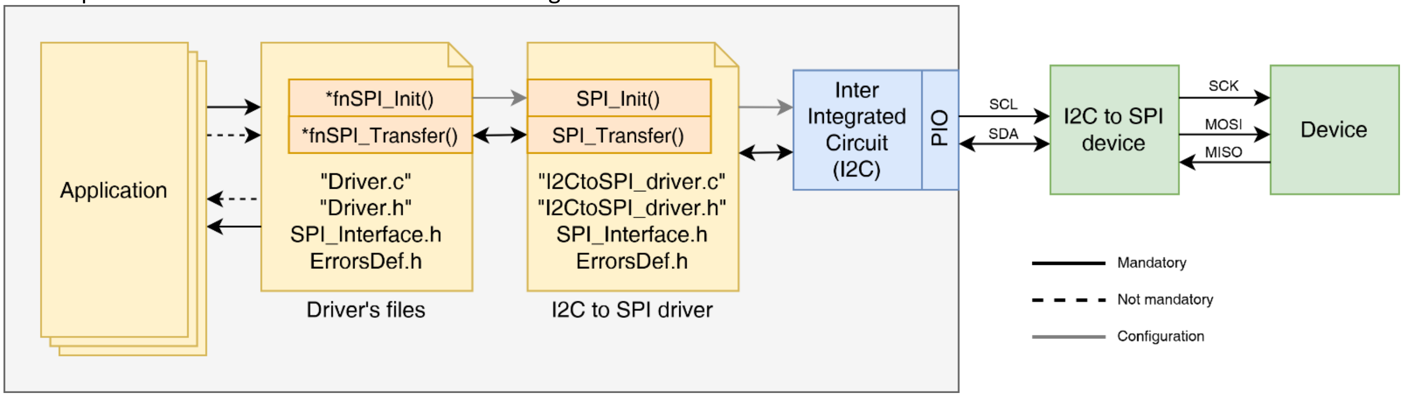


Figure 3 – Interface use with a I2C to SPI device

4. INTERFACE FILE CONFIGURATION

There is no Interface File specific configuration.

5. HOW TO DESIGN A DRIVER THAT USE THIS INTERFACE AS INPUT

Depending on device or peripheral that will be in use, the implementation will differ.

The SPIInterface_Packet.Config indicates to the driver what it can do with the packet to help the driver but may not be used. If not used, the SPI driver will be compatible with all of device's drivers. This configuration is design to be discarded without any problems.

On some device drivers some transformation can be done directly by the SPI driver to reduce the amount of CPU used by the driver, like asking the SPI driver to take care of the endianness transformation or the use of DMA because the driver work by polling.

5.1. Basic use of the SPI_Interface structure

In case of a basic use of a SPI peripheral, the SPIInterface_Packet.Config is not used, and it will be compatible with all of drivers. This configuration is design to be discarded without any problems. The driver design should look like this:

