

Learnings on device drivers: -

Question) file:///C:/Users/sidra/Hello_world/mtb_shared/mtb-hal-cat1/release-v2.6.1/docs/html/group__group__hal__i2c.html

here i2c driver is used in ModusToolbox in Windows OS can I call it as windows device driver

Answer: - In this context, the I2C driver is part of the Hardware Abstraction Layer (HAL) provided by ModusToolbox for use with specific microcontrollers. It abstracts the hardware-specific details of the I2C peripheral, allowing developers to write portable code for embedded systems.

Windows device driver, on the other hand, is a software component that allows the Windows operating system to communicate with hardware devices. These drivers are written for the Windows OS and are used to manage hardware peripherals connected to a PC, such as printers, graphics cards, and USB devices.

Therefore, the I2C driver in ModusToolbox is an embedded system driver and not a Windows device driver.

The screenshot shows a presentation slide with a red background and a yellow square. The slide is titled "exercise:" and lists five tasks for creating an I2C driver for the STM32F407xx. The tasks are:

1. Create `stm32f407xx_i2c_driver.c` and `stm32f407xx_i2c_driver.h`
2. Add I2Cx related details to MCU specific header file
 - I. I2C peripheral register definition structure
 - II. I2Cx base address macros
 - III. I2Cx peripheral definition macros
 - IV. Macros to enable and disable I2Cx peripheral clock
 - V. Bit position definitions of I2C peripheral

On the right side of the slide, there is a sidebar with a checklist of tasks and sections:

- ☒ 182. I2C driver API requirements (7min)
- ☒ 183. I2C handle and configuration str (1min)
- ☒ 184. I2C user configurable macros (5min)
- ☒ 185. I2C API prototypes (3min)
- ☐ 186. Steps for I2C init implementation (4min)
- Section 54: I2C serial clock discuss** (0 / 2 | 13min)
- Section 55: I2C Driver API : I2C Init** (0 / 3 | 23min)
- Section 56: I2C Driver API : I2C Mas**

I am looking for above details in Infineon's PDL I2C driver :-

First 12 macros are for II. I2C Base address macros

Next 12 macros are for III. I2C Peripherals definition macros (need to verify)

```
1261 #define SDHC1_CORE ((SDHC_CORE_Type*) &SDHC1->CORE) /* 0x40471000 */
1262
1263 /*----- SCB -----*/
1264
1265
1266
1267 #define SCB0_BASE 0x40600000UL
1268 #define SCB1_BASE 0x40610000UL
1269 #define SCB2_BASE 0x40620000UL
1270 #define SCB3_BASE 0x40630000UL
1271 #define SCB4_BASE 0x40640000UL
1272 #define SCB5_BASE 0x40650000UL
1273 #define SCB6_BASE 0x40660000UL
1274 #define SCB7_BASE 0x40670000UL
1275 #define SCB8_BASE 0x40680000UL
1276 #define SCB9_BASE 0x40690000UL
1277 #define SCB10_BASE 0x406A0000UL
1278 #define SCB11_BASE 0x406B0000UL
1279 #define SCB12_BASE 0x406C0000UL
1280 #define SCB0 ((CySCB_Type*) SCB0_BASE) /* 0x40600000 */
1281 #define SCB1 ((CySCB_Type*) SCB1_BASE) /* 0x40610000 */
1282 #define SCB2 ((CySCB_Type*) SCB2_BASE) /* 0x40620000 */
1283 #define SCB3 ((CySCB_Type*) SCB3_BASE) /* 0x40630000 */
1284 #define SCB4 ((CySCB_Type*) SCB4_BASE) /* 0x40640000 */
1285 #define SCB5 ((CySCB_Type*) SCB5_BASE) /* 0x40650000 */
1286 #define SCB6 ((CySCB_Type*) SCB6_BASE) /* 0x40660000 */
1287 #define SCB7 ((CySCB_Type*) SCB7_BASE) /* 0x40670000 */
1288 #define SCB8 ((CySCB_Type*) SCB8_BASE) /* 0x40680000 */
1289 #define SCB9 ((CySCB_Type*) SCB9_BASE) /* 0x40690000 */
1290 #define SCB10 ((CySCB_Type*) SCB10_BASE) /* 0x406A0000 */
1291 #define SCB11 ((CySCB_Type*) SCB11_BASE) /* 0x406B0000 */
1292 #define SCB12 ((CySCB_Type*) SCB12_BASE) /* 0x406C0000 */
1293
1294
```

I found only below for scb clock macros

```
834 #define PERI_MS_PPU_FX_GPIO_TEST ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[200]) /* 0x40013A00 */
835 #define PERI_MS_PPU_FX_SMARTIO_PRT8_PRT ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[201]) /* 0x40013A40 */
836 #define PERI_MS_PPU_FX_SMARTIO_PRT9_PRT ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[203]) /* 0x40013AC0 */
837 #define PERI_MS_PPU_FX_LPCOMP ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[204]) /* 0x40013B00 */
838 #define PERI_MS_PPU_FX_CSD0 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[205]) /* 0x40013B40 */
839 #define PERI_MS_PPU_FX_TCPWM0 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[206]) /* 0x40013B80 */
840 #define PERI_MS_PPU_FX_TCPWM1 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[207]) /* 0x40013BC0 */
841 #define PERI_MS_PPU_FX_LCD0 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[208]) /* 0x40013C00 */
842 #define PERI_MS_PPU_FX_USBF0 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[209]) /* 0x40013C40 */
843 #define PERI_MS_PPU_FX_SMIF0 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[210]) /* 0x40013C80 */
844 #define PERI_MS_PPU_FX_SDHC0 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[211]) /* 0x40013CC0 */
845 #define PERI_MS_PPU_FX_SDHC1 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[212]) /* 0x40013D00 */
846 #define PERI_MS_PPU_FX_SCB0 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[213]) /* 0x40013D40 */
847 #define PERI_MS_PPU_FX_SCB1 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[214]) /* 0x40013D80 */
848 #define PERI_MS_PPU_FX_SCB2 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[215]) /* 0x40013DC0 */
849 #define PERI_MS_PPU_FX_SCB3 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[216]) /* 0x40013E00 */
850 #define PERI_MS_PPU_FX_SCB4 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[217]) /* 0x40013E40 */
851 #define PERI_MS_PPU_FX_SCB5 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[218]) /* 0x40013E80 */
852 #define PERI_MS_PPU_FX_SCB6 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[219]) /* 0x40013EC0 */
853 #define PERI_MS_PPU_FX_SCB7 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[220]) /* 0x40013F00 */
854 #define PERI_MS_PPU_FX_SCB8 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[221]) /* 0x40013F40 */
855 #define PERI_MS_PPU_FX_SCB9 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[222]) /* 0x40013F80 */
856 #define PERI_MS_PPU_FX_SCB10 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[223]) /* 0x40013FC0 */
857 #define PERI_MS_PPU_FX_SCB11 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[224]) /* 0x40014000 */
858 #define PERI_MS_PPU_FX_SCB12 ((PERI_MS_PPU_FX_Type*) &PERI_MS->PPU_FX[225]) /* 0x40014040 */
```

