SMART ARM-based Microcontrollers

AT03255: SAM D/R/L/C Serial Peripheral Interface (SERCOM SPI) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the SERCOM module in its SPI mode to transfer SPI data frames. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- SERCOM (Serial Communication Interface)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM D09/D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

Table of Contents

1. Software License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of Atmel may not be used to endorse or promote products derived from this software without specific prior written permission.

4. This software may only be redistributed and used in connection with an Atmel microcontroller product.

THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2. Prerequisites

There are no prerequisites.

3. Module Overview

The Serial Peripheral Interface (SPI) is a high-speed synchronous data transfer interface using three or four pins. It allows fast communication between a master device and one or more peripheral devices.

A device connected to the bus must act as a master or a slave. The master initiates and controls all data transactions. The SPI master initiates a communication cycle by pulling low the Slave Select (SS) pin of the desired slave. The Slave Select pin is active low. Master and slave prepare data to be sent in their respective shift registers, and the master generates the required clock pulses on the SCK line to interchange data. Data is always shifted from master to slave on the Master Out - Slave In (MOSI) line, and from slave to master on the Master In - Slave Out (MISO) line. After each data transfer, the master can synchronize to the slave by pulling the SS line high.

3.1. Driver Feature Macro Definition

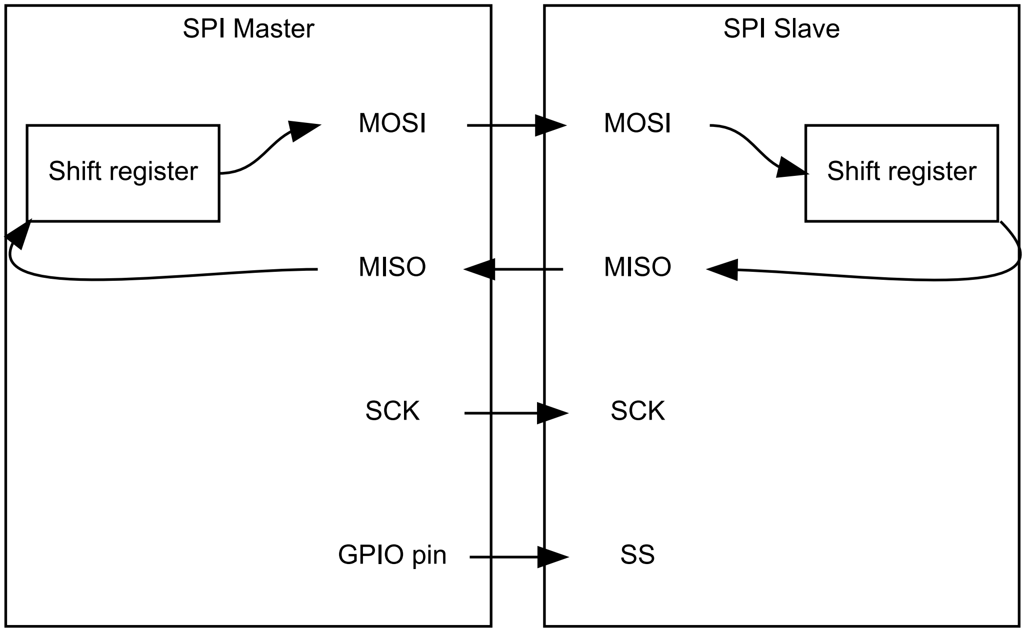
Driver feature macro	Supported devices
FEATURE_SPI_SLAVE_SELECT_LOW_DETECT	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_SPI_HARDWARE_SLAVE_SELECT	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_SPI_ERROR_INTERRUPT	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21
FEATURE_SPI_SYNC_SCHEME_VERSION_2	SAM D21/R21/D10/D11/L21/L22/DA1/C20/C21

Note: The specific features are only available in the driver when the selected device supports those features.

3.2. SPI Bus Connection

In Figure 3-1 SPI Bus Connection on page 8, the connection between one master and one slave is shown.

Figure 3-1 SPI Bus Connection



The different lines are as follows:

- **MISO** Master Input Slave Output. The line where the data is shifted out from the slave and into the master.
- **MOSI** Master Output Slave Input. The line where the data is shifted out from the master and into the slave.
- **SCK** Serial Clock. Generated by the master device.
- **SS** Slave Select. To initiate a transaction, the master must pull this line low.

If the bus consists of several SPI slaves, they can be connected in parallel and the SPI master can use general I/O pins to control separate SS lines to each slave on the bus.

It is also possible to connect all slaves in series. In this configuration, a common SS is provided to `N` slaves, enabling them simultaneously. The MISO from the `N-1` slaves is connected to the MOSI on the next slave. The `Nth` slave connects its MISO back to the master. For a complete transaction, the master must shift `N+1` characters.

3.3. SPI Character Size

The SPI character size is configurable to eight or nine bits.

3.4. Master Mode

When configured as a master, the SS pin will be configured as an output.

3.4.1. Data Transfer

Writing a character will start the SPI clock generator, and the character is transferred to the shift register when the shift register is empty. Once this is done, a new character can be written. As each character is shifted out from the master, a character is shifted in from the slave. If the receiver is enabled, the data is moved to the receive buffer at the completion of the frame and can be read.

3.5. Slave Mode

When configured as a slave, the SPI interface will remain inactive with MISO tri-stated as long as the SS pin is driven high.

3.5.1. Data Transfer

The data register can be updated at any time. As the SPI slave shift register is clocked by SCK, a minimum of three SCK cycles are needed from the time new data is written, until the character is ready to be shifted out. If the shift register has not been loaded with data, the current contents will be transmitted.

If constant transmission of data is needed in SPI slave mode, the system clock should be faster than SCK. If the receiver is enabled, the received character can be read from the receive buffer. When SS line is driven high, the slave will not receive any additional data.

3.5.2. Address Recognition

When the SPI slave is configured with address recognition, the first character in a transaction is checked for an address match. If there is a match, the MISO output is enabled and the transaction is processed. If the address does not match, the complete transaction is ignored.

If the device is asleep, it can be woken up by an address match in order to process the transaction.

Note: In master mode, an address packet is written by the spi_select_slave function if the address_enabled configuration is set in the spi_slave_inst_config struct.

3.6. Data Modes

There are four combinations of SCK phase and polarity with respect to serial data. Table 3-1 SPI Data Modes on page 9 shows the clock polarity (CPOL) and clock phase (CPHA) in the different modes. *Leading edge* is the first clock edge in a clock cycle and *trailing edge* is the last clock edge in a clock cycle.

Table 3-1 SPI Data Modes

Mode	CPOL	CPHA	Leading Edge	Trailing Edge
0	0	0	Rising, Sample	Falling, Setup
1	0	1	Rising, Setup	Falling, Sample
2	1	0	Falling, Sample	Rising, Setup
3	1	1	Falling, Setup	Rising, Sample

3.7. SERCOM Pads

The SERCOM pads are automatically configured as seen in Table 3-2 SERCOM SPI Pad Usages on page 10. If the receiver is disabled, the data input (MISO for master, MOSI for slave) can be used for other purposes.

In master mode, the SS pin(s) must be configured using the spi_slave_inst struct.

Table 3-2 SERCOM SPI Pad Usages

Pin	Master SPI	Slave SPI
MOSI	Output	Input
MISO	Input	Output
SCK	Output	Input
SS	User defined output enable	Input

3.8. Operation in Sleep Modes

The SPI module can operate in all sleep modes by setting the run_in_standby option in the spi_config struct. The operation in slave and master mode is shown in the table below.

run_in_standby	Slave	Master
false	Disabled, all reception is dropped	GCLK is disabled when master is idle, wake on transmit complete
true	Wake on reception	GCLK is enabled while in sleep modes, wake on all interrupts

3.9. Clock Generation

In SPI master mode, the clock (SCK) is generated internally using the SERCOM baudrate generator. In SPI slave mode, the clock is provided by an external master on the SCK pin. This clock is used to directly clock the SPI shift register.

4. Special Considerations

4.1. pinmux Settings

The pin MUX settings must be configured properly, as not all settings can be used in different modes of operation.

5. Extra Information

For extra information, see Extra Information for SERCOM SPI Driver. This includes:

- Acronyms
- Dependencies
- Workarounds Implemented by Driver
- Module History

6. Examples

For a list of examples related to this driver, see Examples for SERCOM SPI Driver.

7. API Overview

7.1. Variable and Type Definitions

7.1.1. Type spi_callback_t

```
typedef void(* spi_callback_t )(struct spi_module *const module)
```

Type of the callback functions

7.2. Structure Definitions

7.2.1. Struct spi_config

Configuration structure for an SPI instance. This structure should be initialized by the spi_get_config_defaults function before being modified by the user application.

Table 7-1 Members

Type	Name	Description
enum spi_character_size	character_size	SPI character size
enum spi_data_order	data_order	Data order
enum gclk_generator	generator_source	GCLK generator to use as clock source
bool	master_slave_select_enable	Enable Master Slave Select
enum spi_mode	mode	SPI mode
union spi_config.mode_specific	mode_specific	Union for slave or master specific configuration
enum spi_signal_mux_setting	mux_setting	MUX setting
uint32_t	pinmux_pad0	PAD0 pinmux
uint32_t	pinmux_pad1	PAD1 pinmux
uint32_t	pinmux_pad2	PAD2 pinmux
uint32_t	pinmux_pad3	PAD3 pinmux
bool	receiver_enable	Enable receiver
bool	run_in_standby	Enabled in sleep modes

Type	Name	Description
bool	select_slave_low_detect_enable	Enable Slave Select Low Detect
enum spi_transfer_mode	transfer_mode	Transfer mode

7.2.2. Union spi_config.mode_specific

Union for slave or master specific configuration

Table 7-2 Members

Type	Name	Description
struct spi_master_config	master	Master specific configuration
struct spi_slave_config	slave	Slave specific configuration

7.2.3. Struct spi_master_config

SPI Master configuration structure.

Table 7-3 Members

Type	Name	Description
uint32_t	baudrate	Baud rate

7.2.4. Struct spi_module

SERCOM SPI driver software instance structure, used to retain software state information of an associated hardware module instance.

