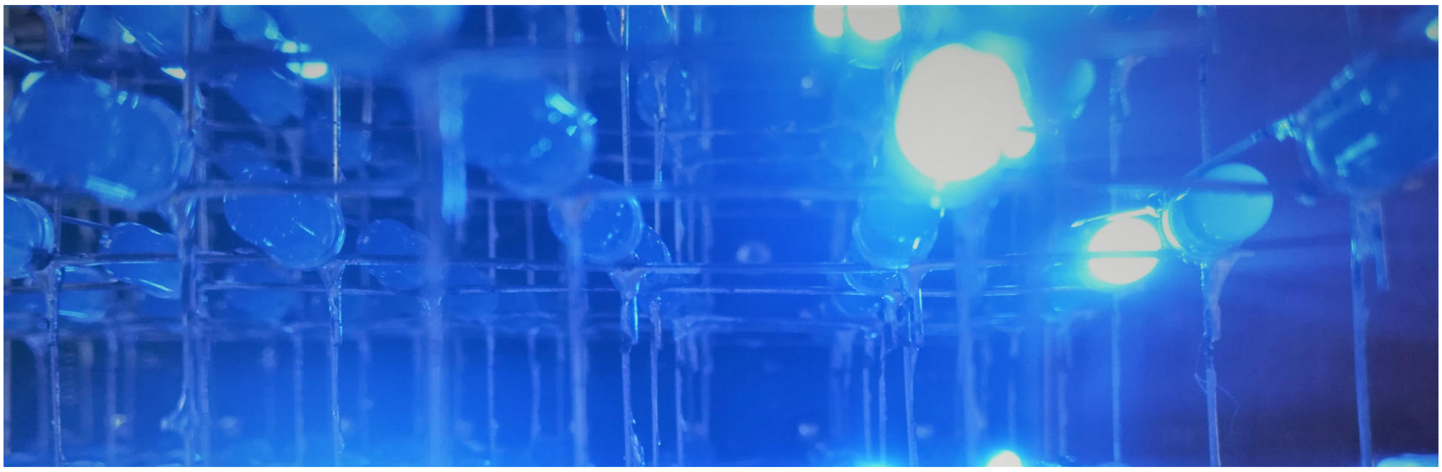

Using STM32 SPI with ChibiOS

Published on August 9, 2018 Updated on March 12, 2019

1 The Serial Peripheral Interface

The **Serial Peripheral Interface** (often shored as **SPI** bus) is a widely used synchronous serial communication peripheral which communicates in full duplex mode using a master-slave architecture with a single master.

As the SPI is a **Synchronous Serial** bus, a clock signal is generated by one of the endpoints and provided to the others through a specific **Serial Clock Line** often shorted as **SCL** or **CLK**. The communication party which generates the clock is named **Master** while other **Slaves**.

In **full-duplex buses**, the communication is simultaneously bi-directional, i.e. data can flow from master to slave and from slave to master at the same time on two separate wires at the same time. The SPI has indeed two data lines:
1. the **Master Output Slave Input**, often shorted as **MOSI** or **SDO** which stands for Serial Data Output,
2. the **Master Input Slave Output**, often shorted as **MISO** or **SDI** which stands for Serial Data Input.

The SPI communication uses an additional wire for each slave to address them during communication. This wire is known as **Chip Select** (CS) or **Slave Select** (SS) more often **nCS** or **nSS** as it works in negate logic.

1.1 SPI highlights

In embedded applications, the SPI could be used to interconnect microcontrollers but more often it is used to connect a micro to an external peripheral/device like Secure Digital Card, LED display, MEMS, etc. In this case, the MCU acts as communication Master while the device as a slave.

Basically, in a single master single slave communication, the bus is composed of 4 wires (i.e. MISO, MOSI, SCL, and nCS) thus the SPI is often nicknamed 4-wire bus. In this scenario, the communication starts with the master lowering the chip select. At this point, the master has to provide a clock signal having a proper frequency: slave and master will send and sample one bit per each clock cycle. For instance, in an 8-bit length communication, the master will provide 8 clock pulses and master and slave will send and receive both 8 bits. The communication ends when the master pulls up the chip select.

