

# ***Actel SmartFusion™ MSS SPI Driver User's Guide***

*Version 2.0*

---

## **Actel Corporation, Mountain View, CA 94043**

© 2010 Actel Corporation. All rights reserved.

Printed in the United States of America

Part Number: 50200191-1

Release: February 2010

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Actel.

Actel makes no warranties with respect to this documentation and disclaims any implied warranties of merchantability or fitness for a particular purpose. Information in this document is subject to change without notice. Actel assumes no responsibility for any errors that may appear in this document.

This document contains confidential proprietary information that is not to be disclosed to any unauthorized person without prior written consent of Actel Corporation.

### **Trademarks**

Actel, Actel Fusion, IGLOO, Libero, Pigeon Point, ProASIC, SmartFusion and the associated logos are trademarks or registered trademarks of Actel Corporation. All other trademarks and service marks are the property of their respective owners.

# Table of Contents

<b>Introduction.....</b>	<b>5</b>
Features .....	5
Supported Hardware IP .....	5
<b>Files Provided .....</b>	<b>7</b>
Documentation .....	7
Driver Source Code .....	7
Example Code.....	7
<b>Driver Deployment .....</b>	<b>9</b>
<b>Driver Configuration.....</b>	<b>11</b>
<b>Application Programming Interface.....</b>	<b>13</b>
Theory of Operation .....	13
Types.....	16
Constant Values .....	18
Data structures .....	18
Global Variables.....	18
Functions .....	19
<b>Product Support.....</b>	<b>41</b>
Customer Service .....	41
Actel Customer Technical Support Center .....	41
Actel Technical Support .....	41
Website .....	41
Contacting the 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.



---

## 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](http://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.

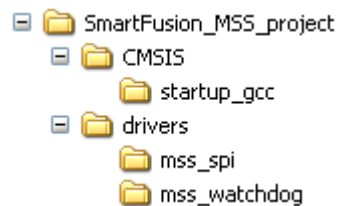


Figure 1 · SmartFusion MSS Project Example



---

## 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, or it defines whether the SPI block will operate in TI synchronous serial mode or in 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, or it defines whether the SPI block will operate in TI synchronous serial mode or in NSC MICROWIRE mode.

### 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 to 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. Alternatively a GPIO can be used to control the state of the target slave device's chip select signal.

A call to 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.

## 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 will be 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()*

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

## 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    = 0x00000004,
    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 , uint16_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 , uint16_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.

### 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\_TI\_MODE
- MSS\_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_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.

### 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 *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 *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.

### 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()* 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:** 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 );
```

### DMA transfer

In this example, the transmit and receive buffers are not specified as part of the call to *MSS\_SPI\_transfer\_block()*. *MSS\_SPI\_transfer\_block()* will only prepare the MSS SPI hardware for a transfer. The MSS SPI transmit hardware FIFO is filled using one DMA channel and a second DMA channel is used to read the content of the MSS SPI receive hardware FIFO. The transmit and receive buffers are specified by two separate calls to *PDMA\_start()* to initiate DMA transfers on the channel used for transmit data and the channel used for receive data.

```
uint8_t master_tx_buffer[MASTER_RX_BUFFER] =
{
    0xC1, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7, 0xC8, 0xC9, 0xCA
};
uint8_t slave_rx_buffer[MASTER_RX_BUFFER] =
{
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A
};
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, 0, 0, 0, 0 );
PDMA_start
(
    PDMA_CHANNEL_1,
    PDMA_SPI0_RX_REGISTER,
    (uint32_t)master_rx_buffer,
    sizeof(master_rx_buffer)
);
PDMA_start
(
    PDMA_CHANNEL_2,
    (uint32_t)master_tx_buffer,
    PDMA_SPI0_TX_REGISTER,
    sizeof(master_tx_buffer)
);
while( PDMA_status(PDMA_CHANNEL_1) == 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()* 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 handling 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 setting the *rx\_buff\_size* parameter to the number of received bytes making up the command.

### 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 transactions or if DMA is used to transfer SPI read data into the MSS SPI slave.

#### **tx\_buff\_size**

The *tx\_buff\_size* parameter specifies 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 or if DMA is used to transfer SPI read data into the MSS SPI slave.

**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. It can also be set to 0 if the MSS SPI slave uses DMA to handle data written to it.

**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. It can also be set to 0 if the MSS SPI slave uses DMA to handle data written to it.

**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. It can also be set to 0 if the MSS SPI slave uses DMA to handle data written to it.

