

Serial_LDD

Embedded Component User Guide

Contents

1	General Information.....	1
2	Properties.....	2
3	Methods.....	6
4	Events.....	19
5	Types and Constants.....	21
6	Typical usage.....	23

1 General Information

Component Level: Logical Device Driver

Category: Logical Device Drivers-Communication

This component encapsulates serial communication interface (UART) and provides methods and events for asynchronous serial communication.

The component provides standard features of asynchronous serial communications (if supported by hardware) and also special functions provided by the hardware like:

- Number of data bits/number of stop bits
- Baud rate
- Parity
- Handshake flow control

The component can work in two basic modes:

- **Polling mode** - no events available, no block transfer functions available
- **Interrupt mode** - user can choose from a set of events for data transfer management

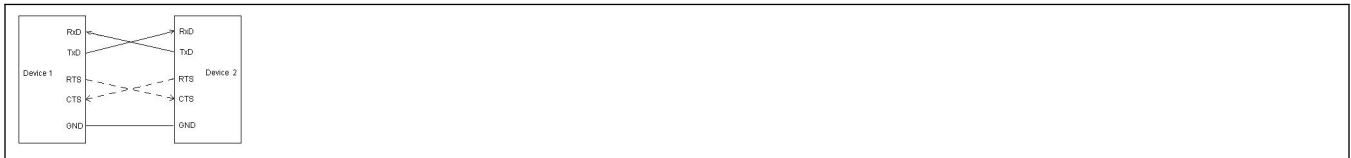
Properties

The component operates upon specified communication buffers to complete block transfer operations. The driver of the component stores all the incoming data to the receive buffer and automatically transmits all the data from the transmit buffer.

The component also offers possibility to control the data transfer with standard RTS/CTS handshake mechanism when supported by HW:

- **CTS** - an input pin. When in low level the device is allowed the send data. When in high level the device has to stop transmitting.
- **RTS** - an output pin. When the device is able to receive data the RTS is set to low level. If the device can't accept new data the RTS is set to high level.

The following picture shows an example of a connection between two DCE devices (RS232C interface) with a possible handshake (dashed line):



The generated driver contains initialization code and selectable methods for the component runtime API generated according to component configuration.

From code sharing perspective, the component driver code can operate in the following modes:

- **Shared mode** - One driver controls all UART channels by one peripheral driver.
- **Distinct mode** - There is generated a separate code for each UART channel.

Please note that capabilities of the component may depend on the capabilities of currently selected peripheral.

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** - The channel used for serial asynchronous communication.
- **Interrupt service/event** - The 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"):

- **Interrupt RxD** - Interrupt from serial receiver.
- **Interrupt RxD priority** - The priority of the interrupt associated with the asynchronous communication.
- *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.

- **Interrupt TxD** - Interrupt from serial transmitter.
- **Interrupt TxD priority** - Priority of the interrupt associated with the asynchronous communication.
- *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.
- **Interrupt Error** - Error interrupt from serial receiver.
- **Interrupt Error priority** - Priority of the interrupt associated with the asynchronous communication.
- *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 setting.
 - **Data width** - Number of data (information) bits.

Properties

- **Parity** - The type of the parity bit (none, hardware odd, hardware even, even (hw or sw), odd (hw or sw)). If odd [even] is selected and the number of "ones" (bits) in the transmitted byte is odd, then the parity bit is 0 [1]. If none is selected, there is no parity bit. The parity bit can be implemented by hardware (if it is supported by the device) or by software.
- **Stop bits** - Number of stop bits (signaling the end of data transmission).
- **Loop mode** - Selects a channel mode.
- **Baud rate** - Communication baud rate. It is necessary to enter both a value and an unit (see Timing Setting Syntax). The setting may be made with the help of the Timing dialog box that opens when clicking on  button.
- **Wakeup condition** - Selects the condition of wake up the SCI.

There are 2 options:

- Address mark wakeup: a 1 in MSB position of received character wakes the receiver.
- Idle line wakeup: An idle character on RxD pin wakes the receiver.
- **Stop in wait mode** - Determines if the component is stopped in wait mode.
- **Idle line mode** - Idle line mode determines when the counter of idle character starts (after a start bit or stop bit).

