CoreSDLC Driver User's Guide

Version 2.0

Table of Contents

Introduction

This document describes the bare metal software driver for CoreSDLC. CoreSDLC is an implementation of the SDLC device communication. The CoreSDLC bare metal software driver is designed for use in systems with no operating system. This driver can be adapted for use as part of an operating system, but the implementation of the adaptation layer between this driver and the operating system's driver model is outside the scope of this driver. This software driver provides programming interfaces (Layer 1 / Link Layer) to configure and access the underlying CoreSDLC hardware. However, implementing any of the (Layer 2 / Network Layer) protocols is not in scope of this driver.

Features

The CoreSDLC driver provides support for the following features:

- Configuring CoreSDLC communication parameters
- Interrupt driven transmit and receive
- Reading statistical parameters of the SDLC node.
- Raw data transmission and reception as test modes.

The CoreSDLC driver is provided as C source code.

Supported Hardware IP

The CoreSDLC bare metal driver can be used with version 3.0.121 of Microsemi’s CoreSDLC IP.

Files Provided

The files provided as part of the CoreSDLC driver fall into three main categories: documentation, driver source code, and example projects. The driver is distributed via the Microsemi SoC Products Group’s Firmware Catalog, which provides access to the documentation for the driver, generates the driver’s source files into an application project, and generate example projects that illustrating how to use the driver. The Firmware Catalog is available from: www.microsemi.com/soc/products/software/firmwarecat/default.aspx

Documentation

The Microsemi Firmware Catalog provides access to these documents for the driver:

- User’s guide (this document)
- Release notes

Driver Source Code

The Firmware Catalog generates the driver’s source code into a drivers\CoreSDLC subdirectory of the selected software project directory. The files making up the driver are detailed below.

core_sdlc.h

This header file contains the public application programming interface (API) of the CoreSDLC software driver. This file should be included in any C source file that uses the CoreSDLC software driver.

coresdlc_regs.h

This file contains the CoreSDLC register definitions required for accessing the core through the hardware abstraction layer (HAL). This file is only used within the software driver implementation and does not need to be directly included in your code.

core_sdlc.c

This C source file contains the implementation of the CoreSDLC software driver.

Example Code

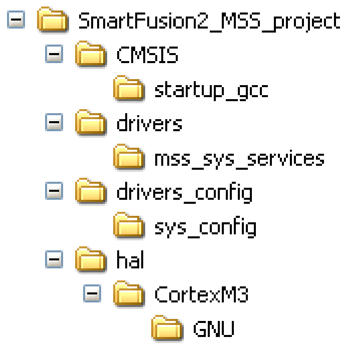
The Firmware Catalog provides access to example projects illustrating the use of the driver. Each example project is self contained and is targeted at a specific processor and software tool chain combination. The example projects are targeted at the FPGA designs in the hardware development tutorials supplied with SoC Product Group’s development boards. The tutorial designs may be found on the Microsemi SoC Development Kit web page.

Note: Make sure that the base addresses for the peripheral drivers used in the example project match the memory map of the targeted hardware design. The base addresses are generally specified in the platform.h file or in the main.c file in the project’s root directory.

Driver Deployment

This driver is intended to be deployed from the Firmware Catalog into a software project by generating the driver's source files into the project directory. The driver uses the SmartFusion2 Cortex Microcontroller Software Interface Standard Hardware Abstraction Layer (CMSIS HAL) to access MSS hardware registers. You must ensure that the SmartFusion2 CMSIS HAL is included in the project settings of the software toolchain used to build your project and that it is generated into your project. The most up-to-date SmartFusion2 CMSIS HAL files can be obtained using the Firmware Catalog.

The following example shows the intended directory structure for a project based on SoftConsole ARM® Cortex™-M3 project targeted at the SmartFusion2 MSS. This project uses the CoreSDLC drivers. Both of these drivers rely on the SmartFusion2 CMSIS HAL for accessing the hardware. The contents of the drivers directory result from generating the source files for each driver into the project. The contents of the *CMSIS* and hal directory result from generating the source files for the SmartFusion2 CMSIS HAL into the project. The contents of the *drivers_config* directory are generated by the Libero project and must be copied into the software project.



```
SmartFusion2_MSS_project
    CMSIS
        startup_gcc
    drivers
        mss_sys_services
    drivers_config
        sys_config
    hal
        CortexM3
            GNU
```

Figure 1 · SmartFusion2 MSS Project Example

Driver Configuration

Your application software should configure the CoreSDLC driver through calls to the *SDLC_init*() function for each CoreSDLC instance in the hardware design. *SDLC_init*() should be called only once for each CoreSDLC instance in the hardware design. The configuration parameters include the CoreSDLC hardware instance base address and pointer to a structure of type *sdlc_cfg_t*. The *sdlc_cfg_t* structure holds all the configuration values to be configured for the CoreSDLC device. *sdlc_cfg_t* structure should be initialized with the desired values before making call to *SDLC_init()* API.

No CoreSDLC hardware configuration parameters are needed by the driver, apart from the CoreSDLC hardware instance base address. Hence, no additional configuration files are required to use the driver.