Note: The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

7.2.5. Struct spi_slave_config

SPI slave configuration structure.

Table 7-4 Members

Type	Name	Description
uint8_t	address	Address
uint8_t	address_mask	Address mask
enum spi_addr_mode	address_mode	Address mode
enum spi_frame_format	frame_format	Frame format
bool	preload_enable	Preload data to the shift register while SS is high

7.2.6. Struct spi_slave_inst

SPI peripheral slave software instance structure, used to configure the correct SPI transfer mode settings for an attached slave. See spi_select_slave.

Table 7-5 Members

Type	Name	Description
uint8_t	address	Address of slave device
bool	address_enabled	Address recognition enabled in slave device
uint8_t	ss_pin	Pin to use as slave select

7.2.7. Struct spi_slave_inst_config

SPI Peripheral slave configuration structure.

Table 7-6 Members

Type	Name	Description
uint8_t	address	Address of slave
bool	address_enabled	Enable address
uint8_t	ss_pin	Pin to use as slave select

7.3. Macro Definitions

7.3.1. Driver Feature Definition

Define SERCOM SPI features set according to different device family.

7.3.1.1. Macro FEATURE_SPI_SLAVE_SELECT_LOW_DETECT

```
#define FEATURE_SPI_SLAVE_SELECT_LOW_DETECT
```

SPI slave select low detection.

7.3.1.2. Macro FEATURE_SPI_HARDWARE_SLAVE_SELECT

```
#define FEATURE_SPI_HARDWARE_SLAVE_SELECT
```

Slave select can be controlled by hardware.

7.3.1.3. Macro FEATURE_SPI_ERROR_INTERRUPT

```
#define FEATURE_SPI_ERROR_INTERRUPT
```

SPI with error detect feature.

7.3.1.4. Macro FEATURE_SPI_SYNC_SCHEME_VERSION_2

```
#define FEATURE_SPI_SYNC_SCHEME_VERSION_2
```

SPI sync scheme version 2.

7.3.2. Macro PINMUX_DEFAULT

```
#define PINMUX_DEFAULT
```

Default pinmux.

7.3.3. Macro PINMUX_UNUSED

```
#define PINMUX_UNUSED
```

Unused pinmux.

7.3.4. Macro SPI_TIMEOUT

```
#define SPI_TIMEOUT
```

SPI timeout value.

7.4. Function Definitions

7.4.1. Driver Initialization and Configuration

7.4.1.1. Function spi_get_config_defaults()

Initializes an SPI configuration structure to default values.

```
void spi_get_config_defaults(
        struct spi_config *const config)
```

This function will initialize a given SPI configuration structure to a set of known default values. This function should be called on any new instance of the configuration structures before being modified by the user application.

The default configuration is as follows:

- Master mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- MUX Setting D
- Character size eight bits
- Not enabled in sleep mode
- Receiver enabled
- Baudrate 100000
- Default pinmux settings for all pads
- GCLK generator 0

Table 7-7 Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

7.4.1.2. Function spi_slave_inst_get_config_defaults()

Initializes an SPI peripheral slave device configuration structure to default values.

```
void spi_slave_inst_get_config_defaults(
        struct spi_slave_inst_config *const config)
```

This function will initialize a given SPI slave device configuration structure to a set of known default values. This function should be called on any new instance of the configuration structures before being modified by the user application.

The default configuration is as follows:

- Slave Select on GPIO pin 10
- Addressing not enabled

Table 7-8 Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

7.4.1.3. Function spi_attach_slave()

Attaches an SPI peripheral slave.

```
void spi_attach_slave(
        struct spi_slave_inst *const slave,
        const struct spi_slave_inst_config *const config)
```

This function will initialize the software SPI peripheral slave, based on the values of the config struct. The slave can then be selected and optionally addressed by the spi_select_slave function.

Table 7-9 Parameters

Data direction	Parameter name	Description
[out]	slave	Pointer to the software slave instance struct
[in]	config	Pointer to the config struct

7.4.1.4. Function spi_init()

Initializes the SERCOM SPI module.

```
enum status_code spi_init(
        struct spi_module *const module,
        Sercom *const hw,
        const struct spi_config *const config)
```

This function will initialize the SERCOM SPI module, based on the values of the config struct.

Table 7-10 Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to the software instance struct
[in]	hw	Pointer to hardware instance
[in]	config	Pointer to the config struct

Returns

Status of the initialization.

Table 7-11 Return Values

Return value	Description
STATUS_OK	Module initiated correctly
STATUS_ERR_DENIED	If module is enabled
STATUS_BUSY	If module is busy resetting
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided

7.4.2. Enable/Disable

7.4.2.1. Function spi_enable()

Enables the SERCOM SPI module.

```
void spi_enable(
        struct spi_module *const module)
```

This function will enable the SERCOM SPI module.

Table 7-12 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

7.4.2.2. Function spi_disable()

Disables the SERCOM SPI module.

```
void spi_disable(
        struct spi_module *const module)
```

This function will disable the SERCOM SPI module.

Table 7-13 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

7.4.2.3. Function spi_reset()

Resets the SPI module.

```
void spi_reset(
        struct spi_module *const module)
```

This function will reset the SPI module to its power on default values and disable it.

Table 7-14 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

7.4.3. Lock/Unlock

7.4.3.1. Function spi_lock()

Attempt to get lock on driver instance.

```
enum status_code spi_lock(
        struct spi_module *const module)
```

This function checks the instance's lock, which indicates whether or not it is currently in use, and sets the lock if it was not already set.

The purpose of this is to enable exclusive access to driver instances, so that, e.g., transactions by different services will not interfere with each other.

Table 7-15 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Table 7-16 Return Values

Return value	Description
STATUS_OK	If the module was locked
STATUS_BUSY	If the module was already locked

7.4.3.2. Function spi_unlock()

Unlock driver instance.

```
void spi_unlock(
        struct spi_module *const module)
```

This function clears the instance lock, indicating that it is available for use.

Table 7-17 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the driver instance to lock

Table 7-18 Return Values

Return value	Description
STATUS_OK	If the module was locked
STATUS_BUSY	If the module was already locked

7.4.4. Ready to Write/Read

7.4.4.1. Function spi_is_write_complete()

Checks if the SPI in master mode has shifted out last data, or if the master has ended the transfer in slave mode.

```
bool spi_is_write_complete(
        struct spi_module *const module)
```

This function will check if the SPI master module has shifted out last data, or if the slave select pin has been drawn high by the master for the SPI slave module.

Table 7-19 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct

Returns

Indication of whether any writes are ongoing.

Table 7-20 Return Values

Return value	Description
true	If the SPI master module has shifted out data, or slave select has been drawn high for SPI slave
false	If the SPI master module has not shifted out data

7.4.4.2. Function spi_is_ready_to_write()

Checks if the SPI module is ready to write data.

```
bool spi_is_ready_to_write(
        struct spi_module *const module)
```

This function will check if the SPI module is ready to write data.

Table 7-21 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct

Returns

Indication of whether the module is ready to read data or not.

Table 7-22 Return Values

Return value	Description
true	If the SPI module is ready to write data
false	If the SPI module is not ready to write data

7.4.4.3. Function spi_is_ready_to_read()

Checks if the SPI module is ready to read data.

```
bool spi_is_ready_to_read(
        struct spi_module *const module)
```

This function will check if the SPI module is ready to read data.

Table 7-23 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct

Returns
Indication of whether the module is ready to read data or not.

Table 7-24 Return Values

Return value	Description
true	If the SPI module is ready to read data
false	If the SPI module is not ready to read data

7.4.5. Read/Write

7.4.5.1. Function spi_write()

Transfers a single SPI character.

```
enum status_code spi_write(
        struct spi_module * module,
        uint16_t tx_data)
```

This function will send a single SPI character via SPI and ignore any data shifted in by the connected device. To both send and receive data, use the spi_transceive_wait function or use the spi_read function after writing a character. The spi_is_ready_to_write function should be called before calling this function.

Note that this function does not handle the SS (Slave Select) pin(s) in master mode; this must be handled from the user application.

Note: In slave mode, the data will not be transferred before a master initiates a transaction.

Table 7-25 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	tx_data	Data to transmit

Returns
Status of the procedure.

Table 7-26 Return Values

Return value	Description
STATUS_OK	If the data was written
STATUS_BUSY	If the last write was not completed

7.4.5.2. Function spi_write_buffer_wait()

Sends a buffer of length SPI characters.

```
enum status_code spi_write_buffer_wait(
        struct spi_module *const module,
        const uint8_t * tx_data,
        uint16_t length)
```

This function will send a buffer of SPI characters via the SPI and discard any data that is received. To both send and receive a buffer of data, use the spi_transceive_buffer_wait function.

Note that this function does not handle the _SS (slave select) pin(s) in master mode; this must be handled by the user application.

Table 7-27 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	tx_data	Pointer to the buffer to transmit
[in]	length	Number of SPI characters to transfer

Returns
Status of the write operation.

Table 7-28 Return Values

Return value	Description
STATUS_OK	If the write was completed
STATUS_ABORTED	If transaction was ended by master before entire buffer was transferred
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_ERR_TIMEOUT	If the operation was not completed within the timeout in slave mode

7.4.5.3. Function spi_read()

Reads last received SPI character.

```
enum status_code spi_read(
        struct spi_module *const module,
        uint16_t * rx_data)
```

This function will return the last SPI character shifted into the receive register by the spi_write function.

Note: The spi_is_ready_to_read function should be called before calling this function.

Note: Receiver must be enabled in the configuration.

Table 7-29 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[out]	rx_data	Pointer to store the received data

Returns
Status of the read operation.

Table 7-30 Return Values

Return value	Description
STATUS_OK	If data was read
STATUS_ERR_IO	If no data is available
STATUS_ERR_OVERFLOW	If the data is overflown

7.4.5.4. Function spi_read_buffer_wait()

Reads buffer of length SPI characters.

```
enum status_code spi_read_buffer_wait(
        struct spi_module *const module,
        uint8_t * rx_data,
        uint16_t length,
        uint16_t dummy)
```

This function will read a buffer of data from an SPI peripheral by sending dummy SPI character if in master mode, or by waiting for data in slave mode.