There are 2 options:

- Starts after start bit: Idle character bit count starts after start bit.
- Starts after stop bit: Idle character bit count starts after stop bit.
- **Transmitter output** - Reverses the polarity of transmitted data.
- **Receiver input** - Reverses the polarity of received data.
- **Break generation length** - The length of generated break characters. The actual length is determined by the selected data width (e.g. value '10/11 bits' means that for 8bit data the break length is 10bits and for 9bit data the break length is 11bits).
- **Receiver** - Enable/Disable the receiver.

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

- **RxD** - Input pin used for the communication.
- **RxD pin signal** - Signal name of RxD pin.
- **Transmitter** - Enable/Disable the transmitter.

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

- **TxD** - Output pin used for the communication.
- **TxD pin signal** - Signal name of TxD pin.

- **Flow control** - Communication flow control.

There are 2 modes:

- None - No flow control. There are no items in this mode.
- Hardware (RTS/CTS) - Hardware supported RTS/CTS flow control. The following items are displayed in this mode:
 - **CTS** - Clear to send handshake. To use this feature the Transmitter must be enabled. **DMA mode:** If DMA controller is available on the selected CPU and the transmitter is configured to use DMA controller then this property is disabled. CTS can't be used in this mode.

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

- **CTS Pin** - Clear-to-send pin.
- **CTS pin signal** - Signal name of CTS pin.
- *Settings only if SW Handshake is supported for CPU*
 - **Interrupt CTS priority** - Priority of the interrupt associated with the asynchronous communication.
 - **Interrupt CTS** - Interrupt from the CTS.
- **RTS** - Ready to send handshake. To use this feature the Receiver must be enabled. **DMA mode:** If DMA controller is available on the selected CPU and the receiver is configured to use DMA controller then this property is disabled. RTS can't be used in this mode.

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

- **RTS Pin** - Ready-to-send pin
- **RTS pin signal** - Signal name of RTS pin
- **RTS enabled** - If an input buffer is enabled and a SW handshake is used then the given number denotes a number of received characters in the buffer when the signal RTS is activated. If a SW handshake is used then the maximum efficiency can be achieved by using the input buffer of a greater size than the specified number for **RTS enabled**. This item is not accessible if the input buffer is disabled.
- **Initialization** - Initial settings (after power-on or reset).
 - **Enabled in init. code** - The component is enabled after power-on or reset (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 initialization code.
 - **OnBlockReceived** - Specifies if OnReceived event is enabled in initialization code.
 - **OnTxComplete** - Specifies if OnTxComplete event is enabled in initialization code.
 - **OnError** - Specifies if OnError event is enabled in initialization code.
 - **OnBreak** - Specifies if OnBreak event is enabled in initialization 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.

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

Deinitializes the device. 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

Enables the device, starts the transmitting and receiving.

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 component does not work in the active clock configuration.

3.4 Disable

Disables the device, stops the transmitting and receiving.

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 component does not work in the active clock configuration.

3.5 SendBlock

Sends a block of characters. The method returns ERR_BUSY when the previous block transmission is not completed. Method CancelBlockTransmission can be used to cancel a transmit operation. This method is available only if the transmitter property is enabled.

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 a buffer from where data will be sent.
- *Size*: uint16_t - Number of characters in the buffer.

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_PARAM_SIZE - Parameter Size is out of expected range.
ERR_DISABLED - The component or device is disabled.
ERR_BUSY - The previous transmit request is pending.

3.6 ReceiveBlock

Specifies the number of data to receive. The method returns ERR_BUSY until the specified number of characters is received. 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* - Number of characters to receive

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_PARAM_SIZE - Parameter Size is out of expected range.
ERR_DISABLED - The component or device is disabled.
ERR_BUSY - The previous receive request is pending.

3.7 CancelBlockTransmission

Immediately cancels the running transmit process started by method SendBlock. Characters already stored in the transmit shift register will be sent.

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 - The component does not work in the active clock configuration.
ERR_DISABLED - The component or device is disabled.

3.8 CancelBlockReception

Immediately cancels the running receive process started by method ReceiveBlock. Characters already stored in the HW FIFO will be lost.

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 - The component does not work in the active clock configuration.
ERR_DISABLED - The component or device is disabled.

