Freescale Semiconductor
User Guide

Document Number:N/A
Rev 1, 12/2013

SPIMaster_LDD

Embedded Component User Guide

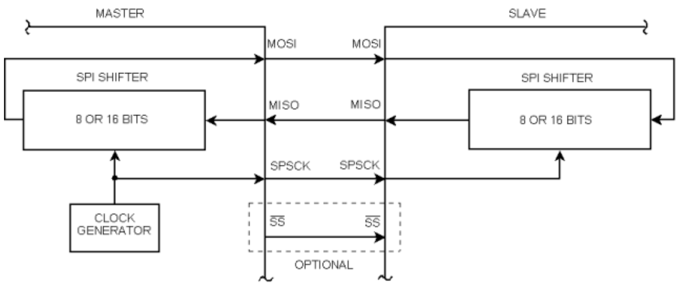
Contents

1 General Information

Component Level: Logical Device Driver

Category: Logical Device Drivers-Communication

This component (in cooperation with similar device as SyncSlave) implements a serial synchronous master-slave communication. Only two devices can communicate at a time - one of them is a MASTER and the other one is a SLAVE.



The SPI bus specifies four logic signals:

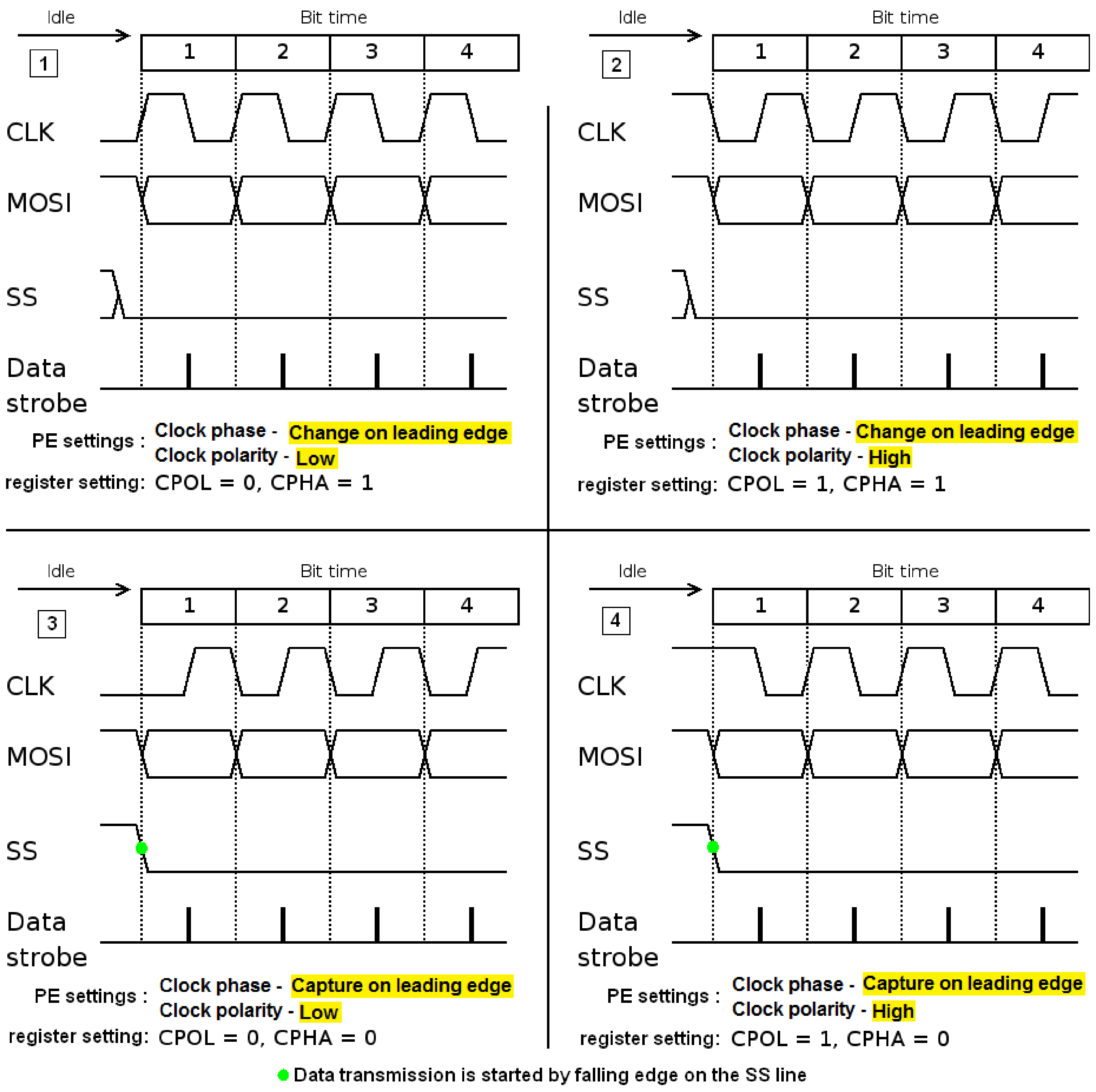
- SPSCK — Serial Clock (output from master)
- MOSI/SIMO — Master Output, Slave Input (output from master)
- MISO/SOMI — Master Input, Slave Output (output from slave)
- SS — Slave Select (active low; output from master)