Note: If address matching is enabled for the slave, the first character received and placed in the buffer will be the address.

Table 7-31 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[out]	rx_data	Data buffer for received data
[in]	length	Length of data to receive
[in]	dummy	8- or 9-bit dummy byte to shift out in master mode

Returns
Status of the read operation.

Table 7-32 Return Values

Return value	Description
STATUS_OK	If the read was completed
STATUS_ABORTED	If transaction was ended by master before the entire buffer was transferred
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_ERR_TIMEOUT	If the operation was not completed within the timeout in slave mode
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_OVERFLOW	If the data is overflown

7.4.5.5. Function spi_transceive_wait()

Sends and reads a single SPI character.

```
enum status_code spi_transceive_wait(
        struct spi_module *const module,
        uint16_t tx_data,
        uint16_t * rx_data)
```

This function will transfer a single SPI character via SPI and return the SPI character that is shifted into the shift register.

In master mode the SPI character will be sent immediately and the received SPI character will be read as soon as the shifting of the data is complete.

In slave mode this function will place the data to be sent into the transmit buffer. It will then block until an SPI master has shifted a complete SPI character, and the received data is available.

Note: The data to be sent might not be sent before the next transfer, as loading of the shift register is dependent on SCK.

Note: If address matching is enabled for the slave, the first character received and placed in the buffer will be the address.

Table 7-33 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	tx_data	SPI character to transmit
[out]	rx_data	Pointer to store the received SPI character

Returns
Status of the operation.

Table 7-34 Return Values

Return value	Description
STATUS_OK	If the operation was completed
STATUS_ERR_TIMEOUT	If the operation was not completed within the timeout in slave mode
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_OVERFLOW	If the incoming data is overflown

7.4.5.6. Function spi_transceive_buffer_wait()

Sends and receives a buffer of length SPI characters.

```
enum status_code spi_transceive_buffer_wait(
        struct spi_module *const module,
        uint8_t * tx_data,
        uint8_t * rx_data,
        uint16_t length)
```

This function will send and receive a buffer of data via the SPI.

In master mode the SPI characters will be sent immediately and the received SPI character will be read as soon as the shifting of the SPI character is complete.

In slave mode this function will place the data to be sent into the transmit buffer. It will then block until an SPI master has shifted the complete buffer and the received data is available.

Table 7-35 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	tx_data	Pointer to the buffer to transmit
[out]	rx_data	Pointer to the buffer where received data will be stored
[in]	length	Number of SPI characters to transfer

Returns
Status of the operation.

Table 7-36 Return Values

Return value	Description
STATUS_OK	If the operation was completed
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_ERR_TIMEOUT	If the operation was not completed within the timeout in slave mode
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_OVERFLOW	If the data is overflown

7.4.5.7. Function spi_select_slave()

Selects slave device.

```
enum status_code spi_select_slave(
        struct spi_module *const module,
        struct spi_slave_inst *const slave,
        bool select)
```

This function will drive the slave select pin of the selected device low or high depending on the select Boolean. If slave address recognition is enabled, the address will be sent to the slave when selecting it.

Table 7-37 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software module struct
[in]	slave	Pointer to the attached slave
[in]	select	Boolean stating if the slave should be selected or deselected

Returns
Status of the operation.

Table 7-38 Return Values

Return value	Description
STATUS_OK	If the slave device was selected
STATUS_ERR_UNSUPPORTED_DEV	If the SPI module is operating in slave mode
STATUS_BUSY	If the SPI module is not ready to write the slave address

7.4.6. Callback Management

7.4.6.1. Function spi_register_callback()

Registers a SPI callback function.

```
void spi_register_callback(
        struct spi_module *const module,
        spi_callback_t callback_func,
        enum spi_callback callback_type)
```

Registers a callback function which is implemented by the user.

Note: The callback must be enabled by spi_enable_callback, in order for the interrupt handler to call it when the conditions for the callback type are met.

Table 7-39 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	callback_func	Pointer to callback function
[in]	callback_type	Callback type given by an enum

7.4.6.2. Function spi_unregister_callback()

Unregisters a SPI callback function.

```
void spi_unregister_callback(
        struct spi_module * module,
        enum spi_callback callback_type)
```

Unregisters a callback function which is implemented by the user.

Table 7-40 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[in]	callback_type	Callback type given by an enum

7.4.6.3. Function spi_enable_callback()

Enables an SPI callback of a given type.

```
void spi_enable_callback(
        struct spi_module *const module,
        enum spi_callback callback_type)
```

Enables the callback function registered by the spi_register_callback. The callback function will be called from the interrupt handler when the conditions for the callback type are met.

Table 7-41 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[in]	callback_type	Callback type given by an enum

7.4.6.4. Function spi_disable_callback()

Disables callback.

```
void spi_disable_callback(
        struct spi_module *const module,
        enum spi_callback callback_type)
```

Disables the callback function registered by the spi_register_callback, and the callback will not be called from the interrupt routine.

Table 7-42 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[in]	callback_type	Callback type given by an enum

7.4.7. Writing and Reading

7.4.7.1. Function spi_write_buffer_job()

Asynchronous buffer write.

```
enum status_code spi_write_buffer_job(
        struct spi_module *const module,
        uint8_t * tx_data,
        uint16_t length)
```

Sets up the driver to write to the SPI from a given buffer. If registered and enabled, a callback function will be called when the write is finished.

Table 7-43 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[out]	tx_data	Pointer to data buffer to receive
[in]	length	Data buffer length

Returns
Status of the write request operation.

Table 7-44 Return Values

Return value	Description
STATUS_OK	If the operation completed successfully
STATUS_ERR_BUSY	If the SPI was already busy with a write operation
STATUS_ERR_INVALID_ARG	If requested write length was zero

7.4.7.2. Function spi_read_buffer_job()

Asynchronous buffer read.

```
enum status_code spi_read_buffer_job(
        struct spi_module *const module,
        uint8_t * rx_data,
        uint16_t length,
        uint16_t dummy)
```

Sets up the driver to read from the SPI to a given buffer. If registered and enabled, a callback function will be called when the read is finished.

Note: If address matching is enabled for the slave, the first character received and placed in the RX buffer will be the address.

Table 7-45 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[out]	rx_data	Pointer to data buffer to receive
[in]	length	Data buffer length
[in]	dummy	Dummy character to send when reading in master mode

Returns
Status of the operation.

Table 7-46 Return Values

Return value	Description
STATUS_OK	If the operation completed successfully
STATUS_ERR_BUSY	If the SPI was already busy with a read operation
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_INVALID_ARG	If requested read length was zero

7.4.7.3. Function spi_transceive_buffer_job()

Asynchronous buffer write and read.

```
enum status_code spi_transceive_buffer_job(
        struct spi_module *const module,
        uint8_t * tx_data,
        uint8_t * rx_data,
        uint16_t length)
```

Sets up the driver to write and read to and from given buffers. If registered and enabled, a callback function will be called when the transfer is finished.

Note: If address matching is enabled for the slave, the first character received and placed in the RX buffer will be the address.

Table 7-47 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct
[in]	tx_data	Pointer to data buffer to send
[out]	rx_data	Pointer to data buffer to receive
[in]	length	Data buffer length

Returns
Status of the operation.

Table 7-48 Return Values

Return value	Description
STATUS_OK	If the operation completed successfully
STATUS_ERR_BUSY	If the SPI was already busy with a read operation
STATUS_ERR_DENIED	If the receiver is not enabled
STATUS_ERR_INVALID_ARG	If requested read length was zero

7.4.7.4. Function spi_abort_job()

Aborts an ongoing job.

```
void spi_abort_job(
        struct spi_module *const module)
```

This function will abort the specified job type.

Table 7-49 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct

7.4.7.5. Function spi_get_job_status()

Retrieves the current status of a job.

```
enum status_code spi_get_job_status(
        const struct spi_module *const module)
```

Retrieves the current status of a job that was previously issued.

Table 7-50 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct

Returns
Current job status.

7.4.7.6. Function spi_get_job_status_wait()

Retrieves the status of job once it ends.

```
enum status_code spi_get_job_status_wait(
        const struct spi_module *const module)
```

Waits for current job status to become non-busy, then returns its value.

Table 7-51 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to SPI software instance struct

Returns
Current non-busy job status.

7.4.8. Function spi_is_syncing()

Determines if the SPI module is currently synchronizing to the bus.

```
bool spi_is_syncing(
        struct spi_module *const module)
```

This function will check if the underlying hardware peripheral module is currently synchronizing across multiple clock domains to the hardware bus. This function can be used to delay further operations on the module until it is ready.

Table 7-52 Parameters

Data direction	Parameter name	Description
[in]	module	SPI hardware module

Returns
Synchronization status of the underlying hardware module.

Table 7-53 Return Values

Return value	Description
true	Module synchronization is ongoing
false	Module synchronization is not ongoing

7.4.9. Function spi_set_baudrate()

Set the baudrate of the SPI module.

```
enum status_code spi_set_baudrate(
        struct spi_module *const module,
        uint32_t baudrate)
```

This function will set the baudrate of the SPI module.

Table 7-54 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	baudrate	The baudrate wanted

Returns
The status of the configuration.

Table 7-55 Return Values

Return value	Description
STATUS_ERR_INVALID_ARG	If invalid argument(s) were provided
STATUS_OK	If the configuration was written

7.5. Enumeration Definitions

7.5.1. Enum spi_addr_mode

For slave mode when using the SPI frame with address format.

Table 7-56 Members

Enum value	Description
SPI_ADDR_MODE_MASK	`address_mask` in the spi_config struct is used as a mask to the register
SPI_ADDR_MODE_UNIQUE	The slave responds to the two unique addresses in `address` and `address_mask` in the spi_config struct
SPI_ADDR_MODE_RANGE	The slave responds to the range of addresses between and including `address` and `address_mask` in the spi_config struct

7.5.2. Enum spi_callback

Callbacks for SPI callback driver.

Note: For slave mode, these callbacks will be called when a transaction is ended by the master pulling Slave Select high.