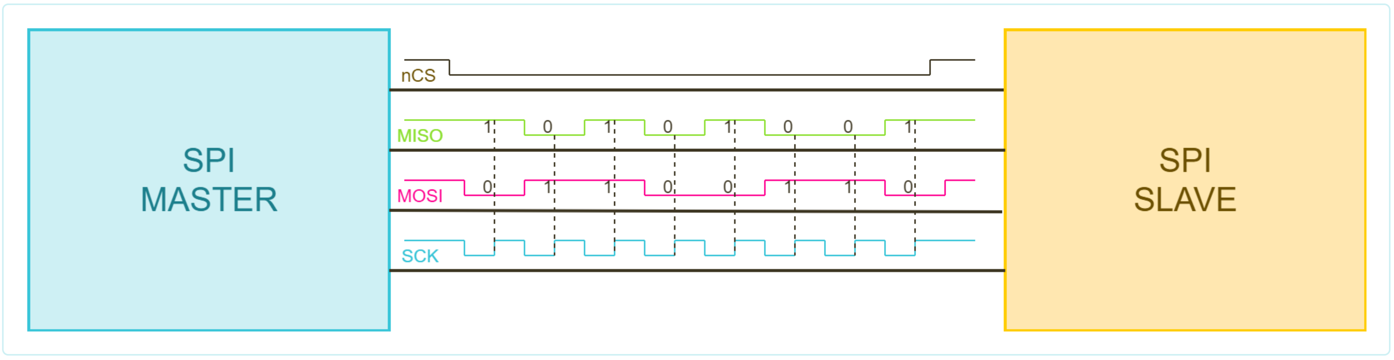

Fig.1 - A diagram of an SPI communication master slave with high clock polarity and low clock phase.

SPI communication is very simple and allows a few settings. One of those configurations is the data length which usually is a multiple of 1 byte (e.g. 8-bit, 16-bit, 32-bit) but it is not always true.

SPI is designed to be independent of clock speed and potentially the clock speed could be pushed at Gigahertz. Anyway, small devices can operate within a specific range of clock rate and such information is often available in their manuals or datasheets. A good master SPI cell should thus be able to change its clock speed and we should thus expect that the clock speed is one of the SPI settings. Indeed, this is usually an option and master has a wide range of allowed clock frequencies in order to be compliant with a wide range of devices.

SPI clock has also another couple of configurations to be taken into account:

- the **Clock polarity** (**CPOL**), which specifies the logic level of the clock signal when the communication bus is its idle state. If CPOL is low, the clock line will be normally low and first clock edge will be the rising one. If CPOL is high, the clock will be normally high and first clock edge will be the falling one.
- the **Clock phase** (**CPHA**) which specifies if data is sampled on the first or second clock edge.

Note that CPHA itself is not able to specify if the data is sampled on rising or falling edge: first and second clock edges are a relative concept. They make sense only whereas CPOL decides. For instance, if CPHA is set as 0, the data will be sampled on the first clock edge. If CPOL is 0, the first edge will be the rising one, otherwise falling one. Thus we can state that:

- if CPOL and CPHA are both 0 the clock is normally low and data is sampled on the rising edge
- if CPOL is 0 and CPHA is 1 the clock is normally low and data is sampled on the falling edge
- if CPOL is 1 and CPHA is 0 the clock is normally high and data is sampled on the falling edge
- if CPOL and CPHA are both 1 the clock is normally high and data is sampled on the rising edge

These 4 CPOL and CPHA combinations and their behavior are shown in the next figure:

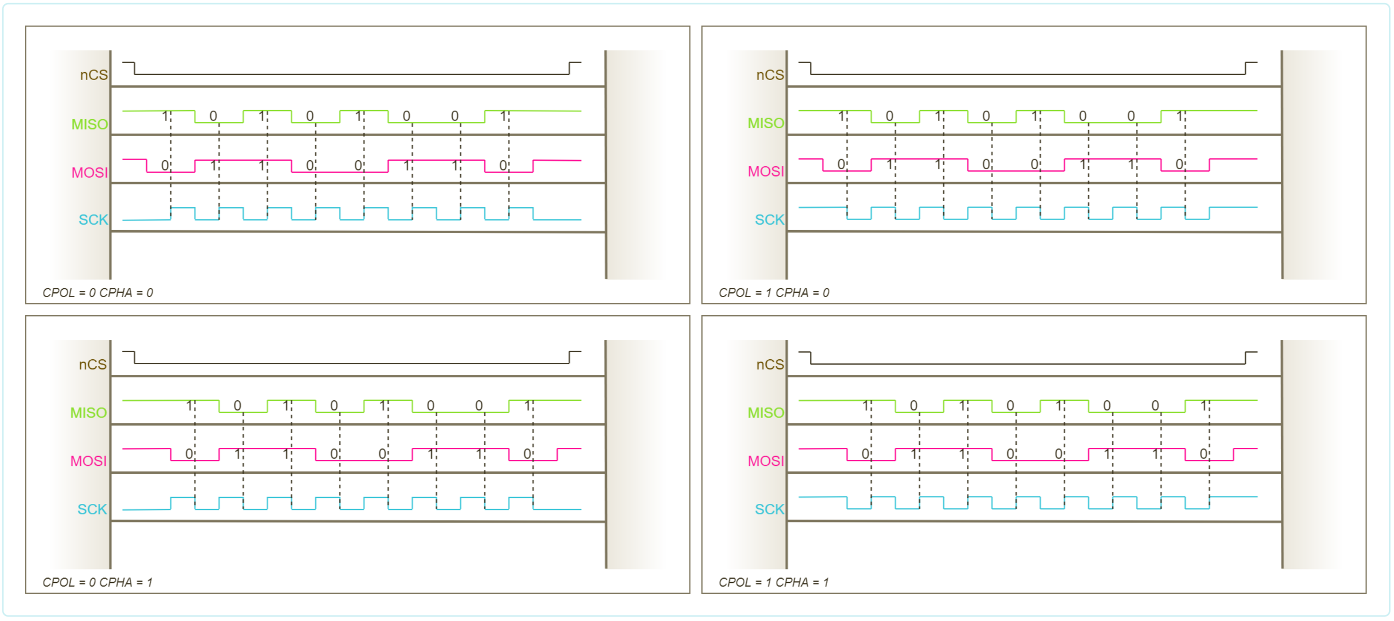

Fig.2 - The behavior of SPI with different clock polarity (CPOL) and clock phase (CPHA) configuration

A good SPI master should be able to operate in every condition since usually slaves are designed to be as cheap as possible with simple circuitry, a small clock speed range, non-configurable clock polarity, and phase. This means we will (almost) always choose master configurations depending on slave needs.

1.2 Multi slave scenarios

There is two way to connect a master with more than a slave: the independent mode and the cooperative mode.

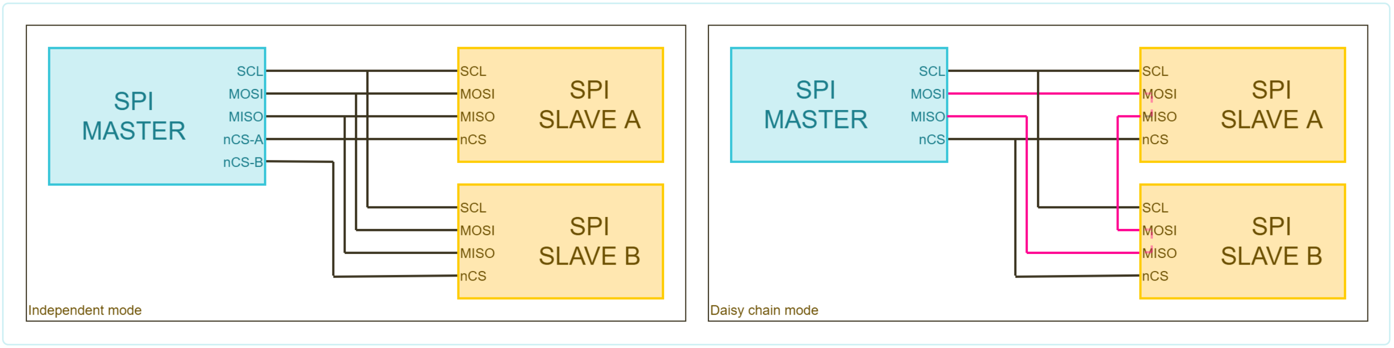

Fig.3 - Two way to connect multiple slaves to a master: on the left the independent mode; on the right the dependent mode (aka daisy chain).

