

UART

Home

Data Structures

Data Structure Index

Data Fields

Data Structures

Here are the data structures with brief descriptions:

[G UART](#)

Handler structure with pointers to dynamic and static parameters

[G UART_CONFIG](#)

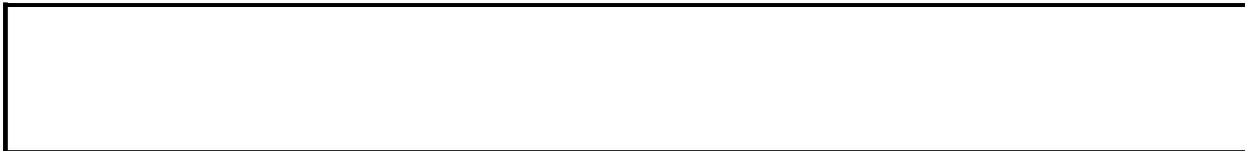
Structure for holding the configuration parameters of [UART](#) channel

[G UART_RUNTIME](#)

Structure to hold the dynamic variables for the [UART](#) communication

[G UART_TX_CONFIG](#)

Structure for transmit pin configuration



UART

[Home](#)

[Data Structures](#)

[Data Structure Index](#)

[Data Fields](#)

[Data Fields](#)

UART Struct Reference

Detailed Description

Handler structure with pointers to dynamic and static parameters.

Definition at line **265** of file **UART.h**.

```
#include <UART.h>
```

Data Fields

XMC_USIC_CH_t *const **channel**

const **UART_CONFIG_t** *const **config**

UART_RUNTIME_t *const **runtime**

Field Documentation

XMC_USIC_CH_t* const UART::channel

USIC channel

Definition at line [267](#) of file [UART.h](#).

Referenced by [UART_ClearFlag\(\)](#), [UART_ClearRXFIFOStatus\(\)](#), [UART_ClearTXFIFOStatus\(\)](#), [UART_DisableEvent\(\)](#), [UART_EnableEvent\(\)](#), [UART_GetFlagStatus\(\)](#), [UART_GetReceivedWord\(\)](#), [UART_GetRXFIFOStatus\(\)](#), [UART_GetTXFIFOStatus\(\)](#), [UART_IsRXFIFOEmpty\(\)](#), [UART_IsTXFIFOFull\(\)](#), [UART_SetBaudrate\(\)](#), [UART_SetRXFIFOTriggerLimit\(\)](#), [UART_SetTXFIFOTriggerLimit\(\)](#), and [UART_TransmitWord\(\)](#).

const UART_CONFIG_t* const UART::config

UART configuration structure pointer

Definition at line [268](#) of file [UART.h](#).

Referenced by [UART_Init\(\)](#), [UART_Receive\(\)](#), [UART_SetBaudrate\(\)](#), [UART_SetRXFIFOTriggerLimit\(\)](#), [UART_SetTXFIFOTriggerLimit\(\)](#), and [UART_Transmit\(\)](#).

UART_RUNTIME_t* const UART::runtime

Pointer to the structure holding all variables,

that can change at runtime

Definition at line [269](#) of file [UART.h](#).

Referenced by [UART_Init\(\)](#), [UART_IsRxBusy\(\)](#), [UART_IsTxBusy\(\)](#), and [UART_SetBaudrate\(\)](#).

The documentation for this struct was generated from the following file:

- [UART.h](#)
-
-

UART

[Home](#)

[Data Structures](#)

[Data Structure Index](#)

[Data Fields](#)

[Data Fields](#)

UART_CONFIG Struct Reference

Detailed Description

Structure for holding the configuration parameters of **UART** channel.

Definition at line **203** of file **UART.h**.

```
#include <UART.h>
```

Data Fields

const XMC_UART_CH_CONFIG_t *const **channel_config**

UART_init_handler **fptr_uart_config**

UART_cbhandler **sync_error_cbhandler**

UART_cbhandler **rx_noise_error_cbhandler**

UART_cbhandler **format_error_bit0_cbhandler**

UART_cbhandler **format_error_bit1_cbhandler**

UART_cbhandler **collision_error_cbhandler**

const **UART_TX_CONFIG_t** * **tx_pin_config**

UART_MODE_t **mode**

UART_TRANSFER_MODE_t **transmit_mode**

UART_TRANSFER_MODE_t **receive_mode**

XMC_USIC_CH_FIFO_SIZE_t **tx_fifo_size**

XMC_USIC_CH_FIFO_SIZE_t **rx_fifo_size**

uint8_t **tx_sr**

Field Documentation

const XMC_UART_CH_CONFIG_t* const UART_CONFIG::channel_

Basic UART configuration from the GUI with baud
, data bits,

frame length, stop bits and parity

Definition at line [205](#) of file [UART.h](#).

UART_cbhandler UART_CONFIG::collision_error_cbhandler

Function pointer to hold the callback
function pointer,

called when collision error is detected

Definition at line [233](#) of file [UART.h](#).

UART_cbhandler UART_CONFIG::format_error_bit0_cbhandler

Function pointer to hold the callback f
unction pointer,

called when format error with stop bit 0 is detected.

Definition at line [229](#) of file [UART.h](#).

UART_cbhandler UART_CONFIG::format_error_bit1_cbhandler

Function pointer to hold the callback f
unction pointer,

called when format error with stop bit 1 is detected.

Definition at line **231** of file **UART.h**.

UART_init_handler UART_CONFIG::fptr_uart_config

Function pointer to configure the MUX values

Definition at line **216** of file **UART.h**.

Referenced by **UART_Init()**.

UART_MODE_t UART_CONFIG::mode

UART operation mode

Definition at line **237** of file **UART.h**.

Referenced by **UART_SetBaudrate()**.

UART_TRANSFER_MODE_t UART_CONFIG::receive_mode

Mode used for receiving data.
Data can be received using

interrupt, DMA or direct(using polling or external APP connection.)

Definition at line **240** of file **UART.h**.

Referenced by **UART_Receive()**.

XMC_USIC_CH_FIFO_SIZE_t UART_CONFIG::rx_fifo_size

Receive FIFO size configuration

Definition at line **243** of file **UART.h**.

Referenced by [UART_SetRXFIFOTriggerLimit\(\)](#).

UART_cbhandler UART_CONFIG::rx_noise_error_cbhandler

Function pointer to hold the callback function pointer,

called when receiver noise is detected

Definition at line [227](#) of file [UART.h](#).

UART_cbhandler UART_CONFIG::sync_error_cbhandler

Function pointer to hold the callback function pointer,

called when synchronization break detected.

Definition at line [225](#) of file [UART.h](#).

UART_TRANSFER_MODE_t UART_CONFIG::transmit_mode

Mode used for transmitting data . Data can be transmitted using

interrupt, DMA or direct(using polling or external APP connection.)

Definition at line [238](#) of file [UART.h](#).

Referenced by [UART_Transmit\(\)](#).

XMC_USIC_CH_FIFO_SIZE_t UART_CONFIG::tx_fifo_size

Transmit FIFO size configuration

Definition at line [242](#) of file [UART.h](#).

Referenced by [UART_SetTXFIFOTriggerLimit\(\)](#).

const [UART_TX_CONFIG_t](#)* [UART_CONFIG::tx_pin_config](#)

Transmit pin configuration to be used during initialization

and while changing baudrate.

Definition at line [235](#) of file [UART.h](#).

Referenced by [UART_SetBaudrate\(\)](#).

uint8_t [UART_CONFIG::tx_sr](#)

Service request number assigned to transmit interrupt

Definition at line [244](#) of file [UART.h](#).

The documentation for this struct was generated from the following file:

- [UART.h](#)



UART

[Home](#)

[Data Structures](#)

[Data Structure Index](#)

[Data Fields](#)

[Data Fields](#)

UART_RUNTIME Struct Reference

Detailed Description

Structure to hold the dynamic variables for the **UART** communication.

Definition at line **250** of file **UART.h**.

```
#include <UART.h>
```

Data Fields

uint8_t* **tx_data**

uint8_t* **rx_data**

uint32_t **tx_data_count**

uint32_t **tx_data_index**

uint32_t **rx_data_count**

uint32_t **rx_data_index**

volatile bool **tx_busy**

volatile bool **rx_busy**

Field Documentation

volatile bool UART_RUNTIME::rx_busy

Status flag to indicate busy when a reception is assigned

Definition at line [259](#) of file [UART.h](#).

Referenced by [UART_IsRxBusy\(\)](#), and [UART_SetBaudrate\(\)](#).

uint8_t* UART_RUNTIME::rx_data

Pointer to the receive data buffer

Definition at line [253](#) of file [UART.h](#).

uint32_t UART_RUNTIME::rx_data_count

Number of bytes of data to be received

Definition at line [256](#) of file [UART.h](#).

uint32_t UART_RUNTIME::rx_data_index

Indicates the number of bytes currently available in the rx_data buffer

Definition at line [257](#) of file [UART.h](#).

volatile bool UART_RUNTIME::tx_busy

Status flag to indicate busy when a transmission is assigned

Definition at line [258](#) of file [UART.h](#).

Referenced by [UART_IsTxBusy\(\)](#), and [UART_SetBaudrate\(\)](#).

uint8_t* UART_RUNTIME::tx_data

Pointer to the transmit data buffer

Definition at line [252](#) of file [UART.h](#).

uint32_t UART_RUNTIME::tx_data_count

Number of bytes of data to be transmitted

Definition at line [254](#) of file [UART.h](#).

uint32_t UART_RUNTIME::tx_data_index

Index to the byte to be transmitted next in the tx_data buffer

Definition at line [255](#) of file [UART.h](#).

The documentation for this struct was generated from the following file:

- [UART.h](#)



UART

[Home](#)

[Data Structures](#)

[Data Structure Index](#)

[Data Fields](#)

[Data Fields](#)

UART_TX_CONFIG **Struct Reference**

Detailed Description

Structure for transmit pin configuration.

Definition at line **182** of file **UART.h**.

```
#include <UART.h>
```

Data Fields

XMC_GPIO_PORT_t *const **port**

const uint8_t **pin**

const XMC_GPIO_CONFIG_t *const **config**

Field Documentation

const XMC_GPIO_CONFIG_t* const UART_TX_CONFIG::config

Pin configuration structure

Definition at line [186](#) of file [UART.h](#).

Referenced by [UART_SetBaudrate\(\)](#).

const uint8_t UART_TX_CONFIG::pin

Pin number in the port

Definition at line [185](#) of file [UART.h](#).

Referenced by [UART_SetBaudrate\(\)](#).

XMC_GPIO_PORT_t* const UART_TX_CONFIG::port

Pointer to the GPIO port base address

Definition at line [184](#) of file [UART.h](#).

Referenced by [UART_SetBaudrate\(\)](#).

The documentation for this struct was generated from the following file:

- [UART.h](#)



UART

Home

Data Structures

Data Structure Index

Data Fields

Data Structure Index

U

U

UART_CONFIG
UART_RUNTIME

UART_TX_CONFIG

UART

U

UART

Home		
Data Structures	Data Structure Index	Data Fields
All	Variables	

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

- channel : [UART](#)
- channel_config : [UART_CONFIG](#)
- collision_error_cbhandler : [UART_CONFIG](#)
- config : [UART_TX_CONFIG](#) , [UART](#)
- format_error_bit0_cbhandler : [UART_CONFIG](#)
- format_error_bit1_cbhandler : [UART_CONFIG](#)
- fptr_uart_config : [UART_CONFIG](#)
- mode : [UART_CONFIG](#)
- pin : [UART_TX_CONFIG](#)
- port : [UART_TX_CONFIG](#)
- receive_mode : [UART_CONFIG](#)
- runtime : [UART](#)
- rx_busy : [UART_RUNTIME](#)
- rx_data : [UART_RUNTIME](#)
- rx_data_count : [UART_RUNTIME](#)
- rx_data_index : [UART_RUNTIME](#)
- rx_fifo_size : [UART_CONFIG](#)
- rx_noise_error_cbhandler : [UART_CONFIG](#)
- sync_error_cbhandler : [UART_CONFIG](#)
- transmit_mode : [UART_CONFIG](#)
- tx_busy : [UART_RUNTIME](#)
- tx_data : [UART_RUNTIME](#)
- tx_data_count : [UART_RUNTIME](#)
- tx_data_index : [UART_RUNTIME](#)
- tx_fifo_size : [UART_CONFIG](#)
- tx_pin_config : [UART_CONFIG](#)
- tx_sr : [UART_CONFIG](#)

--

UART

Home		
Data Structures	Data Structure Index	Data Fields
All	Variables	

- channel : [UART](#)
 - channel_config : [UART_CONFIG](#)
 - collision_error_cbhandler : [UART_CONFIG](#)
 - config : [UART_TX_CONFIG](#) , [UART](#)
 - format_error_bit0_cbhandler : [UART_CONFIG](#)
 - format_error_bit1_cbhandler : [UART_CONFIG](#)
 - fptr_uart_config : [UART_CONFIG](#)
 - mode : [UART_CONFIG](#)
 - pin : [UART_TX_CONFIG](#)
 - port : [UART_TX_CONFIG](#)
 - receive_mode : [UART_CONFIG](#)
 - runtime : [UART](#)
 - rx_busy : [UART_RUNTIME](#)
 - rx_data : [UART_RUNTIME](#)
 - rx_data_count : [UART_RUNTIME](#)
 - rx_data_index : [UART_RUNTIME](#)
 - rx_fifo_size : [UART_CONFIG](#)
 - rx_noise_error_cbhandler : [UART_CONFIG](#)
 - sync_error_cbhandler : [UART_CONFIG](#)
 - transmit_mode : [UART_CONFIG](#)
 - tx_busy : [UART_RUNTIME](#)
 - tx_data : [UART_RUNTIME](#)
 - tx_data_count : [UART_RUNTIME](#)
 - tx_data_index : [UART_RUNTIME](#)
 - tx_fifo_size : [UART_CONFIG](#)
 - tx_pin_config : [UART_CONFIG](#)
 - tx_sr : [UART_CONFIG](#)
-



UART

Home

File List

Globals

File List

Here is a list of all documented files with brief descriptions:

 [UART.c](#)

 [UART.h](#)



UART

Home		
File List	Globals	
Code		

[Functions](#)

UART.c File Reference

Detailed Description

Date

2015-12-17

NOTE: This file is generated by DAVE. Any manual modification done to this file will be lost when the code is regenerated.

Definition in file [UART.c](#).

```
#include "uart.h"
```

Functions

DAVE_APP_VERSION_t **UART_GetAppVersion** ()
Get the **UART** APP version. [More...](#)

UART_STATUS_t **UART_Init** (const **UART_t** *const handle)
Initializes the **UART** module as per the configuration made in UI. [More...](#)

UART_STATUS_t **UART_Transmit** (const **UART_t** *const handle, uint8_t *data_ptr, uint32_t count)
Registers a request for transmitting data over **UART** channel. [More...](#)

UART_STATUS_t **UART_Receive** (const **UART_t** *const handle, uint8_t *data_ptr, uint32_t count)
Registers a request for receiving data over **UART** channel. [More...](#)

UART_STATUS_t **UART_SetBaudrate** (const **UART_t** *handle, uint32_t baud, uint32_t oversampling)
Changes the baudrate of **UART** channel. [More...](#)

Function Documentation

UART_STATUS_t UART_Init (const **UART_t** *const **handle**)

Initializes the **UART** module as per the configuration made in UI.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

UART_STATUS_t: Status of **UART** driver initialization.

UART_STATUS_SUCCESS - on successful initialization.

UART_STATUS_FAILURE - if initialization fails.

UART_STATUS_BUSY - if **UART** channel is busy.

Description:

Initializes IO pins used for the **UART** communication, configures USIC registers based on the settings provided in the GUI.

Calculates divider values PDIV and STEP for a precise baudrate. It also enables configured interrupt flags and service request values.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
int main(void)
{
    UART_STATUS_t init_status;
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        while(1U)
        {
        }
    }
}
```

```

}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **143** of file **UART.c**.

References **UART::config**, **UART_CONFIG::fptr_uart_config**, **UART::runtime**, and **UART_STATUS_SUCCESS**.

```

UART_STATUS_t UART_Receive ( const UART_t *const handle,
                               uint8_t * data_ptr,
                               uint32_t count
                               )

```

Registers a request for receiving data over **UART** channel.

Parameters

- handle** Pointer to UART_t handle structure
- data_ptr** Pointer to data of type uint8_t.
- count** Total no of bytes to be received.
Range: minimum= 1, maximum= maximum value supported by type uint32_t.

Returns

- UART_STATUS_t: Status for receive request.
- UART_STATUS_SUCCESS** if the request is accepted.
- UART_STATUS_BUSY** if a reception is in progress.
- UART_STATUS_BUFFER_INVALID** if the data_ptr is NULL or count is 0.

Description:

Data will be received asynchronously. After the requested number of data bytes are received, optionally, the user configured callback function will be executed. Data reception is accomplished using the receive mode selected in the UI.

Interrupt:

Based on the UI configuration, either standard receive buffer(RBUF) or receive FIFO(OUT) is used for data reception. An interrupt is configured for reading received data from the bus. This function only registers a request to receive a number of data bytes from a USIC channel. If FIFO is configured for reception, the FIFO limit is dynamically configured to optimally utilize the CPU load. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. When all the requested number of data bytes are received, the configured callback function will be executed. If a callback function is not configured, the user has to poll for the value of the variable, *handle->runtime->rx_busy* to be false. The value is updated to *false* when all the requested number of data bytes are received.

DMA:

DMA mode is available only in XMC4x family of microcontrollers. In this mode, a DMA channel is configured for receiving data from standard receive buffer(RBUF) to the user buffer. By calling this API, the DMA channel destination address is configured to the user buffer and the channel is enabled. FIFO will not be used when the receive mode is DMA. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. When all the requested number of data bytes are received, the configured callback function will be executed. If a callback function is not configured, the user has to poll for the value of the variable, *handle->runtime->rx_busy* to be false. The value is updated to *false* when all the requested number of data bytes are received.