Alternative naming conventions are also widely used:

- SCLK, SCK, CLK — Serial Clock (output from master)
- SDI, DI, SI — Serial Data In
- SDO, DO, SO — Serial Data Out
- nCS, CS, nSS, STE — Chip Select, Slave Transmit Enable (active low; output from master)

The difference between the master and slave is that the master generates the clock signal while the slave receives it. Consequently, if the master has no data to send to slave and needs to receive data from the slave, it still has to transmit a character.

Configuring Clock polarity and Clock phase properties


Idle
Bit time
1
1 2 3 4
CLK
MOSI
SS
Data strobe
PE settings : Clock phase - Change on leading edge
Clock polarity - Low
register setting: CPOL = 0, CPHA = 1
Idle
Bit time
2
1 2 3 4
CLK
MOSI
SS
Data strobe
PE settings : Clock phase - Change on leading edge
Clock polarity - High
register setting: CPOL = 1, CPHA = 1
Idle
Bit time
3
1 2 3 4
CLK
MOSI
SS
Data strobe
PE settings : Clock phase - Capture on leading edge
Clock polarity - Low
register setting: CPOL = 0, CPHA = 0
Idle
Bit time
4
1 2 3 4
CLK
MOSI
SS
Data strobe
PE settings : Clock phase - Capture on leading edge
Clock polarity - High
register setting: CPOL = 1, CPHA = 0
● Data transmission is started by falling edge on the SS line

2 Properties

This section describes component's properties. Properties are parameters of the component that influence the generated code. Please see the Processor Expert user manual for more details.

- **Component name** - Name of the component.
- **Device** - Channel used for a serial synchronous communication.

- **Interrupt service/event** - Component may or may not be implemented using interrupts. If this property is set to "Enabled", the component functionality (e.g. sending and receiving data) depends on the interrupt service and will not operate if the CPU interrupts are disabled. For details please refer to chapter "interrupt service in the component's generated code".

  The following items are available only if the group is enabled (the value is "Enabled"):

  - **Input interrupt** - Interrupt from the serial receiver.
  - **Input interrupt priority** - Receiver interrupt priority.
  - *Settings only if component supports interrupt service routine properties.*
    - **ISR Name** - Name of the internal component interrupt service routine (ISR) invoked by this interrupt vector. This property is for information only.
  - *Settings only if compiler supports preserve interrupt preserve registers.*
    - **Interrupt preserve registers** - If this property is set to 'yes' then the "saveall" modifier is generated together with the "#pragma interrupt" statement before the appropriate ISR function definition. This modifier preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT_RESTOREALL routines in the Runtime Library. The "#pragma interrupt called" statement, located before the appropriate event in Events.c, should be removed in this mode.

      If this option is set to 'no', a user must ensure that all registers used by any routine called by the ISR is saved and restored by that routine.

      The "#pragma interrupt called" statement, located before the appropriate event in Events.c, is needed in this mode.
  - **Output interrupt** - Interrupt from serial transmitter.
  - **Output interrupt priority** - Transmitter interrupt priority.
  - *Settings only if component supports interrupt service routine properties.*
    - **ISR Name** - Name of the internal component interrupt service routine (ISR) invoked by this interrupt vector. This property is for information only.
  - *Settings only if compiler supports preserve interrupt preserve registers.*
    - **Interrupt preserve registers** - If this property is set to 'yes' then the "saveall" modifier is generated together with the "#pragma interrupt" statement before the appropriate ISR function definition. This modifier preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT_RESTOREALL routines in the Runtime Library. The "#pragma interrupt called" statement, located before the appropriate event in Events.c, should be removed in this mode.

      If this option is set to 'no', a user must ensure that all registers used by any routine called by the ISR is saved and restored by that routine.

      The "#pragma interrupt called" statement, located before the appropriate event in Events.c, is needed in this mode.
- **Settings** - Common component settings.
  - **Input pin** - Allows enabling or disabling of the input pin signal.

    The following items are available only if the group is enabled (the value is "Enabled"):

    - **Pin** - Input pin for data.
    - **Pin signal** - Signal name of an input pin.
  - **Output pin** - Allows enabling or disabling of the output pin signal.

The following items are available only if the group is enabled (the value is "Enabled"):

- **Pin** - Output pin for data.
- **Pin signal** - Signal name of an output pin.

- **Clock pin** - Group of properties for clock signal.
  - **Pin** - Output pin for the clock.
  - **Pin signal** - Signal name of the clock pin.
- **Chip select list** - List of chip selects allows to prepare chip select pins for various slaves. Item Initial chip select specifies which of chip selects is selected after the initialization. The SelectConfiguration method allows switch chip select in run time.