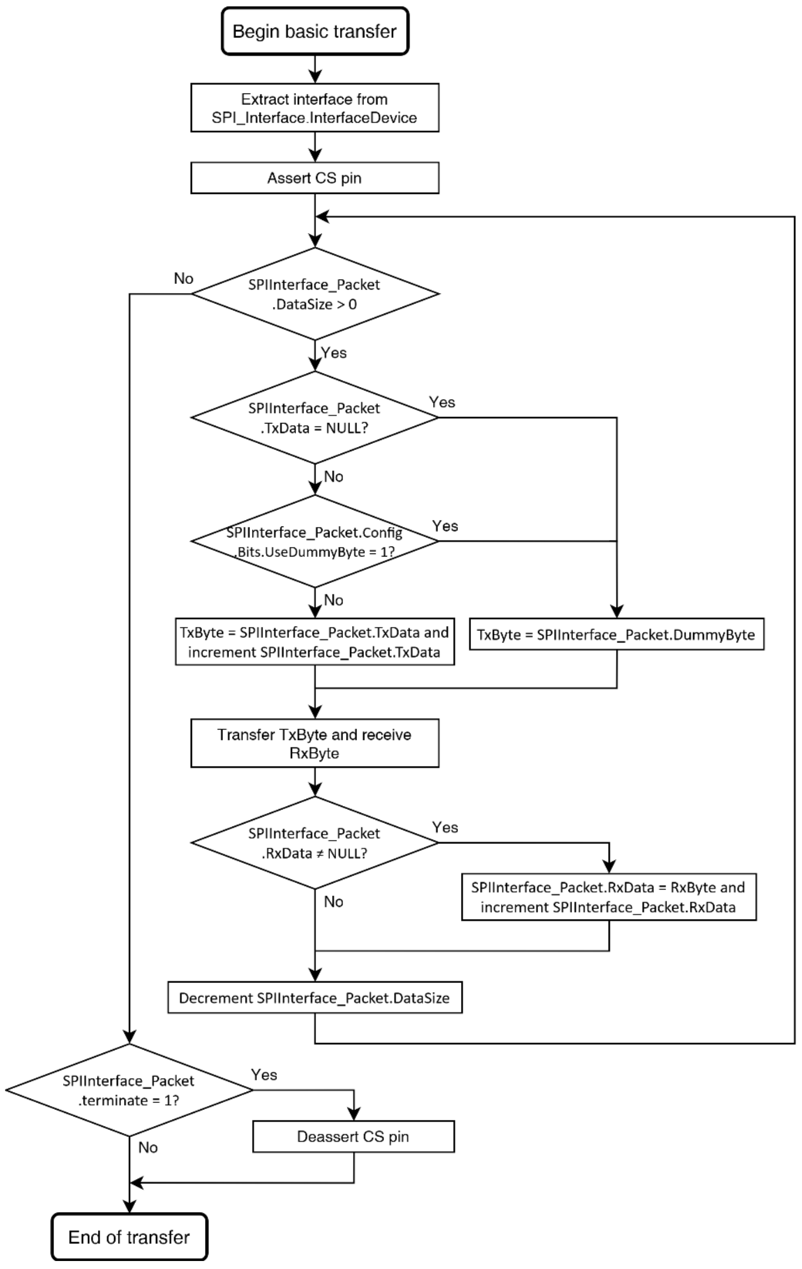

Begin basic transfer
Extract interface from SPI_Interface.InterfaceDevice
Assert CS pin
SPIInterface_Packet .DataSize > 0
No
Yes
SPIInterface_Packet .TxData = NULL?
Yes
No
SPIInterface_Packet.Config .Bits.UseDummyByte = 1?
Yes
No
TxByte = SPIInterface_Packet.TxData and increment SPIInterface_Packet.TxData
TxByte = SPIInterface_Packet.DummyByte
Transfer TxByte and receive RxByte
SPIInterface_Packet .RxData ≠ NULL?
Yes
No
SPIInterface_Packet.RxData = RxByte and increment SPIInterface_Packet.RxData
Decrement SPIInterface_Packet.DataSize
SPIInterface_Packet .terminate = 1?
Yes
No
Deassert CS pin
End of transfer