Direct

In Direct receive mode, neither interrupt nor DMA is used. The API polls the receive flag to read the received data and waits for all the requested number of bytes to be received. Based on FIFO configuration, either RBUF or OUT register is used for reading received data. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. **Note:** *In Direct mode, the API blocks the CPU until the count of bytes requested is received. If this behaviour is not desired, use other APIs like [UART_GetReceivedWord](#), [UART_GetProtocolStatus](#) etc.*

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Pre-condition:
//Receive mode should be configured as "Direct".
//Description:
//Transmits 10 bytes of data after receiving 10 bytes of data.
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t ReadData[10];
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        while(1)
        {
            //Receive 10 bytes of data
            if(UART_Receive(&UART_0, ReadData, 10) ==
            UART_STATUS_SUCCESS)
            {
                //Retransmit the received 10 bytes
                UART_Transmit(&UART_0, ReadData, 10);
            }
        }
    }
}
```

```

else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **211** of file **UART.c**.

References **UART::config**, **UART_CONFIG::receive_mode**, **UART_STATUS_MODE_MISMATCH**, **UART_TRANSFER_MODE_DIRECT**, **UART_TRANSFER_MODE_DMA**, and **UART_TRANSFER_MODE_INTERRUPT**.

```

UART_STATUS_t UART_SetBaudrate ( const UART_t * handle,
                                uint32_t          baud,
                                uint32_t          oversampling
                                )

```

Changes the baudrate of **UART** channel.

Parameters

UART_t * Pointer to the **UART** APP handle.
baud Value of new baudrate.
oversampling Number of samples to be considered for each symbol. 16 is the standard value.

Returns

UART_STATUS_t **UART_STATUS_SUCCESS** if baudrate changed successfully. **UART_STATUS_FAILURE** if baudrate could not be changed.

Description:

The function stops the channel, calculates the clock divider values to achieve the desired baudrate. Sets the divider values and reconfigures the channel as per the configuration in the UI. The channel is enabled at the end of configuration.

Example Usage: *Please disable the receive FIFO in the 'Advanced Settings' tab*

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Precondition:
//Disable receive FIFO in the Advanced settings tab.
//Description:
//Waits for user input of new baudrate value. Input is recognized after
line feed is provided.
//Value is set as the new baudrate and the application waits for any
key to be pressed.
//This helps in reconfiguring the terminal application to the newly set
baudrate. On receiving
//new character, message indicating the successful baudrate change
will be displayed using updated
//value of baudrate.
const uint8_t send_askbaud[] = "Please enter desired baudrate:";
const uint8_t send_data[] = "\nPress 'y' to change baudrate to desired
value:";
const uint8_t send_invalid[] = "\nInvalid value!!";
const uint8_t send_success[] = "\nWe made it...Baudrate changed
successfully :-).\n\n";
uint8_t rec_data[11];
int main(void)
{
DAVE_STATUS_t status;
uint32_t baud;
status = DAVE_Init(); // Initialization of DAVE Apps
if(status == DAVE_STATUS_FAILURE)
{
```

```

XMC_DEBUG(("DAVE Apps initialization failed with status %d\n",
status));
while(1U)
{
}
while(1U)
{
UART_Transmit(&UART_0, send_askbaud, sizeof(send_askbaud)-1);
UART_Receive(&UART_0, rec_data, 10);
while(UART_0.runtime->tx_busy);
while(UART_0.runtime->rx_busy)
{
//If user enters newline character, accept the value
if((UART_0.runtime->rx_data_index > 0) && (UART_0.runtime-
>rx_data[UART_0.runtime->rx_data_index - 1] == 0x0a))
{
//End reception of data on finding newline character
UART_AbortReceive(&UART_0);
}
}
//Add end of string character to the last location
rec_data[UART_0.runtime->rx_data_index] = 0;
//Convert the entered string to number.
baud = atoi(rec_data);
//If the conversion is successful, set the baudrate.
if(baud > 0)
{
//Set the baudrate to USIC channel
if(UART_SetBaudrate(&UART_0, baud, 16) ==
UART_STATUS_SUCCESS)
{
//After changing the baudrate successfully,
//Wait for user to enter a character.
//This wait gives time for the user to change
//the baudrate of the terminal tool used.
UART_Receive(&UART_0, rec_data, 1);

```

```

while(UART_0.runtime->rx_busy);
UART_Transmit(&UART_0, send_success, sizeof(send_success)-1);
}
else
{
UART_Transmit(&UART_0, send_invalid, sizeof(send_invalid)-1);
}
}
else
{
UART_Transmit(&UART_0, send_invalid, sizeof(send_invalid)-1);
}
while(UART_0.runtime->tx_busy);
}
}

```

Definition at line **388** of file **UART.c**.

References **UART::channel**, **UART_TX_CONFIG::config**, **UART::config**, **UART_CONFIG::mode**, **UART_TX_CONFIG::pin**, **UART_TX_CONFIG::port**, **UART::runtime**, **UART_RUNTIME::rx_busy**, **UART_RUNTIME::tx_busy**, **UART_CONFIG::tx_pin_config**, **UART_MODE_LOOPBACK**, **UART_STATUS_BUSY**, and **UART_STATUS_SUCCESS**.

```

UART_STATUS_t UART_Transmit ( const UART_t *const handle,
                                uint8_t * data_ptr,
                                uint32_t count
                                )

```

Registers a request for transmitting data over **UART** channel.

Parameters

handle **UART** APP handle pointer of type **UART_t**
data_ptr Pointer to data of type **uint8_t**.

count Total no of words to be transmitted.
Range: minimum= 1, maximum= maximum supported by uint32_t.

Returns

UART_STATUS_t: Status of transmit request.

UART_STATUS_SUCCESS if the request is accepted.

UART_STATUS_BUSY if a transmission is in progress.

UART_STATUS_BUFFER_INVALID if the data_ptr is NULL or count is 0.

Imp Note: Return value should be validated by user to ensure that the request is registered.

Description:

Transmits data using the **UART** channel. Transmission is accomplished using the transmit mode as configured in the UI.

Interrupt:

The data transmission is accomplished using transmit interrupt. User can configure a callback function in the APP UI. When the data is fully transmitted, the callback function will be executed. If transmit FIFO is enabled, the trigger limit is set to 1. So the transmit interrupt will be generated when all the data in FIFO is moved out of FIFO. The APP handle's runtime structure is used to store the data pointer, count, data index and status of transmission. This function only registers a data transmission request if there is no active transmission in progress. Actual data transmission happens in the transmit interrupt service routine. A trigger is generated for the transmit interrupt to start loading the data to the transmit buffer. If transmit FIFO is configured, the data is filled into the FIFO. Transmit interrupt will be generated subsequently when the transmit FIFO is empty. At this point of time, if there is some more data to be transmitted, it is loaded to the FIFO again. When FIFO is not enabled, data is transmitted one byte at a time. On transmission of each byte an interrupt is generated and the next byte is transmitted in the interrupt service routine. Callback function is executed when all the data

bytes are transmitted. If a callback function is not configured, user has to poll for the value of `tx_busy` flag of the APP handle structure(`handle->runtime->tx_busy`) to check for the completion of data transmission.

DMA:

A DMA channel is configured to provide data to the **UART** transmit buffer. This removes the load off the CPU. This API will only configure and enable the DMA channel by specifying the data buffer and count of bytes to transmit. Rest is taken care without the CPU's intervention. User can configure a callback function in the APP UI. When the transmission is complete, the callback function will be executed. FIFO will not be used in DMA mode. Transmit buffer interrupt is configured for triggering the DMA channel. So each byte is transmitted in the background through the DMA channel. If the callback function is not configured, `handle->runtime->tx_busy` flag can be checked to verify if the transmission is complete. **Direct:**

Data will be transmitted using polling method. Status flags are used to check if data can be transmitted. **Note:** *In Direct mode, the API blocks the CPU until the count of bytes requested is transmitted. If this behaviour is not desired, use other APIs like **UART_TransmitWord**, **UART_GetProtocolStatus** etc.*

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Pre-condition:
//Transmit mode should be configured as "Direct".
//Description:
//Transmits the string "Infineon".
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t Send_Data[] = "Infineon";
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
```

```
{
//Transmit the string.
UART_Transmit(&UART_0, Send_Data, sizeof(Send_Data)-1);
while(1)
{
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **170** of file **UART.c**.

References **UART::config**, **UART_CONFIG::transmit_mode**,
UART_STATUS_MODE_MISMATCH,
UART_TRANSFER_MODE_DIRECT,
UART_TRANSFER_MODE_DMA, and
UART_TRANSFER_MODE_INTERRUPT.

[Go to the source code of this file.](#)

UART

Home		
File List	Globals	
Code		

[Data Structures](#) | [Typedefs](#)

UART.h File Reference

Detailed Description

Date

2015-12-17

NOTE: This file is generated by DAVE. Any manual modification done to this file will be lost when the code is regenerated.

Definition in file [UART.h](#).

```
#include <xmc_scu.h> #include <xmc_gpio.h>
#include <xmc_uart.h>
#include <DAVE_common.h>
#include "uart_conf.h"
#include "uart_extern.h"
```

Data Structures

struct **UART_TX_CONFIG**
Structure for transmit pin configuration. [More...](#)

struct **UART_CONFIG**
Structure for holding the configuration parameters of **UART** channel. [More...](#)

struct **UART_RUNTIME**
Structure to hold the dynamic variables for the **UART** communication. [More...](#)

struct **UART**
Handler structure with pointers to dynamic and static parameters. [More...](#)

Typedefs

typedef void(*	UART_cbhandler)(void)	Function pointer used for c function.	
typedef	UART_STATUS_t (*	UART_init_handler)(void)	Function pointer used for i function.
typedef XMC_UART_CH_STATUS_FLAG_t	UART_PROTOCOL_STA	Wrapper typedefinition for XMC_UART_PROTOCOL	
typedef struct	UART_TX_CONFIG	UART_TX_CONFIG_t	Structure for transmit pin configuration.
typedef struct	UART_CONFIG	UART_CONFIG_t	Structure for holding the c parameters of UART chan
typedef struct	UART_RUNTIME	UART_RUNTIME_t	Structure to hold the dyna variables for the UART communication.
typedef struct	UART	UART_t	Handler structure with poin dynamic and static param

Functions

DAVE_APP_VERSION_t	UART_GetAppVersion (void) Get the UART APP version.
UART_STATUS_t	UART_Init (const UART_t *) Initializes the UART module configuration made in UI. More...
UART_STATUS_t	UART_Receive (const UART_HANDLE_t , uint8_t *data_ptr, uint8_t *length_ptr) Registers a request for receive over UART channel. More...
UART_STATUS_t	UART_Transmit (const UART_HANDLE_t , uint8_t *data_ptr, uint8_t *length_ptr) Registers a request for transmit over UART channel. More...
UART_STATUS_t	UART_SetBaudrate (const UART_HANDLE_t , uint32_t baud, uint8_t oversampling) Changes the baudrate of UART . More...
__STATIC_INLINE uint32_t	UART_GetTXFIFOStatus (const UART_HANDLE_t) Gets the transmit FIFO level. More...
__STATIC_INLINE uint8_t	UART_GetReceivedWord (const UART_HANDLE_t) Provides the received data from the UART .

buffer. [More...](#)

`__STATIC_INLINE void` **UART_TransmitWord** (const `*const handle`, `uint8_t data`)
Transmits a word of data. [More...](#)

`__STATIC_INLINE void` **UART_EnableEvent** (const `*const handle`, `uint32_t event`)
Enables the selected protocol interrupt generation. [More...](#)

`__STATIC_INLINE void` **UART_DisableEvent** (const `*const handle`, `uint32_t event`)
Disables selected events from interrupt. [More...](#)

`__STATIC_INLINE bool` **UART_IsTXFIFOFull** (const `*const handle`)
Checks if the transmit FIFO is full.

`__STATIC_INLINE bool` **UART_IsRXFIFOEmpty** (const `*const handle`)
Checks if the receive FIFO is empty. [More...](#)

`__STATIC_INLINE void` **UART_SetTXFIFOTriggerLimit** (`UART_t *const handle`, `uint8_t limit`)
Configures trigger limit for the transmit FIFO. [More...](#)

`__STATIC_INLINE void` **UART_SetRXFIFOTriggerLimit** (`UART_t *const handle`, `uint8_t limit`)
Configures trigger limit for the receive FIFO. [More...](#)

`__STATIC_INLINE uint32_t` **UART_GetRXFIFOStatus** (
*const handle)
Gets the status of event flag receive FIFO. [More...](#)

`__STATIC_INLINE void` **UART_ClearTXFIFOStatus**
UART_t *const handle, const
flag)
Function clears the specified flag related to transmit FIFO

`__STATIC_INLINE void` **UART_ClearRXFIFOStatus**
UART_t *const handle, const
flag)
Function clears the specified flag related to receive FIFO.
used to clear the status of send receive buffer interrupt, alter
receive buffer interrupt and receive error interrupt flags. [More...](#)

`__STATIC_INLINE uint32_t` **UART_GetFlagStatus** (con
*const handle, uint32_t proto
Provides the status of protocol. [More...](#)

`__STATIC_INLINE void` **UART_ClearFlag** (const **UA**
handle, const uint32_t proto
Clears the event status in the register(PSR_ASCMode). [M](#)

`__STATIC_INLINE bool` **UART_IsTxBusy** (const **UA**
handle)
Checks if the transmission is

More...

`__STATIC_INLINE` bool **UART_IsRxBusy** (const **UART_HandleTypeDef**)
Checks if data reception is in progress.
More...

enum **UART_STATUS** {
 UART_STATUS_SUCCESS,
 UART_STATUS_FAILURE,
 UART_STATUS_BUSY,
 UART_STATUS_BUFFER_FULL,
 UART_STATUS_MODE_MISMATCH
}
Enum to describe the possible status values, returned by **UART_A**

enum **UART_MODE** {
 UART_MODE_FULLDUPLEX,
 UART_MODE_HALFDUPLEX,
 UART_MODE_LOOPBACK
}
Enum used to describe the mode of operation. More...

enum **UART_EVENT** {
 UART_EVENT_SYNC_BREAK,
 UART_EVENT_RX_NOISE,
 UART_EVENT_FORMAT_ERROR,
 UART_EVENT_FORMAT_ERROR,
 UART_EVENT_COLLISION,
 UART_EVENT_MAX
}
Enum used to identify **UART** event callback function. More...

enum **UART_TRANSFER_MODE**
UART_TRANSFER_MODE
UART_TRANSFER_MODE
UART_TRANSFER_MODE
Enum used to identify the tra
used for either transmit or re
function. [More...](#)

typedef enum **UART_STATUS** **UART_STATUS_t**
Enum to describe the possib
values, returned by **UART A**

typedef enum **UART_MODE** **UART_MODE_t**
Enum used to describe the U
of operation.

typedef enum **UART_EVENT** **UART_EVENT_t**
Enum used to identify **UART**
event callback function.

typedef enum **UART_TRANSFER_MODE** **UART_TRANSFER_MODE**
Enum used to identify the tra
used for either transmit or re
function.

[Go to the source code of this file.](#)



UART

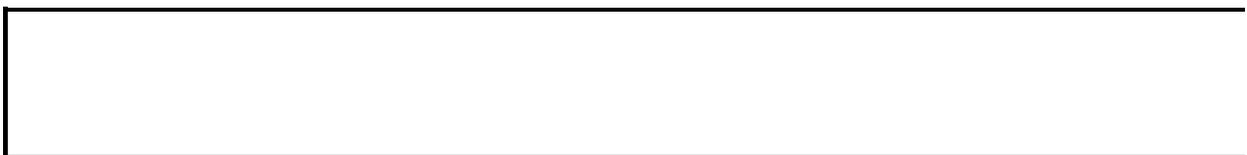
Home					
File List	Globals				
All	Functions	Typedefs	Enumerations	Enumerator	
u					

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- u -

- [UART_cbhandler](#) : [UART.h](#)
- [UART_ClearFlag\(\)](#) : [UART.h](#)
- [UART_ClearRXFIFOStatus\(\)](#) : [UART.h](#)
- [UART_ClearTXFIFOStatus\(\)](#) : [UART.h](#)
- [UART_CONFIG_t](#) : [UART.h](#)
- [UART_DisableEvent\(\)](#) : [UART.h](#)
- [UART_EnableEvent\(\)](#) : [UART.h](#)
- [UART_EVENT](#) : [UART.h](#)
- [UART_EVENT_COLLISION](#) : [UART.h](#)
- [UART_EVENT_FORMAT_ERR0](#) : [UART.h](#)
- [UART_EVENT_FORMAT_ERR1](#) : [UART.h](#)
- [UART_EVENT_MAX](#) : [UART.h](#)
- [UART_EVENT_RX_NOISE](#) : [UART.h](#)
- [UART_EVENT_SYNC_BRK](#) : [UART.h](#)
- [UART_EVENT_t](#) : [UART.h](#)
- [UART_GetAppVersion\(\)](#) : [UART.c](#) , [UART.h](#)
- [UART_GetFlagStatus\(\)](#) : [UART.h](#)
- [UART_GetReceivedWord\(\)](#) : [UART.h](#)
- [UART_GetRXFIFOStatus\(\)](#) : [UART.h](#)
- [UART_GetTXFIFOStatus\(\)](#) : [UART.h](#)
- [UART_Init\(\)](#) : [UART.c](#) , [UART.h](#)
- [UART_init_handler](#) : [UART.h](#)
- [UART_IsRxBusy\(\)](#) : [UART.h](#)

- UART_IsRXFIFOEmpty() : **UART.h**
- UART_IsTxBusy() : **UART.h**
- UART_IsTXFIFOFull() : **UART.h**
- UART_MODE : **UART.h**
- UART_MODE_FULLDUPLEX : **UART.h**
- UART_MODE_HALFDUPLEX : **UART.h**
- UART_MODE_LOOPBACK : **UART.h**
- UART_MODE_t : **UART.h**
- UART_PROTOCOL_STATUS_t : **UART.h**
- UART_Receive() : **UART.c** , **UART.h**
- UART_RUNTIME_t : **UART.h**
- UART_SetBaudrate() : **UART.c** , **UART.h**
- UART_SetRXFIFOTriggerLimit() : **UART.h**
- UART_SetTXFIFOTriggerLimit() : **UART.h**
- UART_STATUS : **UART.h**
- UART_STATUS_BUFFER_INVALID : **UART.h**
- UART_STATUS_BUSY : **UART.h**
- UART_STATUS_FAILURE : **UART.h**
- UART_STATUS_MODE_MISMATCH : **UART.h**
- UART_STATUS_SUCCESS : **UART.h**
- UART_STATUS_t : **UART.h**
- UART_t : **UART.h**
- UART_TRANSFER_MODE : **UART.h**
- UART_TRANSFER_MODE_DIRECT : **UART.h**
- UART_TRANSFER_MODE_DMA : **UART.h**
- UART_TRANSFER_MODE_INTERRUPT : **UART.h**
- UART_TRANSFER_MODE_t : **UART.h**
- UART_Transmit() : **UART.h** , **UART.c**
- UART_TransmitWord() : **UART.h**
- UART_TX_CONFIG_t : **UART.h**



UART

Home				
File List	Globals			
All	Functions	Typedefs	Enumerations	Enumerator

- [UART_ClearFlag\(\)](#) : [UART.h](#)
- [UART_ClearRXFIFOStatus\(\)](#) : [UART.h](#)
- [UART_ClearTXFIFOStatus\(\)](#) : [UART.h](#)
- [UART_DisableEvent\(\)](#) : [UART.h](#)
- [UART_EnableEvent\(\)](#) : [UART.h](#)
- [UART_GetAppVersion\(\)](#) : [UART.c](#) , [UART.h](#)
- [UART_GetFlagStatus\(\)](#) : [UART.h](#)
- [UART_GetReceivedWord\(\)](#) : [UART.h](#)
- [UART_GetRXFIFOStatus\(\)](#) : [UART.h](#)
- [UART_GetTXFIFOStatus\(\)](#) : [UART.h](#)
- [UART_Init\(\)](#) : [UART.c](#) , [UART.h](#)
- [UART_IsRxBusy\(\)](#) : [UART.h](#)
- [UART_IsRXFIFOEmpty\(\)](#) : [UART.h](#)
- [UART_IsTxBusy\(\)](#) : [UART.h](#)
- [UART_IsTXFIFOFull\(\)](#) : [UART.h](#)
- [UART_Receive\(\)](#) : [UART.c](#) , [UART.h](#)
- [UART_SetBaudrate\(\)](#) : [UART.c](#) , [UART.h](#)
- [UART_SetRXFIFOTriggerLimit\(\)](#) : [UART.h](#)
- [UART_SetTXFIFOTriggerLimit\(\)](#) : [UART.h](#)
- [UART_Transmit\(\)](#) : [UART.c](#) , [UART.h](#)
- [UART_TransmitWord\(\)](#) : [UART.h](#)



UART

Home					
File List	Globals				
All	Functions	Typedefs	Enumerations	Enumerator	

- `UART_cbhandler` : [UART.h](#)
- `UART_CONFIG_t` : [UART.h](#)
- `UART_EVENT_t` : [UART.h](#)
- `UART_init_handler` : [UART.h](#)
- `UART_MODE_t` : [UART.h](#)
- `UART_PROTOCOL_STATUS_t` : [UART.h](#)
- `UART_RUNTIME_t` : [UART.h](#)
- `UART_STATUS_t` : [UART.h](#)
- `UART_t` : [UART.h](#)
- `UART_TRANSFER_MODE_t` : [UART.h](#)
- `UART_TX_CONFIG_t` : [UART.h](#)



UART

Home					
File List	Globals				
All	Functions	Typedefs	Enumerations	Enumerator	

- `UART_EVENT` : [UART.h](#)
- `UART_MODE` : [UART.h](#)
- `UART_STATUS` : [UART.h](#)
- `UART_TRANSFER_MODE` : [UART.h](#)



UART

Home				
File List	Globals			
All	Functions	Typedefs	Enumerations	Enumerator

- `UART_EVENT_COLLISION` : [UART.h](#)
- `UART_EVENT_FORMAT_ERR0` : [UART.h](#)
- `UART_EVENT_FORMAT_ERR1` : [UART.h](#)
- `UART_EVENT_MAX` : [UART.h](#)
- `UART_EVENT_RX_NOISE` : [UART.h](#)
- `UART_EVENT_SYNC_BRK` : [UART.h](#)
- `UART_MODE_FULLDUPLEX` : [UART.h](#)
- `UART_MODE_HALFDUPLEX` : [UART.h](#)
- `UART_MODE_LOOPBACK` : [UART.h](#)
- `UART_STATUS_BUFFER_INVALID` : [UART.h](#)
- `UART_STATUS_BUSY` : [UART.h](#)
- `UART_STATUS_FAILURE` : [UART.h](#)
- `UART_STATUS_MODE_MISMATCH` : [UART.h](#)
- `UART_STATUS_SUCCESS` : [UART.h](#)
- `UART_TRANSFER_MODE_DIRECT` : [UART.h](#)
- `UART_TRANSFER_MODE_DMA` : [UART.h](#)
- `UART_TRANSFER_MODE_INTERRUPT` : [UART.h](#)



UART

Home	
File List	Globals
Code	

UART.h

[Go to the documentation of this file.](#)

1