  One Item of the list looks like:

  Chip select 0 - Chip select settings.
  - **Pin** - Chip select pin.
  - **Pin signal** - Signal name of the chip select pin.
  - **Active level** - Determines active level of the chip select pin.
- *Settings only if SPI supports a CS external demultiplexer.*
  - **CS external demultiplexer** - This item enables support and settings for external demultiplexer. It allows to expand the number of chip selects against the device supports directly with the aid of an external decoder. If chip select demultiplexer support is enabled, the chip select list defines address signals connected to the external multiplexer (Chip select 0 is least significant bit) and the Chip select strobe pin specifies a chip select strobe signal for the external demultiplexer. The range of CS addresses depends on the count of Chip select and it is 2^Chip select list size.

    The following items are available only if the group is enabled (the value is "Enabled"):
    - **Chip select strobe pin** - This signal is used as a strobe to extrnal peripheral chip select demultiplexer, preventing glitches on the demultiplexer outputs.
    - **Chip select strobe pin signal** - Signal name of the slave select strobe pin.
- **Attribute set list** - List of attribute sets allows to prepare different settings for various slaves. Item Initial attribute set specifies which of those attribute sets is selected after the initialization. The SelectConfiguration method allows switch attribute sets in run time.

  One Item of the list looks like:

  Attribute set 0 - Group of transfer attributes.
  - **Width** - The number of information bits transferred per character.
  - **MSB first** - The order in which bits are transmitted (MSB: most significant bit, LSB: less significant bit).
  - **Clock polarity** - Selects which edge of DSPI_SCK causes data to change and which edge causes data to be captured. This bit is used in both master and slave mode. For successful communication between serial devices, the devices must have identical clock phase settings.

    There are 2 options:
    - High: The inactive state of clock signal is high.
    - Low: The inactive state of clock signal is low.
  - **Clock phase** - Selects which edge of DSPI_SCK causes data to change and which edge causes data to be captured. This bit is used in both master and slave mode. For successful communication between serial devices, the devices must have identical clock phase settings.

    There are 2 options:

    - Capture on leading edge: Data captured on the leading edge of clock signal and changed on the following edge.
    - Change on leading edge: Data changed on the leading edge of clock signal and captured on the following edge.
  - **Parity** - The type of the parity bit. The parity bit is added to information bits so the total width of transferred bits is (Width + 1).
  - **Chip select toggling** - This items enables chip select toggling between characters transmission.
  - **Clock rate index** - This item selects the clock rate setting from the list specified in Clock rate timing property.
  - *Settings only if SPI supports to define delay between the transmission of data characters.*
    - **Delay between chars index** - This item selects the delay between characters setting from the list specified in Delay between chars timing property.
  - *Settings only if SPI supports to define CS to CLK delay.*
    - **CS to CLK delay index** - This item selects the CS to CLK delay setting from the list specified in CS to CLK delay timing property.
  - *Settings only if SPI supports to define CLK to CS delay.*
    - **CLK to CS delay index** - This item selects the CLK to CS delay setting from the list specified in CLK to CS delay timing property.
- **Clock rate** - Communication shift clock rate. It is necessary to enter a value and an unit (see Timing Setting Syntax). The setting can be made with a help of the Timing dialog box that opens when clicking on button (...).
- *Settings only if SPI supports to define delay between the transmission of data characters.*
  - **Delay between chars** - Controls the time between the transmission of data characters.
- *Settings only if SPI supports to define CS to CLK delay.*
  - **CS to CLK delay** - Controls the time between CS activation and first CLK edge.
- *Settings only if SPI supports to define CLK to CS delay.*
  - **CLK to CS delay** - Controls the time between last CLK edge and CS deactivation.
- **HW input buffer size** - This item sets the size of peripheral's input buffer. The shift register is not included in this value.

  There are 7 options:

  - Max buffer size: Maximal value that is allowed by the device.
  - 0: No buffer, only the shift register.
  - 1: It may be also called as double-buffer. The shift register is not included in this value.
  - 2: It may be also called as 2-item FIFO. The shift register is not included in this value.
  - 4: It may be also called as 4-item FIFO. The shift register is not included in this value.
  - 8: It may be also called as 8-item FIFO. The shift register is not included in this value.
  - 16: It may be also called as 16-item FIFO. The shift register is not included in this value.
- **HW input watermark** - Possible values are 1 to peripheral's input buffer size. This setting ensures characters to be drained from peripheral's buffer timely. Those characters are moved from peripheral's buffer to user's buffer when the selected threshold is reached. Selected value 1 means that characters are read from the peripheral's buffer as soon as there is placed one character into the peripheral's buffer. Selected value 2 means that characters

are read from peripheral's buffer as soon as there are two characters placed in the peripheral's buffer and so on. If selected value gets too near peripheral's input buffer size, characters needn't be read from the peripheral's buffer timely, the next received character causes that peripheral's buffer overruns and the OnError event may be invoked.