Table 7-57 Members

Enum value	Description
SPI_CALLBACK_BUFFER_TRANSMITTED	Callback for buffer transmitted
SPI_CALLBACK_BUFFER_RECEIVED	Callback for buffer received
SPI_CALLBACK_BUFFER_TRANSCEIVED	Callback for buffers transceived
SPI_CALLBACK_ERROR	Callback for error
SPI_CALLBACK_SLAVE_TRANSMISSION_COMPLETE	Callback for transmission ended by master before the entire buffer was read or written from slave
SPI_CALLBACK_SLAVE_SELECT_LOW	Callback for slave select low
SPI_CALLBACK_COMBINED_ERROR	Callback for combined error happen

7.5.3. Enum spi_character_size

SPI character size.

Table 7-58 Members

Enum value	Description
SPI_CHARACTER_SIZE_8BIT	8-bit character
SPI_CHARACTER_SIZE_9BIT	9-bit character

7.5.4. Enum spi_data_order

SPI data order.

Table 7-59 Members

Enum value	Description
SPI_DATA_ORDER_LSB	The LSB of the data is transmitted first
SPI_DATA_ORDER_MSB	The MSB of the data is transmitted first

7.5.5. Enum spi_frame_format

Frame format for slave mode.

Table 7-60 Members

Enum value	Description
SPI_FRAME_FORMAT_SPI_FRAME	SPI frame
SPI_FRAME_FORMAT_SPI_FRAME_ADDR	SPI frame with address

7.5.6. Enum spi_interrupt_flag

Interrupt flags for the SPI module.

Table 7-61 Members

Enum value	Description
SPI_INTERRUPT_FLAG_DATA_REGISTER_EMPTY	This flag is set when the contents of the data register has been moved to the shift register and the data register is ready for new data
SPI_INTERRUPT_FLAG_TX_COMPLETE	This flag is set when the contents of the shift register has been shifted out
SPI_INTERRUPT_FLAG_RX_COMPLETE	This flag is set when data has been shifted into the data register
SPI_INTERRUPT_FLAG_SLAVE_SELECT_LOW	This flag is set when slave select low
SPI_INTERRUPT_FLAG_COMBINED_ERROR	This flag is set when combined error happen

7.5.7. Enum spi_mode

SPI mode selection.

Table 7-62 Members

Enum value	Description
SPI_MODE_MASTER	Master mode
SPI_MODE_SLAVE	Slave mode

7.5.8. Enum spi_signal_mux_setting

Set the functionality of the SERCOM pins. As not all combinations can be used in different modes of operation, proper combinations must be chosen according to the rest of the configuration.

Note: In master operation: DI is MISO, DO is MOSI. In slave operation: DI is MOSI, DO is MISO.

See MUX Settings for a description of the various MUX setting options.

Table 7-63 Members

Enum value	Description
SPI_SIGNAL_MUX_SETTING_A	SPI MUX combination A. DOPO: 0x0, DIPO: 0x0
SPI_SIGNAL_MUX_SETTING_B	SPI MUX combination B. DOPO: 0x0, DIPO: 0x1
SPI_SIGNAL_MUX_SETTING_C	SPI MUX combination C. DOPO: 0x0, DIPO: 0x2
SPI_SIGNAL_MUX_SETTING_D	SPI MUX combination D. DOPO: 0x0, DIPO: 0x3
SPI_SIGNAL_MUX_SETTING_E	SPI MUX combination E. DOPO: 0x1, DIPO: 0x0
SPI_SIGNAL_MUX_SETTING_F	SPI MUX combination F. DOPO: 0x1, DIPO: 0x1
SPI_SIGNAL_MUX_SETTING_G	SPI MUX combination G. DOPO: 0x1, DIPO: 0x2
SPI_SIGNAL_MUX_SETTING_H	SPI MUX combination H. DOPO: 0x1, DIPO: 0x3
SPI_SIGNAL_MUX_SETTING_I	SPI MUX combination I. DOPO: 0x2, DIPO: 0x0
SPI_SIGNAL_MUX_SETTING_J	SPI MUX combination J. DOPO: 0x2, DIPO: 0x1
SPI_SIGNAL_MUX_SETTING_K	SPI MUX combination K. DOPO: 0x2, DIPO: 0x2
SPI_SIGNAL_MUX_SETTING_L	SPI MUX combination L. DOPO: 0x2, DIPO: 0x3
SPI_SIGNAL_MUX_SETTING_M	SPI MUX combination M. DOPO: 0x3, DIPO: 0x0
SPI_SIGNAL_MUX_SETTING_N	SPI MUX combination N. DOPO: 0x3, DIPO: 0x1
SPI_SIGNAL_MUX_SETTING_O	SPI MUX combination O. DOPO: 0x3, DIPO: 0x2
SPI_SIGNAL_MUX_SETTING_P	SPI MUX combination P. DOPO: 0x3, DIPO: 0x3

7.5.9. Enum spi_transfer_mode

SPI transfer mode.

Table 7-64 Members

Enum value	Description
SPI_TRANSFER_MODE_0	Mode 0. Leading edge: rising, sample. Trailing edge: falling, setup
SPI_TRANSFER_MODE_1	Mode 1. Leading edge: rising, setup. Trailing edge: falling, sample
SPI_TRANSFER_MODE_2	Mode 2. Leading edge: falling, sample. Trailing edge: rising, setup
SPI_TRANSFER_MODE_3	Mode 3. Leading edge: falling, setup. Trailing edge: rising, sample

8. Extra Information for SERCOM SPI Driver

8.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Description
SERCOM	Serial Communication Interface
SPI	Serial Peripheral Interface
SCK	Serial Clock
MOSI	Master Output Slave Input
MISO	Master Input Slave Output
SS	Slave Select
DIO	Data Input Output
DO	Data Output
DI	Data Input
DMA	Direct Memory Access

8.2. Dependencies

The SPI driver has the following dependencies:

- System Pin Multiplexer Driver

8.3. Workarounds Implemented by Driver

No workarounds in driver.

8.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Added new features as below: • Slave select low detect • Hardware slave select • DMA support
Edited slave part of write and transceive buffer functions to ensure that second character is sent at the right time
Renamed the anonymous union in `struct` spi_config to `mode_specific`
Initial Release

9. Examples for SERCOM SPI Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM Serial Peripheral Interface (SERCOM SPI) Driver. QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- Quick Start Guide for SERCOM SPI Master - Polled
- Quick Start Guide for SERCOM SPI Slave - Polled
- Quick Start Guide for SERCOM SPI Master - Callback
- Quick Start Guide for SERCOM SPI Slave - Callback
- Quick Start Guide for Using DMA with SERCOM SPI

9.1. Quick Start Guide for SERCOM SPI Master - Polled

In this use case, the SPI on extension header 1 of the Xplained Pro board will be configured with the following settings:

- Master Mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E (see Master Mode Settings)
- 8-bit character size
- Not enabled in sleep mode
- Baudrate 100000
- GLCK generator 0

9.1.1. Setup

9.1.1.1. Prerequisites

There are no special setup requirements for this use-case.

9.1.1.2. Code

The following must be added to the user application:

A sample buffer to send via SPI.

```
static uint8_t buffer[BUF_LENGTH] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
        0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13
};
```

Number of entries in the sample buffer.

```
#define BUF_LENGTH 20
```

GPIO pin to use as Slave Select.

```
#define SLAVE_SELECT_PIN EXT1_PIN_SPI_SS_0
```

A globally available software device instance struct to store the SPI driver state while it is in use.

```
struct spi_module spi_master_instance;
```

A globally available peripheral slave software device instance struct.

```
struct spi_slave_inst slave;
```

A function for configuring the SPI.

```
void configure_spi_master(void)
{
    struct spi_config config_spi_master;
    struct spi_slave_inst_config slave_dev_config;
    /* Configure and initialize software device instance of peripheral
slave */
    spi_slave_inst_get_config_defaults(&slave_dev_config);
    slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
    spi_attach_slave(&slave, &slave_dev_config);
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_master);
    config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
    /* Configure pad 0 for data in */
    config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
    /* Configure pad 2 for data out */
    config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
    spi_init(&spi_master_instance, EXT1_SPI_MODULE, &config_spi_master);

    spi_enable(&spi_master_instance);

}
```

Add to user application `main()`.

```
system_init();
configure_spi_master();
```

9.1.2. Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Set-up the SPI.

   ```
   configure_spi_master();
   ```

   1. Create configuration struct.

      ```
      struct spi_config config_spi_master;
      ```

   2. Create peripheral slave configuration struct.

      ```
      struct spi_slave_inst_config slave_dev_config;
      ```

   3. Create peripheral slave software device instance struct.

      ```
      struct spi_slave_inst slave;
      ```

   4. Get default peripheral slave configuration.

      ```
      spi_slave_inst_get_config_defaults(&slave_dev_config);
      ```

5. Set Slave Select pin.

```
slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
```

6. Initialize peripheral slave software instance with configuration.

```
spi_attach_slave(&slave, &slave_dev_config);
```

7. Get default configuration to edit.

```
spi_get_config_defaults(&config_spi_master);
```

8. Set MUX setting E.

```
config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
```

9. Set pinmux for pad 0 (data in (MISO)).

```
config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
```

10. Set pinmux for pad 1 as unused, so the pin can be used for other purposes.

```
config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
```

11. Set pinmux for pad 2 (data out (MOSI)).

```
config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

12. Set pinmux for pad 3 (SCK).