This driver supports CoreSDLC device configured as primary and secondary node. There is always one primary node in the network, but there may be one or more secondary nodes. The primary node controls operation of the secondary nodes and manages the network. Secondary nodes can send information only if the primary node has given them permission.

This driver can be used for below network configurations.

- Point-to-point, when there is one primary and only one secondary node.
- Multi-drop, when there is one primary and multiple secondary nodes.

Application Programming Interface

This section describes the driver's API. The functions and related data structures described in this section are used by the application programmer to control the CoreSDLC peripheral from their application.

Theory of Operation

The CoreSDLC software driver is designed to allow the control of multiple instances of CoreSDLC. Each instance of CoreSDLC in the hardware design is associated with a single instance of the *sdlc_instance_t* structure in the software. You need to allocate memory for one unique *sdlc_instance_t* structure instance for each CoreSDLC hardware instance. The contents of these data structures are initialized during calls to function *SDLC_Init*(). A pointer to the structure is passed to subsequent driver functions in order to identify the CoreSDLC hardware instance you wish to perform the requested operation on.

Initialization and Configuration

The CoreSDLC driver is initialized and configured by calling the *SDLC_init()* function. The *SDLC_init()* function takes a pointer to a configuration data structure as parameter. This data structure contains all the configuration information required to initialize and configure the CoreSDLC.

The CoreSDLC driver provides the *SDLC_cfg_struct_def_init()* function to initialize the configuration data structure to default value. It is recommended to use this function to retrieve the default configuration then overwrite the defaults with the application specific setting such Mode of operation address length. The *SDLC_init()* function must be called before any other CoreSDLC driver functions. The *SDLC_cfg_struct_def_init()* is the only function which can called before calling *the SDLC_init()* function.

The Following functions are used as part of the initialization and configuration process.

- *SDLC_cfg_struct_def_init()*
- *SDLC_init()*

If you need to individually configure mode parameters of the CoreSDLC driver or to get the current CoreSDLC instance configuration, you can use below APIs at the run time. This API takes *sdlc_instance_t* type as parameter along with the configuration parameter.

- *SDLC_set_mode()*

SDLC_set_mode() sets the mode of the CoreSDLC device. Three modes, Normal (0x00), Raw Transmit (0x01) and Raw receive (0x02) can be set using this API.

- *SDLC_get_cfg()*

The *SDLC_get_cfg()* function is used to read the current node configuration values. This API copies the current configurations into *sdlc_cfg* parameter.

Interrupt Driven Transmit and receive Operations

The CoreSDLC driver transmits or receives frames under interrupt control. SDLC operations are performed using these functions:

- *SDLC_rx_data()*
- *SDLC_tx_data()*
- *SDLC_tx_abort()*

- *SDLC_is_tx_complete()*
- *SDLC_enable_rx()*
- *SDLC_disable_rx()*

- *SDLC_default_txv_handler()*
- *SDLC_default_rxe_handler()*
- *SDLC_default_rdn_handler()*

- *SDLC_enable_interrupt()*
- *SDLC_disable_interrupt()*
- *SDLC_set_interrupt_priority()*

- *SDLC_read_statistic()*
- *SDLC_read_rx_status()*

A CoreSDLC device, irrespective of its use as a primary or a secondary node, can use all of these APIs as needed.

Interrupt Handlers

CoreSDLC driver uses four interrupts to handle different events, Transmit valid (TXV), Receive Valid (RXV), Receive Done (RDN) and Receive Error (RXE). Since each of these events generates independent interrupt signal, four system level (NVIC level) interrupts are needed. Default implementations (default handlers) of all the four interrupts are provided as part of this driver. The default handlers should be called from the system level (NVIC level) interrupt service routine (ISR) assigned to the respective interrupt triggered by the CoreSDLC signals.

First Receive valid interrupt marks the beginning of the frame reception. *SDLC_rx_data()* API should be used when Receive valid event is detected. You should provide the data structure of *sdlc_frame_t* type to this API. This data structure is used to store the data received.

Receive done interrupt indicates the completion of Valid frame reception. Though the frame received was valid, *SDLC_default_rdn_handler*() API checks for frame without control field. In this case CoreSDLC state is changed to SDLC_ERROR and SDLC_ERROR_SHORT_LEN will be returned. This API enables the SDLC reception for receiving next frame.

In case of an error in the reception, *SDLC_default_rxe_handler()* API updates the appropriate status and statistics data structures. This API returns value indicating which error had occurred and re-enables SDLC for receiving next frame.

Transmit valid interrupt occurs when there is place in the transmit FIFO to accept data. SDLC_default_txv_handler() API puts appropriate number of bytes in the FIFO depending on the availability of place in the FIFO and the data remaining to be transmitted.

SDLC_enable_interrupt() and *SDLC_disable_interrupt()* are used to enable or disable interrupts respectively. The parameter value should be logic OR of the interrupt numbers those should be enabled or disabled.

You can choose the priority of interrupt when multiple interrupts occur simultaneously by using *SDLC_set_interrupt_priority()* API. Parameter value '1' steps up the priority of the corresponding interrupt depending on the interrupt number. The parameter value should be logic OR of the interrupt numbers those should be enabled or disabled.