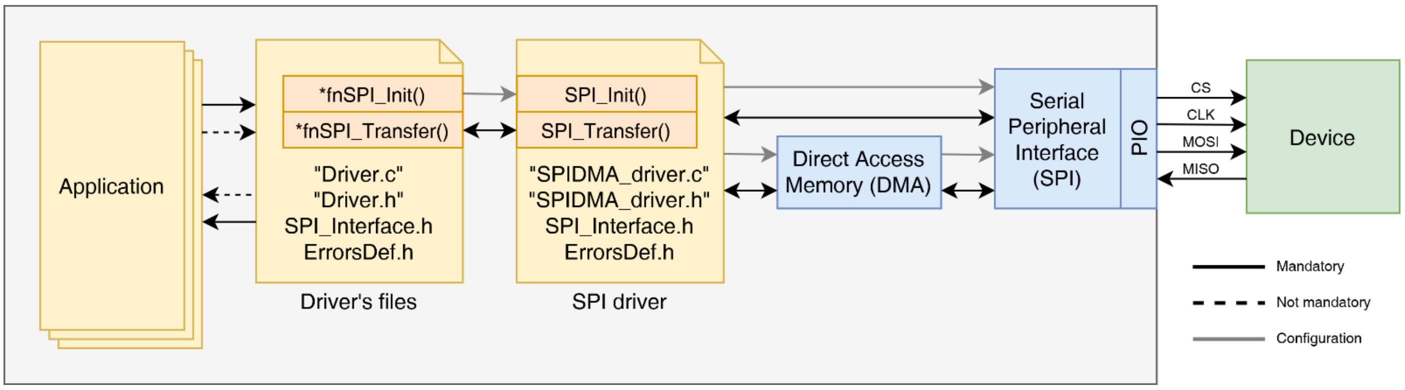
Figure 4 - Basic transfer diagram

5.1.1. Example

Example of a driver .c file:

```
//==============================================================================
// Soft SPI driver interface configuration
//==============================================================================
eERRORRESULT SoftSPI_SPIInit(SPI_Interface *pIntDev, uint8_t chipSelect, eSPIInterface_Mode mode, const uint32_t sckFreq)
{
  SoftSPI_Dev* pDevice = (SoftSPI_Dev*)(pIntDev->InterfaceDevice); // Get the Soft SPI device of this SPI port
  (void)chipSelect; (void)sckFreq; // Not used
  if (SPI_PIN_COUNT_GET(mode) > 1) return ERR__NOT_SUPPORTED;
  if (pDevice->IsConfigured && (mode == pDevice->Mode)) return ERR_OK;
  eERRORRESULT Error;

  //--- Pin configuration of the soft SPI ---
  CS_PIN_High;       // CS = 1
  CS_PIN_Out;        // CS out
  pDevice->IsAsserted = false;

  if (SPI_CPOL_GET(mode) == 0)
       SCK_PIN_Low;  // SCK = 0
  else SCK_PIN_High; // SCK = 1
  SCK_PIN_Out;       // SCK out

  MOSI_PIN_Low;      // MOSI = 0
  MOSI_PIN_Out;      // MOSI out

  MISO_PIN_In;       // MISO in

  pDevice->Mode = mode;
  pDevice->IsConfigured = true;
  return ERR_OK;
}



//==============================================================================
// Software SPI - Transfer data through an SPI communication
//==============================================================================
eERRORRESULT SoftSPI_SPITransfer(SPI_Interface *pIntDev, SPIInterface_Packet* const pPacketDesc)
{
  const bool UseDummyByte = ((pPacketDesc->Config.Value & SPI_USE_DUMMYBYTE_FOR_RECEIVE) == SPI_USE_DUMMYBYTE_FOR_RECEIVE);
  eERRORRESULT Error = ERR_OK;
  bool ForceTerminate = false;
  uint8_t DataToSend, DataRead;

  //--- Transfer data ---
  if (pDevice->IsAsserted == false)             // Start a transfer if not already done
  {
    Error = __SoftSPI_Start(pDevice);
    if (Error != ERR_OK) ForceTerminate = true; // If there is an error while starting the transfer then force terminate the transfer
  }
  if (ForceTerminate == false)
  {
    size_t RemainingBytes = pPacketDesc->DataSize;
    while (RemainingBytes > 0)
    {
      //--- Transmit byte ---
      if ((pPacketDesc->TxData != NULL) && (UseDummyByte == false))
      {
        DataToSend = *(pPacketDesc->TxData);
        ++pPacketDesc->TxData;
      }
      else DataToSend = pPacketDesc->DummyByte;
      //--- Transfer a byte ---
      Error = __SoftSPI_TransferByte(pDevice, DataToSend, &DataRead);
      if (Error != ERR_OK) { ForceTerminate = true; break; } // If there is an error while transferring a byte then force terminate
the transfer
      //--- Received byte ---
      if (pPacketDesc->RxData != NULL)
      {
        *(pPacketDesc->RxData) = DataRead;
        ++pPacketDesc->RxData;
      }
      --RemainingBytes;
    }
  }
  if (pPacketDesc->Terminate || ForceTerminate)
  {
    //--- Stop transfer ---
    eERRORRESULT ErrorStop = __SoftSPI_Stop(pDevice); // Terminate the SPI transfer
    Error = (Error != ERR_OK ? Error : ErrorStop);
  }
  return Error;
}
```