```
63 /*****
```

```
64 * HEADER FILES
```

```
65
```

```
*****/
```

```
66 #ifndef UART_H
```

```
67 #define UART_H
```

```
68
```

```
69 #include <xmc_scu.h>
```

```
70 #include <xmc_gpio.h>
```

```
71 #include <xmc_uart.h>
```

```
72
```

```
73 #if (!((XMC_LIB_MAJOR_VERSION == 2U) && \
```

```
74 (XMC_LIB_MINOR_VERSION >= 1U) && \
```

```
75 (XMC_LIB_PATCH_VERSION >= 10U)))
```

```
76 #error "UART requires XMC Peripheral Library v2.1.10 or higher"
```

```
77 #endif
```

```
78
```

```
79 #include <DAVE_common.h>
```

```
80 #include "uart_conf.h"
```

```
81 #if ((defined UART_TX_DMA_USED) || (defined
```

```
UART_RX_DMA_USED))
```

```
82 #include "../GLOBAL_DMA/global_dma.h"
```

```
83 #endif
```

```
84
```

```
85 /*****
```

```

86 * MACROS
87
*****
88 /*
89 * @brief Represents the maximum data size for DMA transaction*/
90 #define UART_DMA_MAXCOUNT (4095U)
91
92 /*****
93 * ENUMS
94
*****
104 typedef enum UART_STATUS
105 {
106     UART_STATUS_SUCCESS,
108     UART_STATUS_FAILURE,
110     UART_STATUS_BUSY,
112     UART_STATUS_BUFFER_INVALID,
114     UART_STATUS_MODE_MISMATCH
119 } UART_STATUS_t;
120
124 typedef enum UART_MODE
125 {
126     UART_MODE_FULLDUPLEX,
127     UART_MODE_HALFDUPLEX,
128     UART_MODE_LOOPBACK
129 } UART_MODE_t;
130
134 typedef enum UART_EVENT
135 {
136     UART_EVENT_SYNC_BRK,
137     UART_EVENT_RX_NOISE,
138     UART_EVENT_FORMAT_ERR0,
139     UART_EVENT_FORMAT_ERR1,
140     UART_EVENT_COLLISION,
141     UART_EVENT_MAX
142 } UART_EVENT_t;
143

```

```

147 typedef enum UART_TRANSFER_MODE
148 {
149     UART_TRANSFER_MODE_INTERRUPT,
150     UART_TRANSFER_MODE_DMA,
151     UART_TRANSFER_MODE_DIRECT
152 }UART_TRANSFER_MODE_t;
153
154 /*****
155  * DATA STRUCTURES
156  *****/
157
158 /*****
159  * DATA STRUCTURES
160  *****/
161
162 typedef void (*UART_cbhandler)(void);
163
164 typedef UART_STATUS_t (*UART_init_handler)(void);
165
166 typedef XMC_UART_CH_STATUS_FLAG_t
167     UART_PROTOCOL_STATUS_t;
168
169
170
171
172 typedef struct UART_TX_CONFIG
173 {
174     XMC_GPIO_PORT_t *const port;
175     const uint8_t pin;
176     const XMC_GPIO_CONFIG_t *const config;
177 } UART_TX_CONFIG_t;
178
179
180
181
182 #if (defined(UART_TX_DMA_USED) ||
183     defined(UART_RX_DMA_USED))
184
185
186
187
188
189 typedef struct UART_DMA_CONFIG
190 {
191     const XMC_DMA_CH_CONFIG_t * dma_ch_config;
192     uint8_t dma_channel;
193 }UART_DMA_CONFIG_t;
194 #endif
195
196
197
198
199
200
201 typedef struct UART_CONFIG
202 {
203     const XMC_UART_CH_CONFIG_t * const channel_config;
204     #if (defined UART_TX_DMA_USED) || (defined

```

```

UART_RX_DMA_USED)
208 GLOBAL_DMA_t * global_dma;
209 #endif
210 #ifdef UART_TX_DMA_USED
211 const UART_DMA_CONFIG_t * const transmit_dma_config;
212 #endif
213 #ifdef UART_RX_DMA_USED
214 const UART_DMA_CONFIG_t * const receive_dma_config;
215 #endif
216 UART_init_handler fptr_uart_config;
217 #ifdef UART_TX_INTERRUPT_USED
218 UART_cbhandler tx_cbhandler;
220 #endif
221 #ifdef UART_RX_INTERRUPT_USED
222 UART_cbhandler rx_cbhandler;
224 #endif
225 UART_cbhandler sync_error_cbhandler;
227 UART_cbhandler rx_noise_error_cbhandler;
229 UART_cbhandler format_error_bit0_cbhandler;
231 UART_cbhandler format_error_bit1_cbhandler;
233 UART_cbhandler collision_error_cbhandler;
235 const UART_TX_CONFIG_t * tx_pin_config;
237 UART_MODE_t mode;
238 UART_TRANSFER_MODE_t transmit_mode;
240 UART_TRANSFER_MODE_t receive_mode;
242 XMC_USIC_CH_FIFO_SIZE_t tx_fifo_size;
243 XMC_USIC_CH_FIFO_SIZE_t rx_fifo_size;
244 uint8_t tx_sr;
245 } UART_CONFIG_t;
246
250 typedef struct UART_RUNTIME
251 {
252 uint8_t * tx_data;
253 uint8_t * rx_data;
254 uint32_t tx_data_count;
255 uint32_t tx_data_index;
256 uint32_t rx_data_count;

```

```

257 uint32_t rx_data_index;
258 volatile bool tx_busy;
259 volatile bool rx_busy;
260 } UART_RUNTIME_t;
261
265 typedef struct UART
266 {
267   XMC_USIC_CH_t * const channel;
268   const UART_CONFIG_t * const config;
269   UART_RUNTIME_t * const runtime;
271 } UART_t;
272
277 /*****
278  * API Prototypes
279  *****/
280
281 #ifdef __cplusplus
282 extern "C" {
283 #endif
284
325 DAVE_APP_VERSION_t UART_GetAppVersion(void);
326
369 UART_STATUS_t UART_Init(const UART_t *const handle);
370
464 UART_STATUS_t UART_Receive(const UART_t *const handle,
uint8_t* data_ptr, uint32_t count);
465
552 UART_STATUS_t UART_Transmit(const UART_t *const handle,
uint8_t* data_ptr, uint32_t count);
553
554 #if (defined UART_TX_INTERRUPT_USED || defined
UART_TX_DMA_USED)
555
621 UART_STATUS_t UART_AbortTransmit(const UART_t *const
handle);
622 #endif

```

```
623
624 #if (defined UART_RX_INTERRUPT_USED || defined
UART_RX_DMA_USED)
625
692 UART_STATUS_t UART_AbortReceive(const UART_t *const
handle);
693 #endif
694
695 #ifdef UART_RX_INTERRUPT_USED
696
770 UART_STATUS_t UART_StartReceiveIRQ(const UART_t *const
handle, uint8_t* data_ptr, uint32_t count);
771 #endif
772
773 #ifdef UART_TX_INTERRUPT_USED
774
846 UART_STATUS_t UART_StartTransmitIRQ(const UART_t *const
handle, uint8_t* data_ptr, uint32_t count);
847 #endif
848
849 #ifdef UART_TX_DMA_USED
850
919 UART_STATUS_t UART_StartTransmitDMA(const UART_t *const
handle, uint8_t* data_ptr, uint32_t count);
920 #endif
921
922 #ifdef UART_RX_DMA_USED
923
994 UART_STATUS_t UART_StartReceiveDMA(const UART_t *const
handle, uint8_t* data_ptr, uint32_t count);
995 #endif
996
1094 UART_STATUS_t UART_SetBaudrate(const UART_t * handle,
uint32_t baud, uint32_t oversampling);
1095
1153 __STATIC_INLINE uint32_t UART_GetTXFIFOStatus(const
UART_t* const handle)
```

```

1154 {
1155 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1156 return XMC_USIC_CH_TXFIFO_GetEvent(handle->channel);
1157 }
1158
1244 __STATIC_INLINE uint8_t UART_GetReceivedWord(const
UART_t* const handle)
1245 {
1246 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1247 return (uint8_t)XMC_UART_CH_GetReceivedData(handle-
>channel);
1248 }
1249
1305 __STATIC_INLINE void UART_TransmitWord(const UART_t*
const handle, uint8_t data)
1306 {
1307 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1308 XMC_UART_CH_Transmit(handle->channel, (uint16_t)data);
1309 }
1310
1369 __STATIC_INLINE void UART_EnableEvent(const UART_t* const
handle, uint32_t events)
1370 {
1371 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1372 XMC_UART_CH_EnableEvent(handle->channel, events);
1373 }
1374
1432 __STATIC_INLINE void UART_DisableEvent(const UART_t*
const handle, uint32_t events)
1433 {
1434 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1435 XMC_UART_CH_DisableEvent(handle->channel, events);
1436 }
1437
1490 __STATIC_INLINE bool UART_IsTXFIFOFull(const UART_t*
const handle)
1491 {

```

```
1492 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1493 return XMC_USIC_CH_TXFIFO_IsFull(handle->channel);
1494 }
1495
1553 __STATIC_INLINE bool UART_IsRXFIFOEmpty(const UART_t*
const handle)
1554 {
1555 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1556 return XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel);
1557 }
1558
1626 __STATIC_INLINE void UART_SetTXFIFOTriggerLimit(const
UART_t* const handle, uint32_t limit)
1627 {
1628 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1629 XMC_USIC_CH_TXFIFO_SetSizeTriggerLimit(handle->channel,
handle->config->tx_fifo_size, limit);
1630 }
1631
1699 __STATIC_INLINE void UART_SetRXFIFOTriggerLimit(const
UART_t* const handle, uint32_t limit)
1700 {
1701 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1702 XMC_USIC_CH_RXFIFO_SetSizeTriggerLimit(handle->channel,
handle->config->rx_fifo_size, limit);
1703 }
1704
1766 __STATIC_INLINE uint32_t UART_GetRXFIFOStatus(const
UART_t* const handle)
1767 {
1768 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1769 return XMC_USIC_CH_RXFIFO_GetEvent(handle->channel);
1770 }
1771
1828 __STATIC_INLINE void UART_ClearTXFIFOStatus(const
UART_t* const handle, const uint32_t flag)
1829 {
```

```
1830 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1831 XMC_USIC_CH_TXFIFO_ClearEvent(handle->channel, flag);
1832 }
1833
1899 __STATIC_INLINE void UART_ClearRXFIFOStatus(const
UART_t* const handle, const uint32_t flag)
1900 {
1901 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1902 XMC_USIC_CH_RXFIFO_ClearEvent(handle->channel, flag);
1903 }
1904
1979 __STATIC_INLINE uint32_t UART_GetFlagStatus(const UART_t
* const handle, uint32_t protocol_status)
1980 {
1981 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
1982 return (XMC_UART_CH_GetStatusFlag(handle->channel) &
protocol_status);
1983 }
1984
2046 __STATIC_INLINE void UART_ClearFlag(const UART_t *const
handle, const uint32_t protocol_status)
2047 {
2048 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
2049 XMC_UART_CH_ClearStatusFlag(handle->channel,
protocol_status);
2050 }
2051
2116 __STATIC_INLINE bool UART_IsTxBusy(const UART_t *const
handle)
2117 {
2118 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
2119 return (handle->runtime->tx_busy);
2120 }
2121
2181 __STATIC_INLINE bool UART_IsRxBusy(const UART_t *const
handle)
2182 {
```

```
2183 XMC_ASSERT("UART APP handle invalid", (handle != NULL))
2184 return (handle->runtime->rx_busy);
2185 }
2186
2191 #ifdef __cplusplus
2192 }
2193 #endif
2194
2195
2196 /* Include App extern declaration file */
2197 #include "uart_extern.h"
2198
2199 #endif /* UART_H_ */
```

UART

[Home](#)

Data structures

typedef XMC_UART_CH_STATUS_FLAG_t **UART_PROTOCOL_STATUS_FLAG_t**
Wrapper typedefinition for XMC_UART_PROTOCOL_STATUS_FLAG_t

typedef struct **UART_TX_CONFIG** **UART_TX_CONFIG_t**
Structure for transmit pin configuration.

typedef struct **UART_CONFIG** **UART_CONFIG_t**
Structure for holding the configuration parameters of **UART** channel.

typedef struct **UART_RUNTIME** **UART_RUNTIME_t**
Structure to hold the dynamic variables for the **UART** communication.

typedef struct **UART** **UART_t**
Handler structure with pointer to dynamic and static parameters.

Detailed Description

UART

Home	
File List	Globals
Code	

UART.c

[Go to the documentation of this file.](#)

1

```
61 /*****
```

```
62 * HEADER FILES
```

```
63
```

```
*****/
```

```
64 #include "uart.h"
```

```
65
```

```
66 /*****
```

```
67 * MACROS
```

```
68
```

```
*****/
```

```
69
```

```
70 /*****
```

```
71 * LOCAL DATA
```

```
72
```

```
*****/
```

```
73 const XMC_UART_CH_STATUS_FLAG_t  
uart_event_status_flags[UART_EVENT_MAX] = {
```