The **independent mode** is widely used when there are many different slave typologies. In this case, the slaves share SCL, MOSI, and MISO but each slave has its own nCS. In a few words each slave can be selected independently and on each communication, the master lowers the nCS of the slave it wants to communicate with: the chip select acts like a chip enable.

The **cooperative mode** is known also as **daisy chain** and in this case slaves share the same clock and slave select while MOSI and MISO are connected to create a chain of devices: the master output is connected to the first slave input, the first slave output to the second slave input, …, the last slave output to the master input. This kind of connection is particularly suitable with **Serial Input Parallel Output Shift Registers** (**SIPO SR** or simply **SR**): for instance, connecting 4 16-bit SR in a daisy chain, we can obtain a system which acts as a single 64-bit SR.

2 The STM32 SPI

Across the various STM32 subfamilies there are three different hardware version of the SPI cell. We have already explained how to identify the driver version of your STM32 board here reporting them in a table. Anyway, as a quick reference:

- **STM32F1xx**, **STM32F2xx**, **STM32F4xx**, **STM32L0xx** and **STM32L1xx** use SPIv1;
- **STM32F0xx**, **STM32F3xx**, **STM32F7xx** and **STM32L4xx** use SPIv2;
- **STM32H7xx** uses SPIv3.

The differences between various SPI versions are minimal and basically related to exposed configuration registers. STM32 allows:

- both master and slave mode
- to configure the baud-rate
- to communicate in a full-duplex and half-duplex mode using a customization of the SPI named three wire SPI
- to select the data size. Note that possible choices depending on the hardware version. On SPIv1 available choices are 8 or 16-bit, in SPIv2 any size between 4 and 16-bit and in SPIv3 any size between 4 and 32-bit.
- to manage slave select automatically in hardware
- to configure clock phase and polarity
- to do hardware CRC but only in SPIv2 and SPIv3 and in the SPIv3 it is also possible to choose CRC size.

Note that STM32 offers many SPI instances (from 2 in the small packages up to 6 in larger packages). SPI cells are identified with a progressive number (i.e. SPI 1, SPI 2, SPI 3 and so on). Note also that SPI bus relies on GPIO alternate functions to be able to get in touch with the outside world.

3 The ChibiOS SPI driver

The current ChibiOS SPI driver allows only master mode allowing to exploit many of the feature offered by the STM32 SPI cell. It offers again both synchronous and asynchronous API.

> *Each API of the SPI Driver starts with the prefix "spi". Function names are camel-case, pre-processor constants uppercase and variables lowercase.*

3.1 Different driver same approach

Comparing the SPI driver with previously presented Serial Driver or ADC we can notice that there are some certitudes in ChibiOS: one of these is for sure the consistency of design patterns. The SPI driver is organized like every other simple driver of ChibiOS\HAL:

- The whole driver subsystem can be enabled disabled through the proper switch in the *halconf.h*. The switch name is *HAL_USE_SPI*.
- To use the driver we have then to assign it an SPI peripheral acting on *mcuconf.h*.
- Assigning a peripheral a new object will become available: SPID1 on SPI 1 assignation, SPID2 on SPI 2 and so on.
- Each SPIDx object represents a driver which implements a Finite State Machine.
- A driver to be used shall be initialized but this is done automatically on *halInit()*;
- Each driver operation (e.g. a communication exchange) can be done only if the driver has been properly started. This requires a call

to *spiStart()*.

- The *spiStart()* receives as usual two parameters: a pointer to the driver and a pointer to its configuration.
- If the driver is not used it can be stopped through the *spiStop()*.

The following figure represents the state machine of the driver.