3.9 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.

Prototype

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

Parameters

- *DeviceDataPtr*: Pointer to *LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *ErrorPtr*: Pointer to *LDD_SERIAL_TError* - A pointer to the returned set of error flags:
LDD_SERIAL_RX_OVERRUN - Receiver overrun.
LDD_SERIAL_PARITY_ERROR - Parity error (only if HW supports parity feature).
LDD_SERIAL_FRAMING_ERROR - Framing error.
LDD_SERIAL_NOISE_ERROR - Noise error.

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

3.10 GetSentDataNum

Returns the number of sent characters.

Prototype

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

Parameters

Methods

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

Return value

- *Return value: uint16_t* - The number of sent characters.

3.11 GetReceivedDataNum

Returns the number of received characters in the receive buffer.

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* - Number of received characters in the receive buffer.

3.12 GetTxCompleteStatus

Returns whether the transmitter has transmitted all characters and there are no other characters in the transmitter's HW FIFO or the shift register. 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 a peripheral supports this feature.

Prototype

```
bool GetTxCompleteStatus(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

Return value

- *Return value: bool* - Possible values:
 - true - Data block is completely transmitted.
 - false - Data block isn't completely transmitted.

3.13 SetEventMask

Enables/Disables events. This method is available if the interrupt service/event property is enabled and at least one event is enabled.

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* - Mask of events to enable.

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.

3.14 GetEventMask

Returns current event mask. This method is available if the interrupt service/event property is enabled and at least one event is enabled.

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* - The current event mask. The component event masks are defined in the *PE_Types.h* file.

3.15 SelectBaudRate

This method changes the channel communication speed (baud rate). This method is enabled only if the user specifies a list of possible period settings at design time (see Timing dialog box - Runtime setting - from a list of values). Each of these settings constitutes a *mode* and Processor Expert assigns them a *mode identifier*. The prescaler and compare values corresponding to each mode are calculated in design time. The user may switch modes at runtime by referring to a mode identifier. No run-time calculations are performed, all the calculations are performed at design time.

Prototype

```
LDD_TError SelectBaudRate(LDD_TDeviceData *DeviceDataPtr, LDD_SERIAL_TBaudMode Mode)
```

Parameters

- *DeviceDataPtr*: Pointer to *LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *Mode*: *LDD_SERIAL_TBaudMode* - Timing mode to set

Note: Special constant is generated in the components header file for each mode from the list of values.

This constant can be directly passed to the parameter. Format of the constant is:

Methods

<BeanName>_BM_<Timing> e.g. "as1_BM_9600BAUD" for baud rate set to 9600 baud and component name "as1". See header file of the generated code for details.

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_MODE - Invalid ID of the baud rate mode.

3.16 GetSelectedBaudRate

Returns the current selected baud rate ID. This method is enabled only if the user specifies a list of possible period settings in design time (see Timing dialog box - Runtime setting - from a list of values).

Prototype

```
LDD_SERIAL_TBaudMode GetSelectedBaudRate(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

Return value

- *Return value:LDD_SERIAL_TBaudMode* - The current selected baud rate ID.

3.17 SetParity

Sets the parity type.

Prototype

```
LDD_TError SetParity(LDD_TDeviceData *DeviceDataPtr, LDD_SERIAL_TParity Parity)
```

Parameters

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

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_PARITY - Invalid parity.

3.18 GetParity

Returns the parity type currently set.

Prototype

```
LDD_SERIAL_TParity GetParity(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

Return value

- *Return value*: *LDD_SERIAL_TParity* - The parity type currently set.

3.19 SetDataWidth

Sets the number of bits per character.

Prototype

```
LDD_TError SetDataWidth(LDD_TDeviceData *DeviceDataPtr, LDD_SERIAL_TDataWidth DataWidth)
```

Parameters

- *DeviceDataPtr*: Pointer to *LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *DataWidth*: *LDD_SERIAL_TDataWidth* - Character bit length

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_WIDTH - Invalid data width.

3.20 GetDataWidth

Returns the current number of bits per character.

Prototype

```
LDD_SERIAL_TDataWidth GetDataWidth(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

Return value

Methods

- *Return value:* `LDD_SERIAL_TDataWidth` - The current number of bits per character.