I needed port13 base address which I found in

```
main.c
#define HSI0M_PRT10 ((HSI0M_PRT_Type*) &HSI0M->PRT[10]) /* 0x40320000 */
#define HSI0M_PRT11 ((HSI0M_PRT_Type*) &HSI0M->PRT[11]) /* 0x40320040 */
#define HSI0M_PRT12 ((HSI0M_PRT_Type*) &HSI0M->PRT[12]) /* 0x40320080 */
#define HSI0M_PRT13 ((HSI0M_PRT_Type*) &HSI0M->PRT[13]) /* 0x403200C0 */
#define HSI0M_PRT14 ((HSI0M_PRT_Type*) &HSI0M->PRT[14]) /* 0x40320100 */

GPIO
#define GPIO_BASE 0x40310000UL ((GPIO_Type*) GPIO_BASE) /* 0x40310000 */
#define GPIO_PRT0 ((GPIO_PRT_Type*) &GPIO->PRT[0]) /* 0x40310000 */
#define GPIO_PRT1 ((GPIO_PRT_Type*) &GPIO->PRT[1]) /* 0x40310040 */
#define GPIO_PRT2 ((GPIO_PRT_Type*) &GPIO->PRT[2]) /* 0x40310080 */
#define GPIO_PRT3 ((GPIO_PRT_Type*) &GPIO->PRT[3]) /* 0x403100C0 */
#define GPIO_PRT4 ((GPIO_PRT_Type*) &GPIO->PRT[4]) /* 0x40310100 */
#define GPIO_PRT5 ((GPIO_PRT_Type*) &GPIO->PRT[5]) /* 0x40310140 */
#define GPIO_PRT6 ((GPIO_PRT_Type*) &GPIO->PRT[6]) /* 0x40310180 */
#define GPIO_PRT7 ((GPIO_PRT_Type*) &GPIO->PRT[7]) /* 0x403101C0 */
#define GPIO_PRT8 ((GPIO_PRT_Type*) &GPIO->PRT[8]) /* 0x40310200 */
#define GPIO_PRT9 ((GPIO_PRT_Type*) &GPIO->PRT[9]) /* 0x40310240 */
#define GPIO_PRT10 ((GPIO_PRT_Type*) &GPIO->PRT[10]) /* 0x40310280 */
#define GPIO_PRT11 ((GPIO_PRT_Type*) &GPIO->PRT[11]) /* 0x403102C0 */
#define GPIO_PRT12 ((GPIO_PRT_Type*) &GPIO->PRT[12]) /* 0x40310300 */
#define GPIO_PRT13 ((GPIO_PRT_Type*) &GPIO->PRT[13]) /* 0x40310340 */
#define GPIO_PRT14 ((GPIO_PRT_Type*) &GPIO->PRT[14]) /* 0x40310380 */

SMARTIO
#define SMARTIO_BASE 0x40320000UL ((SMARTIO_Type*) SMARTIO_BASE) /* 0x40320000 */
#define SMARTIO_PRT8 ((SMARTIO_PRT_Type*) &SMARTIO->PRT[8]) /* 0x40320000 */
#define SMARTIO_PRT9 ((SMARTIO_PRT_Type*) &SMARTIO->PRT[9]) /* 0x40320040 */

LPCOMP
```

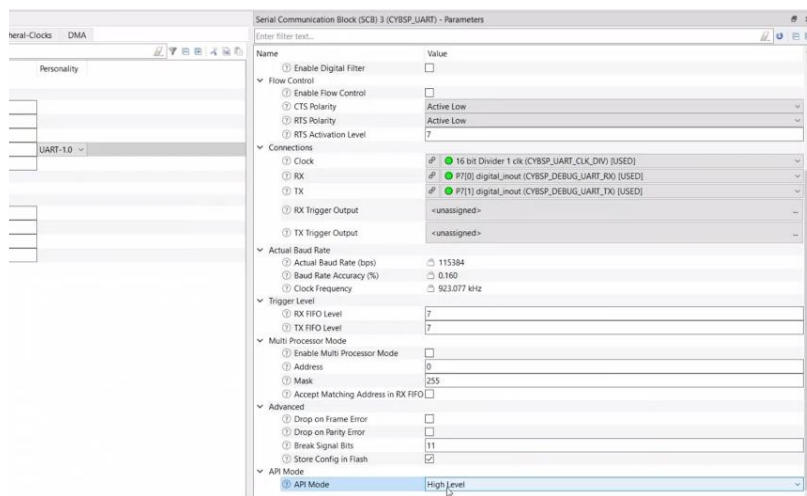
High level and low level APIs:-

Interrupts are configured automatically by the component for us to do the data transfer where in low level where I will be writing from tx and reading from rx fifo and any kind of data management done by the firmware.

let say I want to configure an interrupt on every byte receive so I will have to put it to **low level** and **configure interrupt accordingly and do the data transfer.**

where as in **high level** say I want transmit 100 Bytes of data I simply call high level api that transfers 100 bytes of data and internally component take care of the interrupt to push this data out.

Example:- as he already told us tx fifo has only 16 elements and I want to transfer 100 elements I don't want to do much of firmware tasks I will simply go for high level and say transfer 100 bytes in the component it will automatically transfer 100 bytes for me



From Datasheet of PSoC6: -

Serial Communication Blocks (SCB)

This product line has 13 SCBs:

- Eight can implement either I2C, UART, or SPI.
- Four can implement either I2C or UART.
- One SCB (SCB #8) can operate in system Deep Sleep mode with an external clock; this SCB can be either SPI slave or I2C slave.

UART Mode: This is a full-feature UART operating at up to 8 Mbps. It supports automotive single-wire interface (LIN), infrared interface (IrDA), and SmartCard (ISO7816) protocols, all of which are minor

variants of the basic UART protocol. In addition, it supports the 9-bit multiprocessor mode that allows the addressing of peripherals connected over common Rx and Tx lines. Common UART functions such as **parity error, break detect, and frame error are supported**. A **256-byte FIFO** allows much greater CPU service latencies to be tolerated.

I 2C Mode: The SCB can implement a full multi-master and slave interface (it is capable of multimaster arbitration). This block can operate at speeds of up to 1 Mbps (Fast Mode Plus). It also supports EZI2C, which creates a mailbox address range and effectively reduces I2C communication to reading from and writing to an array in memory. The SCB supports a 256-byte FIFO for receive and transmit.

The I2C peripheral is compatible with I2C standard-mode, Fast Mode, and Fast Mode Plus devices as defined in the NXP I 2C-bus specification and user manual (UM10204). The I2C bus I/O is implemented with GPIO in open-drain modes.

SPI Mode: The SPI mode supports full Motorola SPI, TI Secure Simple Pairing (SSP) (essentially adds a start pulse that is used to synchronize SPI Codecs), and National Microwire (half-duplex form of SPI). The SPI block supports an EZSPI mode in which the data interchange is reduced to reading and writing an array in memory. The SPI interface operates with a 25-MHz clock.

From TRM: -

Serial Communications Block (SCB)

The Serial Communications Block (SCB) supports three serial communication protocols: Serial Peripheral Interface (SPI), Universal Asynchronous Receiver Transmitter (UART), and Inter Integrated Circuit (I2C or IIC). Only one of the protocols is supported by an SCB at any given time.

Features: -

The SCB supports the following features:

- Standard SPI master and slave functionality with Motorola, Texas Instruments, and National Semiconductor protocols
- Standard UART functionality with SmartCard reader, Local Interconnect Network (LIN), and IrDA protocols
- ☐ Standard LIN slave functionality with LIN v1.3 and LIN v2.1/2.2 specification compliance
- Standard I2C master and slave functionality
- Trigger outputs for connection to DMA
- Multiple interrupt sources to indicate status of FIFOs and transfers
- Features available only on Deep Sleep-capable SCB:
 - ☐ EZ mode for SPI and I2C slaves; allows for operation without CPU intervention
 - ☐ CMD_RESP mode for SPI and I2C slaves; allows for operation without CPU intervention
 - ☐ Low-power (Deep Sleep) mode of operation for SPI and I2C slaves (using external clocking)
 - ☐ Deep Sleep wakeup on I2C slave address match or SPI slave selection

Architecture

The operation modes supported by SCB are described in the following sections.

Buffer Modes

Each SCB has 256 bytes of dedicated RAM for transmit and receive operation. This RAM can be configured in three different modes (FIFO, EZ, or CMD_RESP). The following sections give a high-level overview of each mode. The sections on each protocol will provide more details.