Transmitting Data

The frame transmission in Normal mode as well as Raw Transmit mode is initiated by using *SDLC_tx_data()* API with the frame to be transmitted as its parameter. Your application is then free to perform other tasks and inquire later whether transmit has completed by calling the *SDLC_is_tx_complete()* function. *SDLC_tx_data()* API enables transmission, loads data from frame in the transmit FIFO and then enables transmit valid interrupt.

An SDLC Frame consists of the beginning of frame (BOF), Address field, Control field, Info field, CRC field and the end of frame (EOF) flag. BOF, CRC and EOF are added and transmitted by CoreSDLC in normal mode. *SDLC_tx_data()* uses *sdlc_frame_t* parameter to transmit the frame in correct sequence. Make sure that the frame is loaded with the proper node Address, Control and the data buffer before making call to this API.

In Raw Transmit mode, the Address and Control field of the frame are ignored by the driver. Only the data available in the data buffer of the frame will be transmitted. If you want to use this Raw transmit mode for the SDLC Receiver testing, then you must fill the data buffer with BOF, ADDRESS and EOF field at appropriate places along with raw data you want to transmit.

An abort sequence can be transmitted by making a call to the *SDLC_tx_abort()* API.

Receiving Data

Data reception starts with the Receive valid (RV) interrupt occurs. *SDLC_rx_data()* should be called from the RV system level. *SDLC_rx_data()* starts filling up the frame as it keeps receiving the data.

Data reception in Normal mode as well as in Raw receive mode happens in the same manner. In both modes receive frame is updates in the same way, except that the CRC field is also loaded in the data buffer in Raw receive mode.

In case of error in the data reception, Receive Error (RE) interrupt occurs. *SDLC_default_rxe_handler()* should be called from the system level interrupt. *SDLC_default_rxe_handler()* identifies the error occurred and returns a unique value. This return value along with SDLC Status returned by *SDLC_read_rx_status*() should be used in application program to take further action. No error detection is done in Raw receive mode and receive error interrupt is not asserted.

SDLC_default_rdn_handler() API should be called when Receive Done(RDN) interrupt occurs. This event marks the end of valid frame reception and updates the status accordingly. Even though the received frame was valid, CoreSDLC status will be set to SDLC_ERROR when the frame was found to have no control field (number of bytes received < 2 in 8-bit addressing and number of bytes received < 3 in 16-bit addressing).

Note: In this error scenarios mentioned above *SDLC_default_rxe_handler()* will not be called since for CoreSDLC the frame received was still valid. For this reason you should always use *SDLC_read_rx_status*() function to identify if the reception was completed successful.

SDLC_read_rx_status() should also be used to find the exact error occurred along with the values returned by *SDLC_default_rxe_handler()* or *SDLC_default_rdn_handler().*

SDLC_enable_rx() and *SDLC_disable_rx()* are used to enable or disable reception respectively.

SDLC_check_rx_line_idle() is used to know whether receive line is idle (15 consecutive '1' values are received on rxd) or not idle.

SDLC Status and Statistics

SDLC_read_statistics() is used to find the current statistics of the CoreSDLC node. It copies the current statistics of the node in the *sdlc_stat_t* type provided as a parameter. Statistics are applicable in Normal mode only. In Raw transmit and Raw receive modes, no statistics elements are updated. This API should be used only in Normal mode of communication.

SDLC_read_rx_status() is used to know the current status of the CoreSDLC node.

Types

sdlc_states

Prototype

```
enum sdlc_states
    {
        SDLC_IDLE=1,
        SDLC_ACTIVE,
        SDLC_ERROR,
        SDLC_RDN,
        SDLC_TDN
    };
```

Description

This driver uses *sdlc_states* enum to track the current activity undertaken by the SDLC node. Meaning of each state is as follows.

- SDLC_IDLE => Node is initialized but no activity is yet started. Line idle bit is found set.

  Note: Line idle bit is not checked by driver internally. It should be done by user by using *SDLC_check_rx_line_idle()*.
- SDLC_ACTIVE => Node is performing either transmit or receive transaction.
- SDLC_ERROR => Error was detected in last receive operation. No error check for transmittion.
- SDLC_RDN => Frame was received successfully.
- SDLC_TDN => Frame was transmitted successfully.

sdlc_receive_error_code

Prototype

```
enum sdlc_receive_error_code
    {
        SDLC_ERROR_SHORT_LEN=0,
        SDLC_ERROR_ABORT,
        SDLC_ERROR_OVERFLOW,
        SDLC_ERROR_CRC,
        SDLC_ERROR_ALIGNMENT,
    };
```

Description

sdlc_receive_error_code enum indicates which error occurred while receiving the frame. Meaning of each error code is as below.

- SDLC_ERROR_SHORT_LEN => Valid frame was received without control field.
- SDLC_ERROR_ABORT => Abort sequence was detected while receiving frame..
- SDLC_ERROR_OVERFLOW => Overflow error was detected while receiving frame
- SDLC_ERROR_CRC => CRC error was detected while receiving frame.
- SDLC_ERROR_ALIGNMENT => Alignment error was detected while receiving frame

Constant Values

Encoding for Transmit or Receive

The following defines are used to build the elements of *sdlc_cfg_t* type which is used for configuring the CoreSDLC node. These values are also used to provide appropriate configuration parameters to the individual parameter setting APIs.