```
config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

13. Initialize SPI module with configuration.

```
spi_init(&spi_master_instance, EXT1_SPI_MODULE,
&config_spi_master);
```

14. Enable SPI module.

```
spi_enable(&spi_master_instance);
```

9.1.3. Use Case

9.1.3.1. Code

Add the following to your user application `main()`.

```
while (true) {
    /* Infinite loop */
    if(!port_pin_get_input_level(BUTTON_0_PIN)) {
        spi_select_slave(&spi_master_instance, &slave, true);
        spi_write_buffer_wait(&spi_master_instance, buffer, BUF_LENGTH);
        spi_select_slave(&spi_master_instance, &slave, false);
        port_pin_set_output_level(LED_0_PIN, LED0_ACTIVE);
    }
}
```

9.1.3.2. Workflow

1. Select slave.

```
spi_select_slave(&spi_master_instance, &slave, true);
```

2. Write buffer to SPI slave.

```
spi_write_buffer_wait(&spi_master_instance, buffer, BUF_LENGTH);
```

3. Deselect slave.

```
spi_select_slave(&spi_master_instance, &slave, false);
```

4. Light up.

```
port_pin_set_output_level(LED_0_PIN, LED0_ACTIVE);
```

5. Infinite loop.

```
while (true) {
    /* Infinite loop */
    if(!port_pin_get_input_level(BUTTON_0_PIN)) {
        spi_select_slave(&spi_master_instance, &slave, true);
        spi_write_buffer_wait(&spi_master_instance, buffer,
BUF_LENGTH);
        spi_select_slave(&spi_master_instance, &slave, false);
        port_pin_set_output_level(LED_0_PIN, LED0_ACTIVE);
    }
}
```

9.2. Quick Start Guide for SERCOM SPI Slave - Polled

In this use case, the SPI on extension header 1 of the Xplained Pro board will configured with the following settings:

- Slave mode enabled
- Preloading of shift register enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E (see Slave Mode Settings)
- 8-bit character size
- Not enabled in sleep mode
- GLCK generator 0

9.2.1. Setup

9.2.1.1. Prerequisites

The device must be connected to an SPI master which must read from the device.

9.2.1.2. Code

The following must be added to the user application source file, outside any functions:

A sample buffer to send via SPI.

```
static uint8_t buffer_expect[BUF_LENGTH] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
         0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13
};
static uint8_t buffer_rx[BUF_LENGTH] = {0x00};
```

Number of entries in the sample buffer.

```
#define BUF_LENGTH 20
```

A globally available software device instance struct to store the SPI driver state while it is in use.

```
struct spi_module spi_slave_instance;
```

A function for configuring the SPI.

```
void configure_spi_slave(void)
{
    struct spi_config config_spi_slave;
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_slave);
    config_spi_slave.mode = SPI_MODE_SLAVE;
    config_spi_slave.mode_specific.slave.preload_enable = true;
    config_spi_slave.mode_specific.slave.frame_format =
SPI_FRAME_FORMAT_SPI_FRAME;
    config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
    /* Configure pad 0 for data in */
    config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
    /* Configure pad 2 for data out */
    config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
    spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);

    spi_enable(&spi_slave_instance);

}
```

Add to user application `main()`.

```
uint8_t result = 0;

/* Initialize system */
system_init();

configure_spi_slave();
```

9.2.1.3. Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Set-up the SPI.

   ```
   configure_spi_slave();
   ```

   1. Create configuration struct.

      ```
      struct spi_config config_spi_slave;
      ```

   2. Get default configuration to edit.

      ```
      spi_get_config_defaults(&config_spi_slave);
      ```

   3. Set the SPI in slave mode.

      ```
      config_spi_slave.mode = SPI_MODE_SLAVE;
      ```

   4. Enable preloading of shift register.

      ```
      config_spi_slave.mode_specific.slave.preload_enable = true;
      ```

5. Set frame format to SPI frame.

```
config_spi_slave.mode_specific.slave.frame_format =
SPI_FRAME_FORMAT_SPI_FRAME;
```

6. Set MUX setting E.

```
config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
```

7. Set pinmux for pad 0 (data in MOSI).

```
config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
```

8. Set pinmux for pad 1 (slave select).

```
config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
```

9. Set pinmux for pad 2 (data out MISO).

```
config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

10. Set pinmux for pad 3 (SCK).

```
config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

11. Initialize SPI module with configuration.

```
spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);
```

12. Enable SPI module.

```
spi_enable(&spi_slave_instance);
```

9.2.2. Use Case

9.2.2.1. Code

Add the following to your user application `main()`.

```
while(spi_read_buffer_wait(&spi_slave_instance, buffer_rx, BUF_LENGTH,
    0x00) != STATUS_OK) {
    /* Wait for transfer from the master */
}
for (uint8_t i = 0; i < BUF_LENGTH; i++) {
    if(buffer_rx[i] != buffer_expect[i]) {
        result++;
    }
}
while (true) {
    /* Infinite loop */
    if (result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 30000;
        while(delay--) {
        }
    } else {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 600000;
        while(delay--) {
        }
    }
}
```

9.2.2.2. Workflow

1. Read data from SPI master.

```
while(spi_read_buffer_wait(&spi_slave_instance, buffer_rx, BUF_LENGTH,
    0x00) != STATUS_OK) {
    /* Wait for transfer from the master */
}
```

2. Compare the received data with the transmitted data from SPI master.

```
for (uint8_t i = 0; i < BUF_LENGTH; i++) {
    if(buffer_rx[i] != buffer_expect[i]) {
        result++;
    }
}
```

3. Infinite loop. If the data is matched, LED0 will flash slowly. Otherwise, LED will flash quickly.

```
while (true) {
    /* Infinite loop */
    if (result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 30000;
        while(delay--) {
        }
    } else {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 600000;
        while(delay--) {
        }
    }
}
```

9.3. Quick Start Guide for SERCOM SPI Master - Callback

In this use case, the SPI on extension header 1 of the Xplained Pro board will be configured with the following settings:

- Master Mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E (see Master Mode Settings)
- 8-bit character size
- Not enabled in sleep mode
- Baudrate 100000
- GLCK generator 0

9.3.1. Setup

9.3.1.1. Prerequisites

There are no special setup requirements for this use-case.

9.3.1.2. Code

The following must be added to the user application.

A sample buffer to send via SPI.

```
static uint8_t wr_buffer[BUF_LENGTH] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
         0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13
};
static uint8_t rd_buffer[BUF_LENGTH];
```

Number of entries in the sample buffer.

```
#define BUF_LENGTH 20
```

GPIO pin to use as Slave Select.

```
#define SLAVE_SELECT_PIN EXT1_PIN_SPI_SS_0
```

A globally available software device instance struct to store the SPI driver state while it is in use.

```
struct spi_module spi_master_instance;
```

A globally available peripheral slave software device instance struct.

```
struct spi_slave_inst slave;
```

A function for configuring the SPI.

```
void configure_spi_master(void)
{
    struct spi_config config_spi_master;
    struct spi_slave_inst_config slave_dev_config;
    /* Configure and initialize software device instance of peripheral
slave */
    spi_slave_inst_get_config_defaults(&slave_dev_config);
    slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
    spi_attach_slave(&slave, &slave_dev_config);
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_master);
    config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
    /* Configure pad 0 for data in */
    config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
    /* Configure pad 2 for data out */
    config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
    spi_init(&spi_master_instance, EXT1_SPI_MODULE, &config_spi_master);

    spi_enable(&spi_master_instance);

}
```

A function for configuring the callback functionality of the SPI.

```
void configure_spi_master_callbacks(void)
{
    spi_register_callback(&spi_master_instance, callback_spi_master,
            SPI_CALLBACK_BUFFER_TRANSCEIVED);
    spi_enable_callback(&spi_master_instance,
SPI_CALLBACK_BUFFER_TRANSCEIVED);
}
```

A global variable that can flag to the application that the buffer has been transferred.

```
volatile bool transrev_complete_spi_master = false;
```

Callback function.

```
static void callback_spi_master( struct spi_module *const module)
{
    transrev_complete_spi_master = true;
}
```

Add to user application `main()`.

```
/* Initialize system */
system_init();

configure_spi_master();
configure_spi_master_callbacks();
```

9.3.2. Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Set-up the SPI.

   ```
   configure_spi_master();
   ```

   1. Create configuration struct.

      ```
      struct spi_config config_spi_master;
      ```

   2. Create peripheral slave configuration struct.

      ```
      struct spi_slave_inst_config slave_dev_config;
      ```

   3. Get default peripheral slave configuration.

      ```
      spi_slave_inst_get_config_defaults(&slave_dev_config);
      ```

   4. Set Slave Select pin.

      ```
      slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
      ```

   5. Initialize peripheral slave software instance with configuration.

      ```
      spi_attach_slave(&slave, &slave_dev_config);
      ```

   6. Get default configuration to edit.

      ```
      spi_get_config_defaults(&config_spi_master);
      ```

   7. Set MUX setting E.

      ```
      config_spi_master.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
      ```

   8. Set pinmux for pad 0 (data in MISO).

      ```
      config_spi_master.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
      ```

   9. Set pinmux for pad 1 as unused, so the pin can be used for other purposes.

      ```
      config_spi_master.pinmux_pad1 = PINMUX_UNUSED;
      ```

10. Set pinmux for pad 2 (data out MOSI).

```
config_spi_master.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
```

11. Set pinmux for pad 3 (SCK).

```
config_spi_master.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
```

12. Initialize SPI module with configuration.

```
spi_init(&spi_master_instance, EXT1_SPI_MODULE,
&config_spi_master);
```

13. Enable SPI module.

```
spi_enable(&spi_master_instance);
```

3. Setup the callback functionality.

```
configure_spi_master_callbacks();
```

1. Register callback function for buffer transmitted.

```
spi_register_callback(&spi_master_instance, callback_spi_master,
        SPI_CALLBACK_BUFFER_TRANSCEIVED);
```

2. Enable callback for buffer transmitted.

```
spi_enable_callback(&spi_master_instance,
SPI_CALLBACK_BUFFER_TRANSCEIVED);
```

9.3.3. Use Case

9.3.3.1. Code

Add the following to your user application `main()`.

```
while (true) {
    /* Infinite loop */
    if (!port_pin_get_input_level(BUTTON_0_PIN)) {
        spi_select_slave(&spi_master_instance, &slave, true);
        spi_transceive_buffer_job(&spi_master_instance,
wr_buffer,rd_buffer,BUF_LENGTH);
        while (!transrev_complete_spi_master) {
        }
        transrev_complete_spi_master = false;
        spi_select_slave(&spi_master_instance, &slave, false);
    }
}
```

9.3.3.2. Workflow

1. Select slave.

```
spi_select_slave(&spi_master_instance, &slave, true);
```

2. Write buffer to SPI slave.

```
spi_transceive_buffer_job(&spi_master_instance,
wr_buffer,rd_buffer,BUF_LENGTH);
```

3. Wait for the transfer to be complete.

```
while (!transrev_complete_spi_master) {
}
transrev_complete_spi_master = false;
```

4. Deselect slave.

```
spi_select_slave(&spi_master_instance, &slave, false);
```

5. Infinite loop.

```
while (true) {
    /* Infinite loop */
    if (!port_pin_get_input_level(BUTTON_0_PIN)) {
        spi_select_slave(&spi_master_instance, &slave, true);
        spi_transceive_buffer_job(&spi_master_instance,
wr_buffer,rd_buffer,BUF_LENGTH);
        while (!transrev_complete_spi_master) {
        }
        transrev_complete_spi_master = false;
        spi_select_slave(&spi_master_instance, &slave, false);
    }
}
```

9.3.4. Callback

When the buffer is successfully transmitted to the slave, the callback function will be called.

