
SmartFusion™ MSS SPI Driver User's Guide

Version 2.1



Table of Contents

Introduction	3
Features	3
Supported Hardware IP	3
Files Provided	5
Documentation	5
Driver Source Code.....	5
Example Code.....	5
Driver Deployment	7
Driver Configuration	9
Application Programming Interface	11
Theory of Operation	11
Types	14
Constant Values.....	16
Data structures.....	16
Global Variables.....	16
Functions.....	17
Product Support	41
Customer Service.....	41
Customer Technical Support Center.....	41

Introduction

The SmartFusion™ microcontroller subsystem (MSS) includes two serial peripheral interface SPI peripherals for serial communication. This driver provides a set of functions for controlling the MSS SPIs as part of a bare metal system where no operating system is available. These drivers can be adapted for use as part of an operating system, but the implementation of the adaptation layer between this driver and the operating system's driver model is outside the scope of this driver.

Features

The MSS SPI driver provides the following features:

- Support for configuring each MSS SPI peripheral
- SPI master operations
- SPI slave operations

The MSS SPI driver is provided as C source code.

Supported Hardware IP

The MSS SPI bare metal driver can be used with Actel's MSS_SPI IP version 0.2 or higher included in the SmartFusion MSS.

Note: The SPI slave block transfer mode of operation cannot be used on the A2F200 SmartFusion device due to slave mode limitations in the SPI hardware block of this device.

Files Provided

The files provided as part of the MSS SPI driver fall into three main categories: documentation, driver source code, and example projects. The driver is distributed via the Actel Firmware Catalog, which provides access to the documentation for the driver, generates the driver's source files into an application project, and generates example projects that illustrate how to use the driver.

Documentation

The Actel Firmware Catalog provides access to these documents for the driver:

- User's guide (this document)
- A copy of the license agreement for the driver source code
- Release notes

Driver Source Code

The Actel Firmware Catalog generates the driver's source code into a *drivers\mss_spi* subdirectory of the selected software project directory. The files making up the driver are detailed below.

mss_spi.h

This header file contains the public application programming interface (API) of the MSS SPI software driver. This file must be included in any C source file that uses the MSS SPI software driver.

mss_spi.c

This C source file contains the implementation of the MSS SPI software driver.

Example Code

The Actel Firmware Catalog provides access to example projects illustrating the use of the driver. Each example project is self contained and is targeted at a specific processor and software toolchain combination. The example projects are targeted at the FPGA designs in the hardware development tutorials supplied with Actel's development boards. The tutorial designs can be found on the [Actel Development Kit](http://www.actel.com/products/hardware) web page (www.actel.com/products/hardware).

Driver Deployment

This driver is intended to be deployed from the Actel Firmware Catalog into a software project by generating the driver's source files into the project directory. The driver uses the SmartFusion Cortex Microcontroller Software Interface Standard – Peripheral Access Layer (CMSIS-PAL) to access MSS hardware registers. You must ensure that the SmartFusion CMSIS-PAL is either included in the software toolchain used to build your project or is included in your project. The most up-to-date SmartFusion CMSIS-PAL files can be obtained using the Actel Firmware Catalog.

The following example shows the intended directory structure for a SoftConsole ARM® Cortex™-M3 project targeted at the SmartFusion MSS. This project uses the MSS SPI and MSS Watchdog drivers. Both of these drivers rely on SmartFusion CMSIS-PAL for accessing the hardware. The contents of the *drivers* directory result from generating the source files for each driver into the project. The contents of the *CMSIS* directory result from generating the source files for the SmartFusion CMSIS-PAL into the project.



Figure 1 · SmartFusion MSS Project Example

Driver Configuration

The configuration of all features of the MSS SPIs is covered by this driver with the exception of the SmartFusion IOMUX configuration. SmartFusion allows multiple non-concurrent uses of some external pins through IOMUX configuration. This feature allows optimization of external pin usage by assigning external pins for use by either the microcontroller subsystem or the FPGA fabric. The MSS SPIs serial signals are routed through IOMUXes to the SmartFusion device external pins. These IOMUXes are automatically configured correctly by the MSS configurator tool in the hardware flow when the MSS SPIs are enabled in that tool. You must ensure that the MSS SPIs are enabled by the MSS configurator tool in the hardware flow; otherwise the serial inputs and outputs will not be connected to the chip's external pins. For more information on IOMUX, refer to the IOMUX section of the SmartFusion Datasheet.

The base address, register addresses and interrupt number assignment for the MSS SPI blocks are defined as constants in the SmartFusion CMSIS-PAL. You must ensure that the SmartFusion CMSIS-PAL is either included in the software tool chain used to build your project or is included in your project.

Application Programming Interface

This section describes the driver's API. The functions and related data structures described in this section are used by the application programmer to control the MSS SPI peripheral from the user's application.

Theory of Operation

The MSS SPI driver functions are grouped into the following categories:

- Initialization
- Configuration for either master or slave operations
- SPI master frame transfer control
- SPI master block transfer control
- SPI slave frame transfer control
- SPI slave block transfer control
- DMA block transfer

Frame transfers allow the MSS SPI to write or read up to 32 bits of data in a SPI transaction. For example, a frame transfer of 12 bits might be used to read the result of an ADC conversion from a SPI analog to digital converter.

Block transfers allow the MSS SPI to write or read a number of bytes in a SPI transaction. Block transfer transactions allow data transfers in multiples of 8 bits (8, 16, 24, 32, 40...). Block transfers are typically used with byte oriented devices like SPI FLASH devices.

Initialization

The MSS SPI driver is initialized through a call to the *MSS_SPI_init()* function. The *MSS_SPI_init()* function takes only one parameter, a pointer to one of two global data structures used by the driver to store state information for each MSS SPI. A pointer to these data structures is also used as first parameter to any of the driver functions to identify which MSS SPI will be used by the called function. The names of these two data structures are *g_mss_spi0* and *g_mss_spi1*. Therefore any call to an MSS SPI driver function should be of the form *MSS_SPI_function_name(&g_mss_spi0, ...)* or *MSS_SPI_function_name(&g_mss_spi1, ...)*.

The *MSS_SPI_init()* function resets the specified MSS SPI hardware block and clears any pending interrupts from that MSS SPI in the Cortex-M3 NVIC.

The *MSS_SPI_init()* function must be called before any other MSS SPI driver functions can be called.

Configuration

A MSS SPI block can operate either as a master or slave SPI device. There are two distinct functions for configuring a MSS SPI block for master or slave operations.

Master configuration

The *MSS_SPI_configure_master_mode()* function configures the specified MSS SPI block for operations as a SPI master. It must be called once for each remote SPI slave device the MSS SPI block will communicate with. It is used to provide the following information about each SPI slave's communication characteristics:

- The SPI protocol mode
- The SPI clock speed
- The frame bit length

This information is held by the driver and will be used to alter the configuration of the MSS SPI block each time a slave is selected through a call to *MSS_SPI_set_slave_select()*. The SPI protocol mode defines the initial state of the clock signal at the start of a transaction and which clock edge will be used to sample the

data signal (Motorola SPI modes), or it defines whether the SPI block will operate in Texas Instruments (TI) synchronous serial mode or in National Semiconductor (NSC) MICROWIRE mode.