Constant	Description
`SDLC_ENCODING_NRZI`	Transmit or receive will be done using NRZI encoding. Internal clock is selected
`SDLC_ENCODING_NRZ`	Transmit or receive will be done using NRZ encoding. External clock is selected

Table 1 · Encoding options

SDLC communication modes

The following defines are used to build the elements of *sdlc_cfg_t* type which is used for configuring the CoreSDLC node.

Constant	Description
`SDLC_MODE_NORMAL`	SDLC communication in Normal mode
`SDLC_MODE_RAW_TX`	SDLC Communication in Raw Transmit mode
`SDLC_MODE_RAW_RX`	SDLC Communication in Raw Transmit mode

Table 2 · SDLC communication mode Configuration Options

SDLC node address length

The following defines are used to build the elements of *sdlc_cfg_t* type which is used for configuring the CoreSDLC node.

Constant	Description
`SDLC_ADDRESS_LENGTH_8BIT`	Target node address is 8bit
`SDLC_ADDRESS_LENGTH_16BIT`	Target node address is 16bit

Table 3 · SDLC node address length configuration options

SDLC communication CRC type

The following defines are used to build the elements of *sdlc_cfg_t* type which is used for configuring the CoreSDLC node.

Constant	Description
`SDLC_CRC_LENGTH_16BIT`	16 bit CRC is used in SDLC communication

Constant	Description
`SDLC_CRC_LENGTH_32BIT`	32 bit CRC is used in SDLC communication

Table 4 · SDLC communication CRC type configuration options

SDLC communication preamble length

The following defines are used to build the elements of *sdlc_cfg_t* type which is used for configuring the CoreSDLC node.

Constant	Description
`SDLC_PREAMBLE_LENGTH_0BITS`	0-bit length preamble is generated before frame transmission
`SDLC_PREAMBLE_LENGTH_8BITS`	8-bit length preamble is generated before frame transmission
`SDLC_PREAMBLE_LENGTH_32BITS`	32-bit length preamble is generated before frame transmission
`SDLC_PREAMBLE_LENGTH_64BITS`	64-bit length preamble is generated before frame transmission

Table 5 · SDLC communication preamble length selection option

SDLC communication back-to-back frame reception

The following defines are used to build the elements of *sdlc_cfg_t* type which is used for configuring the CoreSDLC node.

Constant	Description
`SDLC_B2B_FRAME_RX_DISABLE`	Back to back frame reception is disabled
`SDLC_B2B_FRAME_RX_ENABLE`	Back to back frame generation is enabled

Table 6 · SDLC communication back to back frame reception selection option

SDLC communication Inter-frame space

The following defines are used to build the elements of *sdlc_cfg_t* type which is used for configuring the CoreSDLC node. Two extreme constant values are provided; you can create and use other IFS values to configure the SDLC node. Please note that the IFS value should always be an even number.

Constant	Description
`SDLC_INTERFRAME_SPACE_2BIT`	Transmitter waits for 2-bit time before starting frame transmission
`SDLC_INTERFRAME_SPACE_254BIT`	Transmitter waits for 254-bit time before starting frame transmission

Table 7 · SDLC communication Inter frame space selection option

SDLC communication idle flag generation

The following defines are used to build the elements of *sdlc_cfg_t* type which is used for configuring the CoreSDLC node.

Constant	Description
`SDLC_IDLE_FLAG_GENERATION_DISABLE`	Idle flag is not generated between two transmit frames
`SDLC_IDLE_FLAG_GENERATION_ENABLE`	Idle flags (01111110) are generated between two transmit frames

Table 8 · SDLC communication idle flag generation selection option

SDLC Interrupts

The following defines are used to enable and disable CoreSDLC interrupts. They are used to build the value of the *irq_mask* parameter for the *SDLC_enable_irq()* and *SDLC_disable_irq()* functions. A bitwise OR of these constants is used to enable or disable multiple interrupts or setting up the priorities of the interrupts.

Constant	Description
`SDLC_RXV_IRQ`	Receive data available interrupt (0x01)
`SDLC_RXE_IRQ`	Receive error detection interrupt (0x02)
`SDLC_RDN_IRQ`	Successful frame reception done (0x04)
`SDLC_TXV_IRQ`	Transmit Valid (TFIFO is not full interrupt) (0x08)

Table 9 · SDLC Interrupt Mask Constants

Data Structures

sdlc_instance_t

Description

This structure is used to identify the various CoreSDLC hardware instances in your system. Your application software should declare one instance of this structure for each instance of CoreSDLC in your system. The function *SDLC_init()* initializes this structure. A pointer to an initialized instance of the structure should be passed as the first parameter to the CoreSDLC driver functions, to identify which CoreSDLC hardware instance should perform the requested operation.

sdlc_frame_t

Description

This structure is used to define a frame used for SDLC transmission and reception. A frame in SDLC consists of beginning of frame (BOF), address field (ADDR), Control field (CONTROL), data field (INFO), CRC and end of frame (EOF). BOF, CRC and EOF are generated by the core. User software needs to provide the ADDR, CONTROL and INFO fields. ADDRESS and CONTROL fields can be provided in the address and control elements of the *sdlc_frame_t* structure. Since the INFO field can be variable in length, *sdlc_frame_t* accepts a pointer to the buffer where INFO field is stored and the size of this buffer as its elements.