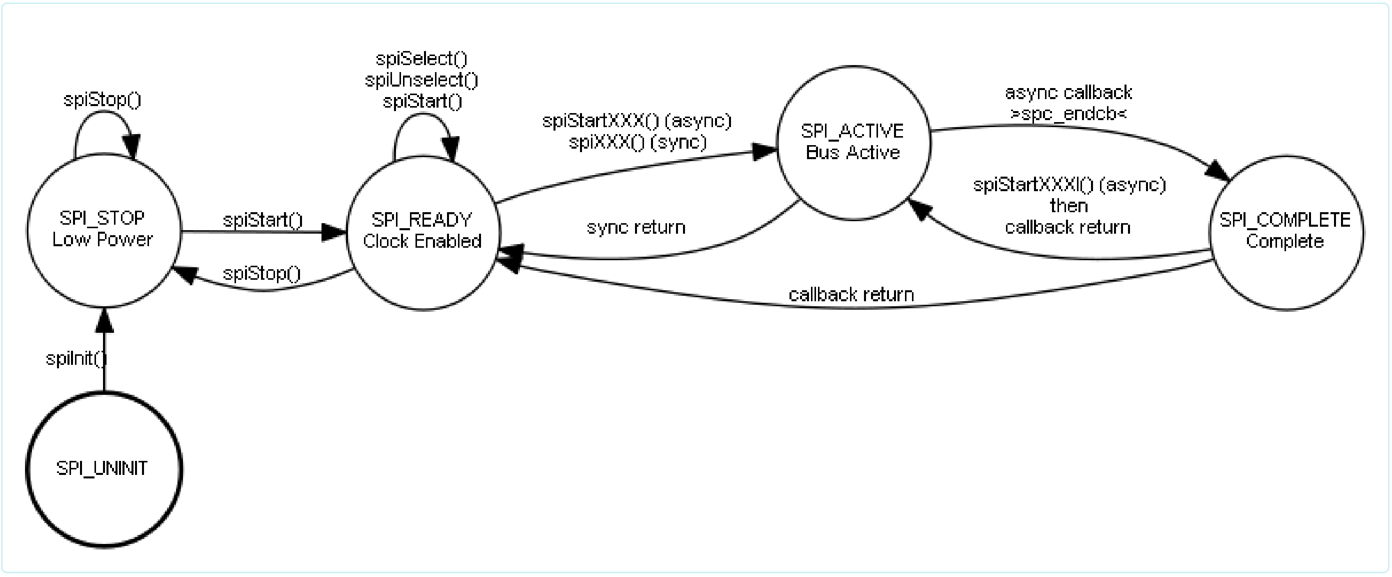

Fig.4 - The SPI Driver state machine

3.2 Configuring the SPI

The SPI configuration structure allows to properly exploit STM32 SPI features by configuring data size, baudrate and clock phase, and polarity. The following structure has been copied out from SPIv2 but is identical to that from SPIv1 and extremely similar to SPIv3.

```
/**
 * @brief   Driver configuration structure.
 */
typedef struct {
#if (SPI_SUPPORTS_CIRCULAR == TRUE) || defined(__DOXYGEN__)
  /**
   * @brief   Enables the circular buffer mode.
   */
  bool                      circular;
#endif
  /**
   * @brief Operation complete callback or @p NULL.
   */
  spicallback_t             end_cb;
#if (SPI_SELECT_MODE == SPI_SELECT_MODE_LINE) || defined(__DOXYGEN__)
  /**
   * @brief The chip select line.
   */
  ioline_t                  ssline;
```

The first parameter is **circular**, a boolean which enables the circular mode. In this mode, the SPI buffer is transmitted continuously. The driver will also execute a callback every time the whole buffer has been sent: the callback can be configured using the second parameter, **end_cb**, which is a pointer to a callable. This callback is particularly suitable for asynchronous APIs usage.

The next parameters are used to identify the chip select of the slave. They depend on *SPI_SELECT_MODE*. According to the configuration we can have:

- one IO line,
- a port identifier and a pad mask,
- a port and a pin identifier (default).