Slave configuration

The *MSS_SPI_configure_slave_mode()* function configures the specified MSS SPI block for operations as a SPI slave. It configures the following SPI communication characteristics:

- The SPI protocol mode
- The SPI clock speed
- The frame bit length

The SPI protocol mode defines the initial state of the clock signal at the start of a transaction and which clock edge will be used to sample the data signal (Motorola SPI modes), or it defines if the SPI block will operate in TI synchronous serial mode

Note: The MSS SPI cannot be configured to operate as a NSC MICROWIRE protocol mode slave.

SPI master frame transfer control

The following functions are used as part of SPI master frame transfers:

- *MSS_SPI_set_slave_select()*
- *MSS_SPI_transfer_frame()*
- *MSS_SPI_clear_slave_select()*

The master must first select the target slave through a call to *MSS_SPI_set_slave_select()*. This causes the relevant slave select line to become asserted while data is clocked out onto the SPI data line.

A call is then made to function *MSS_SPI_transfer_frame()* specifying and the value of the data frame to be sent.

The function *MSS_SPI_clear_slave_select()* can be used after the transfer is complete to prevent this slave select line from being asserted during subsequent SPI transactions. A call to this function is only required if the master is communicating with multiple slave devices.

SPI master block transfer control

The following functions are used as part of SPI master block transfers:

- *MSS_SPI_set_slave_select()*
- *MSS_SPI_clear_slave_select()*
- *MSS_SPI_transfer_block()*

The master must first select the target slave through a call to *MSS_SPI_set_slave_select()*. This causes the relevant slave select line to become asserted while data is clocked out onto the SPI data line.

A call is then made to function *MSS_SPI_transfer_block()*. The parameters of this function specify:

- the number of bytes to be transmitted
- a pointer to the buffer containing the data to be transmitted
- the number of bytes to be received
- a pointer to the buffer where received data will be stored

The number of bytes to be transmitted can be set to zero to indicate that the transfer is purely a block read transfer. The number of bytes to be received can be set to zero to specify that the transfer is purely a block write transfer.

The function *MSS_SPI_clear_slave_select()* can be used after the transfer is complete to prevent this slave select line from being asserted during subsequent SPI transactions. A call to this function is only required if the master is communicating with multiple slave devices.

Note: When performing block transfers using the A2F500 and A2F060 SmartFusion devices, the MSS SPI asserts the target slave device's chip select throughout the transfer in Motorola SPI modes 1 and 3 (i.e. protocol mode configuration is *MSS_SPI_MODE1* or *MSS_SPI_MODE3*), whereas the slave select is de-asserted between each byte transfer in Motorola SPI modes 0 and 2 (i.e. protocol mode configuration is *MSS_SPI_MODE0* or *MSS_SPI_MODE2*). A GPIO may be used to control the target

slave device's chip select in Motorola SPI modes 0 and 2 if de-assertion of the slave select between each byte transfer is not desired.

Note: When performing block transfers using the A2F200 SmartFusion device, it is recommended that you use a GPIO to control the target slave device's chip select input. This ensures reliable operations by preventing the chip select signal from becoming de-asserted, before the transfer completes, as a result of the SPI transmit FIFO becoming empty. This issue may occur where a high priority interrupt is serviced in some other part of the system while the SPI transfer is taking place. This potential problem only exists for block transfers on the A2F200 SmartFusion device.

SPI slave frame transfer control

The following functions are used as part of SPI slave frame transfers:

- *MSS_SPI_set_slave_tx_frame()*
- *MSS_SPI_set_frame_rx_handler()*

The *MSS_SPI_set_slave_tx_frame()* function specifies the frame data that will be returned to the SPI master. The frame data specified through this function is the value that will be read over the SPI bus by the remote SPI master when it initiates a transaction. A call to *MSS_SPI_set_slave_tx_frame()* is only required if the MSS SPI slave is the target of SPI read transactions, i.e. if data is meant to be read from the SmartFusion device over SPI.

The *MSS_SPI_set_frame_rx_handler()* function specifies the receive handler function that is called when a frame of data has been received by the MSS SPI when it is configured as a slave. The receive handler function specified through this call will process the frame data written, over the SPI bus, to the MSS SPI slave by the remote SPI master. The receive handler function must be implemented as part of the application. It is only required if the MSS SPI slave is the target of SPI frame write transactions.

SPI slave block transfer control

The following function is used as part of SPI slave block transfers:

- *MSS_SPI_set_slave_block_buffers()*
- *MSS_SPI_set_cmd_handler()*
- *MSS_SPI_set_cmd_response()*

The *MSS_SPI_set_slave_block_buffers()* function is used to configure a MSS SPI slave for block transfer operations. It specifies:

- The buffer containing the data that will be returned to the remote SPI master
- The buffer where data received from the remote SPI master will be stored
- The handler function that will be called after the receive buffer is filled

The *MSS_SPI_set_cmd_handler()* function specifies a command handler function that will be called by the driver once a specific number of bytes has been received after the SPI chip select signal becoming active. The number of bytes making up the command part of the transaction is specified as part of the parameters to function *MSS_SPI_set_cmd_handler()*. The command handler function is implemented as part of the application making use of the SPI driver and would typically call the *MSS_SPI_set_cmd_response()* function.

The *MSS_SPI_set_cmd_response()* function specifies the data that will be returned to the master. Typically the *MSS_SPI_set_slave_block_buffers()* will have been called as part of the system initialization to specify the data sent to the master while the command bytes are being received. The transmit buffer specified through the call to *MSS_SPI_set_slave_block_buffers()* would also typically include one or more bytes allowing for the turnaround time required for the command handler function to execute and call *MSS_SPI_set_cmd_response()*.

Note: The slave block transfer mode of operation cannot be used on the A2F200 SmartFusion device due to slave mode limitations in the SPI hardware block of this device.

DMA block transfer control

The following functions are used as part of MSS SPI DMA transfers:

- *MSS_SPI_disable()*

- *MSS_SPI_set_transfer_byte_count()*
- *MSS_SPI_enable()*
- *MSS_SPI_tx_done()*

The MSS SPI must first be disabled through a call to function *MSS_SPI_disable()*. The number of bytes to be transferred is then set through a call to function *MSS_SPI_set_transfer_byte_count()*. The DMA transfer is then initiated by a call to the *MSS_PDMA_start()* function provided by the MSS PDMA driver. The actual DMA transfer will only start once the MSS SPI block has been re-enabled through a call to *MSS_SPI_enable()*. The completion of the DMA driven SPI transfer can be detected through a call to *MSS_SPI_tx_done()*. The direction of the SPI transfer, write or read, depends on the DMA channel configuration. A SPI write transfer occurs when the DMA channel is configured to write data to the MSS SPI block. A SPI read transfer occurs when the DMA channel is configured to read data from the MSS SPI block.