9.3.4.1. Workflow

1. Let the application know that the buffer is transmitted by setting the global variable to true.

```
transrev_complete_spi_master = true;
```

9.4. Quick Start Guide for SERCOM SPI Slave - Callback

In this use case, the SPI on extension header 1 of the Xplained Pro board will configured with the following settings:

- Slave mode enabled
- Preloading of shift register enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E (see Slave Mode Settings)
- 8-bit character size
- Not enabled in sleep mode
- GLCK generator 0

9.4.1. Setup

9.4.1.1. Prerequisites

The device must be connected to a SPI master, which must read from the device.

9.4.1.2. Code

The following must be added to the user application source file, outside any functions.

A sample buffer to send via SPI.

```
static uint8_t buffer_rx[BUF_LENGTH] = {0x00,};
static uint8_t buffer_expect[BUF_LENGTH] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
         0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13
};
```

Number of entries in the sample buffer.

```
#define BUF_LENGTH 20
```

A globally available software device instance struct to store the SPI driver state while it is in use.

```
struct spi_module spi_slave_instance;
```

A function for configuring the SPI.

```
void configure_spi_slave(void)
{
    struct spi_config config_spi_slave;
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_slave);
    config_spi_slave.mode = SPI_MODE_SLAVE;
    config_spi_slave.mode_specific.slave.preload_enable = true;
    config_spi_slave.mode_specific.slave.frame_format =
SPI_FRAME_FORMAT_SPI_FRAME;
    config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
    /* Configure pad 0 for data in */
    config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
    /* Configure pad 2 for data out */
    config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
    spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);

    spi_enable(&spi_slave_instance);

}
```

A function for configuring the callback functionality of the SPI.

```
void configure_spi_slave_callbacks(void)
{
    spi_register_callback(&spi_slave_instance, spi_slave_callback,
            SPI_CALLBACK_BUFFER_RECEIVED);
    spi_enable_callback(&spi_slave_instance, SPI_CALLBACK_BUFFER_RECEIVED);
}
```

A global variable that can flag to the application that the buffer has been transferred.

```
volatile bool transfer_complete_spi_slave = false;
```

Callback function.

```
static void spi_slave_callback(struct spi_module *const module)
{
    transfer_complete_spi_slave = true;
}
```

Add to user application `main()`.

```
uint8_t result = 0;

/* Initialize system */
system_init();

configure_spi_slave();
configure_spi_slave_callbacks();
```

9.4.1.3. Workflow

1. Initialize system.

   ```
   system_init();
   ```

2. Set-up the SPI.

   ```
   configure_spi_slave();
   ```

   1. Create configuration struct.

      ```
      struct spi_config config_spi_slave;
      ```

   2. Get default configuration to edit.

      ```
      spi_get_config_defaults(&config_spi_slave);
      ```

   3. Set the SPI in slave mode.

      ```
      config_spi_slave.mode = SPI_MODE_SLAVE;
      ```

   4. Enable preloading of shift register.

      ```
      config_spi_slave.mode_specific.slave.preload_enable = true;
      ```

   5. Set frame format to SPI frame.

      ```
      config_spi_slave.mode_specific.slave.frame_format =
      SPI_FRAME_FORMAT_SPI_FRAME;
      ```

   6. Set MUX setting E.

      ```
      config_spi_slave.mux_setting = EXT1_SPI_SERCOM_MUX_SETTING;
      ```

   7. Set pinmux for pad 0 (data in MOSI).

      ```
      config_spi_slave.pinmux_pad0 = EXT1_SPI_SERCOM_PINMUX_PAD0;
      ```

   8. Set pinmux for pad 1 (slave select).

      ```
      config_spi_slave.pinmux_pad1 = EXT1_SPI_SERCOM_PINMUX_PAD1;
      ```

   9. Set pinmux for pad 2 (data out MISO).

      ```
      config_spi_slave.pinmux_pad2 = EXT1_SPI_SERCOM_PINMUX_PAD2;
      ```

   10. Set pinmux for pad 3 (SCK).

       ```
       config_spi_slave.pinmux_pad3 = EXT1_SPI_SERCOM_PINMUX_PAD3;
       ```

   11. Initialize SPI module with configuration.

       ```
       spi_init(&spi_slave_instance, EXT1_SPI_MODULE, &config_spi_slave);
       ```

12. Enable SPI module.

```
spi_enable(&spi_slave_instance);
```

3. Setup of the callback functionality.

```
configure_spi_slave_callbacks();
```

1. Register callback function for buffer transmitted.

```
spi_register_callback(&spi_slave_instance, spi_slave_callback,
        SPI_CALLBACK_BUFFER_RECEIVED);
```

2. Enable callback for buffer transmitted.

```
spi_enable_callback(&spi_slave_instance,
SPI_CALLBACK_BUFFER_RECEIVED);
```

9.4.2. Use Case

9.4.2.1. Code

Add the following to your user application `main()`.

```
spi_read_buffer_job(&spi_slave_instance, buffer_rx, BUF_LENGTH, 0x00);
while(!transfer_complete_spi_slave) {
    /* Wait for transfer from master */
}
for (uint8_t i = 0; i < BUF_LENGTH; i++) {
    if(buffer_rx[i] != buffer_expect[i]) {
        result++;
    }
}

while (true) {
    /* Infinite loop */
    if (result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 30000;
        while(delay--) {
        }
    } else {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 600000;
        while(delay--) {
        }
    }
}
```

9.4.2.2. Workflow

1. Initiate a read buffer job.

```
spi_read_buffer_job(&spi_slave_instance, buffer_rx, BUF_LENGTH, 0x00);
```

2. Wait for the transfer to be complete.

```
while(!transfer_complete_spi_slave) {
    /* Wait for transfer from master */
}
```

3. Compare the received data with the transmitted data from SPI master.

```
for (uint8_t i = 0; i < BUF_LENGTH; i++) {
    if(buffer_rx[i] != buffer_expect[i]) {
        result++;
    }
}
```

4. Infinite loop. If the data is matched, LED0 will flash slowly. Otherwise, LED will flash quickly.

```
while (true) {
    /* Infinite loop */
    if (result) {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 30000;
        while(delay--) {
        }
    } else {
        port_pin_toggle_output_level(LED_0_PIN);
        /* Add a short delay to see LED toggle */
        volatile uint32_t delay = 600000;
        while(delay--) {
        }
    }
}
```

9.4.3. Callback

When the buffer is successfully transmitted from the master, the callback function will be called.

9.4.3.1. Workflow

1. Let the application know that the buffer is transmitted by setting the global variable to true.

```
transfer_complete_spi_slave = true;
```

9.5. Quick Start Guide for Using DMA with SERCOM SPI

The supported board list:

- SAM D21 Xplained Pro
- SAM R21 Xplained Pro
- SAM L21 Xplained Pro
- SAM L22 Xplained Pro
- SAM DA1 Xplained Pro
- SAM C21 Xplained Pro

This quick start will transmit a buffer data from master to slave through DMA. In this use case the SPI master will be configured with the following settings on SAM Xplained Pro:

- Master Mode enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E
- 8-bit character size
- Not enabled in sleep mode

- Baudrate 100000
- GLCK generator 0

The SPI slave will be configured with the following settings:

- Slave mode enabled
- Preloading of shift register enabled
- MSB of the data is transmitted first
- Transfer mode 0
- SPI MUX Setting E
- 8-bit character size
- Not enabled in sleep mode
- GLCK generator 0

Note that the pinouts on other boards may different, see next sector for details.

9.5.1. Setup

9.5.1.1. Prerequisites

The following connections has to be made using wires:

- SAM D21/DA1 Xplained Pro.
  - **SS_0:** EXT1 PIN15 (PA05) <> EXT2 PIN15 (PA17)
  - **DO/DI**: EXT1 PIN16 (PA06) <> EXT2 PIN17 (PA16)
  - **DI/DO**: EXT1 PIN17 (PA04) <> EXT2 PIN16 (PA18)
  - **SCK:** EXT1 PIN18 (PA07) <> EXT2 PIN18 (PA19)
- SAM R21 Xplained Pro.
  - **SS_0:** EXT1 PIN15 (PB03) <> EXT1 PIN10 (PA23)
  - **DO/DI**: EXT1 PIN16 (PB22) <> EXT1 PIN9 (PA22)
  - **DI/DO**: EXT1 PIN17 (PB02) <> EXT1 PIN7 (PA18)
  - **SCK:** EXT1 PIN18 (PB23) <> EXT1 PIN8 (PA19)
- SAM L21 Xplained Pro.
  - **SS_0:** EXT1 PIN15 (PA05) <> EXT1 PIN12 (PA09)
  - **DO/DI**: EXT1 PIN16 (PA06) <> EXT1 PIN11 (PA08)
  - **DI/DO**: EXT1 PIN17 (PA04) <> EXT2 PIN03 (PA10)
  - **SCK:** EXT1 PIN18 (PA07) <> EXT2 PIN04 (PA11)
- SAM L22 Xplained Pro.
  - **SS_0:** EXT1 PIN15 (PB21) <> EXT2 PIN15 (PA17)
  - **DO/DI**: EXT1 PIN16 (PB00) <> EXT2 PIN17 (PA16)
  - **DI/DO**: EXT1 PIN17 (PB02) <> EXT2 PIN16 (PA18)
  - **SCK:** EXT1 PIN18 (PB01) <> EXT2 PIN18 (PA19)
- SAM C21 Xplained Pro.
  - **SS_0:** EXT1 PIN15 (PA17) <> EXT2 PIN15 (PB03)
  - **DO/DI**: EXT1 PIN16 (PA18) <> EXT2 PIN17 (PB02)
  - **DI/DO**: EXT1 PIN17 (PA16) <> EXT2 PIN16 (PB00)
  - **SCK:** EXT1 PIN18 (PA19) <> EXT2 PIN18 (PB01)