- Masters can only use FIFO mode
- I2C and SPI slaves can use all three modes. Note: EZ Mode and CMD Response Mode are available only on the Deep Sleep-capable SCB

- **UART only uses FIFO mode**

Note: This document discusses hardware implementation of the EZ mode; for the firmware implementation, see the PDL.

FIFO Mode

In this mode the RAM is split into two 128-byte FIFOs, **one for transmit (TX) and one for receive (RX)**. The FIFOs can be configured to be 8 bits x 128 elements or 16 bits x 64 elements; this is done by setting the **BYTE_MODE bit in the SCB control register**.

FIFO mode of operation is available only in Active and Sleep power modes. However, the I2C address or SPI slave select can be used to wake the device from Deep Sleep on the Deep Sleep-capable SCB.

Statuses are provided for both the RX and TX FIFOs. There are multiple interrupt sources available, which indicate the status of the FIFOs, such as full or empty; see “SCB Interrupts” on page 368.

Explanation for **BYTE_MODE bit in the SCB control register**: -

24.1.1 SCB0_CTRL

Generic control

Address: 0x40600000

Retention: Retained

| Bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------|------------|---|---|---|-----------|---|---|---|
| SW Access | None | | | | RW | | | |
| HW Access | None | | | | R | | | |
| Name | None [7:4] | | | | OVS [3:0] | | | |

| Bits | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
|-----------|--------------|----|----|----|----------------|-------------|---|---|
| SW Access | None | | | | RW | None | | |
| HW Access | None | | | | R | None | | |
| Name | None [15:12] | | | | BYTE_- MODE | None [10:8] | | |

| Bits | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|-----------|--------------|----|----|----|----|----|----|------------------|
| SW Access | None | | | | | | | RW |
| HW Access | None | | | | | | | R |
| Name | None [23:17] | | | | | | | ADDR_AC- CEPT |

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
|-----------|---------|--------------|----|----|----|----|--------------|----|
| SW Access | RW | None | | | | | RW | |
| HW Access | R | None | | | | | R | |
| Name | ENABLED | None [30:26] | | | | | MODE [25:24] | |

Bits Name Description

- 31 **ENABLED** IP enabled ('1') or not ('0'). The proper order in which to initialize the IP is as follows:
- Program protocol specific information using SPI_CTRL, UART_CTRL (and UART_TX_CTRL and UART_RX_CTRL) or I2C_CTRL. This includes selection of a submode, master/slave functionality and transmitter/receiver functionality when applicable.
 - Program generic transmitter (TX_CTRL) and receiver (RX_CTRL) information. This includes enabling of the transmitter and receiver functionality.
 - Program transmitter FIFO (TX_FIFO_CTRL) and receiver FIFO (RX_FIFO_CTRL) information.
 - Program CTRL to enable IP, select the specific operation mode and oversampling factor.
- When the IP is enabled, no control information should be changed. Changes should be made AFTER disabling the IP, e.g. to modify the operation mode (from I2C to SPI) or to go from externally to internally clocked. The change takes effect after the IP is re-enabled. Note that disabling the IP will cause re-initialization of the design and associated state is lost (e.g. FIFO content).
- Default Value: 0
- 25 : 24 **MODE** Mode of operation (3: Reserved)
- Default Value: 3
- 0x0: I2C :
Inter-Integrated Circuits (I2C) mode.
- 0x1: SPI :
Serial Peripheral Interface (SPI) mode.
- 0x2: UART :
Universal Asynchronous Receiver/Transmitter (UART) mode.
- 16 **ADDR_ACCEPT** Determines whether a received matching address is accepted in the RX FIFO ('1') or not ('0').

In I2C mode, this field is used to allow the slave to put the received slave address or general call address in the RX FIFO. Note that a received matching address is put in the RX FIFO when ADDR_ACCEPT is '1' for both I2C read and write transfers.

In multi-processor UART receiver mode, this field is used to allow the receiver to put the received address in the RX FIFO. Note: non-matching addresses are never put in the RX FIFO. Default Value: 0

11 BYTE_MODE

Determines the number of bits per FIFO data element:

'0': 16-bit FIFO data elements.

'1': 8-bit FIFO data elements. This mode doubles the amount of FIFO entries, but TX_CTRL.DATA_WIDTH and RX_CTRL.DATA_WIDTH are restricted to [0, 7].

Default Value: 0

UART

The Universal Asynchronous Receiver/Transmitter (UART) protocol is an asynchronous serial interface protocol. UART communication is typically point-to-point. The UART interface consists of two signals:

- TX: Transmitter output
- RX: Receiver input

Additionally, two side-band signals are used to implement flow control in UART. Note that the flow control applies only to TX functionality.

- Clear to Send (CTS): This is an input signal to the transmitter. When active, the receiver signals to the transmitter that it is ready to receive.
- Ready to Send (RTS): This is an output signal from the receiver. When active, it indicates that the receiver is ready to receive data.

Not all SCBs support UART mode; refer to the PSoC 61datasheet/PSoC 62 datasheet for details.

Features

- Supports UART protocol
- ☐ Standard UART
- ☐ Multi-processor mode
- SmartCard (ISO7816) reader
- IrDA
- Supports Local Interconnect Network (LIN)
- ☐ Break detection
- ☐ Baud rate detection
- ☐ Collision detection (ability to detect that a driven bit value is not reflected on the bus, indicating that another component is driving the same bus)
- Data frame size programmable from 4 to 16 bits
- Programmable number of STOP bits, which can be set in terms of half bit periods between 1 and 4
- Parity support (odd and even parity)
- Median filter on RX input
- Programmable oversampling

- Start skipping
- Hardware flow control

General Description

illustrates a standard UART TX and RX.

UART Example



A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit and finally completed by one or more stop bits. The start and stop bits indicate the start and end of data transmission. The parity bit is sent by the transmitter and is used by the receiver to detect single bit errors. Because the interface does not have a clock (asynchronous), the transmitter and receiver use their own clocks; thus, the transmitter and receiver need to agree on the baud rate.

By default, UART supports a data frame width of eight bits. However, this can be configured to any value in the range of 4 to 9. This does not include start, stop, and parity bits. The number of stop bits can be in the range of 1 to 4. The parity bit can be either enabled or disabled. If enabled, the type of parity can be set to either even parity or odd parity. The option of using the parity bit is available only in the Standard UART and SmartCard UART modes. For IrDA UART mode, the parity bit is automatically disabled.

Note: UART interface does not support external clocking operation. Hence, UART operates only in the Active and Sleep system power modes. UART also supports only the FIFO buffer mode.

UART Modes of Operation

Standard Protocol

A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit and finally completed by one or more stop bits. The start bit value is always '0', the data bits values are dependent on the data transferred, the parity bit value is set to a value guaranteeing an even or odd parity over the data bits, and the stop bit value is '1'. The parity bit is generated by the transmitter and can be used by the receiver to detect single bit transmission errors. When not transmitting data, the TX line is '1' – the same value as the stop bits.

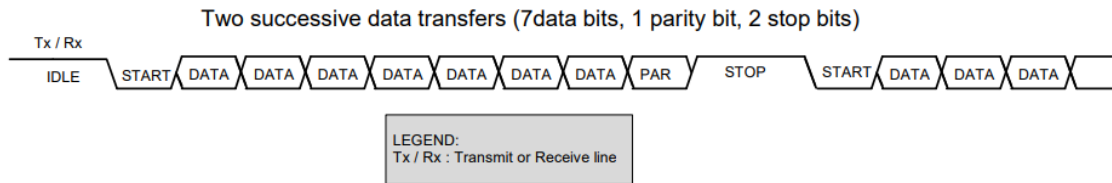
Because the interface does not have a clock, the transmitter and receiver must agree upon the baud rate. The transmitter and receiver have their own internal clocks. The receiver clock runs at a higher frequency than the bit transfer frequency, such that the receiver may oversample the incoming signal.