- *Settings only if the component supports DMA for the selected CPU family.*
    - **Receiver DMA** - Specifies whether DMA transfer will be used for the receiver.

      The following items are available only if the group is enabled (the value is "Enabled"):
        - **DMA channel** - DMA channel transfer interface (for details about settings see Component Inheritance & Component Sharing).
- **HW output buffer size** - This item sets the size of peripheral's output buffer. The shift register is not included in this value.

  There are 7 options:
    - Max buffer size: Maximal value that is allowed by the device.
    - 0: No buffer, only the shift register.
    - 1: It may be also called as double-buffer. The shift register is not included in this value.
    - 2: It may be also called as 2-item FIFO. The shift register is not included in this value.
    - 4: It may be also called as 4-item FIFO. The shift register is not included in this value.
    - 8: It may be also called as 8-item FIFO. The shift register is not included in this value.
    - 16: It may be also called as 16-item FIFO. The shift register is not included in this value.
- **HW output watermark** - Possible values are 1 to peripheral's output buffer size. This setting ensures characters to be delivered to peripheral's buffer timely. Those characters are moved from user's buffer to the peripheral's buffer when the selected threshold (of empty items in the peripheral's buffer) is reached. Selected value 1 means that moved from user's into the peripheral's buffer as soon as place for one character is free in the peripheral's buffer. Selected value 2 means that characters are put into the peripheral's buffer as soon as place for two characters are missing in the peripheral's buffer and so on. If selected value gets too near peripheral's output buffer size, characters needn't be put into the peripheral's buffer timely, peripheral's buffer may underrun and the OnError event may be invoked.
- *Settings only if the component supports DMA for the selected CPU family.*
    - **Transmitter DMA** - Specifies whether DMA transfer will be used for the transmitter.

      The following items are available only if the group is enabled (the value is "Enabled"):
        - **DMA channel** - DMA channel transfer interface for data (for details about settings see Component Inheritance & Component Sharing).

- **Initialization** - Initial settings (after power-on or reset).
    - **Initial chip select** - This item selects initial chip select from the Chip select list. The SelectConfiguration method mustn't be called prior to transmission. If the device supports chip select external demultiplexer and the external demultiplexer is enabled, this item represents the target address. This setting is available only if the Chip select list contains at least one item.
    - **Initial attribute set** - This item selects initial attribute set from the Attribute set list. The SelectConfiguration method mustn't be called prior to transmission.
    - **Enabled in init. code** - Enabled in initialization code.
    -

    - **Auto initialization** - Automated initialization of the component. The component Init method is automatically called from CPU component initialization function PE_low_level_init(). In this mode, the constant <ComponentName>_DeviceData is defined in component header file and it can be used as a device data structure pointer that can be passed as a first parameter to all component methods.

- **Event mask** - This group defines initialization event mask value.
    - **OnBlockSent** - Specifies if OnBlockSent event is enabled in initialzation code.
    - **OnBlockReceived** - Specifies if OnReceived event is enabled in initialzation code.
    - **OnError** - Specifies if OnError event is enabled in initialzation code.

•

- **CPU clock/configuration selection** - Settings for the CPU clock configurations: specifies whether the component is supported or not.

  For details about speed modes please refer to page Speed Modes Support.
    - **Clock configuration 0** - The component is enabled/disabled in the clock configuration 0.
    - **Clock configuration 1** - The component is enabled/disabled in the clock configuration 1.
    - **Clock configuration 2** - The component is enabled/disabled in the clock configuration 2.
    - **Clock configuration 3** - The component is enabled/disabled in the clock configuration 3.
    - **Clock configuration 4** - The component is enabled/disabled in the clock configuration 4.
    - **Clock configuration 5** - The component is enabled/disabled in the clock configuration 5.
    - **Clock configuration 6** - The component is enabled/disabled in the clock configuration 6.
    - **Clock configuration 7** - The component is enabled/disabled in the clock configuration 7.

3 Methods

This section describes component's methods. Methods are user-callable functions/subroutines intended for the component runtime control. Please see the Processor Expert user manual for more details.

3.1 Init

Initializes the device. Allocates memory for the device data structure, allocates interrupt vectors and sets interrupt priority, sets pin routing, sets timing, etc.

If the "Enable in init. code" is set to "yes" value then the device is also enabled(see the description of the Enable() method). In this case the Enable() method is not necessary and needn't to be generated.

This method can be called only once. Before the second call of Init() the Deinit() must be called first.

