SMART ARM-based Microcontrollers

AT03256: SAM D/R/L/C Serial USART (SERCOM USART) 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 USART mode to transfer or receive USART 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 D10/D11
- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM DA1
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information
- Examples
- API Overview

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

To use the USART you need to have a GCLK generator enabled and running that can be used as the SERCOM clock source. This can either be configured in conf_clocks.h or by using the system clock driver.

3. Module Overview

This driver will use one (or more) SERCOM interface(s) in the system and configure it to run as a USART interface in either synchronous or asynchronous mode.

3.1. Driver Feature Macro Definition

Driver Feature Macro	Supported devices
FEATURE_USART_SYNC_SCHEME_V2	SAM D21/R21/D09/D10/D11/L21/L22/DA1/C20/C21
FEATURE_USART_OVER_SAMPLE	SAM D21/R21/D09/D10/D11/L21/L22/DA1/C20/C21
FEATURE_USART_HARDWARE_FLOW_CONTROL	SAM D21/R21/D09/D10/D11/L21/L22/DA1/C20/C21
FEATURE_USART_IRDA	SAM D21/R21/D09/D10/D11/L21/L22/DA1/C20/C21
FEATURE_USART_LIN_SLAVE	SAM D21/R21/D09/D10/D11/L21/L22/DA1/C20/C21
FEATURE_USART_COLLISION_DECTION	SAM D21/R21/D09/D10/D11/L21/L22/DA1/C20/C21
FEATURE_USART_START_FRAME_DECTION	SAM D21/R21/D09/D10/D11/L21/L22/DA1/C20/C21
FEATURE_USART_IMMEDIATE_BUFFER_OVERFLOW_NOTIFICATION	SAM D21/R21/D09/D10/D11/L21/L22/DA1/C20/C21
FEATURE_USART_RS485	SAM C20/C21
FEATURE_USART_LIN_MASTER	SAM L22/C20/C21

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

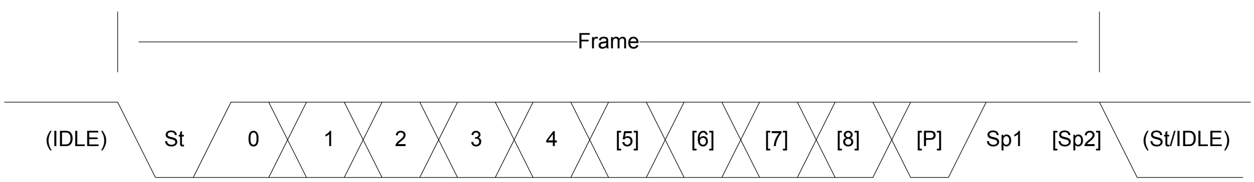
3.2. Frame Format

Communication is based on frames, where the frame format can be customized to accommodate a wide range of standards. A frame consists of a start bit, a number of data bits, an optional parity bit for error detection as well as a configurable length stop bit(s) - see Figure 3-1 USART Frame Overview on page 7. Table 3-1 USART Frame Parameters on page 7 shows the available parameters you can change in a frame.

Table 3-1 USART Frame Parameters

Parameter	Options
Start bit	1
Data bits	5, 6, 7, 8, 9
Parity bit	None, Even, Odd
Stop bits	1, 2

Figure 3-1 USART Frame Overview



3.3. Synchronous Mode

In synchronous mode a dedicated clock line is provided; either by the USART itself if in master mode, or by an external master if in slave mode. Maximum transmission speed is the same as the GCLK clocking the USART peripheral when in slave mode, and the GCLK divided by two if in master mode. In synchronous mode the interface needs three lines to communicate:

- TX (Transmit pin)
- RX (Receive pin)
- XCK (Clock pin)

3.3.1. Data Sampling

In synchronous mode the data is sampled on either the rising or falling edge of the clock signal. This is configured by setting the clock polarity in the configuration struct.

3.4. Asynchronous Mode

In asynchronous mode no dedicated clock line is used, and the communication is based on matching the clock speed on the transmitter and receiver. The clock is generated from the internal SERCOM baudrate generator, and the frames are synchronized by using the frame start bits. Maximum transmission speed is limited to the SERCOM GCLK divided by 16. In asynchronous mode the interface only needs two lines to communicate:

- TX (Transmit pin)

- RX (Receive pin)

3.4.1. Transmitter/receiver Clock Matching

For successful transmit and receive using the asynchronous mode the receiver and transmitter clocks needs to be closely matched. When receiving a frame that does not match the selected baudrate closely enough the receiver will be unable to synchronize the frame(s), and garbage transmissions will result.

3.5. Parity

Parity can be enabled to detect if a transmission was in error. This is done by counting the number of "1" bits in the frame. When using even parity the parity bit will be set if the total number of "1"s in the frame are an even number. If using odd parity the parity bit will be set if the total number of "1"s are odd.

When receiving a character the receiver will count the number of "1"s in the frame and give an error if the received frame and parity bit disagree.

3.6. GPIO Configuration

The SERCOM module has four internal pads; the RX pin can be placed freely on any one of the four pads, and the TX and XCK pins have two predefined positions that can be selected as a pair. The pads can then be routed to an external GPIO pin using the normal pin multiplexing scheme on the SAM.

4. Special Considerations

Never execute large portions of code in the callbacks. These are run from the interrupt routine, and thus having long callbacks will keep the processor in the interrupt handler for an equally long time. A common way to handle this is to use global flags signaling the main application that an interrupt event has happened, and only do the minimal needed processing in the callback.

5. Extra Information

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

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

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

7. API Overview

7.1. Variable and Type Definitions

7.1.1. Type usart_callback_t

```
typedef void(* usart_callback_t )(struct usart_module *const module)
```

Type of the callback functions.

7.2. Structure Definitions

7.2.1. Struct iso7816_config_t

ISO7816 configuration structure.

Table 7-1 Members

Type	Name	Description
bool	enable_inverse	Enable inverse transmission and reception
bool	enabled	
enum iso7816_guard_time	guard_time	Guard time, which lasts two bit times
enum iso7816_inhibit_nack	inhibit_nack	Inhibit Non Acknowledge: • 0: the NACK is generated; • 1: the NACK is not generated.
uint32_t	max_iterations	
enum iso7816_protocol_type	protocol_t	ISO7816 protocol type
enum iso7816_successive_recv_nack	successive_recv_nack	Disable successive NACKs. • 0: NACK is sent on the ISO line as soon as a parity error occurs in the received character. Successive parity errors are counted up to the value in the max_iterations field. These parity errors generate a NACK on the ISO line. As soon as this value is reached, no additional NACK is sent on the ISO line. The ITERATION flag is asserted.

7.2.2. Struct usart_config

Configuration options for USART.

Table 7-2 Members

Type	Name	Description
uint32_t	baudrate	USART baudrate
enum usart_character_size	character_size	USART character size
bool	clock_polarity_inverted	USART Clock Polarity. If true, data changes on falling XCK edge and is sampled at rising edge. If false, data changes on rising XCK edge and is sampled at falling edge.
bool	collision_detection_enable	Enable collision dection
enum usart_dataorder	data_order	USART bit order (MSB or LSB first)
bool	encoding_format_enable	Enable IrDA encoding format
uint32_t	ext_clock_freq	External clock frequency in synchronous mode. This must be set if `use_external_clock` is true.
enum gclk_generator	generator_source	GCLK generator source
bool	immediate_buffer_overflow_notification	Controls when the buffer overflow status bit is asserted when a buffer overflow occurs
struct iso7816_config_t	iso7816_config	Enable ISO7816 for smart card interfacing
enum lin_master_break_length	lin_break_length	LIN Master Break Length
enum lin_master_header_delay	lin_header_delay	LIN master header delay
enum lin_node_type	lin_node	LIN node type
bool	lin_slave_enable	Enable LIN Slave Support
enum usart_signal_mux_settings	mux_setting	USART pin out
enum usart_parity	parity	USART parity