Those are used in software chip select mode. As said chip select can be controlled in hardware as well as in software. When chip select is controlled by software the driver need to know which pin or mask of pins is the chip select. *SPI_SELECT_MODE* can be configured in *halconf.h*. By default, it is set as *SPI_SELECT_MODE_PAD*.

The latest two parameters are hardware depending and represent 2 SPI registers. All the differences between various hardware version are circumscribed here:

- in SPIv1 and SPIv2 we have **cr1** and **cr2** which represent the value of SPI control register 1 (SPI_CR1) and SPI control register 2 (SPI_CR2). Even if they have the same name in both the hardware version, bit fields and related configurations are slightly different.
- in SPIv3 we have **cfg1** and **cfg2** which represent the value of SPI configuration register 1 (SPI_CFG1) and SPI configuration register 2 (SPI_CFG2).

To compose register values we can use the bitmasks already defined in the CMSIS header files. Such files contain all the definition of every bit field of every register of our MCU. It is possible to find under *[chibios_root]\chibios_trunk\os\common\ext\ST\*. Each MCU has is own CMSIS header file as the register map changes with the microcontroller. As example the file for the STM32F401RE is *[chibios_root]\chibios_trunk\os\common\ext\ST\STM32F4xx\stm32f401xe.h.*

Anyway, it can be useful to take a look to SPI demo under testhal folder: here you can find some valid example to understand how to configure SPI. For instance, the following configuration has been copied by the SPI demo for STM32F3 discovery.

```
/*
 * Maximum speed SPI configuration (18MHz, CPHA=0, CPOL=0, MSb first).
 */
const SPIConfig hs_spicfg = {
  false,
  NULL,
  GPIOB,
  12,
  0,
  SPI_CR2_DS_2 | SPI_CR2_DS_1 | SPI_CR2_DS_0
};
```

In this case, the circular mode is disabled, the callback is unused, the chip select pin is PB12 and SPI is configured to work at 18 MHz with CPOL and CPHA both to zero.

Note that to compute the baud rate value user should refer to both SPI CR1 description and clock tree configuration which in ChibiOS can be configured in *mcuconf.h*. In this specific case, the BR bit-field value is set to 0. The register description shows a formula to compute the baud rate which is

$$baudrate = \frac{f_{clk}}{2^{BR+1}}$$

The SPI clock frequency depends on the clock tree configuration. In this demo AHB bus is configured at its maximum speed (i.e 72

MHz), the APB1 and APB2 are configured both to half AHB speed (i.e. 36 MHz). The SPI speed is equal to the speed of the APB bus. Thus in this specific case, we have

$$baudrate = \frac{36MHz}{2^{0+1}} = 18MHz$$

To put in place this configuration user has to call the *spiStart* specifying which SPI Driver we are actually going to use.

```
/**
 * @brief   Configures and activates the SPI peripheral.
 *
 * @param[in] spip      pointer to the @p SPIDriver object
 * @param[in] config    pointer to the @p SPIConfig object
 *
 * @api
 */
void spiStart(SPIDriver *spip, const SPIConfig *config) {

  ...
}
```

SPI driver configuration can be changed on the fly restarting the driver with a new configuration. For instance, this is particularly useful when we have more slaves which work with a different configuration (baud rate, clock polarity and phase, slave select and so on).

3.3 Select and unselect

The SPI offers a couple of API to select or deselect the slave. Basically selecting the slave we are lowering the nCS.

```
/**
 * @brief   Asserts the slave select signal and prepares for transfers.
 *
 * @param[in] spip      pointer to the @p SPIDriver object
 *
 * @api
 */
void spiSelect(SPIDriver *spip) {

  ...
}

/**
 * @brief   Deasserts the slave select signal.
 * @details The previously selected peripheral is unselected.
 *
 * @param[in] spip      pointer to the @p SPIDriver object
 *
 * @api
```

To use these API the driver shall be initialized and nCS shall not be configured as hardware managed. The sequence of SPI transaction would be

```
  spiSelect(&PORTAB_SPI1);            /* Slave Select assertion.          */
  spiExchange(&PORTAB_SPI1, 512,
              txbuf, rxbuf);          /* Atomic transfer operations.      */
  spiUnselect(&PORTAB_SPI1);          /* Slave Select de-assertion.       */
```

Note that a software managed chip select is actually a digital output pin thus nCS pin shall be configured as output push-pull while MISO, MOSI, and SCL are configured in alternate mode (take a look to GPIO article if you are not familiar with).