5.2. DMA use of the SPI_Interface structure



To know if the driver that ask for a transfer needs to use a DMA for this transfer, the user have to check if SPIInterface_Packet.Config.Bits.IsPolling is set to '1'. The driver design should look like this:

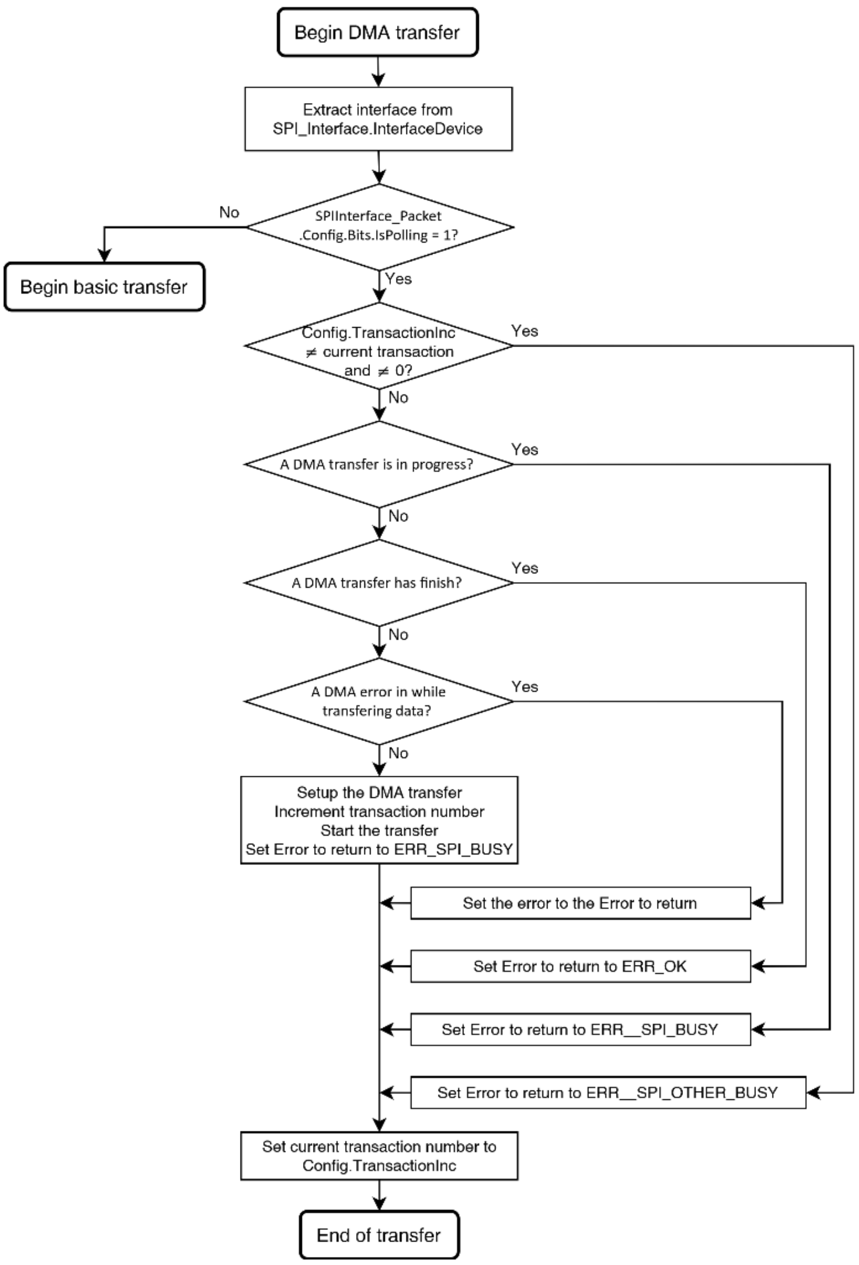

Begin DMA transfer
Extract interface from SPI_Interface.InterfaceDevice
SPIInterface_Packet .Config.Bits.IsPolling = 1?
No
Begin basic transfer
Yes
Config.TransactionInc ≠ current transaction and ≠ 0?
Yes
No
A DMA transfer is in progress?
Yes
No
A DMA transfer has finish?
Yes
No
A DMA error in while transfering data?
Yes
No
Setup the DMA transfer
Increment transaction number
Start the transfer
Set Error to return to ERR_SPI_BUSY
Set the error to the Error to return
Set Error to return to ERR_OK
Set Error to return to ERR__SPI_BUSY
Set Error to return to ERR__SPI_OTHER_BUSY
Set current transaction number to Config.TransactionInc
End of transfer

Figure 5 - DMA transfer diagram

5.3. Specific use of the SPI_Interface structure

There are others configurations to that some drivers can ask to adapt for specific use or to reduce CPU consumption.