Type	Name	Description
uint32_t	pinmux_pad0	PAD0 pinmux. If current USARTx has several alternative multiplexing I/O pins for PAD0, then only one peripheral multiplexing I/O can be enabled for current USARTx PAD0 function. Make sure that no other alternative multiplexing I/O is associated with the same USARTx PAD0.
uint32_t	pinmux_pad1	PAD1 pinmux. If current USARTx has several alternative multiplexing I/O pins for PAD1, then only one peripheral multiplexing I/O can be enabled for current USARTx PAD1 function. Make sure that no other alternative multiplexing I/O is associated with the same USARTx PAD1.
uint32_t	pinmux_pad2	PAD2 pinmux. If current USARTx has several alternative multiplexing I/O pins for PAD2, then only one peripheral multiplexing I/O can be enabled for current USARTx PAD2 function. Make sure that no other alternative multiplexing I/O is associated with the same USARTx PAD2.
uint32_t	pinmux_pad3	PAD3 pinmux. If current USARTx has several alternative multiplexing I/O pins for PAD3, then only one peripheral multiplexing I/O can be enabled for current USARTx PAD3 function. Make sure that no other alternative multiplexing I/O is associated with the same USARTx PAD3.
uint8_t	receive_pulse_length	The minimum pulse length required for a pulse to be accepted by the IrDA receiver

Type	Name	Description
bool	receiver_enable	Enable receiver
enum rs485_guard_time	rs485_guard_time	RS485 guard time
bool	run_in_standby	If true the USART will be kept running in Standby sleep mode
enum usart_sample_adjustment	sample_adjustment	USART sample adjustment
enum usart_sample_rate	sample_rate	USART sample rate
bool	start_frame_detection_enable	Enable start of frame dection
enum usart_stopbits	stopbits	Number of stop bits
enum usart_transfer_mode	transfer_mode	USART in asynchronous or synchronous mode
bool	transmitter_enable	Enable transmitter
bool	use_external_clock	States whether to use the external clock applied to the XCK pin. In synchronous mode the shift register will act directly on the XCK clock. In asynchronous mode the XCK will be the input to the USART hardware module.

7.2.3. Struct usart_module

SERCOM USART 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.

7.3. Macro Definitions

7.3.1. Driver Feature Definition

Define SERCOM USART features set according to different device family.

7.3.1.1. Macro FEATURE_USART_SYNC_SCHEME_V2

```
#define FEATURE_USART_SYNC_SCHEME_V2
```

USART sync scheme version 2.

7.3.1.2. Macro FEATURE_USART_OVER_SAMPLE

```
#define FEATURE_USART_OVER_SAMPLE
```

USART oversampling.

7.3.1.3. Macro FEATURE_USART_HARDWARE_FLOW_CONTROL

```
#define FEATURE_USART_HARDWARE_FLOW_CONTROL
```

USART hardware control flow.

7.3.1.4. Macro FEATURE_USART_IRDA

```
#define FEATURE_USART_IRDA
```

IrDA mode.

7.3.1.5. Macro FEATURE_USART_LIN_SLAVE

```
#define FEATURE_USART_LIN_SLAVE
```

LIN slave mode.

7.3.1.6. Macro FEATURE_USART_COLLISION_DECTION

```
#define FEATURE_USART_COLLISION_DECTION
```

USART collision detection.

7.3.1.7. Macro FEATURE_USART_START_FRAME_DECTION

```
#define FEATURE_USART_START_FRAME_DECTION
```

USART start frame detection.

7.3.1.8. Macro FEATURE_USART_IMMEDIATE_BUFFER_OVERFLOW_NOTIFICATION

```
#define FEATURE_USART_IMMEDIATE_BUFFER_OVERFLOW_NOTIFICATION
```

USART start buffer overflow notification.

7.3.1.9. Macro FEATURE_USART_ISO7816

```
#define FEATURE_USART_ISO7816
```

ISO7816 for smart card interfacing.

7.3.1.10. Macro FEATURE_USART_LIN_MASTER

```
#define FEATURE_USART_LIN_MASTER
```

LIN master mode.

7.3.1.11. Macro FEATURE_USART_RS485

```
#define FEATURE_USART_RS485
```

RS485 mode.

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 USART_TIMEOUT

```
#define USART_TIMEOUT
```

USART timeout value

7.4. Function Definitions

7.4.1. Lock/Unlock

7.4.1.1. Function usart_lock()

Attempt to get lock on driver instance.

```
enum status_code usart_lock(
        struct usart_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 7-3 Parameters

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

Table 7-4 Return Values

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

7.4.1.2. Function usart_unlock()

Unlock driver instance.

```
void usart_unlock(
        struct usart_module *const module)
```

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

Table 7-5 Parameters

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

7.4.2. Writing and Reading

7.4.2.1. Function usart_write_wait()

Transmit a character via the USART.

```
enum status_code usart_write_wait(
        struct usart_module *const module,
        const uint16_t tx_data)
```

This blocking function will transmit a single character via the USART.

Table 7-6 Parameters

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

Returns

Status of the operation.

Table 7-7 Return Values

Return value	Description
STATUS_OK	If the operation was completed
STATUS_BUSY	If the operation was not completed, due to the USART module being busy
STATUS_ERR_DENIED	If the transmitter is not enabled

7.4.2.2. Function usart_read_wait()

Receive a character via the USART.

```
enum status_code usart_read_wait(
        struct usart_module *const module,
        uint16_t *const rx_data)
```

This blocking function will receive a character via the USART.

Table 7-8 Parameters

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

Returns
Status of the operation.

Table 7-9 Return Values

Return value	Description
STATUS_OK	If the operation was completed
STATUS_BUSY	If the operation was not completed, due to the USART module being busy
STATUS_ERR_BAD_FORMAT	If the operation was not completed, due to configuration mismatch between USART and the sender
STATUS_ERR_BAD_OVERFLOW	If the operation was not completed, due to the baudrate being too low or the system frequency being too high
STATUS_ERR_BAD_DATA	If the operation was not completed, due to data being corrupted
STATUS_ERR_DENIED	If the receiver is not enabled

7.4.2.3. Function usart_write_buffer_wait()

Transmit a buffer of characters via the USART.

```
enum status_code usart_write_buffer_wait(
        struct usart_module *const module,
        const uint8_t * tx_data,
        uint16_t length)
```

This blocking function will transmit a block of `length` characters via the USART.

Note: Using this function in combination with the interrupt (`_job`) functions is not recommended as it has no functionality to check if there is an ongoing interrupt driven operation running or not.

Table 7-10 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	tx_data	Pointer to data to transmit
[in]	length	Number of characters to transmit

Note: If using 9-bit data, the array that *tx_data point to should be defined as uint16_t array and should be casted to uint8_t* pointer. Because it is an address pointer, the highest byte is not discarded. For example:

```
#define TX_LEN 3
uint16_t tx_buf[TX_LEN] = {0x0111, 0x0022, 0x0133};
usart_write_buffer_wait(&module, (uint8_t*)tx_buf, TX_LEN);
```

Returns
Status of the operation.

Table 7-11 Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_ERR_INVALID_ARG	If operation was not completed, due to invalid arguments
STATUS_ERR_TIMEOUT	If operation was not completed, due to USART module timing out
STATUS_ERR_DENIED	If the transmitter is not enabled

7.4.2.4. Function usart_read_buffer_wait()

Receive a buffer of length characters via the USART.

```
enum status_code usart_read_buffer_wait(
        struct usart_module *const module,
        uint8_t * rx_data,
        uint16_t length)
```

This blocking function will receive a block of `length` characters via the USART.

Note: Using this function in combination with the interrupt (`*_job`) functions is not recommended as it has no functionality to check if there is an ongoing interrupt driven operation running or not.

Table 7-12 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[out]	rx_data	Pointer to receive buffer
[in]	length	Number of characters to receive

Note: If using 9-bit data, the array that *rx_data point to should be defined as uint16_t array and should be casted to uint8_t* pointer. Because it is an address pointer, the highest byte is not discarded. For example:

```
#define RX_LEN 3
uint16_t rx_buf[RX_LEN] = {0x0,};
usart_read_buffer_wait(&module, (uint8_t*)rx_buf, RX_LEN);
```

Returns
Status of the operation.

Table 7-13 Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_ERR_INVALID_ARG	If operation was not completed, due to an invalid argument being supplied
STATUS_ERR_TIMEOUT	If operation was not completed, due to USART module timing out

Return value	Description
STATUS_ERR_BAD_FORMAT	If the operation was not completed, due to a configuration mismatch between USART and the sender
STATUS_ERR_BAD_OVERFLOW	If the operation was not completed, due to the baudrate being too low or the system frequency being too high
STATUS_ERR_BAD_DATA	If the operation was not completed, due to data being corrupted
STATUS_ERR_DENIED	If the receiver is not enabled

7.4.3. Enabling/Disabling Receiver and Transmitter

7.4.3.1. Function usart_enable_transceiver()

Enable Transceiver.