3.21 SetStopBitLength

Sets the stop bit length.

Prototype

```
LDD_TError SetStopBitLength(LDD_TDeviceData *DeviceDataPtr, LDD_SERIAL_TStopBitLen StopBitLen)
```

Parameters

- *DeviceDataPtr:* Pointer to `LDD_TDeviceData` - Device data structure pointer returned by Init method.
- *StopBitLen:* `LDD_SERIAL_TStopBitLen` - Stop bit length

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_LENGTH - Invalid stop bit length.

3.22 GetStopBitLength

Returns the current stop bit length.

Prototype

```
LDD_SERIAL_TStopBitLen GetStopBitLength(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

Return value

- *Return value:* `LDD_SERIAL_TStopBitLen` - The current stop bit length.

3.23 SetLoopMode

Sets the loop mode operation.

Prototype

```
LDD_TError SetLoopMode(LDD_TDeviceData *DeviceDataPtr, LDD_SERIAL_TLoopMode LoopMode)
```

Parameters

- *DeviceDataPtr*: Pointer to *LDD_TDeviceData* - Device data structure pointer returned by Init method.
- *LoopMode*: *LDD_SERIAL_TLoopMode* - Requested loop mode.

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_MODE - Invalid loop mode.

3.24 GetLoopMode

Returns the loop mode currently set.

Prototype

```
LDD_SERIAL_TLoopMode GetLoopMode(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

Return value

- *Return value*: *LDD_SERIAL_TLoopMode* - The current loop mode.

3.25 GetStats

Returns the driver's receive/transmit statistic information.

Prototype

```
LDD_SERIAL_TStats GetStats(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

Return value

- *Return value*: *LDD_SERIAL_TStats* - Driver's receive/transmit statistic structure.

3.26 ClearStats

Clears the driver's statistic information. This method is available only if the GetStats method is enabled.

Prototype

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

Methods

Parameters

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

3.27 SendBreak

Sends the break character. This method is available only if the Transmitter property is enabled.

Prototype

```
LDD_TError SendBreak(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 component does not work in the active clock configuration.
 - ERR_DISABLED - This component is disabled by user
 - ERR_BUSY - The previous transmit request is pending.

3.28 GetBreak

Tests the internal input break flag, returns it (whether the break has occurred or not) and clears it. This method is available only if the property Receiver is enabled.

Prototype

```
bool GetBreak(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

Return value

- *Return value:bool* - Possible return values:
 - true - The break character was detected.
 - false - The break character was not detected.

3.29 TurnTxOn

Turns on the transmitter. This method is available only if the transmitter property is enabled.

Prototype


```
void TurnTxOn(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

3.30 TurnTxOff

Turns off the transmitter. This method is available only if the transmitter property is enabled.

Prototype

```
void TurnTxOff(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

3.31 TurnRxOn

Turns on the receiver. This method is available only if the receiver property is enabled.

Prototype

```
void TurnRxOn(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

3.32 TurnRxOff

Turns off the receiver. This method is available only if the receiver property is enabled.

Prototype

```
void TurnRxOff(LDD_TDeviceData *DeviceDataPtr)
```

Parameters

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

3.33 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

Methods

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

Possible parameters:

LDD_SERIAL_RX_PIN - Receiver pin mask

LDD_SERIAL_TX_PIN - Transmitter pin mask

LDD_SERIAL_CTS_PIN - CTS pin mask

LDD_SERIAL_RTS_PIN - RTS pin mask

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_PARAM_MASK* - Invalid pin mask parameter.

3.34 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.

3.35 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.

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 component accepted request to change the operation mode.
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.

3.36 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.

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 OnBlockReceived

This event is called when the requested number of data is moved to the input buffer.

Prototype

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

Parameters

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

4.2 OnBlockSent

This event is called after the last character from the output buffer is moved to the transmitter.

Prototype

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

Parameters

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

4.3 OnBreak

This event is called when a break occurs on the input channel.

The event is available only when both Interrupt service/event and Break signal properties are enabled.

Prototype

```
void OnBreak(LDD_TUserData *UserDataPtr)
```

Parameters

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

4.4 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.

The event is available only when the Interrupt service/event property is enabled.