5.3.1. Switch endianness

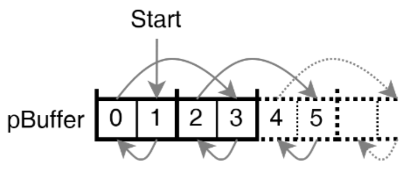
Some devices communicate with data with big-endian, on CPU that use little-endianness, the user or driver need to perform the switch of the endianness which consumes CPU time on big amount of data.

The SPIInterface_Packet.Config.Bits.EndianTransform indicate which transformation need to be performed. This transformation can be quickly applied on basic transfers with almost no CPU change. On DMA transfers, the transformation can be done only on complex DMA with data striding. At the end of transfer, set the SPIInterface_Packet.Config.Bits.EndianResult with the same value as SPIInterface_Packet.Config.Bits.EndianTransform to indicate to the driver that asks for this transfer that the endian transformation have been performed. If the transformation have not been performed, leave the SPIInterface_Packet.Config.Bits.EndianResult to 0 (SPI_NO_ENDIAN_CHANGE).

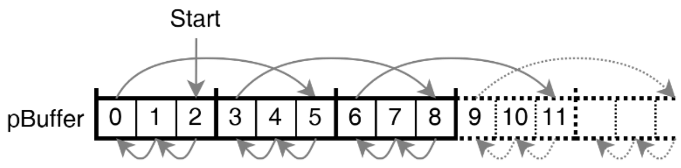
5.3.1.1. Switch endianness on basic transfers

First of all, the user shall verify that the size of buffer is a multiple of the byte size of the endian transformation. The byte size of the endian transformation can be extracted from the value of SPIInterface_Packet.Config.Bits.EndianTransform which is the block size of the transfer.

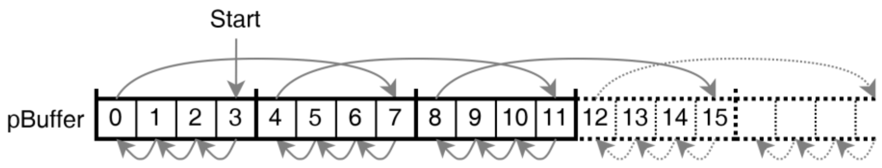
For SPI_SWITCH_ENDIAN_16BITS, this is how the address for the pBuffer should walk to perform the endianness transformation:



For SPI_SWITCH_ENDIAN_24BITS, this is how the address for the pBuffer should walk to perform the endianness transformation:



For SPI_SWITCH_ENDIAN_32BITS, this is how the address for the pBuffer should walk to perform the endianness transformation:



5.3.1.2. Switch endianness on DMA transfers

In this case, refer to the DMA controller of the device MCU/CPU datasheet. There is no specific algorithm to do that, all depends on the DMA controller and the DMA transfer configuration.

6. CONFIGURATION STRUCTURES

6.1. Interface object structure

The SPI_Interface object structure contains all information that is mandatory to communicate with a SPI peripheral.
Source code:

```
typedef struct SPI_Interface SPI_Interface;

struct SPI_Interface
{
  void *InterfaceDevice;
  SPIInit_Func fnSPI_Init;
  SPITransferPacket_Func fnSPI_Transfer;
  uint8_t Channel;
};
```

6.1.1. Data fields

void ***InterfaceDevice**

This is the pointer that will be in the first parameter of all interface call functions.

SPIInit_Func **fnSPI_Init**

This function will be called at driver initialization to configure the interface driver.