```
void usart_enable_transceiver(
        struct usart_module *const module,
        enum usart_transceiver_type transceiver_type)
```

Enable the given transceiver. Either RX or TX.

Table 7-14 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	transceiver_type	Transceiver type

7.4.3.2. Function usart_disable_transceiver()

Disable Transceiver.

```
void usart_disable_transceiver(
        struct usart_module *const module,
        enum usart_transceiver_type transceiver_type)
```

Disable the given transceiver (RX or TX).

Table 7-15 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	transceiver_type	Transceiver type

7.4.4. LIN Master Command and Status

7.4.4.1. Function lin_master_send_cmd()

Sending LIN command.

```
void lin_master_send_cmd(
        struct usart_module *const module,
        enum lin_master_cmd cmd)
```

Sending LIN command.

Table 7-16 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	cmd	Cammand type

7.4.4.2. Function lin_master_transmission_status()

Get LIN transmission status.

```
bool lin_master_transmission_status(
        struct usart_module *const module)
```

Get LIN transmission status.

Table 7-17 Parameters

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

Returns
Status of LIN master transmission.

Table 7-18 Return Values

Return value	Description
true	Data transmission completed
false	Transmission is ongoing

7.4.5. Callback Management

7.4.5.1. Function usart_register_callback()

Registers a callback.

```
void usart_register_callback(
        struct usart_module *const module,
        usart_callback_t callback_func,
        enum usart_callback callback_type)
```

Registers a callback function, which is implemented by the user.

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

Table 7-19 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

7.4.5.2. Function usart_unregister_callback()

Unregisters a callback.

```
void usart_unregister_callback(
        struct usart_module * module,
        enum usart_callback callback_type)
```

Unregisters a callback function, which is implemented by the user.

Table 7-20 Parameters

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

7.4.5.3. Function usart_enable_callback()

Enables callback.

```
void usart_enable_callback(
        struct usart_module *const module,
        enum usart_callback callback_type)
```

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

Table 7-21 Parameters

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

7.4.5.4. Function usart_disable_callback()

Disable callback.

```
void usart_disable_callback(
        struct usart_module *const module,
        enum usart_callback callback_type)
```

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

Table 7-22 Parameters

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

7.4.6. Writing and Reading

7.4.6.1. Function usart_write_job()

Asynchronous write a single char.

```
enum status_code usart_write_job(
        struct usart_module *const module,
        const uint16_t * tx_data)
```

Sets up the driver to write the data given. If registered and enabled, a callback function will be called when the transmit is completed.

Table 7-23 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	tx_data	Data to transfer

Returns
Status of the operation.

Table 7-24 Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_BUSY	If operation was not completed, due to the USART module being busy
STATUS_ERR_DENIED	If the transmitter is not enabled

7.4.6.2. Function usart_read_job()

Asynchronous read a single char.

```
enum status_code usart_read_job(
        struct usart_module *const module,
        uint16_t *const rx_data)
```

Sets up the driver to read data from the USART module to the data pointer given. If registered and enabled, a callback will be called when the receiving is completed.

Table 7-25 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[out]	rx_data	Pointer to where received data should be put

Returns
Status of the operation.

Table 7-26 Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_BUSY	If operation was not completed

7.4.6.3. Function usart_write_buffer_job()

Asynchronous buffer write.

```
enum status_code usart_write_buffer_job(
        struct usart_module *const module,
        uint8_t * tx_data,
        uint16_t length)
```

Sets up the driver to write a given buffer over the USART. If registered and enabled, a callback function will be called.

Table 7-27 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	tx_data	Pointer do data buffer to transmit
[in]	length	Length of the data to transmit

Note: If using 9-bit data, the array that *tx_data point to should be defined as uint16_t array and should be casted to uint8_t* pointer. Because it is an address pointer, the highest byte is not discarded. For example:

```
#define TX_LEN 3
uint16_t tx_buf[TX_LEN] = {0x0111, 0x0022, 0x0133};
usart_write_buffer_job(&module, (uint8_t*)tx_buf, TX_LEN);
```

Returns
Status of the operation.

Table 7-28 Return Values

Return value	Description
STATUS_OK	If operation was completed successfully.
STATUS_BUSY	If operation was not completed, due to the USART module being busy
STATUS_ERR_INVALID_ARG	If operation was not completed, due to invalid arguments
STATUS_ERR_DENIED	If the transmitter is not enabled

7.4.6.4. Function usart_read_buffer_job()

Asynchronous buffer read.

```
enum status_code usart_read_buffer_job(
        struct usart_module *const module,
        uint8_t * rx_data,
        uint16_t length)
```

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

Table 7-29 Parameters

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

Note: If using 9-bit data, the array that *rx_data point to should be defined as uint16_t array and should be casted to uint8_t* pointer. Because it is an address pointer, the highest byte is not discarded. For example:

```
#define RX_LEN 3
uint16_t rx_buf[RX_LEN] = {0x0,};
usart_read_buffer_job(&module, (uint8_t*)rx_buf, RX_LEN);
```

Returns

Status of the operation.

Table 7-30 Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_BUSY	If operation was not completed, due to the USART module being busy
STATUS_ERR_INVALID_ARG	If operation was not completed, due to invalid arguments
STATUS_ERR_DENIED	If the transmitter is not enabled

7.4.6.5. Function usart_abort_job()

Cancels ongoing read/write operation.

```
void usart_abort_job(
        struct usart_module *const module,
        enum usart_transceiver_type transceiver_type)
```

Cancels the ongoing read/write operation modifying parameters in the USART software struct.

Table 7-31 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	transceiver_type	Transfer type to cancel

7.4.6.6. Function usart_get_job_status()

Get status from the ongoing or last asynchronous transfer operation.

```
enum status_code usart_get_job_status(
        struct usart_module *const module,
        enum usart_transceiver_type transceiver_type)
```

Returns the error from a given ongoing or last asynchronous transfer operation. Either from a read or write transfer.

Table 7-32 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to USART software instance struct
[in]	transceiver_type	Transfer type to check

Returns
Status of the given job.

Table 7-33 Return Values

Return value	Description
STATUS_OK	No error occurred during the last transfer
STATUS_BUSY	A transfer is ongoing
STATUS_ERR_BAD_DATA	The last operation was aborted due to a parity error. The transfer could be affected by external noise
STATUS_ERR_BAD_FORMAT	The last operation was aborted due to a frame error
STATUS_ERR_OVERFLOW	The last operation was aborted due to a buffer overflow
STATUS_ERR_INVALID_ARG	An invalid transceiver enum given

7.4.7. Function usart_disable()

Disable module.

```
void usart_disable(
        const struct usart_module *const module)
```

Disables the USART module.

Table 7-34 Parameters

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

7.4.8. Function usart_enable()

Enable the module.

```
void usart_enable(
        const struct usart_module *const module)
```

Enables the USART module.

Table 7-35 Parameters

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

7.4.9. Function usart_get_config_defaults()

Initializes the device to predefined defaults.

```
void usart_get_config_defaults(
        struct usart_config *const config)
```

Initialize the USART device to predefined defaults:

- 8-bit asynchronous USART
- No parity
- One stop bit
- 9600 baud
- Transmitter enabled
- Receiver enabled
- GCLK generator 0 as clock source
- Default pin configuration

The configuration struct will be updated with the default configuration.

Table 7-36 Parameters

Data direction	Parameter name	Description
[in, out]	config	Pointer to configuration struct

7.4.10. Function usart_init()

Initializes the device.