**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.

```

static uint8_t g_spil_tx_buffer_b[SLAVE_TX_BUFFER_SIZE] =
{
    5, 6, 7, 8, 0xA0, 0xA1, 0xA2, 0xA3, 0xA4, 0xA5
};

void spil_block_rx_handler
(
    uint8_t * rx_buff,
    uint16_t rx_size
)
{
    if ( rx_buff[2] == 0x99 )
    {
        PDMA_start
        (
            PDMA_CHANNEL_0,
            (uint32_t)g_spil_tx_buffer_b,
            0x40011014,
            sizeof(g_spil_tx_buffer_b)
        );
    }
}

void setup_slave( void )
{
    uint8_t slave_preamble[8] = { 9, 10, 11, 12, 13, 14, 16, 16 };

    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  
    );  
  
    PDMA_init();  
    PDMA_configure  
    (  
        PDMA_CHANNEL_0,  
        TO_SPI_1,  
        LOW_PRIORITY | BYTE_TRANSFER | INC_SRC_ONE_BYTE  
    );  
  
    MSS_SPI_set_slave_block_buffers  
    (  
        &g_mss_spil,  
        slave_preamble,  
        4,  
        g_spil_rx_buffer,  
        sizeof(g_spil_rx_buffer),  
        spil_block_rx_handler  
    );  
}
```

## 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 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 considering that a transaction has been completed.

### 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, 0x40011014, 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) )
{
    ;
}
```



---

## Product Support

Actel 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 Actel 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.

From Northeast and North Central U.S.A., call **650.318.4480**

From Southeast and Southwest U.S.A., call **650.318.4480**

From South Central U.S.A., call **650.318.4434**

From Northwest U.S.A., call **650.318.4434**

From Canada, call **650.318.4480**

From Europe, call **650.318.4252** or **+44 (0) 1276 401 500**

From Japan, call **650.318.4743**

From the rest of the world, call **650.318.4743**

Fax, from anywhere in the world **650.318.8044**

### Actel Customer Technical Support Center

Actel 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.

### Actel Technical Support

Visit the [Actel Customer Support website \(http://www.actel.com/support/search/default.aspx\)](http://www.actel.com/support/search/default.aspx) for more information and support. Many answers available on the searchable web resource include diagrams, illustrations, and links to other resources on the Actel web site.

### Website

You can browse a variety of technical and non-technical information on Actel's [home page](http://www.actel.com/), at <http://www.actel.com/>.

### Contacting the Customer Technical Support Center

Highly skilled engineers staff the Technical Support Center from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. Several ways of contacting the Center follow:

#### Email

You can communicate your technical questions to our email address and receive answers back by email, fax, or phone. Also, if you have design problems, you can email your design files to receive assistance. We constantly monitor the email account throughout the day. When sending your request to us, please be sure to include your full name, company name, and your contact information for efficient processing of your request.

The technical support email address is [tech@actel.com](mailto:tech@actel.com).

## Phone

Our Technical Support Center answers all calls. The center retrieves information, such as your name, company name, phone number and your question, and then issues a case number. The Center then forwards the information to a queue where the first available application engineer receives the data and returns your call. The phone hours are from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. The Technical Support numbers are:

**650.318.4460**  
**800.262.1060**

Customers needing assistance outside the US time zones can either contact technical support via email ([tech@actel.com](mailto:tech@actel.com)) or contact a local sales office. [Sales office listings](#) can be found at [www.actel.com/company/contact/default.aspx](http://www.actel.com/company/contact/default.aspx).





**Actel is the leader in low-power FPGAs and mixed-signal FPGAs and offers the most comprehensive portfolio of system and power management solutions. Power Matters. Learn more at <http://www.actel.com> .**

**Actel Corporation** • 2061 Stierlin Court • Mountain View, CA 94043 • USA

Phone 650.318.4200 • Fax 650.318.4600 • Customer Service: 650.318.1010 • Customer Applications Center: 800.262.1060

**Actel Europe Ltd.** • River Court, Meadows Business Park • Station Approach, Blackwater • Camberley Surrey GU17 9AB • United Kingdom  
Phone +44 (0) 1276 609 300 • Fax +44 (0) 1276 607 540

**Actel Japan** • EXOS Ebisu Building 4F • 1-24-14 Ebisu Shibuya-ku • Tokyo 150 • Japan

Phone +81.03.3445.7671 • Fax +81.03.3445.7668 • <http://jlp.actel.com>

**Actel Hong Kong** • Room 2107, China Resources Building • 26 Harbour Road • Wanchai • Hong Kong

Phone +852 2185 6460 • Fax +852 2185 6488 • [www.actel.com.cn](http://www.actel.com.cn)