Type

typedef eERRORRESULT (*SPIInit_Func)(SPI_Interface *, uint8_t, eSPIInterface_Mode, const uint32_t)

Initial value / default

This function must point to a function else a ERR__PARAMETER_ERROR is returned by the driver's functions that need to use it.

SPITransferPacket_Func **fnSPI_Transfer**

This function will be called when the driver needs to transfer data over the SPI communication with the device.

Type

typedef eERRORRESULT (*SPITransferPacket_Func)(SPI_Interface *, SPIInterface_Packet* const)

Initial value / default

This function must point to a function else an ERR__PARAMETER_ERROR is returned by the driver's functions that need to use it.

uint8_t **Channel**

SPI channel of the interface device (This is not the ChipSelect).

6.1.2. Driver interface handle functions

```
eERRORRESULT (*SPIInit_Func)(
        SPI_Interface *pIntDev,
        uint8_t chipSelect,
        eSPIInterface_Mode mode,
        const uint32_t sckFreq)
```

Interface function for SPI peripheral initialization. This function will be called at driver initialization to configure the interface driver.

Parameters

Input	*pIntDev	Is the SPI interface container structure used for the interface initialization
Input	chipSelect	Is the Chip Select index to use for the SPI/Dual-SPI/Quad-SPI initialization
Input	mode	Is the mode of the SPI to configure
Input	sckFreq	Is the SCK frequency in Hz to set at the interface initialization

Return

Returns an *eERRORRESULT* value enumerator dependent of how the return error is implemented by the user in the SPI driver. It is recommended, during the implement of the pointer interface function, to return only errors listed in §6.3, and when all when fine, return ERR_OK.

```
eERRORRESULT (*SPITransferPacket_Func)(
       SPI_Interface *pIntDev,
       SPIInterface_Packet* const pPacketDesc)
```

Interface packet function for SPI peripheral transfer. This function will be called when the driver needs to transfer data over the SPI communication with the device.

Parameters

Input	*pIntDev	Is the SPI interface container structure used for the communication
Input	pPacketDesc	Is the packet description to transfer through SPI

Return

Returns an *eERRORRESULT* value enumerator dependent of how the return error is implemented by the user in the SPI driver. It is recommended, during the implement of the pointer interface function, to return only errors listed in §6.3, and when all when fine, return ERR_OK.

6.1.2.1. Enumerators

```
enum eSPIInterface_Mode
```

SPI bit width and mode.

Enumerator

SPI_MODE0	**0x01**	Communication with device with 1 bit per clock (Standard SPI mode 0)
SPI_MODE1	**0x41**	Communication with device with 1 bit per clock (Standard SPI mode 1)
SPI_MODE2	**0x81**	Communication with device with 1 bit per clock (Standard SPI mode 2)
SPI_MODE3	**0xC1**	Communication with device with 1 bit per clock (Standard SPI mode 3)
DUAL_SPI_MODE0	**0x02**	Communication with device with 2 bits per clock (Dual-SPI mode 0)
DUAL_SPI_MODE1	**0x42**	Communication with device with 2 bits per clock (Dual-SPI mode 1)
DUAL_SPI_MODE2	**0x82**	Communication with device with 2 bits per clock (Dual-SPI mode 2)
DUAL_SPI_MODE3	**0xC2**	Communication with device with 2 bits per clock (Dual-SPI mode 3)
QUAD_SPI_MODE0	**0x04**	Communication with device with 4 bits per clock (Quad-SPI mode 0)
QUAD_SPI_MODE1	**0x44**	Communication with device with 4 bits per clock (Quad-SPI mode 1)
QUAD_SPI_MODE2	**0x84**	Communication with device with 4 bits per clock (Quad-SPI mode 2)
QUAD_SPI_MODE3	**0xC4**	Communication with device with 4 bits per clock (Quad-SPI mode 3)

6.2. SPI packet object structure

This is the descriptor of the SPI packet to transfer.

Source code:

```
typedef struct
{
  SPI_Conf Config;
  uint8_t ChipSelect;
  uint8_t DummyByte;
  uint8_t *TxData;
  uint8_t *RxData;
  size_t DataSize;
  bool Terminate;
} SPIInterface_Packet;
```

6.2.1. Data fields

```
SPI_Conf Config
```

Configuration of the SPI transfer.