The transition of a stop bit to a start bit is represented by a change from '1' to '0' on the TX line. This transition can be used by the receiver to synchronize with the transmitter clock. Synchronization at the start of each data transfer allows error-free transmission even in the presence of frequency drift between transmitter and receiver clocks. The required clock accuracy is dependent on the data transfer size.

The stop period or the amount of stop bits between successive data transfers is typically agreed upon between transmitter and receiver, and is typically in the range of 1 to 3-bit transfer periods.

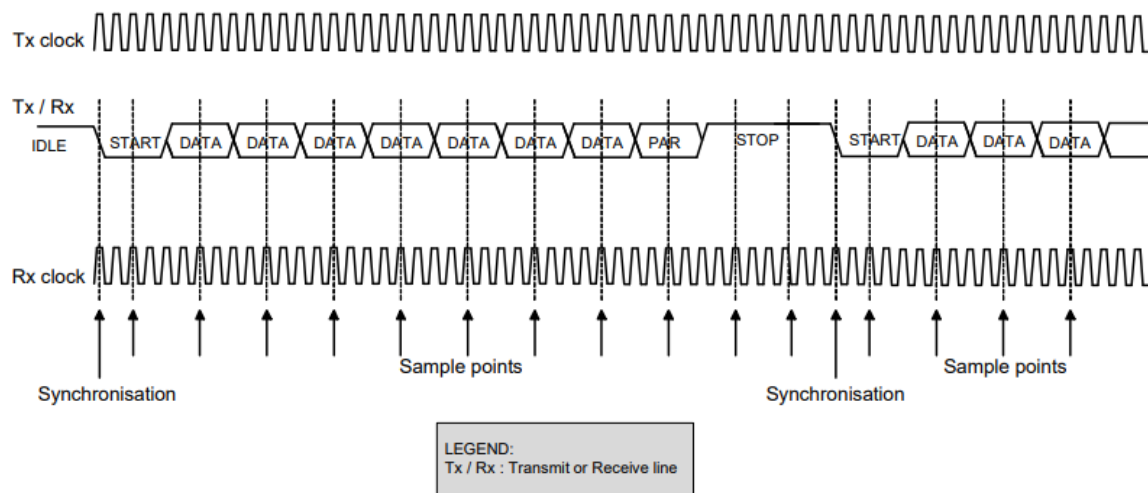
illustrates the UART protocol

Figure 28-15. UART, Standard Protocol Example



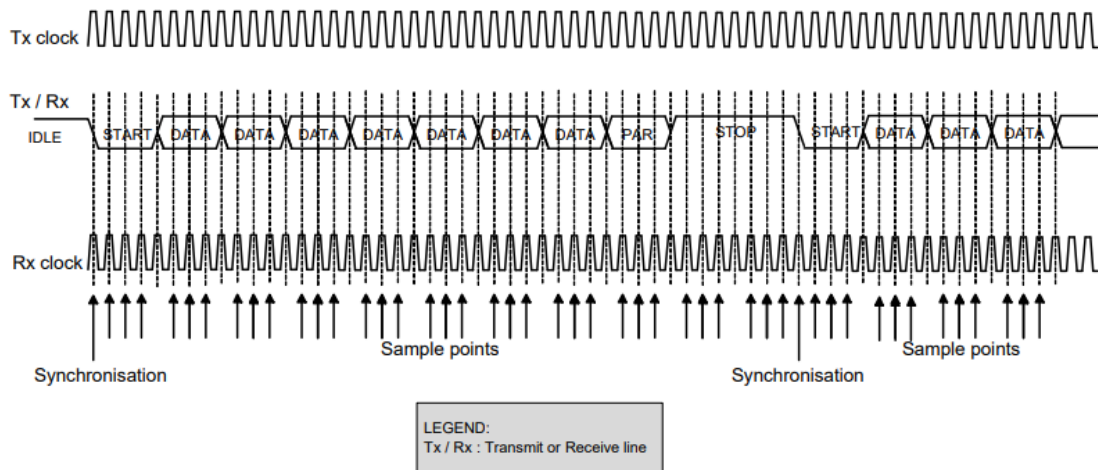
The receiver oversamples the incoming signal; the value of the sample point in the middle of the bit transfer period (on the receiver's clock) is used. Figure 28-16 illustrates this.

Figure 28-16. UART, Standard Protocol Example (Single Sample)



Alternatively, three samples around the middle of the bit transfer period (on the receiver's clock) are used for a majority vote to increase accuracy; this is enabled by enabling the MEDIAN filter in the SCB_RX_CTRL register. Figure 28-17 illustrates this.

Figure 28-17. UART, Standard Protocol (Multiple Samples)



Parity

This functionality adds a parity bit to the data frame and is used to identify single-bit data frame errors. The parity bit is always directly after the data frame bits.

The transmitter calculates the parity bit (when `UART_TX_CTRL.PARITY_ENABLED` is 1) from the data frame bits, such that data frame bits and parity bit have an even (`UART_TX_CTRL.PARITY` is 0) or odd (`UART_TX_CTRL.PARITY` is 1) parity. The receiver checks the parity bit (when `UART_RX_CTRL.PARITY_ENABLED` is 1) from the received data frame bits, such that data frame bits and parity bit have an even (`UART_RX_CTRL.PARITY` is 0) or odd (`UART_RX_CTRL.PARITY` is 1) parity.

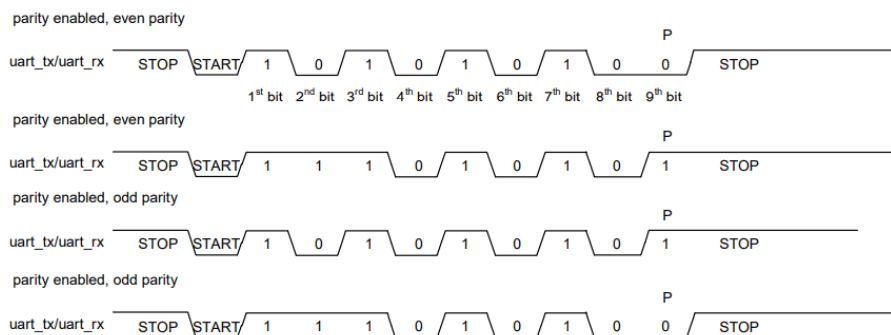
Parity applies to both TX and RX functionality and dedicated control fields are available.

- Transmit functionality: `UART_TX_CTRL.PARITY` and `UART_TX_CTRL.PARITY_ENABLED`.
- Receive functionality: `UART_RX_CTRL.PARITY` and `UART_RX_CTRL.PARITY_ENABLED`.

When a receiver detects a parity error, the data frame is either put in RX FIFO (`UART_RX_CTRL.DROP_ON_PARITY_ERROR` is 0) or dropped (`UART_RX_CTRL.DROP_ON_PARITY_ERROR` is 1).

The following figures illustrate the parity functionality (8-bit data frame).

Figure 28-18. UART Parity Examples



Configuring the SCB as Standard UART Interface

To configure the SCB as a standard UART interface, set various register bits in the following order:

1. Configure the SCB as UART interface by writing '10b' to the MODE field (bits [25:24]) of the SCB_CTRL register.
2. Configure the UART interface to operate as a Standard protocol by writing '00' to the MODE field (bits [25:24]) of the SCB_UART_CTRL register.
3. To enable the UART MP Mode or UART LIN Mode, write '1' to the MP_MODE (bit 10) or LIN_MODE (bit 12) respectively of the SCB_UART_RX_CTRL register.
4. Follow steps 2 to 4 described in "Enabling and Initializing the UART" on page 350.

For more information on these registers, see the registers TRM.

Clocking and Oversampling

The UART protocol is implemented using `clk_scb` as an oversampled multiple of the baud rate. For example, to implement a 100-kHz UART, `clk_scb` could be set to 1 MHz and the oversample factor set to '10'. The oversampling is set using the SCB_CTRL.OVS register field. The oversampling value is `SCB_CTRL.OVS + 1`. In the UART standard sub-mode (including LIN) and the SmartCard submode, the valid range for the OVS field is [7, 15].