Types

mss_spi_protocol_mode_t

Prototype

```
typedef enum __mss_spi_protocol_mode_t {
    MSS_SPI_MODE0      = 0x00000000,
    MSS_SPI_TI_MODE    = 0x01000004,
    MSS_SPI_NSC_MODE   = 0x00000008,
    MSS_SPI_MODE2      = 0x01000000,
    MSS_SPI_MODE1      = 0x02000000,
    MSS_SPI_MODE3      = 0x03000000
} mss_spi_protocol_mode_t;
```

Description

This enumeration is used to define the settings for the SPI protocol mode bits, CPHA and CPOL. It is used as a parameter to the *MSS_SPI_configure_master_mode()* and *MSS_SPI_configure_slave_mode()* functions.

mss_spi_pclk_div_t

Prototype

```
typedef enum __mss_spi_pclk_div_t {
    MSS_SPI_PCLK_DIV_2    = 0,
    MSS_SPI_PCLK_DIV_4    = 1,
    MSS_SPI_PCLK_DIV_8    = 2,
    MSS_SPI_PCLK_DIV_16   = 3,
    MSS_SPI_PCLK_DIV_32   = 4,
    MSS_SPI_PCLK_DIV_64   = 5,
    MSS_SPI_PCLK_DIV_128  = 6,
    MSS_SPI_PCLK_DIV_256  = 7
} mss_spi_pclk_div_t;
```

Description

This enumeration specifies the divider to be applied to the APB bus clock in order to generate the SPI clock. It is used as parameter to the *MSS_SPI_configure_master_mode()* and *MSS_SPI_configure_slave_mode()* functions.

mss_spi_slave_t

Prototype

```
typedef enum __mss_spi_slave_t {  
    MSS_SPI_SLAVE_0 = 0,  
    MSS_SPI_SLAVE_1 = 1,  
    MSS_SPI_SLAVE_2 = 2,  
    MSS_SPI_SLAVE_3 = 3,  
    MSS_SPI_SLAVE_4 = 4,  
    MSS_SPI_SLAVE_5 = 5,  
    MSS_SPI_SLAVE_6 = 6,  
    MSS_SPI_SLAVE_7 = 7,  
    MSS_SPI_MAX_NB_OF_SLAVES = 8  
} mss_spi_slave_t;
```

Description

This enumeration is used to select a specific SPI slave device (0 to 7). It is used as a parameter to the *MSS_SPI_configure_master_mode()*, *MSS_SPI_set_slave_select()*, and *MSS_SPI_clear_slave_select()* functions.

mss_spi_frame_rx_handler_t

Prototype

```
void (*mss_spi_frame_rx_handler_t)(uint32_t rx_frame);
```

Description

This defines the function prototype that must be followed by MSS SPI slave frame receive handler functions. These functions are registered with the MSS SPI driver through the *MSS_SPI_set_frame_rx_handler()* function.

Declaring and Implementing Slave Frame Receive Handler Functions

Slave frame receive handler functions should follow the following prototype:

```
void slave_frame_receive_handler(uint32_t rx_frame);
```

The actual name of the receive handler is unimportant. You can use any name of your choice for the receive frame handler. The *rx_frame* parameter will contain the value of the received frame.

mss_spi_block_rx_handler_t

Prototype

```
void (*mss_spi_block_rx_handler_t)(uint8_t * rx_buff , uint32_t rx_size);
```

Description

This defines the function prototype that must be followed by MSS SPI slave block receive handler functions. These functions are registered with the MSS SPI driver through the *MSS_SPI_set_slave_block_buffers()* function.

Declaring and Implementing Slave Block Receive Handler Functions

Slave block receive handler functions should follow the following prototype:

```
void mss_spi_block_rx_handler ( uint8_t * rx_buff, uint32_t rx_size );
```

The actual name of the receive handler is unimportant. You can use any name of your choice for the receive frame handler. The *rx_buff* parameter will contain a pointer to the start of the received block. The *rx_size* parameter will contain the number of bytes of the received block.

Constant Values

MSS_SPI_BLOCK_TRANSFER_FRAME_SIZE

This constant defines a frame size of 8 bits when configuring an MSS SPI to perform block transfer data transactions.

It must be used as the value for the *frame_bit_length* parameter of function *MSS_SPI_configure_master_mode()* when performing block transfers between the MSS SPI master and the target SPI slave.

This constant must also be used as the value for the *frame_bit_length* parameter of function *MSS_SPI_configure_slave_mode()* when performing block transfers between the MSS SPI slave and the remote SPI master.

Data structures

mss_spi_instance_t

There is one instance of this structure for each of the microcontroller subsystem's SPIs. Instances of this structure are used to identify a specific SPI. A pointer to an instance of the *mss_spi_instance_t* structure is passed as the first parameter to MSS SPI driver functions to identify which SPI should perform the requested operation.

Global Variables

g_mss_spi0

Prototype

```
mss_spi_instance_t g_mss_spi0;
```

Description

This instance of *mss_spi_instance_t* holds all data related to the operations performed by MSS SPI 0. A pointer to *g_mss_spi0* is passed as the first parameter to MSS SPI driver functions to indicate that MSS SPI 0 should perform the requested operation.

g_mss_spi1

Prototype

```
mss_spi_instance_t g_mss_spi1;
```

Description

This instance of *mss_spi_instance_t* holds all data related to the operations performed by MSS SPI 1. A pointer to *g_mss_spi1* is passed as the first parameter to MSS SPI driver functions to indicate that MSS SPI 1 should perform the requested operation.

Functions

MSS_SPI_init

Prototype

```
void MSS_SPI_init  
(  
    mss_spi_instance_t * this_spi  
);
```

Description

The *MSS_SPI_init()* function initializes hardware and data structures of one of the SmartFusion MSS SPIs. The *MSS_SPI_init()* function must be called before any other MSS SPI driver functions can be called.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

Return Value

This function does not return a value.