```
enum status_code usart_init(
        struct usart_module *const module,
        Sercom *const hw,
        const struct usart_config *const config)
```

Initializes the USART device based on the setting specified in the configuration struct.

Table 7-37 Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to USART device
[in]	hw	Pointer to USART hardware instance
[in]	config	Pointer to configuration struct

Returns
Status of the initialization.

Table 7-38 Return Values

Return value	Description
STATUS_OK	The initialization was successful
STATUS_BUSY	The USART module is busy resetting
STATUS_ERR_DENIED	The USART has not been disabled in advance of initialization
STATUS_ERR_INVALID_ARG	The configuration struct contains invalid configuration
STATUS_ERR_ALREADY_INITIALIZED	The SERCOM instance has already been initialized with different clock configuration
STATUS_ERR_BAUD_UNAVAILABLE	The BAUD rate given by the configuration struct cannot be reached with the current clock configuration

7.4.11. Function usart_is_syncing()

Check if peripheral is busy syncing registers across clock domains.

```
bool usart_is_syncing(
        const struct usart_module *const module)
```

Return peripheral synchronization status. If doing a non-blocking implementation this function can be used to check the sync state and hold of any new actions until sync is complete. If this function is not run; the functions will block until the sync has completed.

Table 7-39 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to peripheral module

Returns

Peripheral sync status.

Table 7-40 Return Values

Return value	Description
true	Peripheral is busy syncing
false	Peripheral is not busy syncing and can be read/written without stalling the bus

7.4.12. Function usart_reset()

Resets the USART module.

```
void usart_reset(
        const struct usart_module *const module)
```

Disables and resets the USART module.

Table 7-41 Parameters

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

7.5. Enumeration Definitions

7.5.1. Enum iso7816_guard_time

The value of ISO7816 guard time.

Table 7-42 Members

Enum value	Description
ISO7816_GUARD_TIME_2_BIT	The guard time is 2-bit times
ISO7816_GUARD_TIME_3_BIT	The guard time is 3-bit times
ISO7816_GUARD_TIME_4_BIT	The guard time is 4-bit times
ISO7816_GUARD_TIME_5_BIT	The guard time is 5-bit times
ISO7816_GUARD_TIME_6_BIT	The guard time is 6-bit times
ISO7816_GUARD_TIME_7_BIT	The guard time is 7-bit times

7.5.2. Enum iso7816_inhibit_nack

The value of ISO7816 receive NACK inhibit.

Table 7-43 Members

Enum value	Description
ISO7816_INHIBIT_NACK_DISABLE	The NACK is generated
ISO7816_INHIBIT_NACK_ENABLE	The NACK is not generated

7.5.3. Enum iso7816_protocol_type

ISO7816 protocol type.

Table 7-44 Members

Enum value	Description
ISO7816_PROTOCOL_T_0	ISO7816 protocol type 0
ISO7816_PROTOCOL_T_1	ISO7816 protocol type 1

7.5.4. Enum iso7816_successive_recv_nack

The value of ISO7816 disable successive receive NACK.

Table 7-45 Members

Enum value	Description
ISO7816_SUCCESSIVE_RECV_NACK_DISABLE	The successive receive NACK is enable.
ISO7816_SUCCESSIVE_RECV_NACK_ENABLE	The successive receive NACK is disable.

7.5.5. Enum lin_master_break_length

Length of the break field transmitted when in LIN master mode

Table 7-46 Members

Enum value	Description
LIN_MASTER_BREAK_LENGTH_13_BIT	Break field transmission is 13 bit times
LIN_MASTER_BREAK_LENGTH_17_BIT	Break field transmission is 17 bit times
LIN_MASTER_BREAK_LENGTH_21_BIT	Break field transmission is 21 bit times
LIN_MASTER_BREAK_LENGTH_26_BIT	Break field transmission is 26 bit times

7.5.6. Enum lin_master_cmd

LIN master command enum.

Table 7-47 Members

Enum value	Description
LIN_MASTER_SOFTWARE_CONTROL_TRANSMIT_CMD	LIN master software control transmission command
LIN_MASTER_AUTO_TRANSMIT_CMD	LIN master automatically transmission command

7.5.7. Enum lin_master_header_delay

LIN master header delay between break and sync transmission, and between the sync and identifier (ID) fields. This field is only valid when using automatically transmission command

Table 7-48 Members

Enum value	Description
LIN_MASTER_HEADER_DELAY_0	Delay between break and sync transmission is 1 bit time. Delay between sync and ID transmission is 1 bit time.
LIN_MASTER_HEADER_DELAY_1	Delay between break and sync transmission is 4 bit time. Delay between sync and ID transmission is 4 bit time.
LIN_MASTER_HEADER_DELAY_2	Delay between break and sync transmission is 8 bit time. Delay between sync and ID transmission is 4 bit time.
LIN_MASTER_HEADER_DELAY_3	Delay between break and sync transmission is 14 bit time. Delay between sync and ID transmission is 4 bit time.

7.5.8. Enum lin_node_type

LIN node type.

Table 7-49 Members

Enum value	Description
LIN_MASTER_NODE	LIN master mode
LIN_SLAVE_NODE	LIN slave mode
LIN_INVALID_MODE	Neither LIN master nor LIN slave mode

7.5.9. Enum rs485_guard_time

The value of RS485 guard time.

Table 7-50 Members

Enum value	Description
RS485_GUARD_TIME_0_BIT	The guard time is 0-bit time
RS485_GUARD_TIME_1_BIT	The guard time is 1-bit time
RS485_GUARD_TIME_2_BIT	The guard time is 2-bit times

Enum value	Description
RS485_GUARD_TIME_3_BIT	The guard time is 3-bit times
RS485_GUARD_TIME_4_BIT	The guard time is 4-bit times
RS485_GUARD_TIME_5_BIT	The guard time is 5-bit times
RS485_GUARD_TIME_6_BIT	The guard time is 6-bit times
RS485_GUARD_TIME_7_BIT	The guard time is 7-bit times

7.5.10. Enum usart_callback

Callbacks for the Asynchronous USART driver.

Table 7-51 Members

Enum value	Description
USART_CALLBACK_BUFFER_TRANSMITTED	Callback for buffer transmitted
USART_CALLBACK_BUFFER_RECEIVED	Callback for buffer received
USART_CALLBACK_ERROR	Callback for error
USART_CALLBACK_BREAK_RECEIVED	Callback for break character is received
USART_CALLBACK_CTS_INPUT_CHANGE	Callback for a change is detected on the CTS pin
USART_CALLBACK_START_RECEIVED	Callback for a start condition is detected on the RxD line

7.5.11. Enum usart_character_size

Number of bits for the character sent in a frame.

Table 7-52 Members

Enum value	Description
USART_CHARACTER_SIZE_5BIT	The char being sent in a frame is five bits long
USART_CHARACTER_SIZE_6BIT	The char being sent in a frame is six bits long
USART_CHARACTER_SIZE_7BIT	The char being sent in a frame is seven bits long
USART_CHARACTER_SIZE_8BIT	The char being sent in a frame is eight bits long
USART_CHARACTER_SIZE_9BIT	The char being sent in a frame is nine bits long

7.5.12. Enum usart_dataorder

The data order decides which MSB or LSB is shifted out first when data is transferred.

Table 7-53 Members

Enum value	Description
USART_DATAORDER_MSB	The MSB will be shifted out first during transmission, and shifted in first during reception
USART_DATAORDER_LSB	The LSB will be shifted out first during transmission, and shifted in first during reception

7.5.13. Enum usart_parity

Select parity USART parity mode.

Table 7-54 Members

Enum value	Description
USART_PARITY_ODD	For odd parity checking, the parity bit will be set if number of ones being transferred is even
USART_PARITY_EVEN	For even parity checking, the parity bit will be set if number of ones being received is odd
USART_PARITY_NONE	No parity checking will be executed, and there will be no parity bit in the received frame

7.5.14. Enum usart_sample_adjustment

The value of sample number used for majority voting.

Table 7-55 Members

Enum value	Description
USART_SAMPLE_ADJUSTMENT_7_8_9	The first, middle and last sample number used for majority voting is 7-8-9
USART_SAMPLE_ADJUSTMENT_9_10_11	The first, middle and last sample number used for majority voting is 9-10-11
USART_SAMPLE_ADJUSTMENT_11_12_13	The first, middle and last sample number used for majority voting is 11-12-13
USART_SAMPLE_ADJUSTMENT_13_14_15	The first, middle and last sample number used for majority voting is 13-14-15

7.5.15. Enum usart_sample_rate

The value of sample rate and baudrate generation mode.

Table 7-56 Members

Enum value	Description
USART_SAMPLE_RATE_16X_ARITHMETIC	16x over-sampling using arithmetic baudrate generation
USART_SAMPLE_RATE_16X_FRACTIONAL	16x over-sampling using fractional baudrate generation
USART_SAMPLE_RATE_8X_ARITHMETIC	8x over-sampling using arithmetic baudrate generation
USART_SAMPLE_RATE_8X_FRACTIONAL	8x over-sampling using fractional baudrate generation
USART_SAMPLE_RATE_3X_ARITHMETIC	3x over-sampling using arithmetic baudrate generation

7.5.16. Enum usart_signal_mux_settings

Set the functionality of the SERCOM pins.

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

Table 7-57 Members

Enum value	Description
USART_RX_0_TX_0_XCK_1	MUX setting RX_0_TX_0_XCK_1
USART_RX_0_TX_2_XCK_3	MUX setting RX_0_TX_2_XCK_3
USART_RX_0_TX_0_RTS_2_CTS_3	MUX setting USART_RX_0_TX_0_RTS_2_CTS_3
USART_RX_1_TX_0_XCK_1	MUX setting RX_1_TX_0_XCK_1
USART_RX_1_TX_2_XCK_3	MUX setting RX_1_TX_2_XCK_3
USART_RX_1_TX_0_RTS_2_CTS_3	MUX setting USART_RX_1_TX_0_RTS_2_CTS_3
USART_RX_2_TX_0_XCK_1	MUX setting RX_2_TX_0_XCK_1
USART_RX_2_TX_2_XCK_3	MUX setting RX_2_TX_2_XCK_3
USART_RX_2_TX_0_RTS_2_CTS_3	MUX setting USART_RX_2_TX_0_RTS_2_CTS_3
USART_RX_3_TX_0_XCK_1	MUX setting RX_3_TX_0_XCK_1
USART_RX_3_TX_2_XCK_3	MUX setting RX_3_TX_2_XCK_3
USART_RX_3_TX_0_RTS_2_CTS_3	MUX setting USART_RX_3_TX_0_RTS_2_CTS_3
USART_RX_0_TX_0_XCK_1_TE_2	MUX setting USART_RX_0_TX_0_XCK_1_TE_2
USART_RX_1_TX_0_XCK_1_TE_2	MUX setting USART_RX_1_TX_0_XCK_1_TE_2
USART_RX_2_TX_0_XCK_1_TE_2	MUX setting USART_RX_2_TX_0_XCK_1_TE_2
USART_RX_3_TX_0_XCK_1_TE_2	MUX setting USART_RX_3_TX_0_XCK_1_TE_2

7.5.17. Enum usart_stopbits

Number of stop bits for a frame.

Table 7-58 Members

Enum value	Description
USART_STOPBITS_1	Each transferred frame contains one stop bit
USART_STOPBITS_2	Each transferred frame contains two stop bits

7.5.18. Enum usart_transceiver_type

Select Receiver or Transmitter.

Table 7-59 Members

Enum value	Description
USART_TRANSCEIVER_RX	The parameter is for the Receiver
USART_TRANSCEIVER_TX	The parameter is for the Transmitter

7.5.19. Enum usart_transfer_mode

Select USART transfer mode.

Table 7-60 Members

Enum value	Description
USART_TRANSFER_SYNCHRONOUSLY	Transfer of data is done synchronously
USART_TRANSFER_ASYNCHRONOUSLY	Transfer of data is done asynchronously

8. Extra Information for SERCOM USART 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
USART	Universal Synchronous and Asynchronous Serial Receiver and Transmitter
LSB	Least Significant Bit
MSB	Most Significant Bit
DMA	Direct Memory Access

8.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver
- System clock configuration

8.3. Errata

There are no errata related to this 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 feature as below: • ISO7816
Added new features as below: • LIN master • RS485

Changelog
Added new features as below: • Oversample • Buffer overflow notification • Irda • Lin slave • Start frame detection • Hardware flow control • Collision detection • DMA support
• Added new `transmitter_enable` and `receiver_enable` Boolean values to `struct` usart_config • Altered `usart_write_*` and usart_read_* functions to abort with an error code if the relevant transceiver is not enabled • Fixed `usart_write_buffer_wait()` and `usart_read_buffer_wait()` not aborting correctly when a timeout condition occurs
Initial Release

9. Examples for SERCOM USART Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM Serial USART (SERCOM USART) 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 USART - Basic
- Quick Start Guide for SERCOM USART - Callback
- Quick Start Guide for Using DMA with SERCOM USART
- Quick Start Guide for SERCOM USART LIN

9.1. Quick Start Guide for SERCOM USART - Basic

This quick start will echo back characters typed into the terminal. In this use case the USART will be configured with the following settings:

- Asynchronous mode
- 9600 Baudrate
- 8-bits, No Parity and one Stop Bit
- TX and RX enabled and connected to the Xplained Pro Embedded Debugger virtual COM port

9.1.1. Setup

9.1.1.1. Prerequisites

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

9.1.1.2. Code

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