In UART transmit IrDA sub-mode, this field indirectly specifies the oversampling. Oversampling determines the interface clock per bit cycle and the width of the pulse. This sub-mode has only one valid OVS value—16 (which is a value of 0 in the OVS field of the SCB_CTRL register); the pulse width is roughly 3/16 of the bit period (for all bit rates).

In UART receive IrDA sub-mode (1.2, 2.4, 9.6, 19.2, 38.4, 57.6, and 115.2 kbps), this field indirectly specifies the oversampling. In normal transmission mode, this pulse is approximately 3/16 of the bit period (for all bit rates). In lowpower transmission mode, this pulse is potentially smaller (down to 1.62 μ s typical and 1.41 μ s minimal) than 3/16 of the bit period (for less than 115.2 kbps bit rates).

Pulse widths greater than or equal to two SCB input clock cycles are guaranteed to be detected by the receiver. Pulse widths less than two clock cycles and greater than or equal to one SCB input clock cycle may be detected by the receiver. Pulse widths less than one SCB input clock cycle will not be detected by the receiver. Note that the SCB_RX_CTRL.MEDIAN should be set to '1' for IrDA receiver functionality.

The SCB input clock and the oversampling together determine the IrDA bit rate. Refer to the registers TRM for more details on the OVS values for different baud rates.

Enabling and Initializing the UART

The UART must be programmed in the following order:

1. Program protocol specific information using the UART_TX_CTRL, UART_RX_CTRL, and UART_FLOW_CTRL registers. This includes selecting the submodes of the protocol, transmitter-receiver functionality, and so on.

2. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers. a. Specify the data frame width. b. Specify whether MSb or LSb is the first bit to be transmitted or received.
3. Program the transmitter and receiver FIFOs using the SCB_TX_FIFO_CTRL and SCB_RX_FIFO_CTRL registers, respectively. a. Set the trigger level. b. Clear the transmitter and receiver FIFO and Shift registers.
4. Enable the block (write a '1' to the ENABLE bit of the SCB_CTRL register). After the block is enabled, control bits should not be changed. Changes should be made after disabling the block; for example, to modify the operation mode (from SmartCard to IrDA). The change takes effect only after the block is re-enabled. Note that **re-enabling the block causes re-initialization and the associated state is lost** (for example FIFO content).

I/O Pad Connection

Standard UART Mode

list the use of the I/O pads for the Standard UART mode.

Figure 28-32. Standard UART Mode I/O Pad Connections

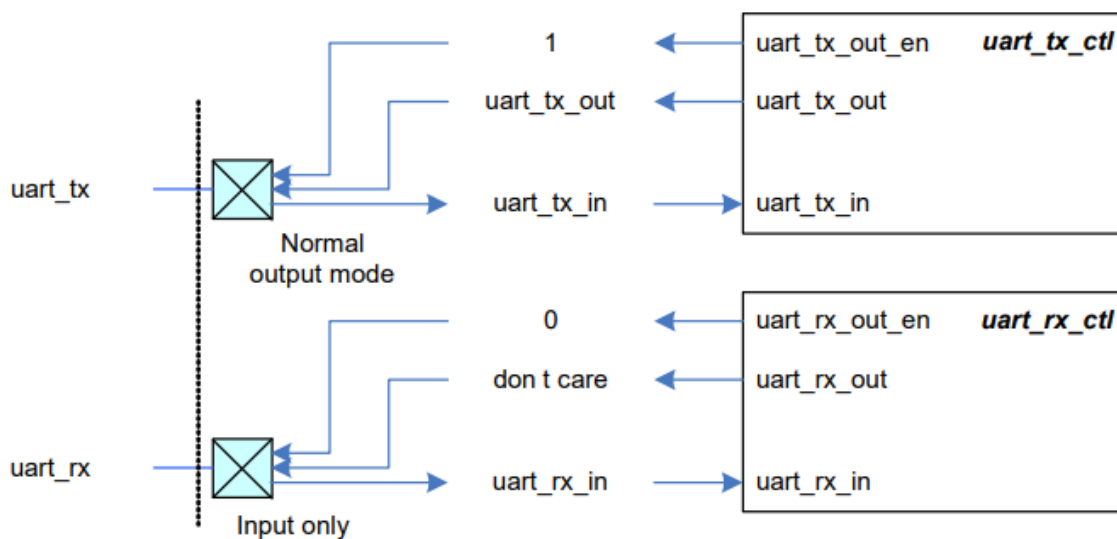


Table 28-7. UART I/O Pad Connection Usage

| I/O Pads | Drive Mode | On-chip I/O Signals | Usage |
|----------|--------------------|-------------------------------|-------------------------|
| uart_tx | Normal output mode | uart_tx_out_en uart_tx_out | Transmit a data element |
| uart_rx | Input only | uart_rx_in | Receive a data element |

UART Registers

The UART interface is controlled using a set of 32-bit registers listed in Table 28-11. For more information on these registers, see the registers TRM.

Table 28-11. UART Registers

| Register Name | Operation |
|-----------------------|---|
| SCB_CTRL | Enables the SCB; selects the type of serial interface (SPI, UART, I ² C) |
| SCB_UART_CTRL | Used to select the sub-modes of UART (standard UART, SmartCard, IrDA), also used for local loop back control. |
| SCB_UART_RX_STATUS | Used to specify the BR_COUNTER value that determines the bit period. This is used to set the accuracy of the SCB clock. This value provides more granularity than the OVS bit in SCB_CTRL register. |
| SCB_UART_TX_CTRL | Used to specify the number of stop bits, enable parity, select the type of parity, and enable retransmission on NACK. |
| SCB_UART_RX_CTRL | Performs same function as SCB_UART_TX_CTRL but is also used for enabling multi processor mode, LIN mode drop on parity error, and drop on frame error. |
| SCB_TX_CTRL | Used to specify the data frame width and to specify whether MSb or LSb is the first bit in transmission. |
| SCB_RX_CTRL | Performs the same function as that of the SCB_TX_CTRL register, but for the receiver. Also decides whether a median filter is to be used on the input interface lines. |
| SCB_UART_FLOW_CONTROL | Configures flow control for UART transmitter. |

SCB Interrupts

SCB supports interrupt generation on various events. The interrupts generated by the SCB block vary depending on the mode of operation.

Table 28-21. SCB Interrupts

| Interrupt | Functionality | Active/Deep Sleep | Registers |
|------------------|---|-------------------|---|
| interrupt_master | I ² C master and SPI master functionality | Active | INTR_M, INTR_M_SET, INTR_M_MASK, INTR_M_MASKED |
| interrupt_slave | I ² C slave and SPI slave functionality | Active | INTR_S, INTR_S_SET, INTR_S_MASK, INTR_S_MASKED |
| interrupt_tx | UART transmitter and TX FIFO functionality | Active | INTR_TX, INTR_TX_SET, INTR_TX_MASK, INTR_TX_MASKED |
| interrupt_rx | UART receiver and RX FIFO functionality | Active | INTR_RX, INTR_RX_SET, INTR_RX_MASK, INTR_RX_MASKED |
| interrupt_i2c_ec | Externally clocked I ² C slave functionality | Deep Sleep | INTR_I2C_EC, INTR_I2C_EC_MASK, INTR_I2C_EC_MASKED |
| interrupt_spi_ec | Externally clocked SPI slave functionality | Deep Sleep | INTR_ISPI_EC, INTR_SPI_EC_MASK, INTR_SPI_EC_MASKED |

Note: To avoid being triggered by events from previous transactions, whenever the firmware enables an interrupt mask register bit, it should clear the interrupt request register in advance.

Note: If the DMA is used to read data out of RX FIFO, the NOT_EMPTY interrupt may never trigger. This can occur when clk_peri (clocking DMA) is running much faster than the clock to the SCB. As a workaround to this issue, set the RX_FIFO_CTRL.TRIGGER_LEVEL to '1' (not 0); this will allow the interrupt to fire.