Example

```
MSS_SPI_init(&g_mss_spi0);
```

MSS_SPI_configure_master_mode

Prototype

```
void MSS_SPI_configure_master_mode
(
    mss_spi_instance_t * this_spi,
    mss_spi_slave_t slave,
    mss_spi_protocol_mode_t protocol_mode,
    mss_spi_pclk_div_t clk_rate,
    uint8_t frame_bit_length
);
```

Description

The *MSS_SPI_configure_master_mode()* function configures the protocol mode, serial clock speed and frame size for a specific target SPI slave device. It is used when the MSS SPI hardware block is used as a SPI master. This function must be called once for each target SPI slave the SPI master is going to communicate with. The SPI master hardware will be configured with the configuration specified by this function during calls to *MSS_SPI_set_slave_select()*.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

slave

The *slave* parameter is used to identify a target SPI slave. The driver will hold the MSS SPI master configuration required to communicate with this slave, as specified by the other function parameters. Allowed values are:

- MSS_SPI_SLAVE_0
- MSS_SPI_SLAVE_1
- MSS_SPI_SLAVE_2
- MSS_SPI_SLAVE_3
- MSS_SPI_SLAVE_4
- MSS_SPI_SLAVE_5
- MSS_SPI_SLAVE_6
- MSS_SPI_SLAVE_7

protocol_mode

Serial peripheral interface operating mode. Allowed values are:

- MSS_SPI_MODE0
- MSS_SPI_MODE1
- MSS_SPI_MODE2
- MSS_SPI_MODE3
- MSS_SPI_TI_MODE
- MSS_SPI_NSC_MODE

clk_rate

Divider value used to generate serial interface clock signal from PCLK. Allowed values are:

- MSS_SPI_PCLK_DIV_2
- MSS_SPI_PCLK_DIV_4
- MSS_SPI_PCLK_DIV_8
- MSS_SPI_PCLK_DIV_16
- MSS_SPI_PCLK_DIV_32
- MSS_SPI_PCLK_DIV_64
- MSS_SPI_PCLK_DIV_128
- MSS_SPI_PCLK_DIV_256

frame_bit_length

Number of bits making up the frame. The maximum frame length is 32 bits. You must use the MSS_SPI_BLOCK_TRANSFER_FRAME_SIZE constant as the value for *frame_bit_length* when configuring the MSS SPI master for block transfer transactions with the target SPI slave.

Return Value

This function does not return a value.

Example

```
MSS_SPI_init(&g_mss_spi0);

MSS_SPI_configure_master_mode
(
    &g_mss_spi0,
    MSS_SPI_SLAVE_0,
    MSS_SPI_MODE2,
    MSS_SPI_PCLK_DIV_64,
    12
);

MSS_SPI_configure_master_mode
(
    &g_mss_spi0,
    MSS_SPI_SLAVE_1,
    MSS_SPI_TI_MODE,
    MSS_SPI_PCLK_DIV_128,
    MSS_SPI_BLOCK_TRANSFER_FRAME_SIZE
);
```

MSS_SPI_configure_slave_mode

Prototype

```
void MSS_SPI_configure_slave_mode
(
    mss_spi_instance_t * this_spi,
    mss_spi_protocol_mode_t protocol_mode,
    mss_spi_pclk_div_t clk_rate,
    uint8_t frame_bit_length
);
```

Description

The *MSS_SPI_configure_slave_mode()* function configure a MSS SPI block for operations as a slave SPI device. It configures the SPI hardware with the selected SPI protocol mode and clock speed.

Note: The MSS SPI cannot be configured to operate as a NSC MICROWIRE protocol mode slave.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

protocol_mode

Serial peripheral interface operating mode. Allowed values are:

- MSS_SPI_MODE0
- MSS_SPI_MODE1
- MSS_SPI_MODE2
- MSS_SPI_MODE3
- MSS_SPI_TI_MODE

clk_rate

Divider value used to generate serial interface clock signal from PCLK. Allowed values are:

- MSS_SPI_PCLK_DIV_2
- MSS_SPI_PCLK_DIV_4
- MSS_SPI_PCLK_DIV_8
- MSS_SPI_PCLK_DIV_16
- MSS_SPI_PCLK_DIV_32
- MSS_SPI_PCLK_DIV_64
- MSS_SPI_PCLK_DIV_128
- MSS_SPI_PCLK_DIV_256

frame_bit_length

Number of bits making up the frame. The maximum frame length is 32 bits. You must use the *MSS_SPI_BLOCK_TRANSFER_FRAME_SIZE* constant as the value for *frame_bit_length* when configuring the MSS SPI master for block transfer transactions with the target SPI slave.

Return Value

This function does not return a value.

Example

```
MSS_SPI_init(&g_mss_spi0);
```

```
MSS_SPI_configure_slave_mode  
(  
    &g_mss_spi0,  
    MSS_SPI_MODE2,  
    MSS_SPI_PCLK_DIV_64,  
    MSS_SPI_BLOCK_TRANSFER_FRAME_SIZE  
);
```

MSS_SPI_enable

Prototype

```
void MSS_SPI_enable
(
    mss_spi_instance_t * this_spi
);
```

Description

The *MSS_SPI_enable()* function is used to re-enable a MSS SPI hardware block after it was disabled using the *SPI_disable()* function.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

Return Value

This function does not return a value.

Example

```
uint32_t transfer_size;
uint8_t tx_buffer[8] = {1, 2, 3, 4, 5, 6, 7, 8};

transfer_size = sizeof(tx_buffer);

MSS_SPI_disable(&g_mss_spi0);
MSS_SPI_set_transfer_byte_count(&g_mss_spi0, transfer_size);
PDMA_start
(
    PDMA_CHANNEL_0,
    (uint32_t)tx_buffer,
    PDMA_SPI1_TX_REGISTER,
    transfer_size
);
MSS_SPI_enable(&g_mss_spi0);

while(!MSS_SPI_tx_done(&g_mss_spi0))
{
    ;
}
```

MSS_SPI_disable

Prototype

```
void MSS_SPI_disable
(
    mss_spi_instance_t * this_spi
);
```

Description

The *MSS_SPI_disable()* function is used to temporarily disable a MSS SPI hardware block. This function is typically used in conjunction with the *SPI_set_transfer_byte_count()* function to setup a DMA controlled SPI transmit transaction as the *SPI_set_transfer_byte_count()* function must only be used when the MSS SPI hardware is disabled.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

Return Value

This function does not return a value.

Example

```
uint32_t transfer_size;
uint8_t tx_buffer[8] = {1, 2, 3, 4, 5, 6, 7, 8};

transfer_size = sizeof(tx_buffer);

MSS_SPI_disable(&g_mss_spi0);
MSS_SPI_set_transfer_byte_count(&g_mss_spi0, transfer_size);
PDMA_start
(
    PDMA_CHANNEL_0,
    (uint32_t)tx_buffer,
    PDMA_SPI1_TX_REGISTER,
    transfer_size
);
MSS_SPI_enable(&g_mss_spi0);

while(!MSS_SPI_tx_done(&g_mss_spi0))
{
    ;
}
```

MSS_SPI_set_slave_select

Prototype

```
void MSS_SPI_set_slave_select
(
    mss_spi_instance_t * this_spi,
    mss_spi_slave_t slave
);
```

Description

The *MSS_SPI_slave_select()* function is used by a MSS SPI master to select a specific slave. This function causes the relevant slave select signal to be asserted while data is clocked out onto the SPI data line. This function also configures the MSS SPI master with the configuration settings necessary for communication with the slave. These configuration settings must be specified in a previous call to the *MSS_SPI_configure_master_mode()* function.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

slave

The *slave* parameter is one of the *mss_spi_slave_t* enumerated constants identifying a slave.

Return Value

This function does not return a value.