```
struct usart_module usart_instance;
```

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

```
void configure_usart(void)
{
    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);

    config_usart.baudrate    = 9600;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;

    while (usart_init(&usart_instance,
            EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
    }

    usart_enable(&usart_instance);
}
```

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

```
configure_usart();
```

9.1.1.3. Workflow

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

   ```
   struct usart_module usart_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. Configure the USART module.
   1. Create a USART module configuration struct, which can be filled out to adjust the configuration of a physical USART peripheral.

      ```
      struct usart_config config_usart;
      ```
   2. Initialize the USART configuration struct with the module's default values.

      ```
      usart_get_config_defaults(&config_usart);
      ```

      Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
   3. Alter the USART settings to configure the physical pinout, baudrate, and other relevant parameters.

      ```
      config_usart.baudrate    = 9600;
      config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
      config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
      config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
      config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
      config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
      ```
   4. Configure the USART module with the desired settings, retrying while the driver is busy until the configuration is stressfully set.

      ```
      while (usart_init(&usart_instance,
              EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
      }
      ```
   5. Enable the USART module.

      ```
      usart_enable(&usart_instance);
      ```

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:

```
uint8_t string[] = "Hello World!\r\n";
usart_write_buffer_wait(&usart_instance, string, sizeof(string));

uint16_t temp;

while (true) {
    if (usart_read_wait(&usart_instance, &temp) == STATUS_OK) {
        while (usart_write_wait(&usart_instance, temp) != STATUS_OK) {
        }
```

```
    }
}
```

9.1.2.2. Workflow

1. Send a string to the USART to show the demo is running, blocking until all characters have been sent.

   ```
   uint8_t string[] = "Hello World!\r\n";
   usart_write_buffer_wait(&usart_instance, string, sizeof(string));
   ```

2. Enter an infinite loop to continuously echo received values on the USART.

   ```
   while (true) {
       if (usart_read_wait(&usart_instance, &temp) == STATUS_OK) {
           while (usart_write_wait(&usart_instance, temp) != STATUS_OK) {
           }
       }
   }
   ```

3. Perform a blocking read of the USART, storing the received character into the previously declared temporary variable.

   ```
   if (usart_read_wait(&usart_instance, &temp) == STATUS_OK) {
   ```

4. Echo the received variable back to the USART via a blocking write.

   ```
   while (usart_write_wait(&usart_instance, temp) != STATUS_OK) {
   }
   ```

9.2. Quick Start Guide for SERCOM USART - Callback

This quick start will echo back characters typed into the terminal, using asynchronous TX and RX callbacks from the USART peripheral. In this use case the USART will be configured with the following settings:

- Asynchronous mode
- 9600 Baudrate
- 8-bits, No Parity and one Stop Bit
- TX and RX enabled and connected to the Xplained Pro Embedded Debugger virtual COM port

9.2.1. Setup

9.2.1.1. Prerequisites

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

9.2.1.2. Code

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

```
struct usart_module usart_instance;
```

```
#define MAX_RX_BUFFER_LENGTH   5

volatile uint8_t rx_buffer[MAX_RX_BUFFER_LENGTH];
```

Copy-paste the following callback function code to your user application:

```
void usart_read_callback(struct usart_module *const usart_module)
{
```

```
    usart_write_buffer_job(&usart_instance,
            (uint8_t *)rx_buffer, MAX_RX_BUFFER_LENGTH);
}

void usart_write_callback(struct usart_module *const usart_module)
{
    port_pin_toggle_output_level(LED_0_PIN);
}
```

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

```
void configure_usart(void)
{
    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);

    config_usart.baudrate    = 9600;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;

    while (usart_init(&usart_instance,
            EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
    }

    usart_enable(&usart_instance);
}

void configure_usart_callbacks(void)
{
    usart_register_callback(&usart_instance,
            usart_write_callback, USART_CALLBACK_BUFFER_TRANSMITTED);
    usart_register_callback(&usart_instance,
            usart_read_callback, USART_CALLBACK_BUFFER_RECEIVED);

    usart_enable_callback(&usart_instance,
USART_CALLBACK_BUFFER_TRANSMITTED);
    usart_enable_callback(&usart_instance, USART_CALLBACK_BUFFER_RECEIVED);
}
```

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

```
configure_usart();
configure_usart_callbacks();
```

9.2.1.3. Workflow

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

   ```
   struct usart_module usart_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. Configure the USART module.

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

   ```
   struct usart_config config_usart;
   ```

2. Initialize the USART configuration struct with the module's default values.

   ```
   usart_get_config_defaults(&config_usart);
   ```

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

3. Alter the USART settings to configure the physical pinout, baudrate, and other relevant parameters.

   ```
   config_usart.baudrate    = 9600;
   config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
   config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
   config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
   config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
   config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
   ```

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

   ```
   while (usart_init(&usart_instance,
           EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
   }
   ```

5. Enable the USART module.

   ```
   usart_enable(&usart_instance);
   ```

3. Configure the USART callbacks.

   1. Register the TX and RX callback functions with the driver.

      ```
      usart_register_callback(&usart_instance,
              usart_write_callback, USART_CALLBACK_BUFFER_TRANSMITTED);
      usart_register_callback(&usart_instance,
              usart_read_callback, USART_CALLBACK_BUFFER_RECEIVED);
      ```

   2. Enable the TX and RX callbacks so that they will be called by the driver when appropriate.

      ```
      usart_enable_callback(&usart_instance,
      USART_CALLBACK_BUFFER_TRANSMITTED);
      usart_enable_callback(&usart_instance,
      USART_CALLBACK_BUFFER_RECEIVED);
      ```

9.2.2. Use Case

9.2.2.1. Code

Copy-paste the following code to your user application:

```
system_interrupt_enable_global();

uint8_t string[] = "Hello World!\r\n";
usart_write_buffer_wait(&usart_instance, string, sizeof(string));

while (true) {
    usart_read_buffer_job(&usart_instance,
            (uint8_t *)rx_buffer, MAX_RX_BUFFER_LENGTH);
}
```

9.2.2.2. Workflow

1. Enable global interrupts, so that the callbacks can be fired.

```
system_interrupt_enable_global();
```

2. Send a string to the USART to show the demo is running, blocking until all characters have been sent.

```
uint8_t string[] = "Hello World!\r\n";
usart_write_buffer_wait(&usart_instance, string, sizeof(string));
```

3. Enter an infinite loop to continuously echo received values on the USART.

```
while (true) {
```

4. Perform an asynchronous read of the USART, which will fire the registered callback when characters are received.

```
usart_read_buffer_job(&usart_instance,
        (uint8_t *)rx_buffer, MAX_RX_BUFFER_LENGTH);
```

9.3. Quick Start Guide for Using DMA with SERCOM USART

The supported board list:

- SAM D21 Xplained Pro
- SAM R21 Xplained Pro
- SAM D11 Xplained Pro
- SAM DA1 Xplained Pro
- SAM L21 Xplained Pro
- SAM L22 Xplained Pro
- SAM C21 Xplained Pro

This quick start will receive eight bytes of data from the PC terminal and transmit back the string to the terminal through DMA. In this use case the USART will be configured with the following settings:

- Asynchronous mode
- 9600 Baudrate
- 8-bits, No Parity and one Stop Bit
- TX and RX enabled and connected to the Xplained Pro Embedded Debugger virtual COM port

9.3.1. Setup

9.3.1.1. Prerequisites

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

9.3.1.2. Code

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

```
struct usart_module usart_instance;
```

```
struct dma_resource usart_dma_resource_rx;
struct dma_resource usart_dma_resource_tx;
```

```
#define BUFFER_LEN    8
static uint16_t string[BUFFER_LEN];
```

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

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

```
static void transfer_done_rx(struct dma_resource* const resource )
{
    dma_start_transfer_job(&usart_dma_resource_tx);
}

static void transfer_done_tx(struct dma_resource* const resource )
{
    dma_start_transfer_job(&usart_dma_resource_rx);
}

static void configure_dma_resource_rx(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

    config.peripheral_trigger = EDBG_CDC_SERCOM_DMAC_ID_RX;
    config.trigger_action = DMA_TRIGGER_ACTON_BEAT;

    dma_allocate(resource, &config);
}

static void setup_transfer_descriptor_rx(DmacDescriptor *descriptor)
{
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.src_increment_enable = false;
    descriptor_config.block_transfer_count = BUFFER_LEN;
    descriptor_config.destination_address =
            (uint32_t)string + sizeof(string);
    descriptor_config.source_address =
            (uint32_t)(&usart_instance.hw->USART.DATA.reg);

    dma_descriptor_create(descriptor, &descriptor_config);
}

static void configure_dma_resource_tx(struct dma_resource *resource)
{
    struct dma_resource_config config;
```

```
    dma_get_config_defaults(&config);

    config.peripheral_trigger = EDBG_CDC_SERCOM_DMAC_ID_TX;
    config.trigger_action = DMA_TRIGGER_ACTON_BEAT;

    dma_allocate(resource, &config);
}

static void setup_transfer_descriptor_tx(DmacDescriptor *descriptor)
{
    struct dma_descriptor_config descriptor_config;

    dma_descriptor_get_config_defaults(&descriptor_config);

    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.dst_increment_enable = false;
    descriptor_config.block_transfer_count = BUFFER_LEN;
    descriptor_config.source_address = (uint32_t)string + sizeof(string);
    descriptor_config.destination_address =
        (uint32_t)(&usart_instance.hw->USART.DATA.reg);

    dma_descriptor_create(descriptor, &descriptor_config);
}

static void configure_usart(void)
{
    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);

    config_usart.baudrate    = 9600;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;

    while (usart_init(&usart_instance,
            EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
    }

    usart_enable(&usart_instance);
}
```

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

```
configure_usart();

configure_dma_resource_rx(&usart_dma_resource_rx);
configure_dma_resource_tx(&usart_dma_resource_tx);

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

dma_add_descriptor(&usart_dma_resource_rx, &example_descriptor_rx);
dma_add_descriptor(&usart_dma_resource_tx, &example_descriptor_tx);

dma_register_callback(&usart_dma_resource_rx, transfer_done_rx,
        DMA_CALLBACK_TRANSFER_DONE);
dma_register_callback(&usart_dma_resource_tx, transfer_done_tx,
        DMA_CALLBACK_TRANSFER_DONE);

dma_enable_callback(&usart_dma_resource_rx,
```

```
        DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&usart_dma_resource_tx,
        DMA_CALLBACK_TRANSFER_DONE);
```

9.3.1.3. Workflow

Create variables

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

   ```
   struct usart_module usart_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 module software instance structures for DMA resources to store the DMA resource state while it is in use.