The following register definitions correspond to the SCB interrupts:

■ **INTR_M:** This register provides the instantaneous status of the interrupt sources. A write of '1' to a bit will clear the interrupt.

■ **INTR_M_SET:** A write of '1' into this register will set the interrupt.

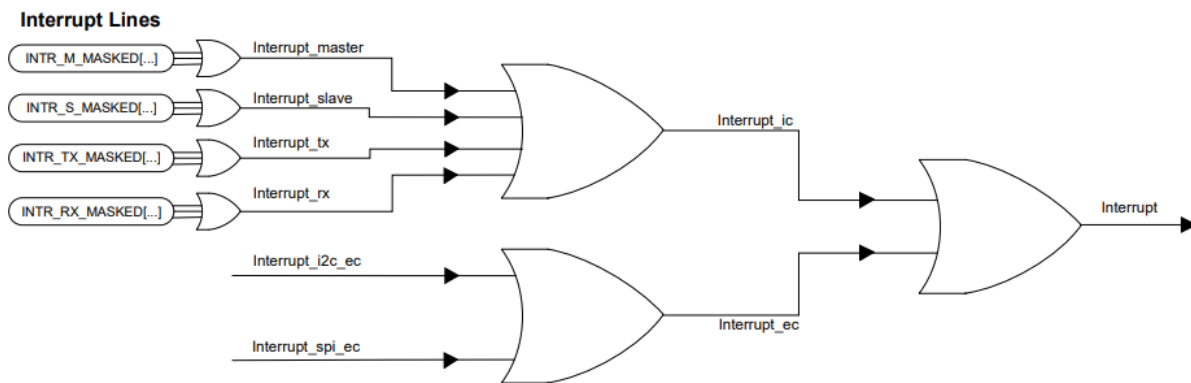
■ **INTR_M_MASK:** The bit in this register masks the interrupt sources. Only the interrupt sources with their masks enabled can trigger the interrupt.

■ **INTR_M_MASKED:** This register provides the instantaneous value of the interrupts after they are masked. It provides logical and corresponding request and mask bits. This is used to understand which interrupt triggered the event.

Note: While registers corresponding to INTR_M are used here, these definitions can be used for INTR_S, INTR_TX, INTR_RX, INTR_I2C_EC, and INTR_SPI_EC.

Figure 28-45 shows the physical interrupt lines. All the interrupts are OR'd together to make one interrupt source that is the OR of all six individual interrupts. All the externally-clocked interrupts make one interrupt line called interrupt_ec, which is the OR'd signal of interrupt_i2c_ec and interrupt_spi_ec. All the internally-clocked interrupts make one interrupt line called interrupt_ic, which is the OR'd signal of interrupt_master, interrupt_slave, interrupt_tx, and interrupt_rx. The Active functionality interrupts are generated synchronously to clk_peri while the Deep Sleep functionality interrupts are generated asynchronously to clk_peri.

Figure 28-45. Interrupt Lines



UART Interrupts

The UART interrupts can be classified as TX interrupts and RX interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources are true. The SCB also provides an interrupt cause register (SCB_INTR_CAUSE) that can be used to determine interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bitfield. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the registers TRM. The UART block generates interrupts on the following events:

■ UART TX

- ☐ TX FIFO has fewer entries than the value specified by TRIGGER_LEVEL in SCB_TX_FIFO_CTRL.
- ☐ TX FIFO not full – TX FIFO is not full. At least one data element can be written into the TX FIFO.
- ☐ TX FIFO empty – The TX FIFO is empty.
- ☐ TX FIFO overflow – Firmware attempts to write to a full TX FIFO.
- ☐ TX FIFO underflow – Hardware attempts to read from an empty TX FIFO. This happens when the SCB is ready to transfer data and EMPTY is '1'.
- ☐ TX NACK – UART transmitter receives a negative acknowledgment in SmartCard mode.
- ☐ TX done – This happens when the UART completes transferring all data in the TX FIFO and the last stop field is transmitted (both TX FIFO and transmit shifter register are empty).
- ☐ TX lost arbitration – The value driven on the TX line is not the same as the value observed on the RX line. This condition event is useful when transmitter and receiver share a TX/RX line. This is the case in LIN or SmartCard modes.

■ UART RX

- ☐ RX FIFO has more entries than the value specified by TRIGGER_LEVEL in SCB_RX_FIFO_CTRL.
- ☐ RX FIFO full – RX FIFO is full. Note that **received data frames are lost when the RX FIFO is full.**
- ☐ RX FIFO not empty – RX FIFO is not empty.
- ☐ RX FIFO overflow – Hardware attempts to write to a full RX FIFO.
- ☐ RX FIFO underflow – Firmware attempts to read from an empty RX FIFO.
- ☐ Frame error in received data frame – UART frame error in received data frame. This can be either a start of stop bit error:
 - Start bit error:** After the beginning of a start bit period is detected (RX line changes from 1 to 0), the middle of the start bit period is sampled erroneously (RX line is '1'). Note: A start bit error is detected before a data frame is received.
 - Stop bit error:** The RX line is sampled as '0', but a '1' was expected. A stop bit error may result in failure to receive successive data frames. Note: A stop bit error is detected after a data frame is received.
- ☐ Parity error in received data frame – If UART_RX_CTL.DROP_ON_PARITY_ERROR is '1', the received frame is dropped. If UART_RX_CTL.DROP_ON_PARITY_ERROR is '0', the received frame is sent to the RX FIFO. In SmartCard sub mode, negatively acknowledged data frames generate a parity error. Note that **firmware can only identify the erroneous data frame in the RX FIFO if it is fast enough to read the data frame before the hardware writes a next data frame into the RX FIFO.**
- ☐ LIN baud rate detection is completed – The receiver software uses the UART_RX_STATUS.BR_COUNTER value to set the clk_scb to guarantee successful receipt of the first LIN data frame (Protected Identifier Field) after the synchronization byte.
- ☐ LIN break detection is successful – The line is '0' for UART_RX_CTRL.BREAK_WIDTH + 1 bit period. Can occur at any time to address unanticipated break fields; that is, "break-in-data" is supported. This feature is supported for the UART standard and LIN submodes. For the UART standard submodes, ongoing

receipt of data frames is not affected; firmware is expected to take proper action. For the LIN submode, possible ongoing receipt of a data frame is stopped and the (partially) received data frame is dropped and baud rate detection is started. Set to '1', when event is detected. Write with '1' to clear bit.

Figure 28-48 and Figure 28-49 show how each of the interrupts are triggered. Figure 28-48 shows the TX buffer and the corresponding interrupts while Figure 28-49 shows all the corresponding interrupts for the RX buffer. The FIFO has 256 split into 128 bytes for TX and 128 bytes for RX instead of the 8 bytes shown in the figures. For more information on [how to implement and clear interrupts see the UART \(SCB_UART_PDL\) datasheet and the PDL.](#)