Example

```
const uint8_t frame_size = 25;
const uint32_t master_tx_frame = 0x0100A0E1;

MSS_SPI_init( &g_mss_spi0 );
MSS_SPI_configure_master_mode
(
    &g_mss_spi0,
    MSS_SPI_SLAVE_0,
    MSS_SPI_MODE1,
    MSS_SPI_PCLK_DIV_256,
    frame_size
);

MSS_SPI_set_slave_select(&g_mss_spi0, MSS_SPI_SLAVE_0);
MSS_SPI_transfer_frame(&g_mss_spi0, master_tx_frame);
MSS_SPI_clear_slave_select(&g_mss_spi0, MSS_SPI_SLAVE_0);
```


MSS_SPI_clear_slave_select

Prototype

```
void MSS_SPI_clear_slave_select
(
    mss_spi_instance_t * this_spi,
    mss_spi_slave_t slave
);
```

Description

The *MSS_SPI_clear_slave_select()* function is used by a MSS SPI master to deselect a specific slave. This function causes the relevant slave select signal to be de-asserted.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

slave

The *slave* parameter is one of the *mss_spi_slave_t* enumerated constants identifying a slave.

Return Value

This function does not return a value.

Example

```
const uint8_t frame_size = 25;
const uint32_t master_tx_frame = 0x0100A0E1;

MSS_SPI_init( &g_mss_spi0 );
MSS_SPI_configure_master_mode
(
    &g_mss_spi0,
    MSS_SPI_SLAVE_0,
    MSS_SPI_MODE1,
    MSS_SPI_PCLK_DIV_256,
    frame_size
);
MSS_SPI_set_slave_select(&g_mss_spi0, MSS_SPI_SLAVE_0);
MSS_SPI_transfer_frame(&g_mss_spi0, master_tx_frame);
MSS_SPI_clear_slave_select(&g_mss_spi0, MSS_SPI_SLAVE_0);
```

MSS_SPI_transfer_frame

Prototype

```
uint32_t MSS_SPI_transfer_frame
(
    mss_spi_instance_t * this_spi,
    uint32_t tx_bits
);
```

Description

The *MSS_SPI_transfer_frame()* function is used by a MSS SPI master to transmit and receive a frame up to 32 bits long. This function is typically used for transactions with a SPI slave where the number of transmit and receive bits is not divisible by 8.

Note: The maximum frame size in NSC Microwire mode is 24 bits organized as an 8 bits command followed by up to 16 bits of data.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

tx_bits

The *tx_bits* parameter is a 32-bit word containing the value that will be transmitted.

Note: The bit length of the value to be transmitted to the slave must be specified as the *frame_bit_length* parameter in a previous call to the *MSS_SPI_configure_master_mode()* function.

Return Value

This function returns a 32-bit word containing the value that is received from the slave.

Example

```
const uint8_t frame_size = 25;
const uint32_t master_tx_frame = 0x0100A0E1;
uint32_t master_rx;

MSS_SPI_init(&g_mss_spi0);
MSS_SPI_configure_master_mode
(
    &g_mss_spi0,
    MSS_SPI_SLAVE_0,
    MSS_SPI_MODE1,
    MSS_SPI_PCLK_DIV_256,
    frame_size
);

MSS_SPI_set_slave_select(&g_mss_spi0, MSS_SPI_SLAVE_0);
master_rx = MSS_SPI_transfer_frame(&g_mss_spi0, master_tx_frame);
MSS_SPI_clear_slave_select(&g_mss_spi0, MSS_SPI_SLAVE_0);
```

MSS_SPI_transfer_block

Prototype

```
void MSS_SPI_transfer_block
(
    mss_spi_instance_t * this_spi,
    const uint8_t * cmd_buffer,
    uint16_t cmd_byte_size,
    uint8_t * rd_buffer,
    uint16_t rd_byte_size
);
```

Description

The *MSS_SPI_transfer_block()* function is used by MSS SPI masters to transmit and receive blocks of data organized as a specified number of bytes. It can be used for:

- Writing a data block to a slave
- Reading a data block from a slave
- Sending a command to a slave followed by reading the outcome of the command in a single SPI transaction. This function can be used alongside peripheral DMA functions to perform the actual moving to and from the SPI hardware block using peripheral DMA.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

cmd_buffer

The *cmd_buffer* parameter is a pointer to the buffer containing the data that will be sent by the master from the beginning of the transfer. This pointer can be null (0) if the master does not need to send a command before reading data or if the command part of the transfer is written to the SPI hardware block using DMA.

cmd_byte_size

The *cmd_byte_size* parameter specifies the number of bytes contained in *cmd_buffer* that will be sent. A value of 0 indicates that no data needs to be sent to the slave. A non-zero value while the *cmd_buffer* pointer is 0 is used to indicate that the command data will be written to the SPI hardware block using DMA.

rd_buffer

The *rd_buffer* parameter is a pointer to the buffer where the data received from the slave after the command has been sent will be stored.

rd_byte_size

The *rd_byte_size* parameter specifies the number of bytes to be received from the slave and stored in the *rd_buffer*. A value of 0 indicates that no data is to be read from the slave. A non-zero value while the *rd_buffer* pointer is null (0) is used to specify the receive size when using DMA to read from the slave.

Note: When using DMA, all bytes received from the slave, including the bytes received while the command is sent, will be read through DMA.

Return Value

This function does not return a value.

Example

Polled write transfer example

```
uint8_t master_tx_buffer[MASTER_TX_BUFFER] =
{
    0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3A
};
MSS_SPI_init(&g_mss_spi0);
MSS_SPI_configure_master_mode
(
    &g_mss_spi0,
    MSS_SPI_SLAVE_0,
    MSS_SPI_MODE1,
    MSS_SPI_PCLK_DIV_256,
    MSS_SPI_BLOCK_TRANSFER_FRAME_SIZE
);

MSS_SPI_set_slave_select(&g_mss_spi0, MSS_SPI_SLAVE_0);
MSS_SPI_transfer_block
(
    &g_mss_spi0,
    master_tx_buffer,
    sizeof(master_tx_buffer),
    0,
    0
);
MSS_SPI_clear_slave_select(&g_mss_spi0, MSS_SPI_SLAVE_0);
```

MSS_SPI_set_slave_tx_frame

Prototype

```
void MSS_SPI_set_slave_tx_frame
(
    mss_spi_instance_t * this_spi,
    uint32_t frame_value
);
```

Description

The *MSS_SPI_set_slave_tx_frame()* function is used by MSS SPI slaves to specify the frame that will be transmitted when a transaction is initiated by the SPI master.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

frame_value

The *frame_value* parameter contains the value of the frame to be sent to the master.

Note: The bit length of the value to be transmitted to the master must be specified as the *frame_bit_length* parameter in a previous call to the *MSS_SPI_configure_slave_mode()* function.

Return Value

This function does not return a value.