   ```
   struct dma_resource usart_dma_resource_rx;
   struct dma_resource usart_dma_resource_tx;
   ```

   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 a buffer to store the data to be transferred /received.

   ```
   #define BUFFER_LEN    8
   static uint16_t string[BUFFER_LEN];
   ```
4. Create DMA transfer descriptors for RX/TX.

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

Configure the USART

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

   ```
   struct usart_config config_usart;
   ```
2. Initialize the USART configuration struct with the module's default values.

   ```
   usart_get_config_defaults(&config_usart);
   ```

   Note: This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
3. Alter the USART settings to configure the physical pinout, baudrate, and other relevant parameters.

   ```
   config_usart.baudrate    = 9600;
   config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
   config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
   config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
   config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
   config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
   ```

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

```
while (usart_init(&usart_instance,
        EDBG_CDC_MODULE, &config_usart) != STATUS_OK) {
}
```

5. Enable the USART module.

```
usart_enable(&usart_instance);
```

Configure DMA

1. Create a callback function of receiver done.

```
static void transfer_done_rx(struct dma_resource* const resource )
{
    dma_start_transfer_job(&usart_dma_resource_tx);
}
```

2. Create a callback function of transmission done.

```
static void transfer_done_tx(struct dma_resource* const resource )
{
    dma_start_transfer_job(&usart_dma_resource_rx);
}
```

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

```
struct dma_resource_config config;
```

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

```
dma_get_config_defaults(&config);
```

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

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

```
config.peripheral_trigger = EDBG_CDC_SERCOM_DMAC_ID_RX;
config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
```

6. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &config);
```

7. 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 descriptor_config;
```

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

```
dma_descriptor_get_config_defaults(&descriptor_config);
```

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

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

```
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.src_increment_enable = false;
```

```
descriptor_config.block_transfer_count = BUFFER_LEN;
descriptor_config.destination_address =
        (uint32_t)string + sizeof(string);
descriptor_config.source_address =
        (uint32_t)(&usart_instance.hw->USART.DATA.reg);
```

10. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

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

```
struct dma_resource_config config;
```

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

```
dma_get_config_defaults(&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 RX Ready trigger causes a beat transfer in this example.

```
config.peripheral_trigger = EDBG_CDC_SERCOM_DMAC_ID_TX;
config.trigger_action = DMA_TRIGGER_ACTON_BEAT;
```

14. Allocate a DMA resource with the configurations.

```
dma_allocate(resource, &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 descriptor_config;
```

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

```
dma_descriptor_get_config_defaults(&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.

```
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.dst_increment_enable = false;
descriptor_config.block_transfer_count = BUFFER_LEN;
descriptor_config.source_address = (uint32_t)string + sizeof(string);
descriptor_config.destination_address =
    (uint32_t)(&usart_instance.hw->USART.DATA.reg);
```

18. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

9.3.2. Use Case

9.3.2.1. Code

Copy-paste the following code to your user application:

```
dma_start_transfer_job(&usart_dma_resource_rx);

while (true) {
}
```

9.3.2.2. Workflow

1. Wait for receiving data.

   ```
   dma_start_transfer_job(&usart_dma_resource_rx);
   ```

2. Enter endless loop.

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

9.4. Quick Start Guide for SERCOM USART LIN

The supported board list:

- SAMC21 Xplained Pro

This quick start will set up LIN frame format transmission according to your configuration `CONF_LIN_NODE_TYPE`. For LIN master, it will send LIN command after startup. For LIN salve, once received a format from LIN master with ID `LIN_ID_FIELD_VALUE`, it will reply four data bytes plus a checksum.

9.4.1. Setup

9.4.1.1. Prerequisites

When verify data transmission between LIN master and slave, two boards are needed: one is for LIN master and the other is for LIN slave. connect LIN master LIN PIN with LIN slave LIN PIN.

9.4.1.2. Code

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

```
static struct usart_module cdc_instance,lin_instance;

#define LIN_ID_FIELD_VALUE 0x64

#define LIN_DATA_LEN 5
static uint8_t rx_buffer[LIN_DATA_LEN]={0};
const static uint8_t tx_buffer[LIN_DATA_LEN]={0x4a,0x55,0x93,0xe5,0xe6};
```

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

```
static void configure_usart_cdc(void)
{

    struct usart_config config_cdc;
    usart_get_config_defaults(&config_cdc);
    config_cdc.baudrate  = 115200;
```

```
    config_cdc.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_cdc.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_cdc.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_cdc.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_cdc.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&cdc_instance, EDBG_CDC_MODULE, &config_cdc);
    usart_enable(&cdc_instance);
}

static void lin_read_callback(struct usart_module *const usart_module)
{

    uint8_t i = 0;

    if (CONF_LIN_NODE_TYPE == LIN_MASTER_NODE) {
        for(i = 0; i < LIN_DATA_LEN; i++){
            if(rx_buffer[i] != tx_buffer[i]) {
                printf("Data error\r\n");
                break;
            }
        }
        if(i == LIN_DATA_LEN){
            printf("Slave response: OK\r\n");
        }
    } else if (CONF_LIN_NODE_TYPE == LIN_SLAVE_NODE) {
        if(rx_buffer[0] == LIN_ID_FIELD_VALUE) {
            usart_enable_transceiver(&lin_instance,USART_TRANSCEIVER_TX);
            printf("Receive ID field from mater: OK \r\n");
            usart_write_buffer_job(&lin_instance,
                (uint8_t *)tx_buffer, LIN_DATA_LEN);
        }
    }
}
static void lin_read_error_callback(struct usart_module *const
usart_module)
{
      printf("Data Read error\r\n");
}

static void configure_usart_lin(void)
{

    struct port_config pin_conf;
    port_get_config_defaults(&pin_conf);
    pin_conf.direction  = PORT_PIN_DIR_OUTPUT;
    port_pin_set_config(LIN_EN_PIN, &pin_conf);

    /* Enable LIN module*/
    port_pin_set_output_level(LIN_EN_PIN, 1);

    struct usart_config config_lin;
    usart_get_config_defaults(&config_lin);

    /* LIN frame format*/
    config_lin.lin_node = CONF_LIN_NODE_TYPE;
    config_lin.transfer_mode = USART_TRANSFER_ASYNCHRONOUSLY;
    config_lin.sample_rate = USART_SAMPLE_RATE_16X_FRACTIONAL;

    config_lin.baudrate = 115200;
    config_lin.mux_setting = LIN_USART_SERCOM_MUX_SETTING;
    config_lin.pinmux_pad0 = LIN_USART_SERCOM_PINMUX_PAD0;
    config_lin.pinmux_pad1 = LIN_USART_SERCOM_PINMUX_PAD1;
```

```
    config_lin.pinmux_pad2 = LIN_USART_SERCOM_PINMUX_PAD2;
    config_lin.pinmux_pad3 = LIN_USART_SERCOM_PINMUX_PAD3;

    /* Disable receiver and transmitter */
    config_lin.receiver_enable  = false;
    config_lin.transmitter_enable = false;

    if (CONF_LIN_NODE_TYPE == LIN_SLAVE_NODE) {
        config_lin.lin_slave_enable = true;
    }

    while (usart_init(&lin_instance,
        LIN_USART_MODULE, &config_lin) != STATUS_OK) {
    }

    usart_enable(&lin_instance);

    usart_register_callback(&lin_instance,
        lin_read_callback, USART_CALLBACK_BUFFER_RECEIVED);
    usart_enable_callback(&lin_instance, USART_CALLBACK_BUFFER_RECEIVED);
      usart_register_callback(&lin_instance,
        lin_read_error_callback, USART_CALLBACK_ERROR);
    usart_enable_callback(&lin_instance, USART_CALLBACK_ERROR);
    system_interrupt_enable_global();
}
```

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

```
system_init();
configure_usart_cdc();
```

9.4.1.3. Workflow

1. Create USART CDC and LIN module software instance structure for the USART module to store the USART driver state while it is in use.

   ```
   static struct usart_module cdc_instance,lin_instance;
   ```

2. Define LIN ID field for header format.

   ```
   #define LIN_ID_FIELD_VALUE 0x64
   ```

   Note: The ID `LIN_ID_FIELD_VALUE` is eight bits as [P1,P0,ID5...ID0], when it's 0x64, the data field length is four bytes plus a checksum byte.