Figure 28-48. TX Interrupt Source Operation

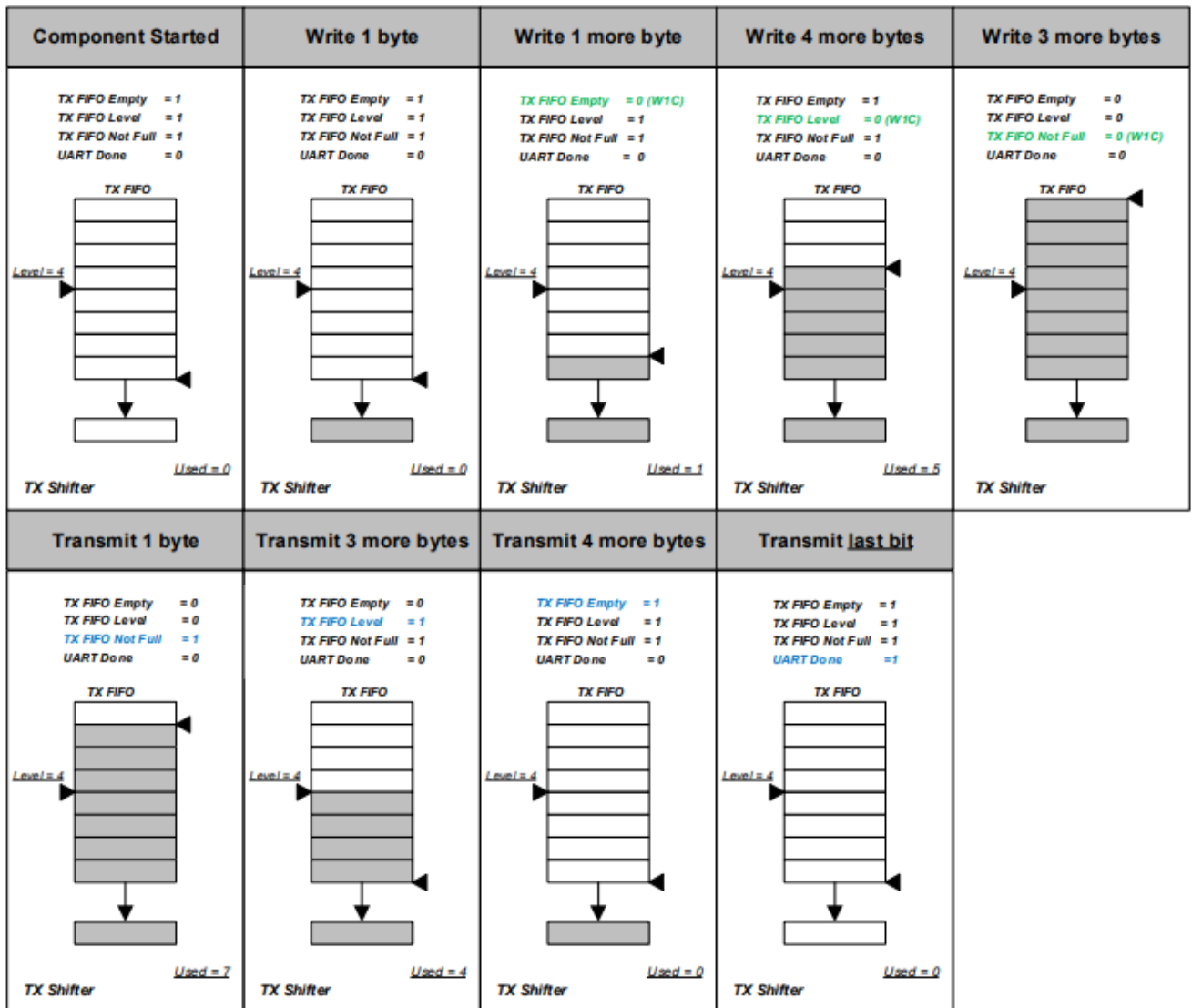
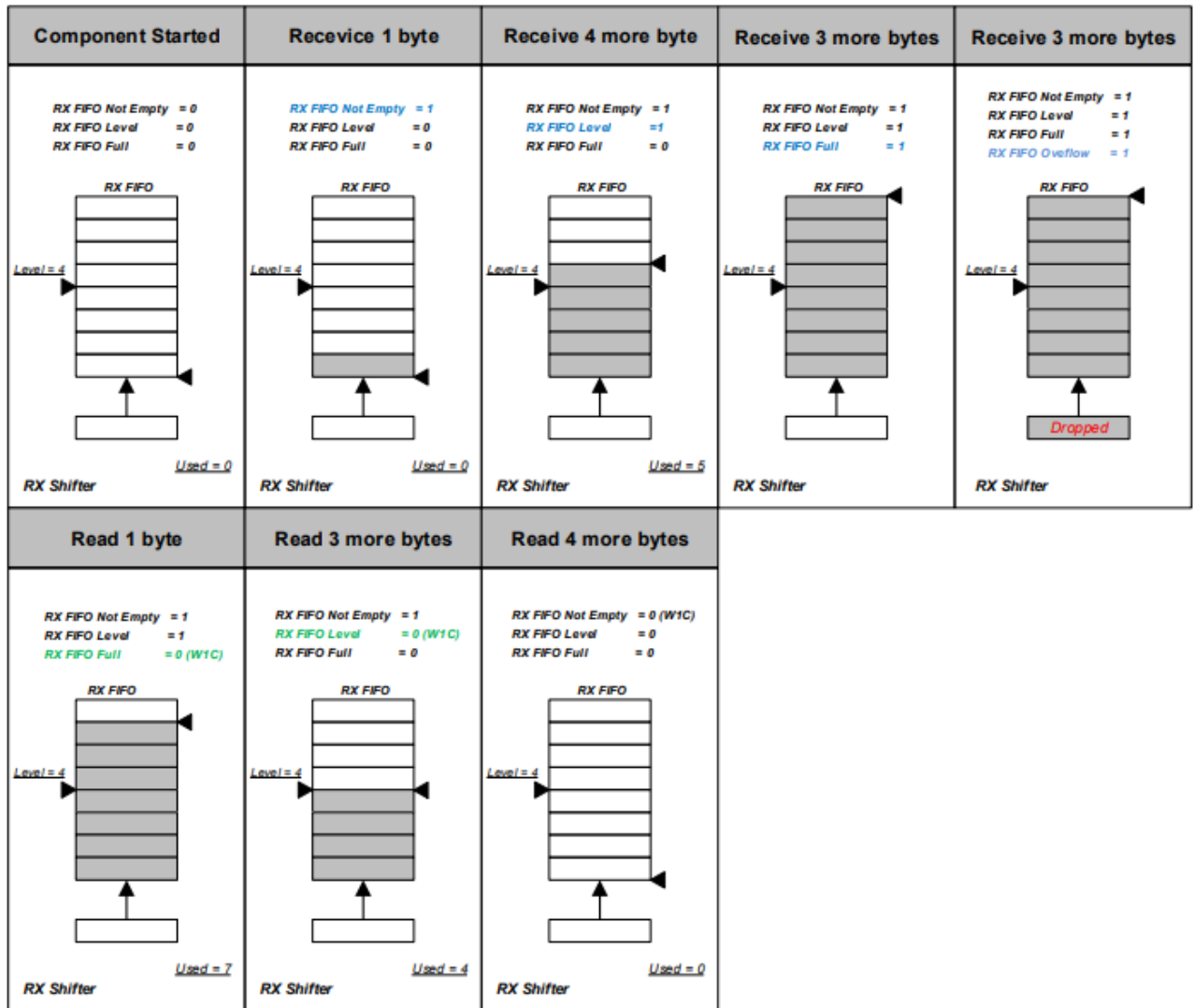


Figure 28-49. RX Interrupt Source Operation



Empty app application in Hello world workspace which is illustrates uart tx and rx using pdl :-

Device configuration settings :-

C:/Users/sidra/Hello_world/Empty_App/bsps/TARGET_APP_CY8CPROTO-062-4343W/config/design.modus* - Device Configurator 4.0

File Edit View Help

CY8C624ABZI-S2D44 LBE5KL1DX/CYW4343WKUBG

Serial Communication Block (SCB) 5 (CYBSP_UART) - Parameters

Enter filter text...

Peripherals Pins Analog-Routing System Peripheral-Clocks DMA

Enter filter text...