Note: Address field of this structure is uint16_t, since the SDLC node address length of then node can be up to 16bit. While operating with 8bit address length, LSB of Address field is transmitted and MSB is ignored.

sdlc_cfg_t

Description

This structure is used to configure the SDLC node parameters. This structure hold all the values needed for successful node configuration and is passed to the *SDLC_init()* function. This structure is also passed to the *SDLC_set_params()* functions, to change the configuration of the node or read the configuration of the node at run time.

sdlc_stat_t

Description

This structure is used to store all the statistical parameters of the node such as successful transmit/receive transactions, receive error, abort frame transmission etc. One instance of this type is created for each hardware instance of the CoreSDLC driver

Functions

SDLC_init

Prototype

```
uint8_t SDLC_init
(
    sdlc_instance_t* this_sdlc,
    sdlc_cfg_t* sdlc_cfg,
    addr_t base_address
);
```

Description

The *SDLC_init()* function initializes the driver's data structures and the CoreSDLC hardware with the configuration passed as parameters. The SDLC_init() function takes a pointer to a configuration data structure of type sdlc_cfg_t as parameter. This configuration data structure contains all the information required to configure the CoreSDLC.

Parameters

this_sdlc

The *this_sdlc* parameter is a pointer to a *sdlc_instance_t* data structure that holds all data regarding CoreSDLC hardware instance being initialized. A pointer to the same data structure must be provided in subsequent calls to the various CoreSDLC driver function in order to identify the CoreSDLC instance that should perform the operation implemented by the called driver function.

cfg

The cfg parameter is a pointer to a *sdlc_cfg_t* structure that holds all the configuration data to be used for initializing the CoreSDLC device. You must initialize this data structure by first calling the *SDLC_cfg_struct_def_init()* function to fill the configuration data structure with default values. You can then overwrite some of the default settings with the ones specific to your application before passing this data structure as parameter to the call to the SDLC_init() function.

base_addr

The *base_address* parameter is the base address in the processor's memory map for the registers of the CoreSDLC instance being initialized.

Return Value

This function returns a zero value when the initialization fails. A non-zero value is returned when the initialization is successful.

SDLC_rx_data

Prototype

```
void SDLC_rx_data
(
    sdlc_instance_t* this_sdlc,
    sdlc_frame_t* rx_frame
)
```

Description

The SDLC_rx_data() function reads available data from receive FIFO when Receive valid(RXV) interrupt occurs. CoreSDLC does the address matching when it receives BOF. If address matches, this API starts loading the data in the frame receive data structure provided as parameters until the reception is done (RDN) or an error occurs in data reception or the data field in the rx_frame is full. This function returns when all the available data from RFIFO is read. However the valid frame reception is still in progress and will complete when RDN interrupt occurs. If *rx_frame* does not have place to hold data anymore then driver sets the CoreSDLC status to SDL_ERROR. Data from RFIFO is still read to avoid overflow error, however data is not loaded in *rx_frame* structure. CoreSDLC status returned by API *SDLC_read_rx_status()* should always be checked to make sure that the reception was done successfully.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

rx_frame

The *rx_frame* parameter is a pointer to a *sdlc_frame_t* type, where the received data will be stored.

Return Value

This function does not return any value.

SDLC_tx_data

Prototype

```
void
SDLC_tx_data
(
    sdlc_instance_t *this_sdlc,
    sdlc_frame_t* tx_frame
)
```

Description

The *SDLC_tx_data()* function is used to transmit a SDLC frame in normal mode. This API is also used to transmit raw data in Raw Transmit mode. This driver uses interrupt method to transmit data. This function first enables transmission, clears TFIFO and loads appropriate number of bytes in TFIFO. It then enables the Transmit Valid (TXV) interrupt. Data transmission starts when the transmission is enabled and data is available in TFIFO. TXV occurs when the transmission is enabled and there is place in TFIFO to accept more data.

Note: All the data may not be transmitted when this function returns. If you need to know the completion of transmission you can use *SDLC_read_rx_status()* API.

Parameters

this_sdlc

The this_sdlc parameter is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

tx_frame

tx_frame is used to pass a SDLC frame to *SDLC_tx_data()* API. User can provide SDLC target node address, control field and the data to be sent in info field and its size.

Return Value

This function does not return any value.

SDLC_tx_abort

Prototype

```
void
SDLC_tx_abort
(
    sdlc_instance_t * this_sdlc
);
```

Description

SDLC_tx_abort() generates abort sequence by going into Raw transmit mode and transmitting 0xFF.

Abort sequence is a pattern of seven consecutive ‘1’. Abort sequence is transmitted to abort the current frame transmission. Driver then returns back to the mode of operation in which it was operating before calling this API.