Example

```
const uint16_t frame_size = 25;
const uint32_t slave_tx_frame = 0x0110F761;
uint32_t master_rx;

MSS_SPI_init(&g_mss_spi1);
MSS_SPI_configure_slave_mode
(
    &g_mss_spi0,
    MSS_SPI_MODE2,
    MSS_SPI_PCLK_DIV_64,
    frame_size
);
MSS_SPI_set_slave_tx_frame(&g_mss_spi1, slave_tx_frame);
```

MSS_SPI_set_slave_block_buffers

Prototype

```
void MSS_SPI_set_slave_block_buffers
(
    mss_spi_instance_t * this_spi,
    const uint8_t * tx_buffer,
    uint32_t tx_buff_size,
    uint8_t * rx_buffer,
    uint32_t rx_buff_size,
    mss_spi_block_rx_handler_t block_rx_handler
);
```

Description

The *MSS_SPI_set_slave_block_buffers()* function is used to configure an MSS SPI slave for block transfer operations. It specifies one or more of the following:

- The data that will be transmitted when accessed by a master.
- The buffer where data received from a master will be stored.
- The handler function that must be called after the receive buffer has been filled.
- The number of bytes that must be received from the master before the receive handler function is called.

These parameters allow the following use cases:

- Slave performing an action after receiving a block of data from a master containing a command. The action will be performed by the receive handler based on the content of the receive data buffer.
- Slave returning a block of data to the master. The type of information is always the same but the actual values change over time. For example, returning the voltage of a predefined set of analog inputs.
- Slave returning data based on a command contained in the first part of the SPI transaction. For example, reading the voltage of the analog input specified by the first data byte by the master. This is achieved by using the *MSS_SPI_set_slave_block_buffers()* function in conjunction with functions *MSS_SPI_set_cmd_handler()* and *MSS_SPI_set_cmd_response()*. Refer to the *MSS_SPI_set_cmd_handler()* function description for details of this use case.

Note: This function cannot be used on the A2F200 SmartFusion device due to slave mode limitations in the SPI hardware block of this device.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

tx_buffer

The *tx_buffer* parameter is a pointer to a buffer containing the data that will be sent to the master. This parameter can be set to 0 if the MSS SPI slave is not intended to be the target of SPI read.

tx_buff_size

The *tx_buff_size* parameter specifies the number of bytes that will be transmitted by the SPI slave. It is the number of bytes contained in the *tx_buffer*. This parameter can be set to 0 if the MSS SPI slave is not intended to be the target of SPI read transactions.

rx_buffer

The *rx_buffer* parameter is a pointer to the buffer where data received from the master will be stored. This parameter can be set to 0 if the MSS SPI slave is not intended to be the target of SPI write or write-read transactions.

rx_buff_size

The *rx_buff_size* parameter specifies the size of the receive buffer. It is also the number of bytes that must be received before the receive handler is called, if a receive handler is specified using the *block_rx_handler* parameter. This parameter can be set to 0 if the MSS SPI slave is not intended to be the target of SPI write or write-read transactions.

block_rx_handler

The *block_rx_handler* parameter is a pointer to a function that will be called when the receive buffer has been filled. This parameter can be set to 0 if the MSS SPI slave is not intended to be the target of SPI write or write-read transactions.

Return Value

This function does not return a value.

Example

Slave Performing Operation Based on Master Command

In this example the SPI slave is configured to receive 10 bytes of data or command from the SPI slave and process the data received from the master.

```
uint32_t nb_of_rx_handler_calls = 0;

void spil_block_rx_handler_b
(
    uint8_t * rx_buff,
    uint16_t rx_size
)
{
    ++nb_of_rx_handler_calls;
}

void setup_slave( void )
{
    uint8_t slave_rx_buffer[10] =
    {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    };

    MSS_SPI_init(&g_mss_spil);
    MSS_SPI_configure_slave_mode
    (
        &g_mss_spi0,
        MSS_SPI_MODE2,
        MSS_SPI_PCLK_DIV_64,
        MSS_SPI_BLOCK_TRANSFER_FRAME_SIZE
    );

    MSS_SPI_set_slave_block_buffers
    (
        &g_mss_spil,
```

```

    0,
    0,
    slave_rx_buffer,
    sizeof(master_tx_buffer),
    spil_block_rx_handler_b
  );
}

```

Slave Responding to Command

In this example the slave will return data based on a command sent by the master. The first part of the transaction is handled using polled mode where each byte returned to the master is written as part of the interrupt service routine. The second part of the transaction, where the slave returns data based on the command value, is sent using a DMA transfer initiated by the receive handler.

The `MSS_SPI_set_slave_block_buffers()` function specifies that five bytes of preamble data will be transmitted before the actual response is sent by the slave. These five bytes will be transmitted while the one byte command, followed by four turnaround bytes, is received. The turnaround bytes give time to the slave to call the command handler and setup the PDMA transfer that will return the response data.

```

#define SLAVE_NB_OF_COMMANDS    4
#define SLAVE_PACKET_LENGTH    8

const uint8_t g_slave_data_array[SLAVE_NB_OF_COMMANDS][SLAVE_PACKET_LENGTH] =
{
    {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
    {0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18},
    {0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28},
    {0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38}
};

void spil_rx_handler_d
(
    uint8_t * rx_buff,
    uint32_t rx_size
)
{
    uint8_t index;
    const uint8_t * p_response;

    index = rx_buff[0];

    if(index < 4)
    {
        p_response = &g_slave_data_array[index][0];
    }
    else
    {
        p_response = &g_slave_data_array[0][0];
    }

    PDMA_start
    (

```



```
        PDMA_CHANNEL_0,
        (uint32_t)p_response,
        PDMA_SPI1_TX_REGISTER,
        SLAVE_PACKET_LENGTH
    );
}

void setup_slave( void )
{
    uint8_t slave_tx_buffer[16] = { 0x12, 0x34, 0x56, 0x78, 0x9A };

    PDMA_init();
    PDMA->RATIO_HIGH_LOW = 0xFF;
    PDMA_configure
    (
        PDMA_CHANNEL_0,
        PDMA_TO_SPI_1,
        PDMA_LOW_PRIORITY | PDMA_BYTE_TRANSFER | PDMA_INC_SRC_ONE_BYTE,
        PDMA_DEFAULT_WRITE_ADJ
    );

    MSS_SPI_init( &g_mss_spil );

    MSS_SPI_configure_slave_mode
    (
        &g_mss_spil,
        MSS_SPI_MODEL,
        MSS_SPI_PCLK_DIV_256,
        MSS_SPI_BLOCK_TRANSFER_FRAME_SIZE
    );

    MSS_SPI_set_slave_block_buffers
    (
        &g_mss_spil,
        slave_tx_buffer,
        5,
        slave_rx_buffer,
        sizeof(slave_rx_buffer),
        spil_rx_handler_d
    );

    MSS_SPI_set_cmd_handler
    (
        &g_mss_spil,
        spil_slave_cmd_handler,
        1
    );
}
```

MSS_SPI_set_frame_rx_handler

Prototype

```
void MSS_SPI_set_frame_rx_handler
(
    mss_spi_instance_t * this_spi,
    mss_spi_frame_rx_handler_t rx_handler
);
```

Description

The *MSS_SPI_set_frame_rx_handler()* function is used by MSS SPI slaves to specify the receive handler function that will be called by the MSS SPI driver interrupt handler when a frame of data is received by the MSS SPI slave.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

rx_handler

The *rx_handler* parameter is a pointer to the frame receive handler that must be called when a frame is received by the MSS SPI slave.