Prototype

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

Parameters

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

4.5 OnTxComplete

This event indicates that the transmitter is finished transmitting all data, preamble, and break characters and is idle. It can be used to determine when it is safe to switch a line driver (e.g. in RS-485 applications).

The event is available only when both Interrupt service/event and Transmitter properties are enabled.

Prototype

```
void OnTxComplete(LDD_TUserData *UserDataPtr)
```

Parameters

- *UserDataPtr:LDD_TUserData* - Pointer to the user or RTOS specific data. This 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

- **ComponentName_TComData** = word User type for communication. Size of this type depends on the communication data width.

- **LDD_SERIAL_TError** : user definition

Error flags.

- **LDD_SERIAL_TSize** : user definition

Buffer size or data length

- **LDD_SERIAL_TParity** = enum { PARITY_UNDEF, PARITY_NONE, PARITY_ODD, PARITY_EVEN, PARITY_MARK, PARITY_SPACE } Parity type

PARITY_UNDEF – Undefined parity type

PARITY_NONE – No parity

PARITY_ODD – Odd parity

PARITY_EVEN – Even parity

PARITY_MARK – Force parity to high

PARITY_SPACE – Force parity to low

- **LDD_SERIAL_TDataWidth** = word Bit length

- **LDD_SERIAL_TStopBitLen** = enum { STOP_BIT_LEN_UNDEF, STOP_BIT_LEN_1, STOP_BIT_LEN_1_5, STOP_BIT_LEN_2 } Stop-bit length

STOP_BIT_LEN_UNDEF – Undefined stop-bit length

STOP_BIT_LEN_1 – Stop-bit length is equal to 1 bit

STOP_BIT_LEN_1_5 – Stop-bit length is equal to 1.5 bit

STOP_BIT_LEN_2 – Stop-bit length is equal to 2 bit

Types and Constants

- **LDD_SERIAL_TBaudMode** = byte Baud rate mode identifier
- **LDD_SERIAL_TStats** = struct { Communication statistics
uint32_t ReceivedChars; – *Number of received characters*
uint32_t SentChars; – *Number of transmitted characters*
uint32_t ReceivedBreaks; – *Number of received break characters*
uint32_t ParityErrors; – *Number of receiver parity errors*
uint32_t FramingErrors; – *Number of receiver framing errors*
uint32_t OverrunErrors; – *Number of receiver overrun errors*
uint32_t NoiseErrors; – *Number of receiver noise errors*
}
- **LDD_SERIAL_TLoopMode** = enum { LOOPMODE_UNDEF, LOOPMODE_NORMAL, LOOPMODE_AUTO_ECHO, LOOPMODE_LOCAL_LOOPBACK, LOOPMODE_REMOTE_LOOPBACK }
Looping modes
LOOPMODE_UNDEF – Undefined looping mode
LOOPMODE_NORMAL – Normal mode (without looping)
LOOPMODE_AUTO_ECHO – Automatic echo mode. Automatically resends received data bit by bit. Transmitter is disabled.
LOOPMODE_LOCAL_LOOPBACK – Local loopback mode. Transmitter output is internally connected to receiver input.
LOOPMODE_REMOTE_LOOPBACK – Remote loopback mode. Receiver automatically resends data bit by bit. Receiver and transmitter are disabled.

Constants

- **ComponentName_BM_<Timing>** - This notation describes constants used to specify several distinctive modes of timing defined in the Baud rate property. The Runtime setting of the Baud rate property must be set to "from list of values".
The <Timing> in the constant name denotes a value of the baudrate, e.g. 9600BAUD. These constants are used as a parameter of the SelectBaudRate method.
- **LDD_SERIAL_RX_PIN** - Receiver pin mask.
- **LDD_SERIAL_TX_PIN** - Transmitter pin mask.
- **LDD_SERIAL_CTS_PIN** - CTS pin mask.
- **LDD_SERIAL_RTS_PIN** - RTS pin mask.
- **LDD_SERIAL_ON_BLOCK_RECEIVED** - OnBlockReceived event mask.
- **LDD_SERIAL_ON_BLOCK_SENT** - OnBlockSent event mask.