Note: Abort sequence can be generated in different ways; this implementation is one of the ways to implement abort sequence.

Parameters

this_sdlc

The this_sdlc parameter is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

Return Value

This function does not return any value.

Example

```
  uint8_t g_CoreSDLC0_tx_buffer[]={0x10,0x20,0x30};
  sdlc_frame_t tx_frame =
  {
      .address = NODE0_RX_ADDRESS,
      .control_field = 0x02,
      .data_buffer=g_CoreSDLC1_tx_buffer,
      .data_buffer_size =sizeof(g_CoreSDLC1_tx_buffer),
  };
  sdlc_instance_t g_CoreSDLC1;
  SDLC_tx_data ( &g_CoreSDLC1, & tx_frame);
  SDLC_tx_abort (&g_CoreSDLC1);
```

SDLC_default_txv_handler

Prototype

```
void
SDLC_default_txv_handler
(
    sdlc_instance_t *this_sdlc
);
```

Description

This API is the default implementation of TXV interrupt handler. TXV interrupt occurs when transmission is enabled and there is place in TFIFO to accept data. This API puts appropriate number of bytes into TFIFO from the frame provided in *SDLC_tx_data()*. This API should be called from the driver’s top level interrupt handler function, from the system level (NVIC level) interrupt handler assigned to the interrupt triggered by the CoreSDLC TXV signal.

It is your responsibility to enable the system level (NVIC level) interrupt connected to the CoreSDLC TXV interrupt signal.

Parameters

this_sdlc

The this_sdlc parameter is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

Return Value

This function does not return any value.

SDLC_default_rxe_handler

Prototype

```
sdlc_receive_error_code_t
SDLC_default_rxe_handler
(
    sdlc_instance_t * this_sdlc
)
```

Description

This *SDLC_default_rxe_handler()* function is the default implementation of the RXE interrupt. RXE interrupt occurs when SDLC is receiving data in normal mode and an error is detected during reception. Different errors such as overflow error, CRC error, abort error, or alignment error can happen. This API detects which error has occurred and returns value accordingly. This API should be called from the driver's top level interrupt handler function, from the system level (NVIC level) interrupt handler assigned to the interrupt triggered by the CoreSDLC RXE signal.

Note:

1. There can be errors in reception even when RXE interrupt is not asserted. These errors are indicated by return value of *SDLC_default_rdn_handler()* API. It is recommended that you always use *SDLC_read_rx_status()* API to know if the last frame received had error or the frame was received without error.
2. Use the return value of this API along value returned by *SDLC_read_rx_status()* to know the exact error occurred.
3. It is your responsibility to enable the system level (NVIC level) interrupt connected to the CoreSDLC RXE interrupt signal.

Parameters

this_sdlc

The this_sdlc parameter is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

Return Value

This function returns a value of type *sdlc_receive_error_code_t*.

SDLC_default_rdn_handler

Prototype

```
size_t
SDLC_default_rdn_handler
(
    sdlc_instance_t * this_sdlc
) ;
```

Description

This API is the default implementation to handle RDN interrupt. RDN interrupt occurs when valid frame is received in normal or Raw receive mode. *SDLC_default_rdn_handler()* API should be called when Receive Done(RDN) interrupt occurs. This event marks the end of valid frame reception. Though the frame received was valid, status of SDLC will be changed to SDLC_ERROR, if the received frame was bigger than the size of the data structure provided by user or the frame does not contain the control field. Driver makes CoreSDLC ready for receiving next frame.

Note: It is your responsibility to enable the system level (NVIC level) interrupt connected to the CoreSDLC RDN interrupt signal.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a sdlc*_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

Return Value

This function returns number of bytes received (including ADDRESS, CONTROL and INFO field) when the status of SDLC is SDLC_RDN. If status of SDLC is SDLC_ERROR then return value indicates the exact error which occurred. Only SDLC_ERROR_SHORT_LEN can occur when RDN interrupt is asserted. No other error type can occur since Received frame was valid for the SDLC core and no RXE interrupt was asserted.

SDLC_read_statistics

Prototype

```
void
SDLC_read_statistics
(
    sdlc_instance_t * this_sdlc,
    sdlc_stat_t * statistics
);
```

Description

The *SDLC_read_statistics()* function is used to read statistical information on the performance of the node at run time. Statistics information is provided only in normal mode of operation.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

statistics

SDLC_read_statistics() API will copy the current statistical parameters to *statistics* parameter when this API is called.

Return Value

This function does not return any value.

SDLC_read_rx_status

Prototype

```
sdlc_states_t
SDLC_read_rx_status
(
    sdlc_instance_t * this_sdlc
);
```

Description

The *SDLC_read_rx_status()* function is used to know the status of the receiver of the CoreSDLC node.

This API returns the status value of type enum *sdlc_states_t*.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

Return Value

This function returns a *sdlc_states_t* type value. This value indicates if the last frame reception was successful or there was error in receiving frame.