Return Value

This function does not return a value.

Example

```
uint32_t g_slave_rx_frame = 0;

void slave_frame_handler(uint32_t rx_frame)
{
    g_slave_rx_frame = rx_frame;
}

int setup_slave( void )
{
    const uint16_t frame_size = 25;
    MSS_SPI_init(&g_mss_spi1);
    MSS_SPI_configure_slave_mode
    (
        &g_mss_spi0,
        MSS_SPI_MODE2,
        MSS_SPI_PCLK_DIV_64,
        frame_size
    );
    MSS_SPI_set_frame_rx_handler(&g_mss_spi1, slave_frame_handler);
}
```

MSS_SPI_set_transfer_byte_count

Prototype

```
void MSS_SPI_set_transfer_byte_count
(
    mss_spi_instance_t * this_spi,
    uint16_t byte_count
);
```

Description

The *MSS_SPI_set_transfer_byte_count()* function is used as part of setting up a SPI transfer using DMA. It specifies the number of bytes that must be transferred before *MSS_SPI_tx_done()* indicates that the transfer is complete.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

byte_count

The *byte_count* parameter specifies the number of bytes that must be transferred by the SPI hardware block before *MSS_SPI_tx_done()* indicates that the transfer is complete.

Return Value

This function does not return a value.

Example

```
uint32_t transfer_size;
uint8_t tx_buffer[8] = {1, 2, 3, 4, 5, 6, 7, 8};

transfer_size = sizeof(tx_buffer);

MSS_SPI_disable(&g_mss_spi0);

MSS_SPI_set_transfer_byte_count(&g_mss_spi0, transfer_size);

PDMA_start(PDMA_CHANNEL_0, (uint32_t)tx_buffer, PDMA_SPI1_TX_REGISTER, transfer_size);

MSS_SPI_enable(&g_mss_spi0);

while(!MSS_SPI_tx_done(&g_mss_spi0))
{
    ;
}
```

MSS_SPI_tx_done

Prototype

```
uint32_t MSS_SPI_tx_done  
(  
    mss_spi_instance_t * this_spi  
);
```

Description

The *MSS_SPI_tx_done()* function is used to find out if a DMA controlled transfer has completed.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block to be initialized. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

Return Value

This function indicates if a SPI transfer has completed. It returns 1 if the number of bytes specified through a previous call to *MSS_SPI_set_transfer_byte_count()* has been sent by the MSS SPI. It returns 0 if some bytes remain to be sent.

Example

```
uint32_t transfer_size;  
uint8_t tx_buffer[8] = {1, 2, 3, 4, 5, 6, 7, 8};  
  
transfer_size = sizeof(tx_buffer);  
  
MSS_SPI_disable(&g_mss_spi0);  
  
MSS_SPI_set_transfer_byte_count(&g_mss_spi0, transfer_size);  
  
PDMA_start  
(  
    PDMA_CHANNEL_0,  
    (uint32_t)tx_buffer,  
    PDMA_SPI1_TX_REGISTER,  
    transfer_size  
);  
  
MSS_SPI_enable(&g_mss_spi0);  
  
while(!MSS_SPI_tx_done(&g_mss_spi0))  
{  
    ;  
}
```

MSS_SPI_set_cmd_handler

Prototype

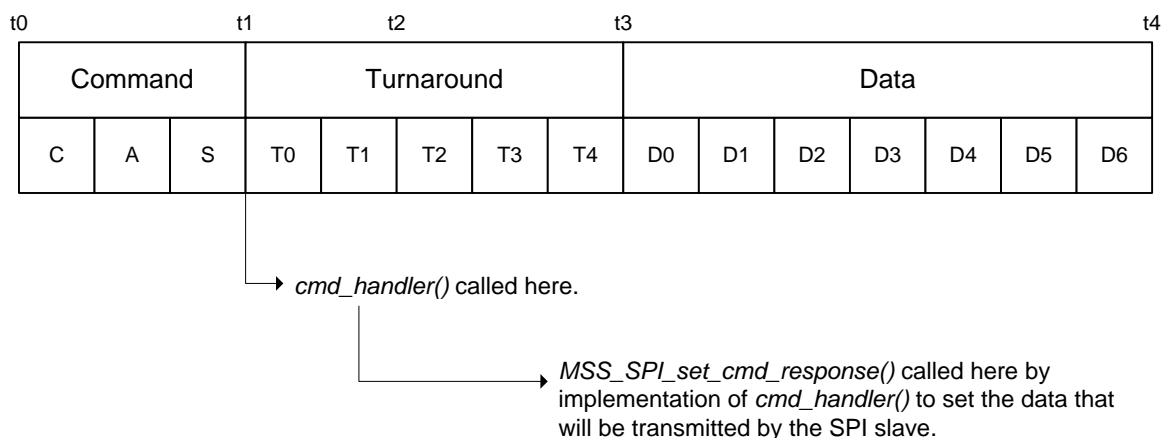
```
void MSS_SPI_set_cmd_handler
(
    mss_spi_instance_t * this_spi,
    mss_spi_block_rx_handler_t cmd_handler,
    uint32_t cmd_size
);
```

Description

The *MSS_SPI_set_cmd_handler()* function specifies a command handler function that will be called when the number of bytes received reaches the command size specified as parameter.

This function is used by SPI slaves performing block transfers. Its purpose is to allow a SPI slave to decide the data that will be returned to the master while a SPI transaction is taking place. Typically, one or more command bytes are sent by the master to request some specific data. The slave interprets the command byte(s) while one or more turn-around bytes are transmitted. The slave adjusts its transmit data buffer based on the command during the turnaround time.

The diagram below provides an example of the use of this function where the SPI slave returns data bytes D0 to D6 based on the value of a command. The 3 bytes long command is made up of a command opcode byte followed by an address byte followed by a size byte. The *cmd_handler()* function specified through an earlier call to *MSS_SPI_set_cmd_handler()* is called by the SPI driver once the third byte is received. The *cmd_handler()* function interprets the command bytes and calls *MSS_SPI_set_cmd_response()* to set the SPI slave's response transmit buffer with the data to be transmitted after the turn around bytes (T0 to T3). The number of turnaround bytes must be sufficient to give enough time for the *cmd_handler()* to execute. The number of turnaround bytes is specified by the protocol used on top of the SPI transport layer so that master and slave agree on the number of turnaround bytes.



Note: This function cannot be used on the A2F200 SmartFusion device due to slave mode limitations in the SPI hardware block of this device.