```
74
```

```
XMC_UART_CH_STATUS_FLAG_SYNCHRONIZATION_BREAK_DETE
```

```
75
```

```
XMC_UART_CH_STATUS_FLAG_RECEIVER_NOISE_DETECTED,
```

```
76
```

```
XMC_UART_CH_STATUS_FLAG_FORMAT_ERROR_IN_STOP_BIT_0,
```

```
77
```

```
XMC_UART_CH_STATUS_FLAG_FORMAT_ERROR_IN_STOP_BIT_1,
```

```

78 XMC_UART_CH_STATUS_FLAG_COLLISION_DETECTED
79 };
80 const XMC_UART_CH_EVENT_t
uart_event_conf_flags[UART_EVENT_MAX] = {
81 XMC_UART_CH_EVENT_SYNCHRONIZATION_BREAK,
82 XMC_UART_CH_EVENT_RECEIVER_NOISE,
83 XMC_UART_CH_EVENT_FORMAT_ERROR,
84 XMC_UART_CH_EVENT_FORMAT_ERROR,
85 XMC_UART_CH_EVENT_COLLISION
86 };
87 /*****
88 * LOCAL ROUTINES
89
*****/
90 #ifdef UART_TX_INTERRUPT_USED
91 /*Function used for handling transmit interrupt.*/
92 void UART_ITransmitHandler(const UART_t * const handle);
93 #endif
94 #ifdef UART_RX_INTERRUPT_USED
95 /*Function used for handling data reception interrupts.*/
96 void UART_IReceiveHandler(const UART_t * const handle);
97 /*Function used for reconfiguring rx FIFO while receiving data.*/
98 static void UART_IReconfigureRxFIFO(const UART_t * const
handle, uint32_t data_size);
99 #endif
100 #ifdef UART_TX_DIRECT_USED
101 /*Function for transmitting data using polling.*/
102 static UART_STATUS_t UART_IStartTransmitPolling (const
UART_t *const handle, uint8_t* data_ptr, uint32_t count);
103 #endif
104 #ifdef UART_RX_DIRECT_USED
105 /*Function for receiving data using polling.*/
106 static UART_STATUS_t UART_IStartReceivePolling (const
UART_t *const handle, uint8_t* data_ptr, uint32_t count);
107 #endif
108 /*Function used for handling protocol related interrupt.*/
109 void UART_IProtocolHandler(const UART_t * const handle);

```

```

110
111
112 /*****
113 * API IMPLEMENTATION
114
*****
115
116 /*
117 * @brief API to retrieve the version of the UART APP.
118 *
119 * @return DAVE_APP_VERSION_t Structure containing major
version, minor version
120 * and patch version.
121 */
122 DAVE_APP_VERSION_t UART_GetAppVersion()
123 {
124 DAVE_APP_VERSION_t version;
125
126 version.major = UART_MAJOR_VERSION;
127 version.minor = UART_MINOR_VERSION;
128 version.patch = UART_PATCH_VERSION;
129
130 return version;
131 }
132
133 /*
134 * @brief Function to initialize the USIC Channel with GUI
configured values.
135 *
136 * @param[in] handle UART APP handle pointer of type UART_t*
137 *
138 * @return UART_STATUS_t
139 * UART_SUCCESS: for successful UART initialization.<BR>
140 * UART_STATUS_FAILURE : If UART initialization fails.<BR>
141 *
142 */
143 UART_STATUS_t UART_Init(const UART_t *const handle)

```

```

144 {
145  UART_STATUS_t status = UART_STATUS_SUCCESS;
146  XMC_ASSERT("UART_Init : UART APP handle invalid", (((handle
147  != NULL)&&
148  (handle->config != NULL)) &&((handle->config->fptr_uart_config !=
149  NULL)&&
150  (handle->runtime != NULL))))
151  /*Initialize the multiplexers required for UART configuration*/
152  status = handle->config->fptr_uart_config();
153  return status;
154 }
155
156 /*
157  * @brief Common function to transmit data.
158  *
159  * @param[in] handle UART APP handle pointer of type UART_t*
160  * @param[in] data_ptr Pointer to data of type uint8_t
161  * @param[in] count Number of uint8_t type bytes to be transmitted
162  *
163  * @return UART_STATUS_t
164  * UART_SUCCESS: If the data is put to transmit.<BR>
165  * UART_STATUS_BUSY : If the channel is busy.<BR>
166  * UART_STATUS_BUFFER_INVALID: Either if buffer is NULL or
167  * count is 0.<BR>
168  * UART_STATUS_MODE_MISMATCH: If the configured mode is
169  * invalid.<BR>
170  *
171  */
172  UART_STATUS_t UART_Transmit(const UART_t *const handle,
173  uint8_t* data_ptr, uint32_t count)
174  {
175  UART_STATUS_t ret_stat = UART_STATUS_MODE_MISMATCH;
176  switch(handle->config->transmit_mode)
177  {

```

```

176 #ifdef UART_TX_INTERRUPT_USED
177 case UART_TRANSFER_MODE_INTERRUPT:
178 ret_stat = UART_StartTransmitIRQ(handle, data_ptr, count);
179 break;
180 #endif
181 #ifdef UART_TX_DMA_USED
182 case UART_TRANSFER_MODE_DMA:
183 ret_stat = UART_StartTransmitDMA(handle, data_ptr, count);
184 break;
185 #endif
186 #ifdef UART_TX_DIRECT_USED
187 case UART_TRANSFER_MODE_DIRECT:
188 ret_stat = UART_IStartTransmitPolling(handle, data_ptr, count);
189 break;
190 #endif
191 default:
192 break;
193 }
194 return ret_stat;
195 }
196
197 /*
198 * @brief Common function to receive data.
199 *
200 * @param[in] handle UART APP handle pointer of type UART_t*
201 * @param[in] data_ptr Pointer to data of type uint8_t
202 * @param[in] count Number of uint8_t type bytes to be received
203 *
204 * @return UART_STATUS_t
205 * UART_SUCCESS: If the data is put to transmit.<BR>
206 * UART_STATUS_BUSY : If the channel is busy.<BR>
207 * UART_STATUS_BUFFER_INVALID: Either if buffer is NULL or
count is 0.<BR>
208 * UART_STATUS_MODE_MISMATCH: If the configured mode is
invalid.<BR>
209 *
210 */

```

```

211 UART_STATUS_t UART_Receive(const UART_t *const handle,
uint8_t* data_ptr, uint32_t count)
212 {
213     UART_STATUS_t ret_stat = UART_STATUS_MODE_MISMATCH;
214
215     switch(handle->config->receive_mode)
216     {
217     #ifdef UART_RX_INTERRUPT_USED
218     case UART_TRANSFER_MODE_INTERRUPT:
219         ret_stat = UART_StartReceiveIRQ(handle, data_ptr, count);
220         break;
221     #endif
222     #ifdef UART_RX_DMA_USED
223     case UART_TRANSFER_MODE_DMA:
224         ret_stat = UART_StartReceiveDMA(handle, data_ptr, count);
225         break;
226     #endif
227     #ifdef UART_RX_DIRECT_USED
228     case UART_TRANSFER_MODE_DIRECT:
229         ret_stat = UART_StartReceivePolling(handle, data_ptr, count);
230         break;
231     #endif
232     default:
233         break;
234     }
235     return ret_stat;
236 }
237
238 #if (defined UART_TX_INTERRUPT_USED || defined
UART_TX_DMA_USED)
239 /*
240 * @brief Common function to abort ongoing transmission.
241 *
242 * @param[in] handle UART APP handle pointer of type UART_t*
243 *
244 * @return UART_STATUS_t
245 * UART_SUCCESS: If the transmission is aborted.<BR>

```

```

246 * UART_STATUS_FAILURE: If the channel is not transmitting.
<BR>
247 * UART_STATUS_MODE_MISMATCH: If the configured mode is
Direct.<BR>
248 *
249 */
250 UART_STATUS_t UART_AbortTransmit(const UART_t *const
handle)
251 {
252     UART_STATUS_t ret_stat = UART_STATUS_SUCCESS;
253     #ifdef UART_TX_DMA_USED
254     const UART_DMA_CONFIG_t * ptr_dma_config = handle->config-
>transmit_dma_config;
255     XMC_DMA_t * ptr_gpdma = handle->config->global_dma->dma;
256     #endif
257
258     XMC_ASSERT("UART_AbortTransmit: UART APP handle invalid",
((handle != NULL)&&
259 (handle->runtime != NULL)))
260
261     /*Reset the user buffer pointer to null*/
262     handle->runtime->tx_busy = false;
263     handle->runtime->tx_data = NULL;
264
265     switch(handle->config->transmit_mode)
266     {
267     #ifdef UART_TX_INTERRUPT_USED
268     case UART_TRANSFER_MODE_INTERRUPT:
269     /*Disable the transmit interrupts*/
270     if (handle->config->tx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
271     {
272     /*Disable the transmit FIFO event*/
273     XMC_USIC_CH_TXFIFO_DisableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD);
274     XMC_USIC_CH_TXFIFO_Flush(handle->channel);
275     }

```

```
276 else
277 {
278 /*Disable the standard transmit event*/
279 XMC_USIC_CH_DisableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
280 }
281 XMC_USIC_CH_SetTransmitBufferStatus(handle->channel,
XMC_USIC_CH_TBUF_STATUS_SET_IDLE);
282 break;
283 #endif
284 #ifdef UART_TX_DMA_USED
285 case UART_TRANSFER_MODE_DMA:
286 /*Disable the standard transmit event*/
287 if (XMC_DMA_CH_IsEnabled(ptr_gpdma, ptr_dma_config-
>dma_channel))
288 {
289 XMC_DMA_CH_Disable(ptr_gpdma, ptr_dma_config-
>dma_channel);
290 while(XMC_DMA_CH_IsEnabled(ptr_gpdma, ptr_dma_config-
>dma_channel)==true)
291 {
292 }
293 XMC_USIC_CH_DisableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
294 }
295 XMC_USIC_CH_SetTransmitBufferStatus(handle->channel,
XMC_USIC_CH_TBUF_STATUS_SET_IDLE);
296 break;
297 #endif
298 default:
299 ret_stat = UART_STATUS_MODE_MISMATCH;
300 break;
301 }
302 return ret_stat;
303 }
304 #endif
305
```

```

306 #if (defined UART_RX_INTERRUPT_USED || defined
UART_RX_DMA_USED)
307 /*
308 * @brief Common function to abort ongoing reception.
309 *
310 * @param[in] handle UART APP handle pointer of type UART_t*
311 *
312 * @return UART_STATUS_t
313 * UART_SUCCESS: If the reception is aborted.<BR>
314 * UART_STATUS_FAILURE : If the channel is not busy.<BR>
315 * UART_STATUS_MODE_MISMATCH: If the configured mode is
Direct.<BR>
316 *
317 */
318 UART_STATUS_t UART_AbortReceive(const UART_t *const
handle)
319 {
320 UART_STATUS_t ret_stat = UART_STATUS_SUCCESS;
321 #ifdef UART_RX_DMA_USED
322 const UART_DMA_CONFIG_t * ptr_dma_config = handle->config-
>receive_dma_config;
323 XMC_DMA_t * ptr_gpdma = handle->config->global_dma->dma;
324 #endif
325 XMC_ASSERT("UART_AbortReceive: UART APP handle invalid",
((handle != NULL)&&
326 (handle->runtime != NULL)))
327
328 /*Reset the user buffer pointer to null*/
329 handle->runtime->rx_busy = false;
330 handle->runtime->rx_data = NULL;
331 switch(handle->config->receive_mode)
332 {
333 #ifdef UART_RX_INTERRUPT_USED
334 case UART_TRANSFER_MODE_INTERRUPT:
335 /*Disable the receive interrupts*/
336 if (handle->config->rx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)

```

```

337 {
338 XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel,
339 ((uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_STANDARD
|
340
(uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_ALTERNATE));
341 }
342 else
343 {
344 XMC_UART_CH_DisableEvent(handle->channel,
345 ((uint32_t)XMC_USIC_CH_EVENT_STANDARD_RECEIVE |
346 (uint32_t)XMC_USIC_CH_EVENT_ALTERNATIVE_RECEIVE));
347 }
348 break;
349 #endif
350 #ifdef UART_RX_DMA_USED
351 case UART_TRANSFER_MODE_DMA:
352 /*Disable the receive interrupts*/
353 if (XMC_DMA_CH_IsEnabled(ptr_gpdma, ptr_dma_config-
>dma_channel))
354 {
355 XMC_DMA_CH_Disable(ptr_gpdma, ptr_dma_config-
>dma_channel);
356 while(XMC_DMA_CH_IsEnabled(ptr_gpdma, ptr_dma_config-
>dma_channel)==true)
357 {
358 }
359 XMC_UART_CH_DisableEvent(handle->channel,
360 ((uint32_t)XMC_USIC_CH_EVENT_STANDARD_RECEIVE |
361 (uint32_t)XMC_USIC_CH_EVENT_ALTERNATIVE_RECEIVE));
362 }
363 break;
364 #endif
365 default:
366 ret_stat = UART_STATUS_MODE_MISMATCH;
367 break;
368 }

```

```

369 return ret_stat;
370 }
371 #endif
372
373 /*
374 * @brief Changes the baudrate of UART channel.
375 *
376 * @param UART_t * Pointer to the UART APP handle.
377 * @param baud Value of new baudrate.
378 * @param oversampling Number of samples to be considered for
each symbol. 16 is the standard value.
379 *
380 * @return UART_STATUS_t UART_STATUS_SUCCESS if
baudrate changed successfully.
381 * UART_STATUS_BUSY if the UART channel is busy.
382 *
383 * \par<b>Description:</b><br>
384 * The function stops the channel, calculates the clock divider
values to achieve the desired baudrate.
385 * Sets the divider values and reconfigures the channel as per the
configuration in the UI. The channel is
386 * enabled at the end of configuration.
387 */
388 UART_STATUS_t UART_SetBaudrate(const UART_t * handle,
uint32_t baud, uint32_t oversampling)
389 {
390     UART_STATUS_t ret_stat = UART_STATUS_BUSY;
391     const UART_TX_CONFIG_t * ptr_tx_conf = handle->config-
>tx_pin_config;
392
393     XMC_ASSERT("UART_SetBaudrate: UART APP handle invalid",
((handle != NULL)&&
394 ((handle->config != NULL) && (handle->runtime != NULL))))
395
396     if ((handle->runtime->tx_busy == false) && (handle->runtime-
>rx_busy == false))
397     {

```

```

398 /* Set UART TX pin as input pin to avoid spikes on the pin.*/
399 if (handle->config->mode != UART_MODE_LOOPBACK)
400 {
401 XMC_GPIO_SetMode(ptr_tx_conf->port, ptr_tx_conf->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
402 }
403 /* Stop the UART channel before changing the baudrate.*/
404 if (XMC_UART_CH_Stop(handle->channel) ==
XMC_UART_CH_STATUS_OK)
405 {
406 /*Change the baudrate*/
407 ret_stat =
(UART_STATUS_t)XMC_UART_CH_SetBaudrate(handle->channel,
baud, oversampling);
408 /*Set the sample point if the baudrate is modified*/
409 if (ret_stat == UART_STATUS_SUCCESS)
410 {
411 XMC_UART_CH_SetSamplePoint(handle->channel, (uint32_t)
(oversampling >> 1U)+1U);
412 }
413 /*Enable UART*/
414 XMC_UART_CH_Start(handle->channel);
415 /* Initialize UART TX pin */
416 if (handle->config->mode != UART_MODE_LOOPBACK)
417 {
418 XMC_GPIO_Init(ptr_tx_conf->port, ptr_tx_conf->pin, ptr_tx_conf-
>config);
419 }
420 }
421 else
422 {
423 ret_stat = UART_STATUS_BUSY;
424 }
425 }
426 return ret_stat;
427 }
428

```

```

429 #ifdef UART_TX_INTERRUPT_USED
430 /*
431 * @brief Registers a request for transmitting data over UART
channel.
432 *
433 * @param[in] UART_t* UART APP handle pointer of type UART_t
434 * @param[in] uint8_t* Pointer to data
435 * @param[in] uint32_t Total no of words to be transmitted.
436 *
437 * @return UART_STATUS_t UART_STATUS_SUCCESS if the
request is accepted.
438 * UART_STATUS_BUSY if a transmission is in progress.
439 * Details of function:
440 * The data transmission is accomplished using transmit interrupt.
User can configure
441 * a callback function in the APP UI. When the data is fully
transmitted, the callback
442 * function will be executed. If transmit FIFO is enabled, the trigger
limit is set to 0.
443 * So the transmit interrupt will be generated when all the data in
FIFO is moved from FIFO.
444 *
445 * <i>Imp Note:</i> Return value should be validated by user to
ensure that the
446 * request is registered.
447 *
448 *
449 */
450 UART_STATUS_t UART_StartTransmitIRQ(const UART_t *const
handle, uint8_t* data_ptr, uint32_t count)
451 {
452     UART_STATUS_t ret_stat = UART_STATUS_MODE_MISMATCH;
453     UART_RUNTIME_t * ptr_runtime = handle->runtime;
454
455     XMC_ASSERT("UART_StartTransmitIRQ: UART APP handle
invalid", ((handle != NULL)&&
456 (handle->runtime != NULL)))

```

```

457
458 if (handle->config->transmit_mode ==
UART_TRANSFER_MODE_INTERRUPT)
459 {
460 ret_stat = UART_STATUS_BUSY;
461 if (ptr_runtime->tx_busy == false)
462 {
463 /*If there is no transmission in progress*/
464 if ((data_ptr != NULL) && (count > 0U))
465 {
466 /*Obtain the address of data, size of data*/
467 ptr_runtime->tx_data = data_ptr;
468 ptr_runtime->tx_data_count = count;
469 /*Initialize to first index and set the busy flag*/
470 ptr_runtime->tx_data_index = 0U;
471 ptr_runtime->tx_busy = true;
472
473 /*Enable the transmit buffer event*/
474 if (handle->config->tx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
475 {
476 /*Clear the transmit FIFO*/
477 XMC_USIC_CH_TXFIFO_Flush(handle->channel);
478 /*Enable transmit buffer interrupt*/
479 XMC_USIC_CH_TXFIFO_EnableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD);
480 }
481 else
482 {
483 XMC_USIC_CH_EnableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
484 }
485 ret_stat = UART_STATUS_SUCCESS;
486 /*Trigger the transmit buffer interrupt*/
487 XMC_USIC_CH_TriggerServiceRequest(handle->channel,
(uint32_t)handle->config->tx_sr);
488 }

```