9.5.1.2. Code

Add to the main application source file, before user definitions and functions according to your board:

For SAM D21 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE  EXT2_SPI_MODULE
#define CONF_MASTER_SS_PIN      EXT2_PIN_SPI_SS_0
#define CONF_MASTER_MUX_SETTING EXT2_SPI_SERCOM_MUX_SETTING
#define CONF_MASTER_PINMUX_PAD0 EXT2_SPI_SERCOM_PINMUX_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 EXT2_SPI_SERCOM_PINMUX_PAD2
#define CONF_MASTER_PINMUX_PAD3 EXT2_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE  EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX   SERCOM1_DMAC_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX   SERCOM0_DMAC_ID_RX
```

For SAM R21 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE  SERCOM3
#define CONF_MASTER_SS_PIN      EXT1_PIN_10
#define CONF_MASTER_MUX_SETTING SPI_SIGNAL_MUX_SETTING_E
#define CONF_MASTER_PINMUX_PAD0 PINMUX_PA22C_SERCOM3_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 PINMUX_PA18D_SERCOM3_PAD2
#define CONF_MASTER_PINMUX_PAD3 PINMUX_PA19D_SERCOM3_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE  EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX   SERCOM3_DMAC_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX   SERCOM5_DMAC_ID_RX
```

For SAM L21 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE  SERCOM2
#define CONF_MASTER_SS_PIN      EXT1_PIN_12
#define CONF_MASTER_MUX_SETTING SPI_SIGNAL_MUX_SETTING_E
#define CONF_MASTER_PINMUX_PAD0 PINMUX_PA08D_SERCOM2_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 PINMUX_PA10D_SERCOM2_PAD2
#define CONF_MASTER_PINMUX_PAD3 PINMUX_PA11D_SERCOM2_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE  EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
```

```
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX   SERCOM2_DMAC_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX   SERCOM0_DMAC_ID_RX
```

For SAM L22 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE  EXT2_SPI_MODULE
#define CONF_MASTER_SS_PIN      EXT2_PIN_SPI_SS_0
#define CONF_MASTER_MUX_SETTING EXT2_SPI_SERCOM_MUX_SETTING
#define CONF_MASTER_PINMUX_PAD0 EXT2_SPI_SERCOM_PINMUX_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 EXT2_SPI_SERCOM_PINMUX_PAD2
#define CONF_MASTER_PINMUX_PAD3 EXT2_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE  EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX   EXT2_SPI_SERCOM_DMAC_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX   EXT1_SPI_SERCOM_DMAC_ID_RX
```

For SAM DA1 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE  EXT2_SPI_MODULE
#define CONF_MASTER_SS_PIN      EXT2_PIN_SPI_SS_0
#define CONF_MASTER_MUX_SETTING EXT2_SPI_SERCOM_MUX_SETTING
#define CONF_MASTER_PINMUX_PAD0 EXT2_SPI_SERCOM_PINMUX_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 EXT2_SPI_SERCOM_PINMUX_PAD2
#define CONF_MASTER_PINMUX_PAD3 EXT2_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE  EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX   SERCOM1_DMAC_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX   SERCOM0_DMAC_ID_RX
```

For SAM C21 Xplained Pro:

```
#define CONF_MASTER_SPI_MODULE  EXT2_SPI_MODULE
#define CONF_MASTER_SS_PIN      EXT2_PIN_SPI_SS_0
#define CONF_MASTER_MUX_SETTING EXT2_SPI_SERCOM_MUX_SETTING
#define CONF_MASTER_PINMUX_PAD0 EXT2_SPI_SERCOM_PINMUX_PAD0
#define CONF_MASTER_PINMUX_PAD1 PINMUX_UNUSED
#define CONF_MASTER_PINMUX_PAD2 EXT2_SPI_SERCOM_PINMUX_PAD2
#define CONF_MASTER_PINMUX_PAD3 EXT2_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_SLAVE_SPI_MODULE  EXT1_SPI_MODULE
#define CONF_SLAVE_MUX_SETTING EXT1_SPI_SERCOM_MUX_SETTING
```

```
#define CONF_SLAVE_PINMUX_PAD0 EXT1_SPI_SERCOM_PINMUX_PAD0
#define CONF_SLAVE_PINMUX_PAD1 EXT1_SPI_SERCOM_PINMUX_PAD1
#define CONF_SLAVE_PINMUX_PAD2 EXT1_SPI_SERCOM_PINMUX_PAD2
#define CONF_SLAVE_PINMUX_PAD3 EXT1_SPI_SERCOM_PINMUX_PAD3
```

```
#define CONF_PERIPHERAL_TRIGGER_TX   SERCOM5_DMAC_ID_TX
#define CONF_PERIPHERAL_TRIGGER_RX   SERCOM1_DMAC_ID_RX
```

Add to the main application source file, outside of any functions:

```
#define BUF_LENGTH 20
```

```
#define TEST_SPI_BAUDRATE             1000000UL
```

```
#define SLAVE_SELECT_PIN CONF_MASTER_SS_PIN
```

```
static const uint8_t buffer_tx[BUF_LENGTH] = {
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A,
        0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14,
};
static uint8_t buffer_rx[BUF_LENGTH];
```

```
struct spi_module spi_master_instance;
struct spi_module spi_slave_instance;
```

```
static volatile bool transfer_tx_is_done = false;
static volatile bool transfer_rx_is_done = false;
```

```
struct spi_slave_inst slave;
```

```
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_tx;
DmacDescriptor example_descriptor_rx;
```

Copy-paste the following setup code to your user application:

```
static void transfer_tx_done(struct dma_resource* const resource )
{
    transfer_tx_is_done = true;
}

static void transfer_rx_done(struct dma_resource* const resource )
{
    transfer_rx_is_done = true;
}

static void configure_dma_resource_tx(struct dma_resource *tx_resource)
{
    struct dma_resource_config tx_config;

    dma_get_config_defaults(&tx_config);

    tx_config.peripheral_trigger = CONF_PERIPHERAL_TRIGGER_TX;
    tx_config.trigger_action = DMA_TRIGGER_ACTON_BEAT;

    dma_allocate(tx_resource, &tx_config);
}
```

```
static void configure_dma_resource_rx(struct dma_resource *rx_resource)
{
    struct dma_resource_config rx_config;

    dma_get_config_defaults(&rx_config);

    rx_config.peripheral_trigger = CONF_PERIPHERAL_TRIGGER_RX;
    rx_config.trigger_action = DMA_TRIGGER_ACTON_BEAT;

    dma_allocate(rx_resource, &rx_config);
}

static void setup_transfer_descriptor_tx(DmacDescriptor *tx_descriptor)
{
    struct dma_descriptor_config tx_descriptor_config;

    dma_descriptor_get_config_defaults(&tx_descriptor_config);

    tx_descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
    tx_descriptor_config.dst_increment_enable = false;
    tx_descriptor_config.block_transfer_count = sizeof(buffer_tx)/
sizeof(uint8_t);
    tx_descriptor_config.source_address = (uint32_t)buffer_tx +
sizeof(buffer_tx);
    tx_descriptor_config.destination_address =
        (uint32_t)(&spi_master_instance.hw->SPI.DATA.reg);

    dma_descriptor_create(tx_descriptor, &tx_descriptor_config);
}

static void setup_transfer_descriptor_rx(DmacDescriptor *rx_descriptor)
{
    struct dma_descriptor_config rx_descriptor_config;

    dma_descriptor_get_config_defaults(&rx_descriptor_config);

    rx_descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
    rx_descriptor_config.src_increment_enable = false;
    rx_descriptor_config.block_transfer_count = sizeof(buffer_rx)/
sizeof(uint8_t);
    rx_descriptor_config.source_address =
        (uint32_t)(&spi_slave_instance.hw->SPI.DATA.reg);
    rx_descriptor_config.destination_address =
        (uint32_t)buffer_rx + sizeof(buffer_rx);

    dma_descriptor_create(rx_descriptor, &rx_descriptor_config);
}

static void configure_spi_master(void)
{
    struct spi_config config_spi_master;
    struct spi_slave_inst_config slave_dev_config;
    /* Configure and initialize software device instance of peripheral
slave */
    spi_slave_inst_get_config_defaults(&slave_dev_config);
    slave_dev_config.ss_pin = SLAVE_SELECT_PIN;
    spi_attach_slave(&slave, &slave_dev_config);
    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_master);
    config_spi_master.mode_specific.master.baudrate = TEST_SPI_BAUDRATE;
    config_spi_master.mux_setting = CONF_MASTER_MUX_SETTING;
```

```
    /* Configure pad 0 for data in */
    config_spi_master.pinmux_pad0 = CONF_MASTER_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_master.pinmux_pad1 = CONF_MASTER_PINMUX_PAD1;
    /* Configure pad 2 for data out */
    config_spi_master.pinmux_pad2 = CONF_MASTER_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_master.pinmux_pad3 = CONF_MASTER_PINMUX_PAD3;
    spi_init(&spi_master_instance, CONF_MASTER_SPI_MODULE,
&config_spi_master);

    spi_enable(&spi_master_instance);

}

static void configure_spi_slave(void)
{
    struct spi_config config_spi_slave;

    /* Configure, initialize and enable SERCOM SPI module */
    spi_get_config_defaults(&config_spi_slave);
    config_spi_slave.mode = SPI_MODE_SLAVE;
    config_spi_slave.mode_specific.slave.preload_enable = true;
    config_spi_slave.mode_specific.slave.frame_format =
SPI_FRAME_FORMAT_SPI_FRAME;
    config_spi_slave.mux_setting = CONF_SLAVE_MUX_SETTING;
    /* Configure pad 0 for data in */
    config_spi_slave.pinmux_pad0 = CONF_SLAVE_PINMUX_PAD0;
    /* Configure pad 1 as unused */
    config_spi_slave.pinmux_pad1 = CONF_SLAVE_PINMUX_PAD1;
    /* Configure pad 2 for data out */
    config_spi_slave.pinmux_pad2 = CONF_SLAVE_PINMUX_PAD2;
    /* Configure pad 3 for SCK */
    config_spi_slave.pinmux_pad3 = CONF_SLAVE_PINMUX_PAD3;
    spi_init(&spi_slave_instance, CONF_SLAVE_SPI_MODULE,
&config_spi_slave);

    spi_enable(&spi_slave_instance);