For instance, this code has been copied by a demo for STM32 Nucleo-64 F401RE where SPI pins are not configured at board level

```
/*
 * SPID1 I/O pins setup.(Overwriting board.h configurations)
 */
palSetPadMode(PORT_SPI1_SCK, PIN_SPI1_SCK,
              PAL_MODE_ALTERNATE(5) | PAL_STM32_OSPEED_HIGHEST);    /* New SCK */
palSetPadMode(PORT_SPI1_MISO, PIN_SPI1_MISO,
              PAL_MODE_ALTERNATE(5) | PAL_STM32_OSPEED_HIGHEST);    /* New MISO*/
palSetPadMode(PORT_SPI1_MOSI, PIN_SPI1_MOSI,
              PAL_MODE_ALTERNATE(5) | PAL_STM32_OSPEED_HIGHEST);    /* New MOSI*/
palSetPadMode(PORT_SPI1_CS, PIN_SPI1_CS,
              PAL_MODE_OUTPUT_PUSHPULL | PAL_STM32_OSPEED_HIGHEST); /* New CS*/
```

3.4 Communication API

The SPI driver offers both synchronous and asynchronous communication functions. We already introduced these concepts but let me recall them.

A **blocking function** blocks the execution of the calling thread until the operation is completed. If the function has some internal wait they are not polled, instead, the calling thread is suspended and the RTOS executes other threads. Such functions are also known as **synchronous functions** as the calling thread remains in sync with the operation.

A **non-blocking function** launches the operation and returns to the calling thread which continues to be executed. Such operation is executed in the background, usually in hardware, launching an interrupt on operation complete event. Such IRQ usually is exposed by driver API in the form of a callback. Such functions are also known as **asynchronous functions**.

3.4.1 Exchange

As the SPI is actually a full duplex, basically each transaction is bidirectional. Thus maybe the *spiStartExchange* and the *spiExhange* can be considered the main communication functions of the SPI driver.

```
/**
 * @brief   Exchanges data on the SPI bus.
 * @details This asynchronous function starts a simultaneous transmit/receive
 *          operation.
 * @pre     A slave must have been selected using @p spiSelect() or
 *          @p spiSelectI().
 * @post    At the end of the operation the configured callback is invoked.
 * @note    The buffers are organized as uint8_t arrays for data sizes below
 *          or equal to 8 bits else it is organized as uint16_t arrays.
 *
 * @param[in] spip      pointer to the @p SPIDriver object
 * @param[in] n         number of words to be exchanged
 * @param[in] txbuf     the pointer to the transmit buffer
 * @param[out] rxbuf    the pointer to the receive buffer
 *
 * @api
 */
void spiStartExchange(SPIDriver *spip, size_t n,
                      const void *txbuf, void *rxbuf) {
```

The first is asynchronous while the second is synchronous. Both functions receive four parameters:

- A pointer to the SPI driver
- The transaction size expressed in words whereas the word size is defined by the SPI configuration
- a pointer to a transmission buffer having the proper size and pre-filled before the exchange call
- a pointer to a receiver buffer having the proper size which will be filled by exchange call with SPI incoming data.

The synchronous call (i.e *spiExhange*) puts the calling thread in a sleep state until the transaction is completed or aborted. The asynchronous call instead starts the communication and returns to the caller. In both cases when the operation is completed the *end_cb* (when it is not equal to NULL) is executed.