```

489 else
490 {
491 ret_stat = UART_STATUS_BUFFER_INVALID;
492 }
493 }
494 }
495 return ret_stat;
496 }
497 #endif
498
499 #ifdef UART_RX_INTERRUPT_USED
500 /*
501 * @brief Registers a request to receive data over UART channel.
502 *
503 * @param[in] UART_t* UART APP handle pointer of type UART_t
504 * @param[in] uint8_t* Pointer to data array
505 * @param[in] uint32_t Total no of bytes to be read.
506 *
507 * @return UART_STATUS_t UART_STATUS_SUCCESS if the
request is accepted.
508 * UART_STATUS_BUSY if a reception is in progress.
509 * Details of function:
510 * This function registers the receive request by configuring the
UART
511 * receive FIFO/Standard buffer (depending on the user
configuration). The data
512 * is received asynchronously. When the requested number of data
bytes are received,
513 * optionally, the user configured callback function will be executed.
If a callback
514 * function is not configured on the APP UI, the user has to poll for
the status of
515 * rx_busy variable of the APP handle structure.
516 *
517 * <i>Imp Note:</i> Return value should be validated by user to
ensure that the
518 * request is registered.

```

```

519 *
520 *
521 */
522 UART_STATUS_t UART_StartReceiveIRQ(const UART_t *const
handle, uint8_t* data_ptr, uint32_t count)
523 {
524     UART_STATUS_t ret_stat = UART_STATUS_MODE_MISMATCH;
525     UART_RUNTIME_t * ptr_runtime = handle->runtime;
526
527     XMC_ASSERT("UART_StartReceiveIRQ: UART APP handle
invalid", ((handle != NULL)&&
528 (handle->runtime != NULL)))
529
530     if (handle->config->receive_mode ==
UART_TRANSFER_MODE_INTERRUPT)
531     {
532         ret_stat = UART_STATUS_BUSY;
533         if (ptr_runtime->rx_busy == false)
534         {
535             /*If no active reception in progress*/
536             if ((data_ptr != NULL) && (count > 0U))
537             {
538                 /*Obtain the address of data buffer and
539                 * number of data bytes to be received*/
540                 ptr_runtime->rx_data = data_ptr;
541                 ptr_runtime->rx_data_count = count;
542                 ptr_runtime->rx_busy = true;
543                 ptr_runtime->rx_data_index = 0U;
544
545                 if (handle->config->rx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
546                 {
547                     /*Clear the receive FIFO, configure the trigger lime
548                     * and enable the receive events*/
549                     XMC_USIC_CH_RXFIFO_Flush(handle->channel);
550
551                     /*Configure the FIFO trigger limit based on the required data size*/

```

```

552 UART_IReconfigureRxFIFO(handle, count);
553
554 XMC_USIC_CH_RXFIFO_EnableEvent(handle->channel,
555 (uint32_t)
556 ((uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_STANDARD |
557 (uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_ALTERNATE));
558 }
559 else
560 {
561 XMC_USIC_CH_EnableEvent(handle->channel,
562 (uint32_t)
563 ((uint32_t)XMC_USIC_CH_EVENT_STANDARD_RECEIVE |
564 (uint32_t)XMC_USIC_CH_EVENT_ALTERNATIVE_RECEIVE));
565 }
566 ret_stat = UART_STATUS_SUCCESS;
567 }
568 else
569 {
570 ret_stat = UART_STATUS_BUFFER_INVALID;
571 }
572 }
573 return ret_stat;
574 }
575 #endif
576
577 #ifdef UART_TX_DMA_USED
578 /*
579 * @brief Registers a request for transmitting data over UART
580 * channel using DMA.
581 *
582 * @param[in] UART_t* UART APP handle pointer of type UART_t
583 * @param[in] uint8_t* Pointer to data
584 * @param[in] uint32_t Total no of words to be transmitted.
585 *
586 * @return UART_STATUS_t UART_STATUS_SUCCESS if the

```

request is accepted.

584 * UART_STATUS_BUSY if a transmission is in progress.

585 * Details of function:

586 * The data transmission is accomplished using a DMA channel.

User can configure

587 * a callback function in the APP UI. When the data is fully transmitted, the callback

588 * function will be executed.

589 * *Imp Note:* Return value should be validated by user to ensure that the

590 * request is registered.

591 *

592 *

593 */

594 UART_STATUS_t UART_StartTransmitDMA(const UART_t *const handle, uint8_t* data_ptr, uint32_t count)

595 {

596 UART_STATUS_t ret_stat = UART_STATUS_MODE_MISMATCH;

597 UART_RUNTIME_t * ptr_runtime = handle->runtime;

598 const UART_DMA_CONFIG_t * ptr_dma_config = handle->config->transmit_dma_config;

599 XMC_DMA_t * ptr_gpdma = handle->config->global_dma->dma;

600

601 XMC_ASSERT("UART_StartTransmitDMA: UART APP handle invalid", (((handle != NULL)&&

602 (handle->runtime != NULL))&&(handle->config != NULL)))

603

604 if (handle->config->transmit_mode == UART_TRANSFER_MODE_DMA)

605 {

606 ret_stat = UART_STATUS_BUSY;

607 if (ptr_runtime->tx_busy == false)

608 {

609 */*If there is no transmission in progress*/*

610 if ((data_ptr != NULL) && ((count > 0U) &&(count <= UART_DMA_MAXCOUNT)))

611 {

```

612 /*Obtain the address of data, size of data*/
613 ptr_runtime->tx_data = data_ptr;
614 ptr_runtime->tx_data_count = count;
615 /*Initialize to first index and set the busy flag*/
616 ptr_runtime->tx_data_index = 0U;
617 ptr_runtime->tx_busy = true;
618
619 /*Enable transmit event generation*/
620 XMC_UART_CH_EnableEvent(handle->channel,
(uint32_t)XMC_UART_CH_EVENT_TRANSMIT_BUFFER);
621 ret_stat = UART_STATUS_SUCCESS;
622
623 /*Enable DMA channel*/
624 XMC_DMA_CH_SetBlockSize(ptr_gpdma, ptr_dma_config-
>dma_channel, count);
625 XMC_DMA_CH_SetSourceAddress(ptr_gpdma, ptr_dma_config-
>dma_channel, (uint32_t)data_ptr);
626 XMC_DMA_CH_SetDestinationAddress(ptr_gpdma,
ptr_dma_config->dma_channel,
627 (uint32_t)&(handle->channel->TBUF[0]));
628 XMC_DMA_CH_Enable(ptr_gpdma, ptr_dma_config-
>dma_channel);
629 }
630 else
631 {
632 ret_stat = UART_STATUS_BUFFER_INVALID;
633 }
634 }
635 }
636 return ret_stat;
637 }
638 #endif
639
640 #ifdef UART_RX_DMA_USED
641 /*
642 * @brief Registers a request to receive data over UART channel
using DMA.

```

```

643 *
644 * @param[in] UART_t* UART APP handle pointer of type UART_t
645 * @param[in] uint8_t* Pointer to data array
646 * @param[in] uint32_t Total no of bytes to be read.
647 *
648 * @return UART_STATUS_t UART_STATUS_SUCCESS if the
request is accepted.
649 * UART_STATUS_BUSY if a reception is in progress.
650 * Details of function:
651 * This function registers the receive request by configuring the
UART
652 * receive Standard buffer and the DMA channel. The data
653 * is received asynchronously. When the requested number of data
bytes are received,
654 * optionally, the user configured callback function will be executed.
655 *
656 * <i>Imp Note:</i> Return value should be validated by user to
ensure that the
657 * request is registered.
658 *
659 *
660 */
661 UART_STATUS_t UART_StartReceiveDMA(const UART_t *const
handle, uint8_t* data_ptr, uint32_t count)
662 {
663     UART_STATUS_t ret_stat = UART_STATUS_MODE_MISMATCH;
664     UART_RUNTIME_t * ptr_runtime = handle->runtime;
665     const UART_DMA_CONFIG_t * ptr_dma_config = handle->config-
>receive_dma_config;
666     XMC_DMA_t * ptr_gpdma = handle->config->global_dma->dma;
667
668     XMC_ASSERT("UART_StartReceiveDMA: UART APP handle
invalid", (((handle != NULL)&&
669 (handle->runtime != NULL)) && (handle->config != NULL)))
670
671     if (handle->config->receive_mode ==
UART_TRANSFER_MODE_DMA)

```

```

672 {
673 ret_stat = UART_STATUS_BUSY;
674 if (ptr_runtime->rx_busy == false)
675 {
676 /*If no active reception in progress*/
677 if ((data_ptr != NULL) && ((count > 0U) && (count <=
UART_DMA_MAXCOUNT)))
678 {
679 /*Obtain the address of data buffer and
680 * number of data bytes to be received*/
681 ptr_runtime->rx_data = data_ptr;
682 ptr_runtime->rx_data_count = count;
683 ptr_runtime->rx_busy = true;
684 ptr_runtime->rx_data_index = 0U;
685
686 XMC_USIC_CH_EnableEvent(handle->channel,
687 (uint32_t)
((uint32_t)XMC_USIC_CH_EVENT_STANDARD_RECEIVE |
(uint32_t)XMC_USIC_CH_EVENT_ALTERNATIVE_RECEIVE));
688 ret_stat = UART_STATUS_SUCCESS;
689
690 /*Enable DMA channel*/
691 XMC_DMA_CH_SetBlockSize(ptr_gpdma, ptr_dma_config-
>dma_channel, count);
692 XMC_DMA_CH_SetSourceAddress(ptr_gpdma, ptr_dma_config-
>dma_channel, (uint32_t)&(handle->channel->RBUF));
693 XMC_DMA_CH_SetDestinationAddress(ptr_gpdma,
ptr_dma_config->dma_channel, (uint32_t)data_ptr);
694 XMC_DMA_CH_Enable(ptr_gpdma, ptr_dma_config-
>dma_channel);
695 }
696 else
697 {
698 ret_stat = UART_STATUS_BUFFER_INVALID;
699 }
700 }
701 }

```

```

702 return ret_stat;
703 }
704 #endif
705
706 #ifdef UART_TX_DIRECT_USED
707 /*
708 * Polling method to transmit data.
709 * @param[in] UART_t* handle UART APP handle pointer
710 * @param[in] uint8_t* Pointer to data array
711 * @param[in] uint32_t number of bytes to be transmitted.
712 *
713 * @return UART_STATUS_t Status of transmit request handling.
714 *
715 * Description:
716 * Transmits data by blocking the CPU until all data is sent.
Transmission
717 * cannot be aborted since it is blocking implementation. Based on
FIFO selection,
718 * either TBUF or IN register is updated with the data.
719 *
720 */
721 static UART_STATUS_t UART_IStartTransmitPolling(const UART_t
*const handle, uint8_t* data_ptr, uint32_t count)
722 {
723     UART_STATUS_t ret_stat = UART_STATUS_BUFFER_INVALID;
724     uint32_t loc_index;
725
726     XMC_ASSERT("UART_Transmit: UART APP handle invalid",
(((handle != NULL)&&
727 (handle->runtime != NULL))&&(handle->config != NULL)))
728
729     if ((data_ptr != NULL) && (count > 0U))
730     {
731         ret_stat = UART_STATUS_BUSY;
732         if (handle->runtime->tx_busy == false)
733         {
734             handle->runtime->tx_busy = true;

```

```

735 if (handle->config->tx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
736 {
737 /*Clear the transmit FIFO*/
738 XMC_USIC_CH_TXFIFO_Flush(handle->channel);
739 }
740 /*Loop through each byte*/
741 for (loc_index = 0U; loc_index < count; loc_index++)
742 {
743 /*If FIFO is enabled, FIFO filling status should be checked
744 * to avoid overflow error*/
745 if (handle->config->tx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
746 {
747 /*Wait if transmit FIFO is full*/
748 while (XMC_USIC_CH_TXFIFO_IsFull(handle->channel) == true)
749 {
750 }
751 }
752 XMC_UART_CH_Transmit(handle->channel,
(uint16_t)data_ptr[loc_index]);
753 }
754
755 if (handle->config->tx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
756 {
757 /*Wait till FIFO is empty*/
758 while (XMC_USIC_CH_TXFIFO_IsEmpty(handle->channel) ==
false)
759 {
760 }
761 }
762 ret_stat = UART_STATUS_SUCCESS;
763 handle->runtime->tx_busy = false;
764 }
765 }
766 return ret_stat;

```

```

767 }
768 #endif
769
770 #ifdef UART_RX_DIRECT_USED
771 /*
772 * Polling method to receive data.
773 * @param[in] UART_t* handle UART APP handle pointer
774 * @param[in] uint8_t* Pointer to data array
775 * @param[in] uint32_t number of bytes to be received.
776 *
777 * @return UART_STATUS_t Status of receive request handling.
778 *
779 * Description:
780 * Receives data by blocking the CPU until all data is received.
Reception
781 * cannot be aborted since it is blocking implementation. Based on
FIFO selection,
782 * either RBUF or OUT register will be read.
783 *
784 */
785 static UART_STATUS_t UART_IStartReceivePolling(const UART_t
*const handle, uint8_t* data_ptr, uint32_t count)
786 {
787     UART_STATUS_t ret_stat = UART_STATUS_BUFFER_INVALID;
788     uint32_t loc_index;
789     uint32_t loc_status;
790
791     XMC_ASSERT("UART_Receive: UART APP handle invalid",
((handle != NULL)&&
792 (handle->runtime != NULL)))
793
794     if ((data_ptr != NULL) && (count > 0U))
795     {
796         ret_stat = UART_STATUS_BUSY;
797         if (handle->runtime->rx_busy == false)
798         {
799             handle->runtime->rx_busy = true;

```

```

800 if (handle->config->rx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
801 {
802 /*Clear the receive FIFO, configure the trigger lime
803 * and enable the receive events*/
804 XMC_USIC_CH_RXFIFO_Flush(handle->channel);
805 }
806 for (loc_index = 0U; loc_index < count; loc_index++)
807 {
808 /*If receive FIFO is configured, wait for FIFO to get data.*/
809 if (handle->config->rx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
810 {
811 /*Wait if FIFO empty*/
812 while(XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel) ==
true)
813 {
814 }
815 }
816 else
817 {
818 /*Wait for RIF or AIF flag update*/
819 loc_status = XMC_UART_CH_GetStatusFlag(handle->channel);
820 while (!(loc_status &
((uint32_t)XMC_UART_CH_STATUS_FLAG_ALTERNATIVE_RECEIVE_
|
821
(uint32_t)XMC_UART_CH_STATUS_FLAG_RECEIVE_INDICATION)))
822 {
823 loc_status = XMC_UART_CH_GetStatusFlag(handle->channel);
824 }
825 /*Clear the detected event.
826 * Both events should not be cleared at once, otherwise if 2 bytes
are received, only
827 * one byte will be read.*/
828 XMC_UART_CH_ClearStatusFlag(handle->channel,
829

```

```

((uint32_t)XMC_UART_CH_STATUS_FLAG_RECEIVE_INDICATION |
(uint32_t)XMC_UART_CH_STATUS_FLAG_ALTERNATIVE_RECEIVE_I
830 }
831 data_ptr[loc_index] =
(uint8_t)XMC_UART_CH_GetReceivedData(handle->channel);
832 }
833 ret_stat = UART_STATUS_SUCCESS;
834 handle->runtime->rx_busy = false;
835 }
836 }
837 return ret_stat;
838 }
839 #endif
840
841 #ifdef UART_TX_INTERRUPT_USED
842 /*
843 * Transmit interrupt handler for the APP.
844 * This is a common interrupt handling function called for different
instances of the APP.
845 *
846 * * param[in] handle UART APP handle pointer of type UART_t*
847 *
848 * * return void
849 */
850 void UART_ITransmitHandler(const UART_t * const handle)
851 {
852     UART_RUNTIME_t * ptr_runtime = handle->runtime;
853
854     if (ptr_runtime->tx_data_index < ptr_runtime->tx_data_count)
855     {
856         if (handle->config->tx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
857         {
858             /*When Transmit FIFO is enabled*/
859             /*Fill the transmit FIFO */
860             while (XMC_USIC_CH_TXFIFO_IsFull(handle->channel) == false)
861             {

```

```

862 if (ptr_runtime->tx_data_index < ptr_runtime->tx_data_count)
863 {
864 /*Load the FIFO byte by byte till either FIFO is full or all data is
loaded*/
865 XMC_UART_CH_Transmit(handle->channel,
(uint16_t)ptr_runtime->tx_data[ptr_runtime->tx_data_index]);
866 (ptr_runtime->tx_data_index)++;
867 }
868 else
869 {
870 break;
871 }
872 }
873 }
874 else
875 {
876 /*When Transmit FIFO is disabled*/
877 XMC_UART_CH_Transmit(handle->channel,
(uint16_t)ptr_runtime->tx_data[ptr_runtime->tx_data_index]);
878 (ptr_runtime->tx_data_index)++;
879 }
880 }
881 else
882 {
883 if (XMC_USIC_CH_TXFIFO_IsEmpty(handle->channel) == true)
884 {
885 if (handle->config->tx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
886 {
887 /*Disable the transmit FIFO event*/
888 XMC_USIC_CH_TXFIFO_DisableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD);
889 }
890 else
891 {
892 /*Disable the standard transmit event*/
893 XMC_USIC_CH_DisableEvent(handle->channel,

```

```

(uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
894 }
895
896 /*Wait for the transmit buffer to be free to ensure that all data is
transmitted*/
897 while (XMC_USIC_CH_GetTransmitBufferStatus(handle-
>channel) == XMC_USIC_CH_TBUF_STATUS_BUSY)
898 {
899
900 }
901 /*All data is transmitted*/
902 ptr_runtime->tx_busy = false;
903 ptr_runtime->tx_data = NULL;
904
905 if (handle->config->tx_cbhandler != NULL)
906 {
907 /*Execute the callback function provided in the UART APP UI*/
908 handle->config->tx_cbhandler();
909 }
910 }
911 }
912 }
913 #endif
914
915 #ifdef UART_RX_INTERRUPT_USED
916 /*
917 * Receive interrupt handler for the APP.
918 * This is a common interrupt handling function for different
instances of the UART APP.
919 *
920 * param[in] handle UART APP handle pointer of type UART_t*
921 *
922 * return void
923 */
924 void UART_IReceiveHandler(const UART_t * const handle)
925 {
926 UART_RUNTIME_t * ptr_runtime = handle->runtime;

```