}
```

Add to user application initialization (typically the start of `main()`):

```
configure_spi_master();
configure_spi_slave();

configure_dma_resource_tx(&example_resource_tx);
configure_dma_resource_rx(&example_resource_rx);

setup_transfer_descriptor_tx(&example_descriptor_tx);
setup_transfer_descriptor_rx(&example_descriptor_rx);

dma_add_descriptor(&example_resource_tx, &example_descriptor_tx);
dma_add_descriptor(&example_resource_rx, &example_descriptor_rx);

dma_register_callback(&example_resource_tx, transfer_tx_done,
        DMA_CALLBACK_TRANSFER_DONE);
dma_register_callback(&example_resource_rx, transfer_rx_done,
        DMA_CALLBACK_TRANSFER_DONE);

dma_enable_callback(&example_resource_tx, DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource_rx, DMA_CALLBACK_TRANSFER_DONE);
```

9.5.1.3. Workflow

1. Create a module software instance structure for the SPI module to store the SPI driver state while it is in use.

```
struct spi_module spi_master_instance;
struct spi_module spi_slave_instance;
```

   Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Create a module software instance structure for DMA resource to store the DMA resource state while it is in use.

```
struct dma_resource example_resource_tx;
struct dma_resource example_resource_rx;
```

   Note: This should never go out of scope as long as the module is in use. In most cases, this should be global.

3. Create transfer done flag to indication DMA transfer done.

```
static volatile bool transfer_tx_is_done = false;
static volatile bool transfer_rx_is_done = false;
```

4. Define the buffer length for TX/RX.

```
#define BUF_LENGTH 20
```

5. Create buffer to store the data to be transferred.

```
static const uint8_t buffer_tx[BUF_LENGTH] = {
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A,
        0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14,
};
static uint8_t buffer_rx[BUF_LENGTH];
```

6. Create the SPI module configuration struct, which can be filled out to adjust the configuration of a physical SPI peripheral.

```
struct spi_config config_spi_master;
```

```
struct spi_config config_spi_slave;
```

7. Initialize the SPI configuration struct with the module's default values.

```
spi_get_config_defaults(&config_spi_master);
```

```
spi_get_config_defaults(&config_spi_slave);
```

   Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

8. Alter the SPI settings to configure the physical pinout, baudrate, and other relevant parameters.

```
config_spi_master.mux_setting = CONF_MASTER_MUX_SETTING;
```

```
config_spi_slave.mux_setting = CONF_SLAVE_MUX_SETTING;
```

9. Configure the SPI module with the desired settings, retrying while the driver is busy until the configuration is stressfully set.

```
spi_init(&spi_master_instance, CONF_MASTER_SPI_MODULE,
&config_spi_master);
```

```
spi_init(&spi_slave_instance, CONF_SLAVE_SPI_MODULE,
&config_spi_slave);
```

10. Enable the SPI module.

```
spi_enable(&spi_master_instance);
```

```
spi_enable(&spi_slave_instance);
```

11. Create the DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_resource_config tx_config;
```

```
struct dma_resource_config rx_config;
```

12. Initialize the DMA resource configuration struct with the module's default values.

```
dma_get_config_defaults(&tx_config);
```

```
dma_get_config_defaults(&rx_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

13. Set extra configurations for the DMA resource. It is using peripheral trigger. SERCOM TX empty and RX complete trigger causes a beat transfer in this example.

```
tx_config.peripheral_trigger = CONF_PERIPHERAL_TRIGGER_TX;
tx_config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
```

```
rx_config.peripheral_trigger = CONF_PERIPHERAL_TRIGGER_RX;
rx_config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
```

14. Allocate a DMA resource with the configurations.

```
dma_allocate(tx_resource, &tx_config);
```

```
dma_allocate(rx_resource, &rx_config);
```

15. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```
struct dma_descriptor_config tx_descriptor_config;
```

```
struct dma_descriptor_config rx_descriptor_config;
```

16. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```
dma_descriptor_get_config_defaults(&tx_descriptor_config);
```

```
dma_descriptor_get_config_defaults(&rx_descriptor_config);
```

Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

17. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```
tx_descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
tx_descriptor_config.dst_increment_enable = false;
tx_descriptor_config.block_transfer_count = sizeof(buffer_tx)/
sizeof(uint8_t);
tx_descriptor_config.source_address = (uint32_t)buffer_tx +
sizeof(buffer_tx);
tx_descriptor_config.destination_address =
    (uint32_t)(&spi_master_instance.hw->SPI.DATA.reg);
```

```
rx_descriptor_config.beat_size = DMA_BEAT_SIZE_BYTE;
rx_descriptor_config.src_increment_enable = false;
rx_descriptor_config.block_transfer_count = sizeof(buffer_rx)/
sizeof(uint8_t);
rx_descriptor_config.source_address =
    (uint32_t)(&spi_slave_instance.hw->SPI.DATA.reg);
rx_descriptor_config.destination_address =
    (uint32_t)buffer_rx + sizeof(buffer_rx);
```

18. Create the DMA transfer descriptor.

```
dma_descriptor_create(tx_descriptor, &tx_descriptor_config);
```

```
dma_descriptor_create(rx_descriptor, &rx_descriptor_config);
```

9.5.2. Use Case

9.5.2.1. Code

Copy-paste the following code to your user application:

```
spi_select_slave(&spi_master_instance, &slave, true);

dma_start_transfer_job(&example_resource_rx);
dma_start_transfer_job(&example_resource_tx);

while (!transfer_rx_is_done) {
    /* Wait for transfer done */
}

spi_select_slave(&spi_master_instance, &slave, false);

while (true) {
}
```

9.5.2.2. Workflow

1. Select the slave.

```
spi_select_slave(&spi_master_instance, &slave, true);
```

2. Start the transfer job.

```
dma_start_transfer_job(&example_resource_rx);
dma_start_transfer_job(&example_resource_tx);
```

3. Wait for transfer done.

```
while (!transfer_rx_is_done) {
    /* Wait for transfer done */
}
```

4. Deselect the slave.

```
spi_select_slave(&spi_master_instance, &slave, false);
```

5. Enter endless loop.

```
while (true) {
}
```

10. MUX Settings

The following lists the possible internal SERCOM module pad function assignments for the four SERCOM pads in both SPI Master and SPI Slave modes. They are combinations of DOPO and DIPO in CTRLA. Note that this is in addition to the physical GPIO pin MUX of the device, and can be used in conjunction to optimize the serial data pin-out.

10.1. Master Mode Settings

The following table describes the SERCOM pin functionalities for the various MUX settings, whilst in SPI Master mode.

Note: If MISO is unlisted, the SPI receiver must not be enabled for the given MUX setting.

Combination	DOPO / DIPO	SERCOM PAD[0]	SERCOM PAD[1]	SERCOM PAD[2]	SERCOM PAD[3]
A	0x0 / 0x0	MOSI	SCK	-	-
B	0x0 / 0x1	MOSI	SCK	-	-
C	0x0 / 0x2	MOSI	SCK	MISO	-
D	0x0 / 0x3	MOSI	SCK	-	MISO
E	0x1 / 0x0	MISO	-	MOSI	SCK
F	0x1 / 0x1	-	MISO	MOSI	SCK
G	0x1 / 0x2	-	-	MOSI	SCK
H	0x1 / 0x3	-	-	MOSI	SCK
I	0x2 / 0x0	MISO	SCK	-	MOSI
J	0x2 / 0x1	-	SCK	-	MOSI
K	0x2 / 0x2	-	SCK	MISO	MOSI
L	0x2 / 0x3	-	SCK	-	MOSI
M	0x3 / 0x0	MOSI	-	-	SCK
N	0x3 / 0x1	MOSI	MISO	-	SCK
O	0x3 / 0x2	MOSI	-	MISO	SCK
P	0x3 / 0x3	MOSI	-	-	SCK

10.2. Slave Mode Settings

The following table describes the SERCOM pin functionalities for the various MUX settings, whilst in SPI Slave mode.

Note: If MISO is unlisted, the SPI receiver must not be enabled for the given MUX setting.

Combination	DOPO / DIPO	SERCOM PAD[0]	SERCOM PAD[1]	SERCOM PAD[2]	SERCOM PAD[3]
A	0x0 / 0x0	MISO	SCK	/SS	-
B	0x0 / 0x1	MISO	SCK	/SS	-
C	0x0 / 0x2	MISO	SCK	/SS	-
D	0x0 / 0x3	MISO	SCK	/SS	MOSI
E	0x1 / 0x0	MOSI	/SS	MISO	SCK
F	0x1 / 0x1	-	/SS	MISO	SCK
G	0x1 / 0x2	-	/SS	MISO	SCK
H	0x1 / 0x3	-	/SS	MISO	SCK
I	0x2 / 0x0	MOSI	SCK	/SS	MISO
J	0x2 / 0x1	-	SCK	/SS	MISO
K	0x2 / 0x2	-	SCK	/SS	MISO
L	0x2 / 0x3	-	SCK	/SS	MISO
M	0x3 / 0x0	MISO	/SS	-	SCK
N	0x3 / 0x1	MISO	/SS	-	SCK
O	0x3 / 0x2	MISO	/SS	MOSI	SCK
P	0x3 / 0x3	MISO	/SS	-	SCK

11. Document Revision History

Doc. Rev.	Date	Comments
42115E	12/2015	Add SAM L21/L22, SAM DA1, SAM D09, and SAM C21 support
42115D	12/2014	Add SAM R21/D10/D11 support
42115C	01/2014	Add SAM D21 support
42115B	11/2013	Replaced the pad multiplexing documentation with a condensed table
42115A	06/2013	Initial release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**