SDLC_enable_interrupt

Prototype

```
void
SDLC_enable_interrupt
(
    sdlc_instance_t * this_sdlc,
    uint8_t irq_mask
);
```

Description

The *SDLC_enable_interrupt()* function enables the CoreSDLC interrupts specified by the *irq_mask* parameter. The *irq_mask* parameter identifies the CoreSDLC interrupts by bit position, as defined in the interrupt enable register (IEN1) of CoreSDLC. The CoreSDLC interrupts and their identifying *irq_mask* bit positions are as follows:

• Receive valid interrupt enable (RXV)	(*irq_mask* bit 0)
• Receive error interrupt enable (RXE)	(*irq_mask* bit 1)
• Transmit valid interrupt enable (TXV)	(*irq_mask* bit 3)

When an *irq_mask* bit position is set to 1, this function enables the corresponding CoreSDLC interrupt.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a sdlc*_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

irq_mask

The *irq_mask* parameter is used to select which of the CoreSDLC's interrupts you want to enable. The allowed value for the *irq_mask* parameter is one of the following constants or a bitwise OR of more than one:

• SDLC_RXV_IRQ	((uint8_t) 0x01)
• SDLC_RXE_IRQ	((uint8_t) 0x02)
• SDLC_TXV_IRQ	((uint8_t) 0x08)

Return Value

This function does not return any value.

Example

```
SDLC_enable_interrupt(&g_CoreSDLC1, SDLC_RXV_IRQ| SDLC_RXE_IRQ);
```

SDLC_disable_Interrupt

Prototype

```
void
SDLC_disable_interrupt
(
    sdlc_instance_t * this_sdlc,
    uint8_t irq_mask
);
```

Description

The *SDLC*_disable_Interrupt*()* function disables the CoreSDLC interrupts specified by the *irq_mask* parameter. The *irq_mask* parameter identifies the CoreSDLC interrupts by bit position, as defined in the interrupt enable register (IEN1) of CoreSDLC. The CoreSDLC interrupts and their identifying bit positions are as follows:

• Receive valid interrupt enable (RXV)	(*irq_mask* bit 0)
• Receive error interrupt enable (RXE)	(*irq_mask* bit 1)
• Transmit valid interrupt enable (TXV)	(*irq_mask* bit 3)

When an *irq_mask* bit position is set to 1, this function disables the corresponding CoreSDLC interrupt.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

irq_mask

The *irq_mask* parameter is used to select which of the CoreSDLC's interrupts you want to enable. The allowed value for the *irq_mask* parameter is one of the following constants or a bitwise OR of more than one:

• SDLC_RXV_IRQ	((uint8_t) 0x01)
• SDLC_RXE_IRQ	((uint8_t) 0x02)

- SDLC_TXV_IRQ ((uint8_t) 0x08)

Return Value

This function does not return any value.

Example

```
  SDLC_disable_interrupt(&g_CoreSDLC1, SDLC_RXV_IRQ | SDLC_RXE_IRQ);
```

SDLC_set_interrupt_priority

Prototype

```
void
SDLC_set_interrupt_priority
(
    sdlc_instance_t * this_sdlc,
    uint8_t irq_prio_mask
);
```

Description

The *SDLC_set_interrupt_priority()* function sets the priority of the interrupt compared to other interrupts in CoreSDLC depending on the parameter *int_prio_mask.* The *int_prio_mask* parameter identifies the CoreSDLC interrupts by bit position, as defined in the interrupt priority register (IPN1) of CoreSDLC. The CoreSDLC interrupts and their identifying bit positions are as follows:

Receive error interrupt enable (RXE)	(*irq_mask* bit 0)
Receive valid interrupt enable (RXV)	(*irq_mask* bit 1)
Transmit valid interrupt enable (TXV)	(*irq_mask* bit 3)

When an *irq_prio_mask* bit position is set to 1, this function sets the priority of the corresponding CoreSDLC interrupt.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a sdlc*_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

irq_prio_mask

The *irq_prio_mask* parameter is used to select which of the CoreSDLC's interrupts you want to change the priority. The allowed value for the *irq_prio_mask* parameter is one of the following constants or a bitwise OR of more than one:

SDLC_RXE_IRQ	((uint8_t) 0x01)
SDLC_RXV_IRQ	((uint8_t) 0x02)
SDLC_TXV_IRQ	((uint8_t) 0x08)

Return Value

This function does not return any value.

Example

```
  SDLC_set_interrupt_priority(&g_CoreSDLC1, SDLC_RXV_IRQ | SDLC_RXE_IRQ);
```

SDLC_cfg_struct_def_init

Prototype

```
void
SDLC_cfg_struct_def_init
(
    sdlc_cfg_t * sdlc_cfg
);
```

Description

The *SDLC_cfg_struct_def_init()* function is used to initializes a *sdlc_cfg_t* configuration data structure to default values. This default configuration can then be used as parameter to *SDLC_init().* Typically the default configuration would be modified to suit the application before being passed to *SDLC_init().*

Parameters

sdlc_cfg

The *sdlc_cfg* parameter is a pointer to a *sdlc_cfg_t* structure that holds all the configuration data to be used for initializing the CoreSDLC device.

Return Value

This function does not return any value.