3. Define LIN RX/TX buffer.

   ```
   #define LIN_DATA_LEN 5
   static uint8_t rx_buffer[LIN_DATA_LEN]={0};
   const static uint8_t tx_buffer[LIN_DATA_LEN]={0x4a,
   0x55,0x93,0xe5,0xe6};
   ```

   Note: For `tx_buffer` and `rx_buffer`, the last byte is for checksum.

4. Configure the USART CDC for output message.

   ```
   static void configure_usart_cdc(void)
   {

       struct usart_config config_cdc;
       usart_get_config_defaults(&config_cdc);
       config_cdc.baudrate  = 115200;
       config_cdc.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
   ```

```
    config_cdc.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_cdc.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_cdc.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_cdc.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&cdc_instance, EDBG_CDC_MODULE, &config_cdc);
    usart_enable(&cdc_instance);
}
```

5. Configure the USART LIN module.

```
static void lin_read_callback(struct usart_module *const usart_module)
{

    uint8_t i = 0;

    if (CONF_LIN_NODE_TYPE == LIN_MASTER_NODE) {
        for(i = 0; i < LIN_DATA_LEN; i++){
            if(rx_buffer[i] != tx_buffer[i]) {
                printf("Data error\r\n");
                break;
            }
        }
        if(i == LIN_DATA_LEN){
            printf("Slave response: OK\r\n");
        }
    } else if (CONF_LIN_NODE_TYPE == LIN_SLAVE_NODE) {
        if(rx_buffer[0] == LIN_ID_FIELD_VALUE) {

usart_enable_transceiver(&lin_instance,USART_TRANSCEIVER_TX);
            printf("Receive ID field from mater: OK \r\n");
            usart_write_buffer_job(&lin_instance,
                (uint8_t *)tx_buffer, LIN_DATA_LEN);
        }
    }
}
static void lin_read_error_callback(struct usart_module *const
usart_module)
{
      printf("Data Read error\r\n");
}

static void configure_usart_lin(void)
{

    struct port_config pin_conf;
    port_get_config_defaults(&pin_conf);
    pin_conf.direction  = PORT_PIN_DIR_OUTPUT;
    port_pin_set_config(LIN_EN_PIN, &pin_conf);

    /* Enable LIN module*/
    port_pin_set_output_level(LIN_EN_PIN, 1);

    struct usart_config config_lin;
    usart_get_config_defaults(&config_lin);

    /* LIN frame format*/
    config_lin.lin_node = CONF_LIN_NODE_TYPE;
    config_lin.transfer_mode = USART_TRANSFER_ASYNCHRONOUSLY;
    config_lin.sample_rate = USART_SAMPLE_RATE_16X_FRACTIONAL;

    config_lin.baudrate = 115200;
    config_lin.mux_setting = LIN_USART_SERCOM_MUX_SETTING;
    config_lin.pinmux_pad0 = LIN_USART_SERCOM_PINMUX_PAD0;
```

```
    config_lin.pinmux_pad1 = LIN_USART_SERCOM_PINMUX_PAD1;
    config_lin.pinmux_pad2 = LIN_USART_SERCOM_PINMUX_PAD2;
    config_lin.pinmux_pad3 = LIN_USART_SERCOM_PINMUX_PAD3;

    /* Disable receiver and transmitter */
    config_lin.receiver_enable  = false;
    config_lin.transmitter_enable = false;

    if (CONF_LIN_NODE_TYPE == LIN_SLAVE_NODE) {
        config_lin.lin_slave_enable = true;
    }

    while (usart_init(&lin_instance,
        LIN_USART_MODULE, &config_lin) != STATUS_OK) {
    }

    usart_enable(&lin_instance);

    usart_register_callback(&lin_instance,
        lin_read_callback, USART_CALLBACK_BUFFER_RECEIVED);
    usart_enable_callback(&lin_instance,
USART_CALLBACK_BUFFER_RECEIVED);
      usart_register_callback(&lin_instance,
        lin_read_error_callback, USART_CALLBACK_ERROR);
    usart_enable_callback(&lin_instance, USART_CALLBACK_ERROR);
    system_interrupt_enable_global();
}
```

Note: The LIN frame format can be configured as master or slave, refer to `CONF_LIN_NODE_TYPE`.

9.4.2. Use Case

9.4.2.1. Code

Copy-paste the following code to your user application:

```
configure_usart_lin();

if (CONF_LIN_NODE_TYPE == LIN_MASTER_NODE) {
    printf("LIN Works in Master Mode\r\n");
    if (lin_master_transmission_status(&lin_instance)) {
        usart_enable_transceiver(&lin_instance,USART_TRANSCEIVER_TX);
        lin_master_send_cmd(&lin_instance,LIN_MASTER_AUTO_TRANSMIT_CMD);
        usart_write_wait(&lin_instance,LIN_ID_FIELD_VALUE);
        usart_enable_transceiver(&lin_instance,USART_TRANSCEIVER_RX);
        while(1) {
            usart_read_buffer_job(&lin_instance,
            (uint8_t *)rx_buffer, 5);
        }
    }
} else {
    printf("LIN Works in Slave Mode\r\n");
    usart_enable_transceiver(&lin_instance,USART_TRANSCEIVER_RX);
    while(1) {
        usart_read_buffer_job(&lin_instance,
        (uint8_t *)rx_buffer, 1);
    }
}
```

9.4.2.2. Workflow

1. Set up USART LIN module.

```
configure_usart_lin();
```

2. For LIN master, sending LIN command. For LIN slaver, start reading data .

```
if (CONF_LIN_NODE_TYPE == LIN_MASTER_NODE) {
    printf("LIN Works in Master Mode\r\n");
    if (lin_master_transmission_status(&lin_instance)) {
        usart_enable_transceiver(&lin_instance,USART_TRANSCEIVER_TX);

lin_master_send_cmd(&lin_instance,LIN_MASTER_AUTO_TRANSMIT_CMD);
        usart_write_wait(&lin_instance,LIN_ID_FIELD_VALUE);
        usart_enable_transceiver(&lin_instance,USART_TRANSCEIVER_RX);
        while(1) {
            usart_read_buffer_job(&lin_instance,
            (uint8_t *)rx_buffer, 5);
        }
    }
} else {
    printf("LIN Works in Slave Mode\r\n");
    usart_enable_transceiver(&lin_instance,USART_TRANSCEIVER_RX);
    while(1) {
        usart_read_buffer_job(&lin_instance,
        (uint8_t *)rx_buffer, 1);
    }
}
```

10. SERCOM USART MUX Settings

The following lists the possible internal SERCOM module pad function assignments, for the four SERCOM pads when in USART mode. 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.

When TX and RX are connected to the same pin, the USART will operate in half-duplex mode if both one transmitter and several receivers are enabled.

Note: When RX and XCK are connected to the same pin, the receiver must not be enabled if the USART is configured to use an external clock.

MUX/Pad	PAD 0	PAD 1	PAD 2	PAD 3
RX_0_TX_0_XCK_1	TX / RX	XCK	-	-
RX_0_TX_2_XCK_3	RX	-	TX	XCK
RX_1_TX_0_XCK_1	TX	RX / XCK	-	-
RX_1_TX_2_XCK_3	-	RX	TX	XCK
RX_2_TX_0_XCK_1	TX	XCK	RX	-
RX_2_TX_2_XCK_3	-	-	TX / RX	XCK
RX_3_TX_0_XCK_1	TX	XCK	-	RX
RX_3_TX_2_XCK_3	-	-	TX	RX / XCK

11. Document Revision History

Doc. Rev.	Date	Comments
42118F	12/2015	Added support for SAM L21/L22, SAM DA1, SAM D09, and SAM C20/C21
42118E	12/2014	Added support for SAM R21 and SAM D10/D11
42118D	01/2014	Added support for SAM D21
42118C	10/2013	Replaced the pad multiplexing documentation with a condensed table
42118B	06/2013	Corrected documentation typos
42118A	06/2013	Initial release

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**