Type

union	SPI_Conf

```
bool ChipSelect
```

Is the Chip Select index to use for the SPI/Dual-SPI/Quad-SPI transfer.

```
uint8_t DummyByte
```

Is the byte to use for receiving data (used with flag SPI_Conf.Bits.UseDummyByte = 1 in SPIInterface_Packet.Config when receiving data or SPIInterface_Packet.TxData is NULL).

*uint8_t** **TxData**

Is the data to send through the interface (used with flag SPI_Conf.Bits.UseDummyByte = 0 in SPIInterface_Packet.Config when receiving data).

*uint8_t** **RxData**

Is where the data received through the interface will be stored. This parameter can be nulled by the driver if no received data is expected.

size_t **DataSize**

Is the size of the data to send and receive through the interface.

bool **Terminate**

Ask to terminate the current transfer. If 'true', deassert the ChipSelect pin at the end of transfer else leave the pin asserted.

6.2.2. Structures and unions

This is the SPI configuration to apply to the packet to transfer. Source code:

```
typedef union SPI_Conf
{
  uint16_t Value;
  struct
  {
    uint16_t UseDummyByte   : 1;
    uint16_t                : 2;
    uint16_t IsPolling      : 1;
    uint16_t EndianResult   : 3;
    uint16_t EndianTransform: 3;
    uint16_t TransactionInc : 6;
  } Bits;
} SPI_Conf;
```

6.2.2.1. Data fields

uint16_t **Value**

This is the value of SPI_Conf.

bool **Bits.UseDummyByte**:1

Use dummy byte for receiving: 'true' = use the DummyByte member for all bytes to receive ; 'false' = Use TxData for all bytes to receive.

bool **Bits.IsPolling**:1

Polling indication for DMA use: 'true' = The driver uses polling for this transfer and the SPI peripheral can use DMA (non-blocking transfer) ; 'false' = No polling therefore do not use DMA (blocking transfer).

eSPI_EndianTransform **Bits.EndianResult**:3

If the transfer changes the endianness, the peripheral that do the transfer will say it here.

eSPI_EndianTransform **Bits.EndianTransform**:3

The driver that asks for the transfer needs an endian change from little to big-endian or big to little-endian.

uint8_t **Bits.TransactionInc**:6

Current transaction number (managed by the SPI+DMA driver). When a new DMA transaction is initiate, set this value to '0', the SPI+DMA driver will return an incremental number. This is for knowing that the transaction has been accepted or the bus is busy with another transaction.

6.2.2.2. Enumerators

`enum eSPI_EndianTransform`

Transfer type of the packet.

Enumerator

SPI_NO_ENDIAN_CHANGE	**0x0**	Do not change endianness therefore read/write byte at the same order as received/sent
SPI_SWITCH_ENDIAN_16BITS	**0x2**	Switch endianness per read/write 16-bits data received/sent
SPI_SWITCH_ENDIAN_24BITS	**0x3**	Switch endianness per read/write 24-bits data received/sent
SPI_SWITCH_ENDIAN_32BITS	**0x4**	Switch endianness per read/write 32-bits data received/sent

6.3. Function's return error enumerator

`enum eERRORRESULT`

There is only one error code at the same time returned by the functions. The only code that indicates that all went fine is ERR_OK.

Enumerator

ERR_OK	**0**	Succeeded
ERR__SPI_PARAMETER_ERROR	**200**	SPI parameter error
ERR__SPI_COMM_ERROR	**201**	SPI communication error
ERR__SPI_CONFIG_ERROR	**202**	SPI configuration error
ERR__SPI_TIMEOUT	**203**	SPI communication timeout
ERR__SPI_INVALID_DATA	**204**	SPI invalid data
ERR__SPI_FREQUENCY_ERROR	**205**	SPI frequency error
ERR__SPI_OVERFLOW_ERROR	**206**	SPI overflow error
ERR__SPI_UNDERFLOW_ERROR	**207**	SPI underflow error
ERR__SPI_BUSY	**208**	SPI busy
ERR__SPI_OTHER_BUSY	**209**	SPI busy by other transfer