Example

```
        sdlc_cfg_t g_SDLC1_cfg;
        SDLC_cfg_struct_def_init(& g_SDLC1_cfg);
```

SDLC_get_cfg

Prototype

```
void
SDLC_get_cfg
(
    sdlc_instance_t* this_sdlc,
    sdlc_cfg_t * sdlc_cfg
);
```

Description

The *SDLC_get_cfg()* function is used to read the current node configuration values. This API copies the current configurations into sdlc_cfg parameter.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a sdlc*_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

sdlc_cfg

The *sdlc_cfg* parameter is a pointer to a *sdlc_cfg_t* structure that holds all the configuration data to be used for initializing the CoreSDLC device.

Return Value

This function does not return any value.

Example

```
sdlc_cfg_t current_cfg_values;
SDLC_get_cfg (&g_CoreSDLC1,&current_cfg_values);
```

SDLC_set_mode

Prototype

```
void SDLC_set_mode
(
    sdlc_instance_t * this_sdlc,
    uint8_t mode
);
```

Description

SDLC_set_mode() sets the mode of the CoreSDLC device. Three modes, Normal (0x00), Raw Transmit (0x01) and Raw receive (0x02) can be set using this API.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a sdlc*_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

mode

Mode = 0x00	=>	Normal mode
Mode = 0x01	=>	Raw Tx Mode
Mode = 0x02	=>	Raw Rx mode

Return Value

This function does not return any value.

SDLC_enable_rx

Prototype

```
void
SDLC_enable_rx
(
    sdlc_instance_t * this_sdlc
);
```

Description

SDLC_enable_rx() is used to enable the reception.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

Return Value

This function does not return any value.

SDLC_disable_rx

Prototype

```
void
SDLC_disable_rx
(
    sdlc_instance_t * this_sdlc
);
```

Description

SDLC_disable_rx() is used to disable the reception.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

Return Value

This function does not return any value.

SDLC_check_rx_line_idle_status

Prototype

```
uint8_t
SDLC_check_rx_line_idle
(
    sdlc_instance_t * this_sdlc
);
```

Description

SDLC_check_rx_line_idle() is used to know whether receive line is idle (15 consecutive '1' values are received on rxd) or not idle.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

Return Value

This function returns non-zero value, if Receive line is idle. Zero value is returned when receive line is not idle.

SDLC_is_tx_complete

Prototype

```
uint8_t
SDLC_is_tx_complete
(
    sdlc_instance_t * this_sdlc
);
```

Description

The *SDLC_is_tx_complete()* function is used to find out if the interrupt driven transmit previously initiated through a call to *SDLC_tx_data()* is complete. This is typically used to find out when it is safe to reuse or release the memory buffer holding transmit data. This routine returns the information about the transmit state.

Parameters

this_sdlc

The *this_sdlc parameter* is a pointer to a *sdlc_instance_t* structure that holds all data regarding this instance of the CoreSDLC.

Return Value

This function return a non-zero value if transmit has completed, otherwise it returns zero.

Product Support

Microsemi SoC Products Group backs its products with various support services, including Customer Service, Customer Technical Support Center, a website, electronic mail, and worldwide sales offices. This appendix contains information about contacting Microsemi SoC Products Group and using these support services.

Customer Service

Contact Customer Service for non-technical product support, such as product pricing, product upgrades, update information, order status, and authorization.

From North America, call **800.262.1060**
From the rest of the world, call **650.318.4460**
Fax, from anywhere in the world **408.643.6913**

Customer Technical Support Center

Microsemi SoC Products Group staffs its Customer Technical Support Center with highly skilled engineers who can help answer your hardware, software, and design questions about Microsemi SoC Products. The Customer Technical Support Center spends a great deal of time creating application notes, answers to common design cycle questions, documentation of known issues and various FAQs. So, before you contact us, please visit our online resources. It is very likely we have already answered your questions.

Technical Support

Visit the Microsemi SoC Products Group Customer Support website for more information and support (http://www.microsemi.com/soc/support/search/default.aspx). Many answers available on the searchable web resource include diagrams, illustrations, and links to other resources on website.

Website

You can browse a variety of technical and non-technical information on the Microsemi SoC Products Group home page, at http://www.microsemi.com/soc/

Contacting the Customer Technical Support Center

Highly skilled engineers staff the Technical Support Center. The Technical Support Center can be contacted by email or through the Microsemi SoC Products Group website.

Email

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

The technical support email address is soc_tech@microsemi.com.

My Cases

Microsemi SoC Products Group customers may submit and track technical cases online by going to My Cases.

Outside the U.S.

Customers needing assistance outside the US time zones can either contact technical support via email (soc_tech@microsemi.com) or contact a local sales office. Sales office listings can be found at www.microsemi.com/soc/company/contact/default.aspx.

ITAR Technical Support

For technical support on RH and RT FPGAs that are regulated by International Traffic in Arms Regulations (ITAR), contact us via soc_tech_itar@microsemi.com. Alternatively, within My Cases, select **Yes** in the ITAR drop-down list. For a complete list of ITAR-regulated Microsemi FPGAs, visit the ITAR web page.

Microsemi Corporate Headquarters
2381 Morse Avenue, Irvine, CA 92614
Phone; 949.221.7100 · Fax: 949.756.0308
www.microsemi.com

5-02-00284-0/03.15