```

927
928 if (handle->config->rx_fifo_size !=
XMC_USIC_CH_FIFO_DISABLED)
929 {
930 /*When Receive FIFO is enabled*/
931 while (XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel) ==
false)
932 {
933 if (ptr_runtime->rx_data_index < ptr_runtime->rx_data_count)
934 {
935 /*Read all the content of Receive FIFO */
936 ptr_runtime->rx_data[ptr_runtime->rx_data_index] =
(uint8_t)XMC_UART_CH_GetReceivedData(handle->channel);
937 (ptr_runtime->rx_data_index)++;
938 }
939
940 if (ptr_runtime->rx_data_index == ptr_runtime->rx_data_count)
941 {
942 /*Reception complete*/
943 ptr_runtime->rx_busy = false;
944 /*Disable both standard receive and alternative receive FIFO
events*/
945 XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel,
946 (uint32_t)
((uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_STANDARD |
947
(uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_ALTERNATE));
948 if (handle->config->rx_cbhandler != NULL)
949 {
950 /*Execute the 'End of reception' callback function*/
951 handle->config->rx_cbhandler();
952 }
953 break;
954 }
955 }
956 /*Set the trigger limit if data still to be received*/
957 if (ptr_runtime->rx_data_index < ptr_runtime->rx_data_count)

```

```

958 {
959 UART_IReconfigureRxFIFO(handle,
960 (uint32_t)(ptr_runtime->rx_data_count - ptr_runtime-
>rx_data_index));
961 }
962 }
963 else
964 {
965 /*When RxFIFO is disabled*/
966 if (ptr_runtime->rx_data_index < ptr_runtime->rx_data_count)
967 {
968 ptr_runtime->rx_data[ptr_runtime->rx_data_index] =
(uint8_t)XMC_UART_CH_GetReceivedData(handle->channel);
969 (ptr_runtime->rx_data_index)++;
970 }
971
972 if (ptr_runtime->rx_data_index == ptr_runtime->rx_data_count)
973 {
974 /*Reception complete*/
975 ptr_runtime->rx_busy = false;
976 /*Disable both standard receive and alternative receive FIFO
events*/
977 XMC_USIC_CH_DisableEvent(handle->channel,
978 (uint32_t)
((uint32_t)XMC_USIC_CH_EVENT_ALTERNATIVE_RECEIVE |
(uint32_t)XMC_USIC_CH_EVENT_STANDARD_RECEIVE));
979
980 if (handle->config->rx_cbhandler != NULL)
981 {
982 /*Execute the 'End of reception' callback function*/
983 handle->config->rx_cbhandler();
984 }
985 }
986 }
987 }
988
989 /*

```

```

990 * A local function to reconfigure Receive FIFO with the given size
and trigger limit.
991 * Size is needed because the FIFO should be disabled before
changing the trigger limit by
992 * clearing the FIFO size.
993 *
994 * param[in] UART_t * pointer to the UART APP handle
995 * param[in] uint8_t number of bytes to be received.
996 *
997 * return void.
998 */
999 static void UART_IReconfigureRxFIFO(const UART_t * const
handle, uint32_t data_size)
1000 {
1001     uint32_t fifo_size;
1002     uint32_t ret_limit_val = 0U;
1003
1004     /*Get FIFO size in bytes*/
1005     fifo_size = (uint32_t)(0x01UL << (uint8_t)(handle->config-
>rx_fifo_size));
1006     /*If data size is more than FIFO size, configure the limit to the
FIFO size*/
1007     if (data_size < fifo_size)
1008     {
1009         ret_limit_val = (uint32_t)(data_size - 1U);
1010     }
1011     else
1012     {
1013         ret_limit_val = (uint32_t)(fifo_size - 1U);
1014     }
1015     /*Set the limit value*/
1016     XMC_USIC_CH_RXFIFO_SetSizeTriggerLimit(handle->channel,
1017     handle->config->rx_fifo_size, ret_limit_val);
1018 }
1019 #endif
1020
1021 #ifdef UART_PROTOCOL_EVENT_USED

```

```

1022 /*
1023 * Protocol interrupt handling function.
1024 * The function is common for different instances of the UART
APP.
1025 *
1026 * param[in] handle UART APP handle pointer of type UART_t*
1027 *
1028 * return void
1029 */
1030 void UART_IProtocolHandler(const UART_t * const handle)
1031 {
1032 /*Protocol status value to check which event occurred*/
1033 uint32_t psr_status = XMC_UART_CH_GetStatusFlag(handle-
>channel);
1034 /*Protocol event configuration to check which event is
1035 * configured for interrupt generation and hence callback*/
1036 uint32_t pcr_conf = handle->channel->PCR_ASCMode;
1037 /*Array of callback functions in the order of events*/
1038 const UART_cbhandler callback_arr[UART_EVENT_MAX] = {
1039 handle->config->sync_error_cbhandler,
1040 handle->config->rx_noise_error_cbhandler,
1041 handle->config->format_error_bit0_cbhandler,
1042 handle->config->format_error_bit1_cbhandler,
1043 handle->config->collision_error_cbhandler
1044 };
1045 UART_EVENT_t loc_index;
1046
1047 for (loc_index = UART_EVENT_SYNC_BRK; loc_index <
UART_EVENT_MAX; loc_index++)
1048 {
1049 /*Check if event is configured for interrupt generation and event
has occurred*/
1050 if ((pcr_conf & (uint32_t)uart_event_conf_flags[loc_index]) &&
1051 (psr_status & (uint32_t)uart_event_status_flags[loc_index]))
1052 {
1053 XMC_UART_CH_ClearStatusFlag(handle->channel,
(uint32_t)uart_event_status_flags[(uint32_t)loc_index]);

```

```
1054 /*Call the callback function if it is valid*/
1055 if ((callback_arr[(uint32_t)loc_index] != NULL))
1056 {
1057     callback_arr[(uint32_t)loc_index]();
1058 }
1059 /*Process only one event*/
1060 break;
1061 }
1062 }
1063 }
1064 #endif
```

UART

Home

Enumerations

enum **UART_STATUS** {
 UART_STATUS_SUCCESS,
 UART_STATUS_FAILURE,
 UART_STATUS_BUSY,
 UART_STATUS_BUFFER_OVERFLOW,
 UART_STATUS_MODE_MISMATCH
}

Enum to describe the possible values, returned by **UART_A**

enum **UART_MODE** {
 UART_MODE_FULLDUPLEX,
 UART_MODE_HALFDUPLEX,
 UART_MODE_LOOPBACK
}

Enum used to describe the mode of operation. More...

enum **UART_EVENT** {
 UART_EVENT_SYNC_BREAK,
 UART_EVENT_RX_NOISE,
 UART_EVENT_FORMAT_ERROR,
 UART_EVENT_FORMAT_ERROR,
 UART_EVENT_COLLISION,
 UART_EVENT_MAX
}

Enum used to identify **UART** event callback function. More...

UART_TRANSFER_MODE

enum **UART_TRANSFER_MODE**
UART_TRANSFER_MODE
UART_TRANSFER_MODE
Enum used to identify the tra
used for either transmit or re
function. [More...](#)

typedef enum **UART_STATUS** **UART_STATUS_t**
Enum to describe the possib
values, returned by **UART A**

typedef enum **UART_MODE** **UART_MODE_t**
Enum used to describe the **U**
of operation.

typedef enum **UART_EVENT** **UART_EVENT_t**
Enum used to identify **UART**
event callback function.

typedef enum **UART_TRANSFER_MODE** **UART_TRANSFER_MODE**
Enum used to identify the tra
used for either transmit or re
function.

Detailed Description

Enumeration Type Documentation

enum **UART_EVENT**

Enum used to identify **UART** protocol event callback function.

Enumerator	
<i>UART_EVENT_SYNC_BRK</i>	Synchronization break detected event
<i>UART_EVENT_RX_NOISE</i>	Receiver noise detected event
<i>UART_EVENT_FORMAT_ERR0</i>	Frame format error at stop bit 0 event
<i>UART_EVENT_FORMAT_ERR1</i>	Frame format error at stop bit 1 event
<i>UART_EVENT_COLLISION</i>	Data collision detected in half duplex mode event
<i>UART_EVENT_MAX</i>	Indicates number of UART events supported

Definition at line **134** of file **UART.h**.

enum **UART_MODE**

Enum used to describe the **UART** Mode of operation.

Enumerator	
<i>UART_MODE_FULLDUPLEX</i>	Full Duplex mode selected
<i>UART_MODE_HALFDUPLEX</i>	Half Duplex mode selected
<i>UART_MODE_LOOPBACK</i>	LoopBack mode selected

Definition at line **124** of file **UART.h**.

enum **UART_STATUS**

Enum to describe the possible status values, returned by **UART** APIs.

Enumerator	
<i>UART_STATUS_SUCCESS</i>	Indicates App initialization state successful
<i>UART_STATUS_FAILURE</i>	Unknown error
<i>UART_STATUS_BUSY</i>	UART Busy
<i>UART_STATUS_BUFFER_INVALID</i>	Buffer provided or the buffer size is invalid
<i>UART_STATUS_MODE_MISMATCH</i>	API invoked by a handle

configured with different mode. e.g, If `UART_StartTransmitDMA` is invoked for an instance which has transmit mode configured as "Interrupt", will return this status.

Definition at line **104** of file **UART.h**.

enum UART_TRANSFER_MODE

Enum used to identify the transfer type used for either transmit or receive function.

Enumerator	
<code>UART_TRANSFER_MODE_INTERRUPT</code>	Implement data transmit or receive using interrupts
<code>UART_TRANSFER_MODE_DMA</code>	Implement data transmit or receive using DMA
<code>UART_TRANSFER_MODE_DIRECT</code>	This configuration exposes signals for external APP connection

Definition at line **147** of file **UART.h**.



UART

[Home](#)

[Code](#) >

Code Directory Reference

Files

file [UART.c](#) [code]

file [UART.h](#) [code]



UART

[Home](#)

Methods

DAVE_APP_VERSION_t **UART_GetAppVersion** (void)
Get the **UART** APP version. [More...](#)

UART_STATUS_t **UART_Init** (const **UART_t** *const handle)
Initializes the **UART** module as per the configuration made in UI. [More...](#)

UART_STATUS_t **UART_Receive** (const **UART_t** *const handle, uint8_t *data_ptr, uint32_t count)
Registers a request for receiving data over **UART** channel. [More...](#)

UART_STATUS_t **UART_Transmit** (const **UART_t** *const handle, uint8_t *data_ptr, uint32_t count)
Registers a request for transmitting data over **UART** channel. [More...](#)

UART_STATUS_t **UART_SetBaudrate** (const **UART_t** *handle, uint32_t baud, uint32_t oversampling)
Changes the baudrate of **UART** channel. [More...](#)

__STATIC_INLINE uint32_t **UART_GetTXFIFOStatus** (const **UART_t** *const handle)
Gets the transmit FIFO event flags. [More...](#)

<p><code>__STATIC_INLINE uint8_t</code></p>	<p>UART_GetReceivedWord (const UART_t *const handle) Provides the received data from receive buffer. More...</p>
<p><code>__STATIC_INLINE void</code></p>	<p>UART_TransmitWord (const UART_t *const handle, uint8_t data) Transmits a word of data. More...</p>
<p><code>__STATIC_INLINE void</code></p>	<p>UART_EnableEvent (const UART_t *const handle, uint32_t events) Enables the selected protocol events for interrupt generation. More...</p>
<p><code>__STATIC_INLINE void</code></p>	<p>UART_DisableEvent (const UART_t *const handle, uint32_t events) Disables selected events from generating interrupt. More...</p>
<p><code>__STATIC_INLINE bool</code></p>	<p>UART_IsTXFIFOFull (const UART_t *const handle) Checks if the transmit FIFO is full. More...</p>
<p><code>__STATIC_INLINE bool</code></p>	<p>UART_IsRXFIFOEmpty (const UART_t *const handle) Checks if the receive FIFO is empty. More...</p>
<p><code>__STATIC_INLINE void</code></p>	<p>UART_SetTXFIFOTriggerLimit (const UART_t *const handle, uint32_t limit) Configures trigger limit for the transmit FIFO. More...</p>

<p><code>__STATIC_INLINE void</code></p>	<p>UART_SetRXFIFOTriggerLimit (const UART_t *const handle, uint32_t limit) Configures trigger limit for the receive FIFO. More...</p>
<p><code>__STATIC_INLINE uint32_t</code></p>	<p>UART_GetRXFIFOStatus (const UART_t *const handle) Gets the status of event flags related to receive FIFO. More...</p>
<p><code>__STATIC_INLINE void</code></p>	<p>UART_ClearTXFIFOStatus (const UART_t *const handle, const uint32_t flag) Function clears the specified FIFO event flag related to transmit FIFO. More...</p>
<p><code>__STATIC_INLINE void</code></p>	<p>UART_ClearRXFIFOStatus (const UART_t *const handle, const uint32_t flag) Function clears the specified FIFO event flag related to receive FIFO. It should be used to clear the status of standard receive buffer interrupt, alternative receive buffer interrupt and receive buffer error interrupt flags. More...</p>
<p><code>__STATIC_INLINE uint32_t</code></p>	<p>UART_GetFlagStatus (const UART_t *const handle, uint32_t protocol_status) Provides the status of protocol events. More...</p>
<p><code>__STATIC_INLINE void</code></p>	<p>UART_ClearFlag (const UART_t *const handle, const uint32_t protocol_status) Clears the event status in the</p>

register(PSR_ASCMode). [More...](#)

`__STATIC_INLINE bool` **UART_IsTxBusy** (const **UART_t** *const handle)
Checks if the transmission is in progress. [More...](#)

`__STATIC_INLINE bool` **UART_IsRxBusy** (const **UART_t** *const handle)
Checks if data reception is in progress. [More...](#)

Detailed Description

Methods

Function Documentation

```
__STATIC_INLINE void UART_ClearFlag ( const UART_t *const handle,
                                       const uint32_t protocol_status )
```

Clears the event status in the register(PSR_ASCMode).

Parameters

handle **UART** APP handle pointer of type **UART_t**
protocol_status Event whose status is to be cleared.
Range: Use type
XMC_UART_CH_STATUS_FLAG_t for input.
Multiple events can be combined using *OR*
operation.

Returns

None

Description:

Clears a given protocol event flag bit using the PSCR register. This function is an inline wrapper for the API provided by xmc_uart.h file. The user should mask the input value based on the events to be cleared.

Example Usage:

```
#include <DAVE.h>
//Precondition:
//Configure receive mode as direct and disable receive FIFO and
transmit FIFO.
//Description:
//Transmits each received byte of data.
int main(void)
{
DAVE_STATUS_t init_status;
```

```

uint16_t ReceiveData = 0;
init_status = DAVE_Init();
if(init_status == DAVE_STATUS_SUCCESS)
{
while(1U)
{
//Check if data is received
if(UART_GetFlagStatus(&UART_0,
(XMC_UART_CH_STATUS_FLAG_RECEIVE_INDICATION |
XMC_UART_CH_STATUS_FLAG_ALTERNATIVE_RECEIVE_INDICATI
{
//Read the received data
ReceiveData =
XMC_UART_CH_GetReceivedData(UART_0.channel);
//Transmit the received data
XMC_UART_CH_Transmit(UART_0.channel,(const
uint16_t)ReceiveData);
//Clear the receive flags
UART_ClearFlag(&UART_0,
(XMC_UART_CH_STATUS_FLAG_RECEIVE_INDICATION |
XMC_UART_CH_STATUS_FLAG_ALTERNATIVE_RECEIVE_INDICATI
}
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **2046** of file **UART.h**.

References **UART::channel**.

```
__STATIC_INLINE void UART_ClearRXFIFOStatus ( const UART_t *  
                                              const uint32_t  
                                              )
```

Function clears the specified FIFO event flag related to receive FIFO. It should be used to clear the status of standard receive buffer interrupt, alternative receive buffer interrupt and receive buffer error interrupt flags.

Parameters

handle **UART** APP handle pointer of type **UART_t**
flag Value with event bits at the bit positions in TRBSR register to be cleared.
Range: Use type XMC_USIC_CH_RXFIFO_EVENT_t for providing events. Multiple events can be input by using *OR* operation.

Returns

None

Description:

Function clears a status bit in TRBSR using the TRBSCR register. The function does not mask the input value to clear only receive buffer events. So user should appropriately mask the input value before calling the function.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation  
(includes SFR declaration)  
//Precondition: Configure transmit mode and receive mode as direct.  
//Description: Receives data of 10 bytes and transmits the same  
bytwise.  
int main(void)  
{  
  UART_STATUS_t init_status;
```

```

uint8_t ReadData[10];
uint8_t index = 0;
init_status = (UART_STATUS_t)UART_Init(&UART_0);
if(init_status == UART_STATUS_SUCCESS)
{
//Configure receive FIFO to generate event when one byte is received.
UART_SetRXFIFOTriggerLimit(&UART_0, 0);
while(1U)
{
//Check if receive event is generated
if(UART_GetRXFIFOStatus(&UART_0) &
XMC_USIC_CH_RXFIFO_EVENT_STANDARD)
{
//Clear receive event
UART_ClearRXFIFOStatus(&UART_0,
XMC_USIC_CH_RXFIFO_EVENT_STANDARD);
//Read received data from FIFO
ReadData[index] =
(uint8_t)XMC_USIC_CH_RXFIFO_GetData((XMC_USIC_CH_t
*)&UART_0.channel);
//Transmit received data
UART_Transmit(&UART_0, &ReadData[index], 1);
index++;
index = index % 10;
}
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **1899** of file **UART.h**.

References **UART::channel**.

```
__STATIC_INLINE void UART_ClearTXFIFOStatus ( const UART_t *  
                                              const uint32_t  
                                              )
```

Function clears the specified FIFO event flag related to transmit FIFO.

Parameters

handle **UART** APP handle pointer of type **UART_t**

flag Value with event bits at their bit positions in TRBSR register to be cleared.

Range: Use type XMC_USIC_CH_TXFIFO_EVENT_t. Multiple events can be combined using *OR* operation.