- **LDD_SERIAL_ON_BREAK** - OnBreak event mask.
- **LDD_SERIAL_ON_TXCOMPLETE** - OnTxComplete event mask.
- **LDD_SERIAL_ON_ERROR** - OnError event mask.

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 Serial_LDD component

1. Block reception/transmission, with interrupt service
2. Block reception/transmission, without interrupt service (polling)

Block reception/transmission, with interrupt service

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

The following example demonstrates a simple "Hello world" application. The program waits until 'e' character is received and then sends text "Hello world".

Required component setup :

- *Interrupt service/event* : Enabled
- *DataWidth* : 8 bits
- *Receiver* : Enabled
- *Transmitter* : Enabled
- *Enabled in init. code* : yes
- *Methods to enable* : SendBlock, ReceiveBlock
- *Events to enable* : OnBlockReceived

Content of ProcessorExpert.c:

```
volatile bool DataReceivedFlg = FALSE;
char OutData[] = "Hello world";
char InpData[10];
LDD_TError Error;
LDD_TDeviceData *MySerialPtr;

void main(void)
{
    . . .
    MySerialPtr = AS1_Init(NULL); /* Initialization of AS1 component */
    for(;;) {
        Error = AS1_ReceiveBlock(MySerialPtr, InpData, 1U); /* Start reception of one character */
    } /* Wait until 'e' character is received */
    while (!DataReceivedFlg) {
    }
    if (InpData[0] == 'e') {
        Error = AS1_SendBlock(MySerialPtr, OutData, sizeof(OutData)); /* Send block of characters */
    }
}
```

Typical usage

```
    }
    DataReceivedFlg = FALSE;
}
}
```

Content of Event.c:

```
extern volatile bool DataReceivedFlg;

void AS1_OnBlockReceived(LDD_TUserData *UserDataPtr)
{
    DataReceivedFlg = TRUE; /* Set DataReceivedFlg flag */
}
```

Block reception/transmission, without interrupt service (polling)

The simplest mode of the Serial 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 a simple "Hello world" application. The program waits until 'e' character is received and then sends text "Hello world".

Required component setup :

- *Interrupt service/event* : Disabled
- *DataWidth* : 8 bits
- *Receiver* : Enabled
- *Transmitter* : Enabled
- *Enabled in init. code* : yes
- *Methods to enable* : SendBlock, ReceiveBlock
- *Events to enable* : OnBlockReceived

Content of ProcessorExpert.c:

```
volatile bool DataReceivedFlg = FALSE;
char OutData[] = "Hello world";
char InpData[10];
LDD_TError Error;
LDD_TDeviceData *MySerialPtr;

void main(void)
{
    . . .
    MySerialPtr = AS1_Init(NULL); /* Initialization of AS1 component */
    for(;;) {
        Error = AS1_ReceiveBlock(MySerialPtr, InpData, 1U); /* Start reception of one character */
    /*
    while (!DataReceivedFlg) { /* Wait until 'e' character is
received */
        AS1_Main(MySerialPtr);
    }
    if (InpData[0] == 'e') {
        Error = AS1_SendBlock(MySerialPtr, OutData, sizeof(OutData)); /* Send block of characters */
    }
    DataReceivedFlg = FALSE;
}
}
```

Content of Event.c:

```
extern volatile bool DataReceivedFlg;
```



```
void AS1_OnBlockReceived(LDD_TUserData *UserDataPtr)
{
    DataReceivedFlg = TRUE; /* Set DataReceivedFlg flag */
}
```

How to Reach Us:

Home Page:

freescale.com

Web Support:

freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, Altivec, C-5, CodeTest, CodeWarrior, ColdFire, ColdFire+, C-Ware, Energy Efficient Solutions logo, Kinetis, mobileGT, PowerQUICC, Processor Expert, QorIQ, Qorivva, StarCore, Symphony, and VortiQa are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Airfast, BeeKit, BeeStack, CoreNet, Flexis, Layerscape, MagniV, MXC, Platform in a Package, QorIQ Qonverge, QUICC Engine, Ready Play, SafeAssure, SafeAssure logo, SMARTMOS, Tower, TurboLink, Vybrid, and Xtrinsic are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© 2013 Freescale Semiconductor, Inc.

Document Number N/A
Revision 1, 12/2013