Prototype

```
LDD_TDeviceData* Init(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr: Pointer to LDD_TUserData* - Pointer to the user or RTOS specific data. This pointer will be passed as an event or callback parameter.

Return value

- *Return value:LDD_TDeviceData** - Device data structure pointer.

3.2 Deinit

This method deinitializes the device. It switches off the device, frees the device data structure memory, interrupts vectors, etc.

Prototype

```
void Deinit(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

3.3 Enable

This method enables SPI device. This method is intended to be used together with Disable() method to temporary switch On/Off the device after the device is initialized. This method is required if the Enabled in init. code property is set to "no" value.

Prototype

```
LDD_TError Enable(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - The device doesn't work in the active clock configuration

3.4 Disable

Disables the SPI device. When the device is disabled, some component methods should not be called. If so, error ERR_DISABLED may be reported. This method is intended to be used together with Enable() method to temporary switch on/off the device after the device is initialized. This method is not required. The Deinit() method can be used to switch off and uninstall the device.

Prototype

```
LDD_TError Disable(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

ERR_OK - OK

ERR_SPEED - The device doesn't work in the active clock configuration

3.5 SetEventMask

Enables/disables event(s). The events contained within the mask are enabled. Events not contained within the mask are disabled. The component event masks are defined in the PE_Types.h file. Note: Event that are not generated (See the "Method" tab in the Component inspector) are not handled by this method. In this case the method returns ERR_PARAM_MASK error code. See also method GetEventMask.

Prototype

```
LDD_TError SetEventMask(LDD_TDeviceData *DeviceDataPtr, LDD_TEventMask EventMask)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *EventMask:LDD_TEventMask* - Current EventMask. The component event masks are defined in the PE_Types.h file.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - The component does not work in the active clock configuration.

  ERR_DISABLED - The component or device is disabled.

  ERR_PARAM_MASK - Invalid event mask.

3.6 GetEventMask

Returns current events mask. Note: Event that are not generated (See the "Method" tab in the Component inspector) are not handled by this method. See also method SetEventMask.

Prototype

```
LDD_TEventMask GetEventMask(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TEventMask* - Current EventMask. The component event masks are defined in the PE_Types.h file.

3.7 SendBlock

This method sends a block of characters. The method returns ERR_BUSY when the previous block transmission is not completed. The method CancelBlockTransmission can be used to cancel a transmit operation.

Prototype

```
LDD_TError SendBlock(LDD_TDeviceData *DeviceDataPtr, LDD_TData *BufferPtr, uint16_t Size)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferPtr: Pointer to LDD_TData* - Pointer to the block of data to send.
- *Size:uint16_t* - Number of characters in the buffer.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_DISABLED - Component is disabled

  ERR_BUSY - The previous transmit request is pending

3.8 ReceiveBlock

This method specifies the number of data to receive. The method returns ERR_BUSY until the specified number of characters is received. The method CancelBlockReception can be used to cancel a running receive operation.

Prototype

```
LDD_TError ReceiveBlock(LDD_TDeviceData *DeviceDataPtr, LDD_TData *BufferPtr, uint16_t Size)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *BufferPtr: Pointer to LDD_TData* - Pointer to A buffer where received characters will be stored.
- *Size:uint16_t* - Size of the block

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_DISABLED - Component is disabled

  ERR_BUSY - The previous receive request is pending

3.9 GetSentDataNum

Returns the number of sent characters. This method is available only if method SendBlock is enabled.

Prototype

```
uint16_t GetSentDataNum(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:uint16_t* - The number of characters in the output buffer.

3.10 GetReceivedDataNum

Returns the number of received characters in the receive buffer. This method is available only if the ReceiveBlock method is enabled.

Prototype

```
uint16_t GetReceivedDataNum(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:uint16_t* - The number of characters in the input buffer.

3.11 GetBlockSentStatus

This method returns whether the transmitter is finished transmitting all data block. The status flag is accumulated, after calling this method the status is returned and cleared (set to "false" state). This method is available only if method SendBlock is enabled.

Prototype

```
bool GetBlockSentStatus(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:bool* - Return value:

  true - Data block is completely transmitted

  false - Data block isn't completely transmitted.

3.12 GetBlockReceivedStatus

This method returns whether the receiver is finished reception of all data block. The status flag is accumulated, after calling this method the status is returned and cleared (set to "false" state). This method is available only if method ReceiveBlock is enabled.

Prototype

```
bool GetBlockReceivedStatus(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:bool* - Return value:

  true - Data block is completely received

  false - Data block isn't completely received

3.13 CancelBlockTransmission

Immediately cancels running transmit process. Unsent data will never been sent. This method is available only if the SendBlock method is enabled.

Prototype

```
LDD_TError CancelBlockTransmission(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_DISABLED - Component is disabled

3.14 CancelBlockReception

Immediately cancels the running receive process started by the ReceiveBlock method. Characters already stored in the HW buffer will be lost. This method is available only if the ReceiveBlock method is enabled.

Prototype

```
LDD_TError CancelBlockReception(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - This device does not work in the active clock configuration

ERR_DISABLED - Component is disabled

3.15 GetError

This method returns a set of asserted flags. The flags are accumulated in the set. After calling this method the set is returned and cleared. This method is enabled when SPI device support error detect.