3.4.2 Send and Receive

The driver offers also some functions to communicate in one way: the *spiStartSend*, the *spiStartReceive*, and their related synchronous version *spiSend* and *spiReceive*

```
/**
 * @brief   Sends data over the SPI bus.
 * @details This asynchronous function starts a transmit operation.
 * @pre     A slave must have been selected using @p spiSelect() or
 *          @p spiSelectI().
 * @post    At the end of the operation the configured callback is invoked.
 * @note    The buffers are organized as uint8_t arrays for data sizes below
 *          or equal to 8 bits else it is organized as uint16_t arrays.
 *
 * @param[in] spip      pointer to the @p SPIDriver object
 * @param[in] n         number of words to send
 * @param[in] txbuf     the pointer to the transmit buffer
 *
 * @api
 */
void spiStartSend(SPIDriver *spip, size_t n, const void *txbuf) {

  ...
}
```

This is actually a trick because the SPI always act in full duplex mode. Indeed the *spiSend* is actually a *spiExchange* which ignores received data. Similarly, the *spiRead* is actually a *spiExchange* which uses dummy data as transmission data. Of course the same is for asynchronous API.

3.4.3 Ignore and Abort

The driver offers also some functions to ignore data on the bus: the *spiStartIgnore* and *spiIgnore*.

```
/**
 * @brief   Ignores data on the SPI bus.
 * @details This asynchronous function starts the transmission of a series of
 *          idle words on the SPI bus and ignores the received data.
 * @pre     A slave must have been selected using @p spiSelect() or
 *          @p spiSelectI().
 * @post    At the end of the operation the configured callback is invoked.
 *
 * @param[in] spip      pointer to the @p SPIDriver object
 * @param[in] n         number of words to be ignored
 *
 * @api
 */
void spiStartIgnore(SPIDriver *spip, size_t n) {
  ...
}

/**
 * @brief   Ignores data on the SPI bus.
 * @details This synchronous function performs the transmission of a series of
```

These functions can be useful in certain cases (e.g where the slave expects some data to push forward an internal state machine but the master is not interested in receiving data or is not transmitting anything).

The asynchronous API requires also another function which eventually is able to stop the ongoing activities: the *spiAbort*.

```
/**
 * @brief   Aborts the ongoing SPI operation, if any.
 *
 * @param[in] spip      pointer to the @p SPIDriver object
 *
 * @api
 */
void spiAbort(SPIDriver *spip) {
  ...
}
```

This function aborts ongoing operations if any.

3.5 Mutual exclusion

The driver offers also a couple of API to acquire and release the bus: *spiAcquireBus* and *spiReleaseBus*.

```
/**
 * @brief   Gains exclusive access to the SPI bus.
 * @details This function tries to gain ownership to the SPI bus, if the bus
 *          is already being used then the invoking thread is queued.
 * @pre     In order to use this function the option @p SPI_USE_MUTUAL_EXCLUSION
 *          must be enabled.
 *
 * @param[in] spip      pointer to the @p SPIDriver object
 *
 * @api
 */
void spiAcquireBus(SPIDriver *spip) {
  ...
}

/**
 * @brief   Releases exclusive access to the SPI bus.
 * @pre     In order to use this function the option @p SPI_USE_MUTUAL_EXCLUSION
 *          must be enabled.
```

Such functions are useful when the same SPI driver is used from two threads because they guarantee that none will use that SPI driver until it is acquired avoiding misbehavior due to concurrent access. Note that these API requires *SPI_USE_MUTUAL_EXCLUSION* is set to *TRUE* in the *halconf.h* file.

4 Further readings and Hands-on

We have already planned a collection of example and exercises for the SPI driver. If you are interested in, follow us on Facebook to be updated on our articles. Anyway, at this moment you could refer to the SPI demo under *testhal* to give it a try.

SPI is used by many devices and we wrote many articles about them. For sure the quick reference could be:

- STM32, ChibiOS and a 8×8 LED Matrix
- 7-segment display and STM32 using ChibiOS
- A Radio Frequency transceiver library: nRF24L01 and ChibiOS/RT

You can also try the demo under *testex* for STM32F3 Discovery which uses L3GD20: this is a 3-axis gyroscope connected to the MCU through the SPI.

5 Previous and next

This article is part of a series of articles which are meant to be tutorials. I have composed them to be read in sequence. Here the previous and next article of this series:

- PWM in hardware with STM32 Timer and ChibiOS (Previous)
- Using STM32 I2C with ChibiOS (Next)

Be the first to reply at Using STM32 SPI with ChibiOS

Leave a Reply

Enter your comment here...