Returns

None

Description:

Function clears a status bit in TRBSR register using the TRBSCR register. But the function does not mask the input value with the bit positions restricted to transmit FIFO status bits. User should ensure that the input value is appropriately masked.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation  
(includes SFR declaration)  
//Precondition: Configure transmit mode as direct.  
//Description: Transmits the string "Infineon" bitwise.  
int main(void)  
{  
  UART_STATUS_t init_status;  
  uint8_t Send_Data[] = "Infineon";
```

```

uint8_t index = 0;
init_status = (UART_STATUS_t)UART_Init(&UART_0);
if(init_status == UART_STATUS_SUCCESS)
{
while(index < sizeof(Send_Data))
{
//Put every byte to FIFO.
XMC_USIC_CH_TXFIFO_PutData(UART_0.channel,
(uint16_t)Send_Data[index]);
index++;
//Wait for FIFO transmit standar buffer interrupt to fill it again with
remaining data
while((UART_GetTXFIFOStatus(&UART_0) &
XMC_USIC_CH_TXFIFO_EVENT_STANDARD) == 0);
UART_ClearTXFIFOStatus(&UART_0,
XMC_USIC_CH_TXFIFO_EVENT_STANDARD);
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **1828** of file **UART.h**.

References **UART::channel**.

```

__STATIC_INLINE void UART_DisableEvent ( const UART_t *const
                                         uint32_t
                                         )

```

Disables selected events from generating interrupt.

Parameters

handle **UART** APP handle pointer of type **UART_t**

events Events to be disabled from generating interrupt.
: Use type to select the event. Multiple events can be combined using the bitwise OR operation.

Returns

None

Description:

Events are disabled by clearing their respective bits in either CCR, TBCTR or RBCTR.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
// Precondition:
// Add INTERRUPT APP and connect the UART APP protocol
interrupt signal to the INTERRUPT APP
// irq signal. Provide the callback function name in INTERRUPT APP
as "ProtocolInterrupt".
//
// Description: The example configures protocol interrupt for data loss
detection.
// When the data loss interrupt occurs, the receive FIFO is cleared.
After the receive FIFO
// is cleared, the channel can receive few bytes till the FIFO gets filled.
int main(void)
{
DAVE_STATUS_t status;
status = DAVE_Init();
if(status == DAVE_STATUS_FAILURE)
{
XMC_DEBUG("DAVE APPs initialization failed\n");
```

```

while(1U)
{
}
}
//Enable interrupt generation when data loss is detected
UART_EnableEvent(&UART_0,
XMC_UART_CH_EVENT_DATA_LOST);
while(1U)
{
}
}
void ProtocolInterrupt()
{
uint8_t txt_msg[]="Receiver data loss detected";
UART_Transmit(&UART_0, txt_msg, sizeof(txt_msg));
UART_DisableEvent(&UART_0,
XMC_UART_CH_EVENT_DATA_LOST);
//Clear receive FIFO so that data will be received.
XMC_USIC_CH_RXFIFO_Flush(UART_0.channel);
UART_EnableEvent(&UART_0,
XMC_UART_CH_EVENT_DATA_LOST);
}

```

Definition at line **1432** of file **UART.h**.

References **UART::channel**.

```

__STATIC_INLINE void UART_EnableEvent ( const UART_t *const
                                         uint32_t
                                         )

```

Enables the selected protocol events for interrupt generation.

Parameters

handle **UART** APP handle pointer of type **UART_t**

events Protocol events to be enabled for interrupt generation.

: Use type to select the event. Multiple events can be combined using the bitwise OR operation.

Returns

None

Description:

Enables the events by configuring CCR or PCR register based on the event. When the event is enabled, an interrupt can be generated on occurrence of the event. The API should be used only for *Direct* mode related events. Using this API for non *Direct* mode may not yield expected result.

Example Usage:

```
* #include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
// Precondition:
// Disable receive FIFO.
// Add INTERRUPT APP and connect the UART APP protocol
interrupt signal to the INTERRUPT APP
// irq signal. Provide the callback function name in INTERRUPT APP
as "ProtocolInterrupt".
//
// Description:
// Generates an event when data loss is detected and transmits a
relevent message.
// To generate the event, transmit more than 2 bytes of data to the
UART channel .
int main(void)
{
DAVE_STATUS_t status;
status = DAVE_Init();
if(status == DAVE_STATUS_FAILURE)
{
XMC_DEBUG("DAVE APPs initialization failed\n");
while(1U)
{
```

```

}
}
//Enable the interrupt generation when data loss is detected.
UART_EnableEvent(&UART_0,
XMC_UART_CH_EVENT_DATA_LOST);
while(1U)
{
}
}
//Protocol interrupt handler
void ProtocolInterrupt()
{
uint8_t txt_msg[]="Receiver data loss detected";
//Transmit the message to indicate data loss
UART_Transmit(&UART_0, txt_msg, sizeof(txt_msg));
}

```

Definition at line **1369** of file **UART.h**.

References **UART::channel**.

DAVE_APP_VERSION_t UART_GetAppVersion (void)

Get the **UART** APP version.

Returns

DAVE_APP_VERSION_t APP version information (major, minor and patch number)

Example Usage:

```

//Description:
//Transmits the text "UART APP supported.", if the UART APP version
is v4.1.x, where x can be any value.
#include <DAVE.h>
int main(void)
{

```

```

UART_STATUS_t init_status;
DAVE_APP_VERSION_t uart_version;
uint8_t valid_str[] = "UART APP supported.";
init_status = (UART_STATUS_t)UART_Init(&UART_0);
if(init_status == UART_STATUS_SUCCESS)
{
uart_version = UART_GetAppVersion();
if((uart_version.major == 4) &&
(uart_version.minor == 1))
{
UART_Transmit(&UART_0, valid_str, sizeof(valid_str));
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **122** of file **UART.c**.

```

__STATIC_INLINE uint32_t UART_GetFlagStatus ( const UART_t *c
                                             uint32_t
                                             )

```

Provides the status of protocol events.

Parameters

handle	UART APP handle pointer of type UART_t
protocol_status	Event whose status is to be read.
Range:	Use type XMC_UART_CH_STATUS_FLAG_t for input.

Multiple events can be combined using *OR* operation.

Returns

uint32_t: Status of selected protocol events read from PSR_ASCMode register.

Range: Use type XMC_UART_CH_STATUS_FLAG_t for comparing the return value with event bitmasks. Status of multiple events can be checked by combining enum values using *OR* operation while comparing.

Description:

Reads the protocol status bits from the register PSR_ASCMode and compares the values with the input value of selected events. Returns the masked value of selected events with the status register value. This function is an inline wrapper for the API provided by xmc_uart.h file.

Example Usage:

```
#include <DAVE.h>
//Precondition: Configure transmit mode as Interrupt
//Description: Transmits the string "Infineon", waits for transmit buffer to
go idle and then receives 10 bytes.
//Transmits the received 10 bytes.
int main(void)
{
DAVE_STATUS_t init_status;
uint8_t Send_Data[] = "Infineon";
uint8_t ReceiveData[10] = {0};
init_status = DAVE_Init();
if(init_status == DAVE_STATUS_SUCCESS)
{
//Send the first string.
//Check if the request to transmit is accepted.
while(UART_Transmit(&UART_0,Send_Data, sizeof(Send_Data)) ==
UART_STATUS_BUSY)
{
```

```

}
while(1U)
{
//Check if transmit buffer is idle
if(UART_GetFlagStatus(&UART_0,
XMC_UART_CH_STATUS_FLAG_TRANSMISSION_IDLE))
{
//Check if receive request is successful
if(UART_Receive(&UART_0, ReceiveData, 10) ==
UART_STATUS_SUCCESS)
{
//Wait for reception of 10 bytes
while(UART_0.runtime->rx_busy)
{
}
//Transmit the received data.
UART_Transmit(&UART_0, ReceiveData, 10);
}
}
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **1979** of file **UART.h**.

References **UART::channel**.

```

__STATIC_INLINE uint8_t UART_GetReceivedWord ( const UART_t

```

Provides the received data from receive buffer.

Parameters

handle **UART** APP handle pointer of type **UART_t**

Returns

uint8_t: Data read from RBUF.

Description:

This can be used in receive mode "Direct" to read the received data. If Rx FIFO is not configured, function reads the value of RBUF register. Otherwise it reads the data from OUTF register. User can poll for receive event or configure an interrupt by connecting the external INTERRUPT APP to receive event signals. This API can be used inside the ISR to read the received data.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Pre-condition:
//Configure transmit mode and receive mode as "Direct" with transmit
FIFO and receive FIFO enabled
//Description:
//Transmits the string "Infineon", receives 10 bytes and retransmits the
received 10 bytes.
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t Send_Data[] = "Infineon";
    uint8_t Rec_Data[10];
    uint8_t index = 0;
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        //Transmit the string "Infineon"
        while(index < sizeof(Send_Data))
```

```

{
UART_TransmitWord(&UART_0,Send_Data[index]);
index++;
//Wait for transmit buffer interrupt to fill it again with remaining data
while((UART_GetTXFIFOStatus(&UART_0) &
XMC_USIC_CH_TXFIFO_EVENT_STANDARD) == 0);
UART_ClearTXFIFOStatus(&UART_0,
XMC_USIC_CH_TXFIFO_EVENT_STANDARD);
}
//Configure receive FIFO trigger limit to 9.
UART_SetRXFIFOTriggerLimit(&UART_0, 9);
//Receive 10 bytes input
index = 0;
//Wait till 10 bytes are received
while(!(UART_GetRXFIFOStatus(&UART_0) &
(XMC_USIC_CH_RXFIFO_EVENT_STANDARD |
XMC_USIC_CH_RXFIFO_EVENT_ALTERNATE)))
{
Rec_Data[index] = UART_GetReceivedWord(&UART_0);
index++;
if(index == 10)
{
break;
}
}
//Transmit the received data
index = 0;
while(index < 10)
{
UART_TransmitWord(&UART_0,Rec_Data[index]);
index++;
//Wait for transmit buffer interrupt to fill it again with remaining data
while((UART_GetTXFIFOStatus(&UART_0) &
XMC_USIC_CH_TXFIFO_EVENT_STANDARD) == 0);
UART_ClearTXFIFOStatus(&UART_0,
XMC_USIC_CH_TXFIFO_EVENT_STANDARD);
}
}

```

```

}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **1244** of file **UART.h**.

References **UART::channel**.

__STATIC_INLINE uint32_t UART_GetRXFIFOStatus (const UART_

Gets the status of event flags related to receive FIFO.

Parameters

handle **UART** APP handle pointer of type **UART_t**

Returns

uint32_t: Status of standard receive buffer event, alternative receive buffer event and receive buffer error event in their bit positions in TRBSR register.

Range: Use type XMC_USIC_CH_RXFIFO_EVENT_t for event bitmasks. Multiple events' status can be combined for comparison using *OR* operation.

Description:

It provides the status of standard receive buffer event, alternative receive buffer event and receive buffer error event. Function masks the TRBSR register with the bitmask of SRBI, ARBI and RBERI flags. User has to mask the bits of interest before checking the status.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Precondition: Configure transmit mode and receive mode as direct.
//Description: Receives data of 10 bytes and retransmits it.
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t ReadData[10];
    uint8_t index = 0;
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        //Configure the receive FIFO event to generate when one byte is
        received.
        UART_SetRXFIFOTriggerLimit(&UART_0, 0);
        while(1U)
        {
            //Check if receive FIFO event is generated
            if(UART_GetRXFIFOStatus(&UART_0) &
            XMC_USIC_CH_RXFIFO_EVENT_STANDARD)
            {
                UART_ClearRXFIFOStatus(&UART_0,
                XMC_USIC_CH_RXFIFO_EVENT_STANDARD);
                //Read received data
                ReadData[index] =
                (uint8_t)XMC_USIC_CH_RXFIFO_GetData((XMC_USIC_CH_t
                *)&UART_0.channel);
                //Transmit received data
                UART_Transmit(&UART_0, &ReadData[index], 1);
                index++;
                index = index % 10;
            }
        }
    }
    else
```

```
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **1766** of file **UART.h**.

References **UART::channel**.

__STATIC_INLINE uint32_t UART_GetTXFIFOStatus (const UART_t

Gets the transmit FIFO event flags.

Parameters

handle **UART** APP handle pointer of type **UART_t**

Returns

uint32_t: Status of the STBI and TBERI bits in TRBSR register in their bit positions.

Range: Use type XMC_USIC_CH_TXFIFO_EVENT_t for the bitmask of events.

Description:

Function reads the value of TRBSR register. It masks the standard transmit buffer interrupt flag and transmit buffer error flag before providing the value. User has to mask the bits of interest before checking the status.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Pre-condition:
```

```

//Configure transmit mode as "Direct" with transmit FIFO enabled.
//Description:
//Transmits the string "Infineon".
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t Send_Data[] = "Infineon";
    uint8_t index = 0;
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        //Clear the Tx FIFO standard transmit buffer event.
        UART_ClearTXFIFOStatus(&UART_0,
            XMC_USIC_CH_TXFIFO_EVENT_STANDARD);
        //Iterate for the length of the string "Infineon"
        while(index < sizeof(Send_Data)-1)
        {
            //Put the character in the transmit FIFO.
            XMC_USIC_CH_TXFIFO_PutData((XMC_USIC_CH_t
                *)UART_0.channel,(uint16_t)Send_Data[index]);
            index++;
            //Wait for FIFO transmit standard buffer interrupt to fill it again with
            remaining data
            while((UART_GetTXFIFOStatus(&UART_0) &
                XMC_USIC_CH_TXFIFO_EVENT_STANDARD) == 0);
            UART_ClearTXFIFOStatus(&UART_0,
                XMC_USIC_CH_TXFIFO_EVENT_STANDARD);
        }
    }
    else
    {
        XMC_DEBUG("main: Application initialization failed");
        while(1U)
        {
        }
    }
    return 1U;
}

```

```
}
```

Definition at line **1153** of file **UART.h**.

References **UART::channel**.

UART_STATUS_t UART_Init (const UART_t *const handle)

Initializes the **UART** module as per the configuration made in UI.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

UART_STATUS_t: Status of **UART** driver initialization.

UART_STATUS_SUCCESS - on successful initialization.

UART_STATUS_FAILURE - if initialization fails.

UART_STATUS_BUSY - if **UART** channel is busy.

Description:

Initializes IO pins used for the **UART** communication, configures USIC registers based on the settings provided in the GUI.

Calculates divider values PDIV and STEP for a precise baudrate. It also enables configured interrupt flags and service request values.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation  
(includes SFR declaration)  
int main(void)  
{  
    UART_STATUS_t init_status;  
    init_status = (UART_STATUS_t)UART_Init(&UART_0);  
    if(init_status == UART_STATUS_SUCCESS)  
{
```

```

while(1U)
{
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **143** of file **UART.c**.

References **UART::config**, **UART_CONFIG::fptr_uart_config**, **UART::runtime**, and **UART_STATUS_SUCCESS**.

__STATIC_INLINE bool UART_IsRxBusy (const UART_t *const handle

Checks if data reception is in progress.

Parameters

handle **UART** APP handle pointer of type **UART_t**

Returns

bool: Status of data reception.

Range: true - if reception is ongoing.

false- if reception is not active.

Description:

Indicates if the communication channel is configured for receiving data, initiated using UART_Receive, UART_StartReceiveIRQ or UART_StartReceiveDMA API.

Example Usage:

```

#include <DAVE.h>
//Pre-condition:
//Transmit mode and receive mode should be configured as
"Interrupt".
//Description:
//Receives 10 bytes of data and transmits the same.
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t ReadData[10];
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        while(1)
        {
            //Start reception of 10 bytes. The status will be returned success, if the
            channel is not busy.
            if(UART_StartReceiveIRQ(&UART_0, ReadData, 10) ==
            UART_STATUS_SUCCESS)
            {
                //Wait till the data is received.
                while(UART_IsRxBusy(&UART_0))
                {
                }
                //Transmit the received data.
                UART_Transmit(&UART_0, ReadData, 10);
                while(UART_IsTxBusy(&UART_0))
                {
                }
            }
        }
    }
    else
    {
        XMC_DEBUG("main: Application initialization failed");
        while(1U)
        {

```

```
}  
}  
return 1U;  
}
```

Definition at line **2181** of file **UART.h**.

References **UART::runtime**, and **UART_RUNTIME::rx_busy**.

__STATIC_INLINE bool UART_IsRXFIFOEmpty (const UART_t *con

Checks if the receive FIFO is empty.

Parameters

handle **UART** APP handle pointer of type **UART_t**

Returns

bool Status of receive FIFO filling level. : - if receive FIFO is empty.
- if receive FIFO still has data.

Description:

Checks the status using the register TRBSR. Can be used while reading data from the receive FIFO.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation  
(includes SFR declaration)  
// Precondition:  
// Receive mode should be "Direct"  
//  
// Description:  
// Receives 10 bytes and transmits the received 10 bytes.  
uint8_t send_text[] = "Enter 10 bytes:";  
uint8_t rec_data[10];  
int main(void)
```

```

{
DAVE_STATUS_t status;
uint32_t loc_index;
status = DAVE_Init(); // Initialization of DAVE APPs
if(status == DAVE_STATUS_FAILURE)
{
XMC_DEBUG("DAVE APPs initialization failed\n");
while(1U)
{
}
}
UART_Transmit(&UART_0, send_text, sizeof(send_text));
for(loc_index = 0; loc_index < sizeof(rec_data); loc_index++)
{
//Wait when Rx FIFO is empty
while(UART_IsRXFIFOEmpty(&UART_0))
{
}
rec_data[loc_index] = UART_GetReceivedWord(&UART_0);
}
//Transmit the received data
UART_Transmit(&UART_0, rec_data, sizeof(rec_data));
while(1U)
{
}
}

```

Definition at line **1553** of file **UART.h**.

References **UART::channel**.

__STATIC_INLINE bool UART_IsTxBusy (const UART_t *const har

Checks if the transmission is in progress.

Parameters

handle UART APP handle pointer of type **UART_t**

Returns

bool: Status of data transmission.

Range: true - if transmission is ongoing.

false- if transmission is not active.

Description:

Indicates if the communication channel is busy in transmitting data provided using UART_Transmit, UART_StartTransmitIRQ or UART_StartTransmitDMA API.

Example Usage:

```
#include <DAVE.h>
//Pre-condition:
//Transmit mode should be configured as "Interrupt".
//Description:
//Initiates the transmission of one string, aborts the transmission
immediately and
//starts transmission of another string. The receiver might see traces
of first string followed,
//by the complete second string.
int main(void)
{
    UART_STATUS_t init_status;
    //String1
    uint8_t Send_Data[] = "Infineon DAVE application.";
    //String2
    uint8_t NewData[] = "New data message";
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        //Initiate transmission of first string.
        UART_Transmit(&UART_0, Send_Data, sizeof(Send_Data));
        //When the first string is being transmitted,
        if(UART_IsTxBusy(&UART_0))
        {
```

```

//Stop the transmission of first string.
if(UART_AbortTransmit(&UART_0) == UART_STATUS_SUCCESS)
{
//Start the transmission of second string
UART_Transmit(&UART_0, NewData, sizeof(NewData));
//Wait till the transmission is finished.
while(UART_IsTxBusy(&UART_0));
}
}
while(1)
{
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **2116** of file **UART.h**.