Prototype

```
LDD_TError GetError(LDD_TDeviceData *DeviceDataPtr, LDD_SPIMASTER_TError *ErrorPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *ErrorPtr: Pointer to LDD_SPIMASTER_TError* - A pointer to the returned set of error flags:

  LDD_SPIMASTER_RX_OVERFLOW - Receiver overflow

  LDD_SPIMASTER_PARITY_ERROR - Parity error (only if HW supports parity feature)

Return value

- *Return value:LDD_TError* - Error code (if GetError did not succeed), possible codes:

  ERR_OK - OK

  ERR_SPEED - This device does not work in the active clock configuration

  ERR_DISABLED - Component is disabled

3.16 SelectConfiguration

This method selects attributes of communication from the Attribute set list and select a chip select from the Chip select list. Once any configuration is selected, a transmission can be started multiple times. This method is available if number of chip selects or number of attribute set is greater than 1. If the device doesn't support chip select functionality the ChipSelect parameter is ignored.

Prototype

```
 LDD_TError SelectConfiguration(LDD_TDeviceData *DeviceDataPtr, uint8_t ChipSelect, uint8_t
AttributeSet)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *ChipSelect:uint8_t* - Chip select index from the Chip select list.
- *AttributeSet:uint8_t* - Communication attribute index from the Attribute set list

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - OK

  ERR_SPEED - This device does not work in the active clock configuration

ERR_DISABLED - Component is disabled

ERR_PARAM_CHIP_SELECT - Chip select index is out of range

ERR_PARAM_ATTRIBUTE_SET - Attribute set index is out of range

ERR_BUSY - Transmission is in progress

3.17 Main

This method is available only in the polling mode (Interrupt service/event = 'no'). If interrupt service is disabled this method replaces the interrupt handler. This method should be called if Receive/SendBlock was invoked before in order to run the reception/transmission. The end of the receiving/transmitting is indicated by OnBlockSent or OnBlockReceived event.

Prototype

```
void Main(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

3.18 ForceReceiver

Copy chars from the receiver FIFO to user buffer defined by method ReceiveBlock. This method is possible use for flush Rx FIFO, when size of user Rx buffer defined by method ReceiveBlock isn't multiple of Rx FIFO watermark value. This method is available only when ReceiveBlock method is enabled and value of Rx watermark is greater than 1.

Prototype

```
void ForceReceiver(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

3.19 GetStats

Returns communication statistics (e.g. sent character count, parity error count).

Prototype

```
LDD_SPIMASTER_TStats GetStats(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_SPIMASTER_TStats* - The actual communication statistics.

3.20 ClearStats

Clears the communication statistics. This method is available only if the GetStats method is enabled.

Prototype

```
 void ClearStats(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

3.21 SetOperationMode

This method requests to change the component's operation mode. Upon a request to change the operation mode, the component will finish a pending job first and then notify a caller that an operation mode has been changed. When no job is pending (ERR_OK), the component changes an operation mode immediately and notify a caller about this change.

Prototype

```
 LDD_TError SetOperationMode(LDD_TDeviceData *DeviceDataPtr, LDD_TDriverOperationMode
OperationMode, LDD_TCallback ModeChangeCallback, LDD_TCallbackParam
*ModeChangeCallbackParamPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *OperationMode:LDD_TDriverOperationMode* - Requested driver operation mode.
- *ModeChangeCallback:LDD_TCallback* - Callback to notify the upper layer once a mode has been changed.
- *ModeChangeCallbackParamPtr: Pointer to LDD_TCallbackParam* - Pointer to callback parameter to notify the upper layer once a mode has been changed.

Return value

- *Return value:LDD_TError* - Error code, possible codes:

  ERR_OK - The change operation mode request has been accepted, callback will notify an application as soon as the mode is changed.

  ERR_SPEED - The component does not work in the active clock configuration.

  ERR_DISABLED - This component is disabled by user.

  ERR_PARAM_MODE - Invalid operation mode.

  ERR_BUSY - Job is running and the driver can't detect job end by itself. The approximate end of the job can be detected by method GetDriverState. The actual transmission/reception finishes later. It depends on component settings (data width, timing, size of HW buffer, etc.).

3.22 GetDriverState

This method returns the current driver status.

Prototype

```
LDD_TDriverState GetDriverState(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.

Return value

- *Return value:LDD_TDriverState* - The current driver status mask.

  Following status masks defined in PE_Types.h can be used to check the current driver status.

  PE_LDD_DRIVER_DISABLED_IN_CLOCK_CONFIGURATION - 1 - Driver is disabled in the current mode; 0 - Driver is enabled in the current mode.

  PE_LDD_DRIVER_DISABLED_BY_USER - 1 - Driver is disabled by the user; 0 - Driver is enabled by the user.