Parameters

this_spi

The `this_spi` parameter is a pointer to an `mss_spi_instance_t` structure identifying the MSS SPI hardware block used. There are two such data structures, `g_mss_spi0` and `g_mss_spi1`, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the `g_mss_spi0` or `g_mss_spi1` global data structure defined within the SPI driver.

cmd_handler

The `cmd_handler` parameter is a pointer to a function with prototype:

```
void cmd_handler(uint8_t * rx_buff, uint32_t rx_size);
```

It specifies the function that will be called when the number of bytes specified by parameter `cmd_size` has been received.

cmd_size

The `cmd_size` parameter specifies the number of bytes that must be received before the command handler function specified by `cmd_handler` is called.

Return Value

This function does not return a value.

Example

The example below demonstrates how to configure SPI1 to implement the protocol given as example in the diagram above. The `configure_slave()` function configures SPI1. It sets the receive and transmit buffers. The transmit buffer specified through the call to `MSS_SPI_set_slave_block_buffers()` specifies the data that will be returned to the master in bytes between `t0` and `t3`. These are the bytes that will be sent to the master while the master transmits the command and dummy bytes. The `spi1_slave_cmd_handler()` function will be called by the driver at time `t1` after the 3 command bytes have been received. The `spi1_block_rx_handler()` function will be called by the driver at time `t4` when the transaction completes when the slave select signal becomes de-asserted.

```
#define COMMAND_SIZE          3
#define NB_OF_DUMMY_BYTES    4
#define MAX_TRANSACTION_SIZE  16

uint8_t slave_tx_buffer[COMMAND_SIZE + NB_OF_DUMMY_BYTES];
uint8_t slave_rx_buffer[MAX_TRANSACTION_SIZE];

void configure_slave(void)
{
    MSS_SPI_init( &g_mss_spi1 );

    MSS_SPI_configure_slave_mode
    (
        &g_mss_spi1,
        MSS_SPI_MODE1,
        MSS_SPI_PCLK_DIV_256,
        MSS_SPI_BLOCK_TRANSFER_FRAME_SIZE
    );

    MSS_SPI_set_slave_block_buffers
    (
        &g_mss_spi1,
        slave_tx_buffer,
        COMMAND_SIZE + NB_OF_DUMMY_BYTES,
```

```
        slave_rx_buffer,  
        sizeof(slave_rx_buffer),  
        spil_block_rx_handler  
    );  
  
MSS_SPI_set_cmd_handler  
(  
    &g_mss_spil,  
    spil_slave_cmd_handler,  
    COMMAND_SIZE  
);  
}  
  
void spil_slave_cmd_handler  
(  
    uint8_t * rx_buff,  
    uint32_t rx_size  
)  
{  
    uint8_t command;  
    uint8_t address;  
    uint8_t size;  
  
    uint8_t * p_response;  
    uint32_t response_size;  
  
    command = rx_buff[0];  
    address = rx_buff[1];  
    size = rx_buff[2];  
  
    p_response = get_response_data(command, address, size, &response_size);  
  
    MSS_SPI_set_cmd_response(&g_mss_spil, p_response, response_size);  
}  
  
void spil_block_rx_handler  
(  
    uint8_t * rx_buff,  
    uint32_t rx_size  
)  
{  
    process_rx_data(rx_buff, rx_size);  
}
```

MSS_SPI_set_cmd_response

Prototype

```
void MSS_SPI_set_cmd_response  
(  
    mss_spi_instance_t * this_spi,  
    const uint8_t * resp_tx_buffer,  
    uint32_t resp_buff_size  
);
```

Description

The *MSS_SPI_set_cmd_response()* function specifies the data that will be returned to the master. See the description of *MSS_SPI_set_cmd_handler()* for details.

Note: This function cannot be used on the A2F200 SmartFusion device due to slave mode limitations in the SPI hardware block of this device.

Parameters

this_spi

The *this_spi* parameter is a pointer to an *mss_spi_instance_t* structure identifying the MSS SPI hardware block used. There are two such data structures, *g_mss_spi0* and *g_mss_spi1*, associated with MSS SPI 0 and MSS SPI 1 respectively. This parameter must point to either the *g_mss_spi0* or *g_mss_spi1* global data structure defined within the SPI driver.

resp_tx_buffer

The *resp_tx_buffer* parameter is a pointer to the buffer containing the data that must be returned to the host in the data phase of a SPI transaction.

resp_buff_size

The *resp_buff_size* parameter specifies the size of the buffer pointed to by the *resp_tx_buffer* parameter.

Return Value

This function does not return a value.

Product Support

Microsemi backs its products with various support services including Customer Service, a Customer Technical Support Center, a web site, an FTP site, electronic mail, and worldwide sales offices. This appendix contains information about contacting Microsemi and using these support services.

Customer Service

Contact Customer Service for non-technical product support, such as product pricing, product upgrades, update information, order status, and authorization.

Actel's customer service representatives are available Monday through Friday, from 8 AM to 5 PM Pacific Time, to answer non-technical questions.

Phone: +1 650.318.2470

Customer Technical Support Center

Microsemi staffs its Customer Technical Support Center with highly skilled engineers who can help answer your hardware, software, and design questions. The Customer Technical Support Center spends a great deal of time creating application notes and answers to FAQs. So, before you contact us, please visit our online resources. It is very likely we have already answered your questions.

Technical Support

Microsemi customers can receive technical support on Microsemi SoC products by calling Technical Support Hotline anytime Monday through Friday. Customers also have the option to interactively submit and track cases online at [My Cases](#) or submit questions through [email](#) anytime during the week.

Web: www.actel.com/mycases

Phone (North America): 1.800.262.1060

Phone (International): +1 650.318.4460

Email: soc_tech@microsemi.com

ITAR Technical Support

Microsemi customers can receive ITAR technical support on Microsemi SoC products by calling ITAR Technical Support Hotline: Monday through Friday, from 9 AM to 6 PM Pacific Time. Customers also have the option to interactively submit and track cases online at [My Cases](#) or submit questions through [email](#) anytime during the week.

Web: www.actel.com/mycases

Phone (North America): 1.888.988.ITAR

Phone (International): +1 650.318.4900

Email: soc_tech_itar@microsemi.com



Microsemi Corporate Headquarters
2381 Morse Avenue, Irvine, CA 92614
Phone; 949.221.7100 · Fax: 949.756.0308
www.microsemi.com

Microsemi Corporation (NASDAQ: MSCC) offers the industry's most comprehensive portfolio of semiconductor technology. Committed to solving the most critical system challenges, Microsemi's products include high-performance, high-reliability analog and RF devices, mixed signal integrated circuits, FPGAs and customizable SoCs, and complete subsystems. Microsemi serves leading system manufacturers around the world in the defense, security, aerospace, enterprise, commercial, and industrial markets. Learn more at www.microsemi.com.

© 2011 Microsemi Corporation. All rights reserved. Microsemi and the Microsemi logo are trademarks of Microsemi Corporation. All other trademarks and service marks are the property of their respective owners.