References **UART::runtime**, and **UART_RUNTIME::tx_busy**.

__STATIC_INLINE bool UART_IsTXFIFOFull (const UART_t *const

Checks if the transmit FIFO is full.

Parameters

handle **UART** APP handle pointer of type **UART_t**

Returns

bool Status of transmit FIFO filling level. : - if transmit FIFO is

full.

- if transmit FIFO is not full.

Description:

Checks the status using the register TRBSR. Can be used while filling data to the transmit FIFO.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
// Precondition:
// Transmit mode should be "Direct"
//Description:
//Transmits a string using FIFO.
uint8_t send_data[] = "Infineon Technologies";
int main(void)
{
    DAVE_STATUS_t status;
    uint32_t loc_index;
    status = DAVE_Init(); // Initialization of DAVE APPs
    if(status == DAVE_STATUS_FAILURE)
    {
        XMC_DEBUG("DAVE APPs initialization failed\n");
        while(1U)
        {
        }
    }
    for(loc_index = 0; loc_index < sizeof(send_data); loc_index++)
    {
        //Wait when Tx FIFO is full
        while(UART_IsTXFIFOFull(&UART_0))
        {
        }
        UART_TransmitWord(&UART_0, send_data[loc_index]);
    }
    while(1U)
    {
```

```
}  
}
```

Definition at line [1490](#) of file [UART.h](#).

References [UART::channel](#).

```
UART_STATUS_t UART_Receive ( const UART_t *const handle,  
                               uint8_t * data_ptr,  
                               uint32_t count  
                               )
```

Registers a request for receiving data over [UART](#) channel.

Parameters

handle Pointer to UART_t handle structure

data_ptr Pointer to data of type uint8_t.

count Total no of bytes to be received.

Range: minimum= 1, maximum= maximum value supported by type uint32_t.

Returns

UART_STATUS_t: Status for receive request.

[UART_STATUS_SUCCESS](#) if the request is accepted.

[UART_STATUS_BUSY](#) if a reception is in progress.

[UART_STATUS_BUFFER_INVALID](#) if the data_ptr is NULL or count is 0.

Description:

Data will be received asynchronously. After the requested number of data bytes are received, optionally, the user configured callback function will be executed. Data reception is accomplished using the receive mode selected in the UI.

Interrupt:

Based on the UI configuration, either standard receive buffer(RBUF) or receive FIFO(OUT) is used for data reception.

An interrupt is configured for reading received data from the bus. This function only registers a request to receive a number of data bytes from a USIC channel. If FIFO is configured for reception, the FIFO limit is dynamically configured to optimally utilize the CPU load. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. When all the requested number of data bytes are received, the configured callback function will be executed. If a callback function is not configured, the user has to poll for the value of the variable, *handle->runtime->rx_busy* to be false. The value is updated to *false* when all the requested number of data bytes are received.

DMA:

DMA mode is available only in XMC4x family of microcontrollers. In this mode, a DMA channel is configured for receiving data from standard receive buffer(RBUF) to the user buffer. By calling this API, the DMA channel destination address is configured to the user buffer and the channel is enabled. FIFO will not be used when the receive mode is DMA. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. When all the requested number of data bytes are received, the configured callback function will be executed. If a callback function is not configured, the user has to poll for the value of the variable, *handle->runtime->rx_busy* to be false. The value is updated to *false* when all the requested number of data bytes are received.

Direct

In Direct receive mode, neither interrupt nor DMA is used. The API polls the receive flag to read the received data and waits for all the requested number of bytes to be received. Based on FIFO configuration, either RBUF or OUT register is used for reading received data. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. **Note:** *In Direct mode, the API blocks the CPU until the count of bytes requested is received. If this behaviour is not desired, use other APIs like*

UART_GetReceivedWord, UART_GetProtocolStatus etc.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Pre-condition:
//Receive mode should be configured as "Direct".
//Description:
//Transmits 10 bytes of data after receiving 10 bytes of data.
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t ReadData[10];
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        while(1)
        {
            //Receive 10 bytes of data
            if(UART_Receive(&UART_0, ReadData, 10) ==
                UART_STATUS_SUCCESS)
            {
                //Retransmit the received 10 bytes
                UART_Transmit(&UART_0, ReadData, 10);
            }
        }
    }
    else
    {
        XMC_DEBUG("main: Application initialization failed");
        while(1)
        {
        }
    }
    return 1;
}
```

Definition at line **211** of file **UART.c**.

References **UART::config**, **UART_CONFIG::receive_mode**, **UART_STATUS_MODE_MISMATCH**, **UART_TRANSFER_MODE_DIRECT**, **UART_TRANSFER_MODE_DMA**, and **UART_TRANSFER_MODE_INTERRUPT**.

```
UART_STATUS_t UART_SetBaudrate ( const UART_t * handle,  
                                uint32_t           baud,  
                                uint32_t           oversampli  
                                )
```

Changes the baudrate of **UART** channel.

Parameters

UART_t * Pointer to the **UART** APP handle.
baud Value of new baudrate.
oversampling Number of samples to be considered for each symbol. 16 is the standard value.

Returns

UART_STATUS_t **UART_STATUS_SUCCESS** if baudrate changed successfully. **UART_STATUS_FAILURE** if baudrate could not be changed.

Description:

The function stops the channel, calculates the clock divider values to achieve the desired baudrate. Sets the divider values and reconfigures the channel as per the configuration in the UI. The channel is enabled at the end of configuration.

Example Usage: *Please disable the receive FIFO in the 'Advanced Settings' tab*

```
#include <DAVE.h> //Declarations from DAVE Code Generation
```

```

(includes SFR declaration)
//Precondition:
//Disable receive FIFO in the Advanced settings tab.
//Description:
//Waits for user input of new baudrate value. Input is recognized after
line feed is provided.
//Value is set as the new baudrate and the application waits for any
key to be pressed.
//This helps in reconfiguring the terminal application to the newly set
baudrate. On receiving
//new character, message indicating the successful baudrate change
will be displayed using updated
//value of baudrate.
const uint8_t send_askbaud[] = "Please enter desired baudrate:";
const uint8_t send_data[] = "\nPress 'y' to change baudrate to desired
value:";
const uint8_t send_invalid[] = "\nInvalid value!!";
const uint8_t send_success[] = "\nWe made it...Baudrate changed
successfully :-).\n\n";
uint8_t rec_data[11];
int main(void)
{
DAVE_STATUS_t status;
uint32_t baud;
status = DAVE_Init(); // Initialization of DAVE Apps
if(status == DAVE_STATUS_FAILURE)
{
XMC_DEBUG(("DAVE Apps initialization failed with status %d\n",
status));
while(1U)
{
}
}
while(1U)
{
UART_Transmit(&UART_0, send_askbaud, sizeof(send_askbaud)-1);
UART_Receive(&UART_0, rec_data, 10);

```

```

while(UART_0.runtime->tx_busy);
while(UART_0.runtime->rx_busy)
{
//If user enters newline character, accept the value
if((UART_0.runtime->rx_data_index > 0) && (UART_0.runtime->
rx_data[UART_0.runtime->rx_data_index - 1] == 0x0a))
{
//End reception of data on finding newline character
UART_AbortReceive(&UART_0);
}
}
//Add end of string character to the last location
rec_data[UART_0.runtime->rx_data_index] = 0;
//Convert the entered string to number.
baud = atoi(rec_data);
//If the conversion is successful, set the baudrate.
if(baud > 0)
{
//Set the baudrate to USIC channel
if(UART_SetBaudrate(&UART_0, baud, 16) ==
UART_STATUS_SUCCESS)
{
//After changing the baudrate successfully,
//Wait for user to enter a character.
//This wait gives time for the user to change
//the baudrate of the terminal tool used.
UART_Receive(&UART_0, rec_data, 1);
while(UART_0.runtime->rx_busy);
UART_Transmit(&UART_0, send_success, sizeof(send_success)-1);
}
else
{
UART_Transmit(&UART_0, send_invalid, sizeof(send_invalid)-1);
}
}
else
{

```

```
UART_Transmit(&UART_0, send_invalid, sizeof(send_invalid)-1);
}
while(UART_0.runtime->tx_busy);
}
}
```

Definition at line **388** of file **UART.c**.

References **UART::channel**, **UART_TX_CONFIG::config**, **UART::config**, **UART_CONFIG::mode**, **UART_TX_CONFIG::pin**, **UART_TX_CONFIG::port**, **UART::runtime**, **UART_RUNTIME::rx_busy**, **UART_RUNTIME::tx_busy**, **UART_CONFIG::tx_pin_config**, **UART_MODE_LOOPBACK**, **UART_STATUS_BUSY**, and **UART_STATUS_SUCCESS**.

```
__STATIC_INLINE void UART_SetRXFIFOTriggerLimit ( const UART
                                                    uint32_t
                                                    )
```

Configures trigger limit for the receive FIFO.

Parameters

handle **UART** APP handle pointer of type **UART_t**
limit Value of receive FIFO filling level, transition above which the interrupt should be generated.
: 0 to receive FIFO size.
e.g, If receive FIFO size is 16, and limit is configured as 8, FIFO receive buffer interrupt will be generated when the FIFO filling level rises from 8 to 9.

Returns

None

Description:

Receive FIFO trigger limit is configured by setting its value in the RBCTR register. Receive FIFO is configured to generate

interrupt when the FIFO filling level rises above the trigger limit.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Precondition:
//Configure receive mode as "Direct"
//Add an instance of the INTERRUPT APP and connect the UART
event_fifo_receive_buffer_interrupt
//signal to the INTERRUPT sr_irq signal.
//Provide the callback function name in INTERRUPT APP as "rx_cb"
//Description:
uint8_t Rec_Data[10];
uint8_t index = 0;
int main(void)
{
    UART_STATUS_t init_status;
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        //Configure receive interrupt generation when 10 bytes are received
        UART_SetRXFIFOTriggerLimit(&UART_0, 9);
        //Wait for the data to be received.
        while(index < 10);
        //Transmit the received data
        UART_Transmit(&UART_0, Rec_Data, 10);
        //Wait for transmission to finish
        while(UART_0.runtime->tx_busy);
    }
    else
    {
        XMC_DEBUG("main: Application initialization failed");
        while(1U)
        {
        }
    }
}
```

```

return 1U;
}
//INTERRUPT APP callback function
void rx_cb()
{
while((index < 10) &&
(!XMC_USIC_CH_RXFIFO_IsEmpty(UART_0.channel)))
{
//Read data from FIFO
Rec_Data[index] = UART_GetReceivedWord(&UART_0);
index++;
}
}
}

```

Definition at line **1699** of file **UART.h**.

References **UART::channel**, **UART::config**, and **UART_CONFIG::rx_fifo_size**.

```

__STATIC_INLINE void UART_SetTXFIFOTriggerLimit ( const UART
                                                    uint32_t
                                                    )

```

Configures trigger limit for the transmit FIFO.

Parameters

handle **UART** APP handle pointer of type **UART_t**

limit Value of transmit FIFO filling level, transition below which the interrupt should be generated.
: 0 to transmit FIFO size.
e.g, If transmit FIFO size is 16, and limit is configured as 8, FIFO standard transmit buffer interrupt will be generated when the FIFO filling level drops from 8 to 7.

Returns

None

Description:

Transmit FIFO trigger limit is configured by setting its value in the TBCTR register. Transmit FIFO is configured to generate interrupt when the FIFO filling level drops below the trigger limit.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Precondition:
//Configure receive mode as "Direct"
//Add an instance of the INTERRUPT APP and connect the UART
event_fifo_transmit_buffer_interrupt
//signal to the INTERRUPT sr_irq signal.
//Provide the callback function name in INTERRUPT APP as "tx_cb"
//Description:
//Transmits the string "Infineon" using FIFO. Configures the FIFO to
generate event when the FIFO
//is empty. Puts one byte to the FIFO when the event is generated.
uint8_t Send_Data[] = "Infineon";
uint8_t index = 0;
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t Rec_Data[10];
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        //Configure transmit interrupt generation when the transmit FIFO is
empty
        UART_SetTXFIFOTriggerLimit(&UART_0, 1);
        //Put one word to the FIFO
        UART_TransmitWord(&UART_0,Send_Data[index]);
        index++;
    }
    else
```

```

{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
//INTERRUPT APP callback function
void tx_cb()
{
if(index < sizeof(Send_Data))
{
//Put one word to the FIFO
UART_TransmitWord(&UART_0,Send_Data[index]);
index++;
}
}
}

```

Definition at line **1626** of file **UART.h**.

References **UART::channel**, **UART::config**, and **UART_CONFIG::tx_fifo_size**.

```

UART_STATUS_t UART_Transmit ( const UART_t *const handle,
                                uint8_t * data_ptr,
                                uint32_t count
                                )

```

Registers a request for transmitting data over **UART** channel.

Parameters

- handle** **UART** APP handle pointer of type **UART_t**
- data_ptr** Pointer to data of type **uint8_t**.
- count** Total no of words to be transmitted.

Range: minimum= 1, maximum= maximum supported by uint32_t.

Returns

UART_STATUS_t: Status of transmit request.

UART_STATUS_SUCCESS if the request is accepted.

UART_STATUS_BUSY if a transmission is in progress.

UART_STATUS_BUFFER_INVALID if the data_ptr is NULL or count is 0.

Imp Note: Return value should be validated by user to ensure that the request is registered.

Description:

Transmits data using the **UART** channel. Transmission is accomplished using the transmit mode as configured in the UI.

Interrupt:

The data transmission is accomplished using transmit interrupt. User can configure a callback function in the APP UI. When the data is fully transmitted, the callback function will be executed. If transmit FIFO is enabled, the trigger limit is set to 1. So the transmit interrupt will be generated when all the data in FIFO is moved out of FIFO. The APP handle's runtime structure is used to store the data pointer, count, data index and status of transmission. This function only registers a data transmission request if there is no active transmission in progress. Actual data transmission happens in the transmit interrupt service routine. A trigger is generated for the transmit interrupt to start loading the data to the transmit buffer. If transmit FIFO is configured, the data is filled into the FIFO. Transmit interrupt will be generated subsequently when the transmit FIFO is empty. At this point of time, if there is some more data to be transmitted, it is loaded to the FIFO again. When FIFO is not enabled, data is transmitted one byte at a time. On transmission of each byte an interrupt is generated and the next byte is transmitted in the interrupt service routine. Callback function is executed when all the data bytes are transmitted. If a callback function is not configured,

user has to poll for the value of `tx_busy` flag of the APP handle structure(`handle->runtime->tx_busy`) to check for the completion of data transmission.

DMA:

A DMA channel is configured to provide data to the **UART** transmit buffer. This removes the load off the CPU. This API will only configure and enable the DMA channel by specifying the data buffer and count of bytes to transmit. Rest is taken care without the CPU's intervention. User can configure a callback function in the APP UI. When the transmission is complete, the callback function will be executed. FIFO will not be used in DMA mode. Transmit buffer interrupt is configured for triggering the DMA channel. So each byte is transmitted in the background through the DMA channel. If the callback function is not configured, `handle->runtime->tx_busy` flag can be checked to verify if the transmission is complete. **Direct:**

Data will be transmitted using polling method. Status flags are used to check if data can be transmitted. **Note:** *In Direct mode, the API blocks the CPU until the count of bytes requested is transmitted. If this behaviour is not desired, use other APIs like **UART_TransmitWord**, **UART_GetProtocolStatus** etc.*

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Pre-condition:
//Transmit mode should be configured as "Direct".
//Description:
//Transmits the string "Infineon".
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t Send_Data[] = "Infineon";
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
```

```

//Transmit the string.
UART_Transmit(&UART_0, Send_Data, sizeof(Send_Data)-1);
while(1)
{
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}

```

Definition at line **170** of file **UART.c**.

References **UART::config**, **UART_CONFIG::transmit_mode**, **UART_STATUS_MODE_MISMATCH**, **UART_TRANSFER_MODE_DIRECT**, **UART_TRANSFER_MODE_DMA**, and **UART_TRANSFER_MODE_INTERRUPT**.

```

__STATIC_INLINE void UART_TransmitWord ( const UART_t *const
                                         uint8_t
                                         )

```

Transmits a word of data.

Parameters

handle **UART** APP handle pointer of type **UART_t**
data Data to be transmitted.

Returns

None

Description:

Transmits a byte of data through the **UART** channel. If Tx FIFO is configured, the data is placed in the IN[0] register of the USIC channel used. If Tx FIFO is not configured, API waits for the TBUF to be free and then places the data in the TBUF register. User can poll for receive event or configure interrupt by connecting an external INTERRUPT APP. This API can be used inside the ISR to read the received data.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation
(includes SFR declaration)
//Precondition:
//Configure transmit mode and receive mode as "Direct"
//Description:
//Transmits the string "Infinon"
int main(void)
{
    UART_STATUS_t init_status;
    uint8_t Send_Data[] = "Infineon";
    uint8_t Rec_Data[10];
    uint8_t index = 0;
    init_status = (UART_STATUS_t)UART_Init(&UART_0);
    if(init_status == UART_STATUS_SUCCESS)
    {
        while(index < sizeof(Send_Data))
        {
            UART_TransmitWord(&UART_0,Send_Data[index]);
            index++;
            //Wait for transmit buffer interrupt to fill it again with remaining data
            while(((UART_GetTXFIFOStatus(&UART_0) &
            XMC_USIC_CH_TXFIFO_EVENT_STANDARD) == 0);
            UART_ClearTXFIFOStatus(&UART_0,
            XMC_USIC_CH_TXFIFO_EVENT_STANDARD);
        }
    }
}
```

```
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **1305** of file **UART.h**.

References **UART::channel**.