  PE_LDD_DRIVER_BUSY - 1 - Driver is the BUSY state; 0 - Driver is in the IDLE state.

3.23 ConnectPin

This method reconnects the requested pin associated with the selected peripheral in this component. This method is only available for CPU derivatives and peripherals that support the runtime pin sharing with other internal on-chip peripherals.

Prototype

```
LDD_TError ConnectPin(LDD_TDeviceData *DeviceDataPtr, LDD_TPinMask PinMask)
```

Parameters

- *DeviceDataPtr: Pointer to LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *PinMask:LDD_TPinMask* - Mask for the requested pins. The peripheral pins are reconnected according to this mask.

  Possible parameters:

  LDD_SPIMASTER_INPUT_PIN - Input pin

  LDD_SPIMASTER_OUTPUT_PIN - Output pin

  LDD_SPIMASTER_CLK_PIN - Clock pin

  LDD_SPIMASTER_CS_X_PIN - Chip select pin

  LDD_SPIMASTER_CSS_PIN - Chip select strobe pin (only if supported by hardware)

Return value

- *Return value:LDD_TError* - Error code, possible values:

  ERR_OK - OK

  ERR_SPEED - The device doesn't work in the active clock configuration

  ERR_PARAM_MASK - Invalid pin mask

4 Events

This section describes component's events. Events are call-back functions called when an important event occurs. For more general information on events, please see the Processor Expert user manual.

4.1 OnBlockSent

This event is called after the last character from the output buffer is moved to the transmitter. This event is available only if the SendBlock method is enabled.

Prototype

```
void OnBlockSent(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. The pointer is passed as the parameter of Init method.

4.2 OnBlockReceived

This event is called when the requested number of data is moved to the input buffer. This method is available only if the ReceiveBlock method is enabled.

Prototype

```
void OnBlockReceived(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. The pointer is passed as the parameter of Init method.

4.3 OnError

This event is called when a channel error (not the error returned by a given method) occurs. The errors can be read using GetError method. This event is enabled when SPI device supports error detect.

Prototype

```
void OnError(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. The pointer is passed as the parameter of Init method.

5 Types and Constants

This section contains definitions of user types and constants. User types are derived from basic types and they are designed for usage in the driver interface. They are declared in the generated code.

Type Definitions

- **LDD_SPIMASTER_TError** = uint32_t Error flags.

- ***ComponentName*_TDMATransferId** = enum { enum } DMA transfer identification

- **LDD_SPIMASTER_TStats** = struct { Communication statistics

  uint32_t RxChars; – *Number of received characters*

  uint32_t TxChars; – *Number of transmitted characters*

  uint32_t RxParityErrors; – *Number of receiver parity errors, which have occured*

  uint32_t RxOverruns; – *Number of receiver overruns, which have occured*

  }

Constants

- **LDD_SPIMASTER_INPUT_PIN** - Input pin mask
- **LDD_SPIMASTER_OUTPUT_PIN** - Output pin mask
- **LDD_SPIMASTER_CLK_PIN** - Clock pin mask
- **LDD_SPIMASTER_CS_0_PIN** - Chip select 0 pin mask
- **LDD_SPIMASTER_CS_1_PIN** - Chip select 1 pin mask
- **LDD_SPIMASTER_CS_2_PIN** - Chip select 2 pin mask
- **LDD_SPIMASTER_CS_3_PIN** - Chip select 3 pin mask
- **LDD_SPIMASTER_CS_4_PIN** - Chip select 4 pin mask
- **LDD_SPIMASTER_CS_5_PIN** - Chip select 5 pin mask
- **LDD_SPIMASTER_CS_6_PIN** - Chip select 6 pin mask
- **LDD_SPIMASTER_CS_7_PIN** - Chip select 7 pin mask
- **LDD_SPIMASTER_CSS_PIN** - Chip select strobe pin mask
- **LDD_SPIMASTER_ON_BLOCK_RECEIVED** - OnBlockReceived event mask

- **LDD_SPIMASTER_ON_BLOCK_SENT** - OnBlockSent event mask
- **LDD_SPIMASTER_ON_ERROR** - OnError event mask
- **LDD_SPIMASTER_RX_OVERFLOW** - Receiver overflow
- **LDD_SPIMASTER_PARITY_ERROR** - Parity error
- **LDD_SPIMASTER_RX_DMA_ERROR** - Receive DMA channel error
- **LDD_SPIMASTER_TX_DMA_ERROR** - Transmit DMA channel error

6 Typical usage

This section contains examples of a typical usage of the component in user code. For general information please see the section Component Code Typical Usage in Processor Expert user manual.

Examples of typical settings and usage of the component

- Block reception/transmission, with interrupt service
- Block reception/transmission, without interrupt service (polling)

Block reception/transmission, with interrupt service

The most of applications use communication in the interrupt mode when an application is asynchronously notified by a driver about transmission/reception events.

The following example demonstrates reception/transmission from/to a slave that is single on the bus (input, output, clock pins). The program receive and transmit 4 characters.

Required component setup:

- *Interrupt service/event*: Enabled
- *Input pin*: Enabled
- *Output pin*: Enabled
- *Attribute set list*: 1
- *Attribute set 0*: configured as slave requires
- *Width*: 8 bits
- *Initial attribute set*: 0
- *Enabled in init. code*: yes
- *Event mask/OnBlockReceived*: Enabled
- *Event mask/OnError*: Enabled
- Methods: SendBlock, ReceiveBlock, GetError
- Events: OnBlockReceived, OnError

Content of ProcessorExpert.c:

```
#define BLOCK_SIZE 4

uint8_t OutData[BLOCK_SIZE] = "0123";
uint8_t InpData[BLOCK_SIZE];
volatile bool DataReceivedFlag = FALSE;
volatile LDD_SPIMASTER_TError ComError = 0U;
LDD_TError Error;
LDD_TDeviceData *MySPIPtr;

void main(void)
{
  ...
  MySPIPtr = SM1_Init(NULL);                               /* Initialization of SM1 component */
  Error = SM1_ReceiveBlock(MySPIPtr, InpData, BLOCK_SIZE); /* Request data block reception */
  Error = SM1_SendBlock(MySPIPtr, OutData, BLOCK_SIZE);    /* Start transmission/reception */
  while (!DataReceivedFlag) {};                            /* Wait until data block is transmitted/
received */
}
```

Content of Event.c:

```
extern volatile bool DataReceivedFlag;
extern volatile LDD_SPIMASTER_TError ComError;
extern LDD_TError Error;
extern LDD_TDeviceData *MySPIPtr;

void SM1_OnBlockReceived(LDD_TUserData *UserDataPtr)
{
  DataReceivedFlag = TRUE;                                 /* Set Data received flag */
}

void SM1_OnError(LDD_TUserData *UserDataPtr)
{
  Error = SM1_GetError(MySPIPtr, (LDD_SPIMASTER_TError *)&ComError); /* Get communication errors if
occured */
}
```

Block reception/transmission, without interrupt service (polling)

The simplest mode of the component is setting with Interrupt service/event disabled (so called polling mode). The driver doesn't use the interrupts in this mode, but provides events capability like in the interrupt mode. Main method of the component simulates the interrupt driven behavior.

The following example demonstrates transmission of data blocks if two slaves are connected to the bus (input, output, clock pins). The program receive and transmit 4 characters with each slave.

Required component setup:

- *Interrupt service/event*: Disabled
- *Input pin*: Enabled
- *Output pin*: Enabled
- *Attribute set list*: 2
- *Attribute set 0 and 1*: configured as slaves require
- *Width*: 8 bits
- *Enabled in init. code*: yes
- *Event mask/OnError*: Enabled
- Methods: SendBlock, ReceiveBlock, GetBlockReceivedStatus, GetError
- Events: OnError

Content of ProcessorExpert.c:

```
#define BLOCK_SIZE 4

uint8_t OutData[BLOCK_SIZE] = "0123";
uint8_t InpData[BLOCK_SIZE];
volatile LDD_SPIMASTER_TError ComError = 0U;
LDD_TError Error;
LDD_TDeviceData *MySPIPtr;

void main(void)
{
  ...
  MySPIPtr = SM1_Init(NULL);                              /* Initialization of SM1 component */
  SM1_SelectConfiguration(MySPIPtr, 1U, 1U);              /* Select chip select 1 and attribute set 1 */
  Error = SM1_ReceiveBlock(MySPIPtr, InpData, BLOCK_SIZE); /* Request data block reception */
  Error = SM1_SendBlock(MySPIPtr, OutData, BLOCK_SIZE);    /* Start transmission/reception */
  while (!SM1_GetBlockReceivedStatus(MySPIPtr)) {          /* Wait until data block is transmitted/
received */
    SM1_Main(MySPIPtr);
  }
  SM1_SelectConfiguration(MySPIPtr, 0U, 0U);              /* Select chip select 0 and attribute set 0 */
  Error = SM1_ReceiveBlock(MySPIPtr, InpData, BLOCK_SIZE); /* Request data block reception */
  Error = SM1_SendBlock(MySPIPtr, OutData, BLOCK_SIZE);    /* Start transmission/reception */
  while (!SM1_GetBlockReceivedStatus(MySPIPtr)) {          /* Wait until data block is transmitted/
received */
    SM1_Main(MySPIPtr);
  }
}
```

Content of Event.c:

```
extern volatile LDD_SPIMASTER_TError ComError;
extern LDD_TError Error;
extern LDD_TDeviceData *MySPIPtr;

void SM1_OnError(LDD_TUserData *UserDataPtr)
{
  Error = SM1_GetError(MySPIPtr, (LDD_SPIMASTER_TError *)&ComError); /* Get communication errors if
occured */
}
```

Document Number N/A
Revision 1, 12/2013