Resource Name(s) Personality

> Analog

Communication

☐ Inter-IC Sound Bus (I2S) 0 audioss_0_i2s_0

☐ Inter-IC Sound Bus (I2S) 1 audioss_1_i2s_0

☐ Quad Serial Memory Interface (QSPI) 0 smif_0

☐ SD Host Controller (SDHC) 0 sdhc_0

☐ SD Host Controller (SDHC) 1 sdhc_1

☐ Serial Communication Block (SCB) 0 scb_0

☐ Serial Communication Block (SCB) 1 scb_1

☐ Serial Communication Block (SCB) 2 scb_2

☐ Serial Communication Block (SCB) 3 scb_3

☐ Serial Communication Block (SCB) 4 scb_4

☒ Serial Communication Block (SCB) 5 CYBSP_UART UART-3.0 v

☐ Serial Communication Block (SCB) 6 scb_6

☐ Serial Communication Block (SCB) 7 scb_7

☐ Serial Communication Block (SCB) 8 scb_8

☐ Serial Communication Block (SCB) 9 scb_9

☐ Serial Communication Block (SCB) 10 scb_10

☐ Serial Communication Block (SCB) 11 scb_11

☐ Serial Communication Block (SCB) 12 scb_12

☐ Universal Serial Bus (USB) 0 usb_0

Digital

☐ PDM-PCM Converter 0 audioss_0_pdm_0

Timer, Counter, and PWM (TCPWM) 0

☐ TCPWM[0] 32-bit Counter 0 tcpwm_0_cnt_0

☐ TCPWM[0] 32-bit Counter 1 tcpwm_0_cnt_1

☐ TCPWM[0] 32-bit Counter 2 tcpwm_0_cnt_2

☐ TCPWM[0] 32-bit Counter 3 tcpwm_0_cnt_3

☐ TCPWM[0] 32-bit Counter 4 tcpwm_0_cnt_4

☐ TCPWM[0] 32-bit Counter 5 tcpwm_0_cnt_5

Serial Communication Block (SCB) 5 (CYBSP_UART) - Parameters

Enter filter text...

Name Value

Overview

Configuration Help [Open UART \(SCB\) Documentation](#)

General

Com Mode Standard

Baud Rate (bps) 115200

Oversample 8

Bit Order LSB First

Data Width 8 bits

Parity None

Stop Bits 1 bit

Enable Digital Filter ☐

Support RS-485

TX-Enable ☐

Flow Control

Connections

Clock 16 bit Divider 1 clk [USED]

RX P5[0] digital_inout (CYBSP_DEBUG_UART_RX, CYBSP_I2S_MCLK) [USED]

TX P5[1] digital_inout (CYBSP_DEBUG_UART_TX, CYBSP_I2S_TX_SCK) [USED]

RX Trigger Output <unassigned>

TX Trigger Output <unassigned>

Actual Baud Rate

Actual Baud Rate (bps) 114678

Baud Rate Accuracy (%) 0.453

Clock Frequency 917.431 kHz

Trigger Level

RX FIFO Level 4

TX FIFO Level 4

Multi Processor Mode

Advanced

API Mode

API Mode High Level

Code Preview :-

/* NOTE: This is a preview only. It combines elements of the

* cycfg_peripherals.c and cycfg_peripherals.h files located in the folder

* C:/Users/sidra/Hello_world/Empty_App/bsps/TARGET_APP_CY8CPROTO-062-4343W/config/GeneratedSource.

*/

```
#include "cy_scb_uart.h"
```

```
#include "cy_sysclk.h"
```

```
#if defined (CY_USING_HAL)
```

```
#include "cyhal_hwmgr.h"
```

```
#endif //defined (CY_USING_HAL)
```

```
#define CYBSP_UART_HW SCB5
```

```
#define CYBSP_UART_IRQ scb_5_interrupt_IRQn
```

```
const cy_stc_scb_uart_config_t CYBSP_UART_config =
```

```
{
```

```
.uartMode = CY_SCB_UART_STANDARD,
```

```
.enableMutliProcessorMode = false,
```

```
.smartCardRetryOnNack = false,
```

```
.irdaInvertRx = false,
```

```
.irdaEnableLowPowerReceiver = false,
```

```
.oversample = 8,
```

```
.enableMsbFirst = false,
```

```

.dataWidth = 8UL,
.parity = CY_SCB_UART_PARITY_NONE,
.stopBits = CY_SCB_UART_STOP_BITS_1,
.enableInputFilter = false,
.breakWidth = 11UL,
.dropOnFrameError = false,
.dropOnParityError = false,
.receiverAddress = 0x0UL,
.receiverAddressMask = 0x0UL,
.acceptAddrInFifo = false,
.enableCts = false,
.ctsPolarity = CY_SCB_UART_ACTIVE_LOW,
.rtsRxFifoLevel = 0UL,
.rtsPolarity = CY_SCB_UART_ACTIVE_LOW,
.rxFifoTriggerLevel = 4UL,
.rxFifoIntEnableMask = 0UL,
.txFifoTriggerLevel = 4UL,
.txFifoIntEnableMask = 0UL,
};
#if defined (CY_USING_HAL)
const cyhal_resource_inst_t CYBSP_UART_obj =
{
.type = CYHAL_RSC_SCB,
.block_num = 5U,
.channel_num = 0U,
};
#endif //defined (CY_USING_HAL)

void init_cycfg_peripherals(void)
{
Cy_SysClk_PeriphAssignDivider(PCLK_SCB5_CLOCK, CY_SYSClk_DIV_16_BIT, 1U);
}

void reserve_cycfg_peripherals(void)
{
#if defined (CY_USING_HAL)
cyhal_hwmgr_reserve(&CYBSP_UART_obj);
#endif //defined (CY_USING_HAL)
}

```

C:\Users\sidra\Hello_world\Empty_App\bsps\TARGET_APP_CY8CPROTO-062-4343W\conf\design.module - Device Configurator 4.0

File Edit View Help

CY8C624ABZI-S2D44 LBEE5KL1DX/CYW4343WKUBG

Peripherals Pins Analog-Routing System Peripheral-Clocks

Enter filter text...

| Resource | Name(s) | Personality |
|--|-----------------|----------------------|
| > 8 bit | | |
| 16 bit | | |
| <input checked="" type="checkbox"/> 16 bit Divider 0 | peri_0_div_16_0 | Peripheral Clock-3.0 |
| <input checked="" type="checkbox"/> 16 bit Divider 1 | peri_0_div_16_1 | Peripheral Clock-3.0 |
| <input type="checkbox"/> 16 bit Divider 2 | peri_0_div_16_2 | |
| <input type="checkbox"/> 16 bit Divider 3 | peri_0_div_16_3 | |
| <input type="checkbox"/> 16 bit Divider 4 | peri_0_div_16_4 | |
| <input type="checkbox"/> 16 bit Divider 5 | peri_0_div_16_5 | |
| <input type="checkbox"/> 16 bit Divider 6 | peri_0_div_16_6 | |
| <input type="checkbox"/> 16 bit Divider 7 | peri_0_div_16_7 | |

16 bit Divider 1 - Parameters - Device Configurator 4.0

Enter filter text...

| Name | Value |
|--------------------|--|
| Overview | |
| Configuration Help | Open Peripherals Clock Dividers Documentation |
| General | |
| Source Clock | CLK_PERI (100 MHz ± 2.4%) |
| Divider | 109 |
| Frequency | 917.4 kHz ± 2.4% |
| Start on Reset | <input checked="" type="checkbox"/> |
| Peripherals | <input checked="" type="checkbox"/> Serial Communication Block (SCB) 5 clock (CYBSP_UART) [USED] |

```

/* NOTE: This is a preview only. It combines elements of the
* cycfg_clocks.c and cycfg_clocks.h files located in the folder
* C:/Users/sidra/Hello_world/Empty_App/bsps/TARGET_APP_CY8CPROTO-062-4343W/config/GeneratedSource.
*/

```

```

#include "cy_sysclk.h"
#if defined (CY_USING_HAL)
#include "cyhal_hwmgr.h"
#endif //defined (CY_USING_HAL)

#define peri_0_div_16_1_HW CY_SYSClk_DIV_16_BIT
#define peri_0_div_16_1_NUM 1U

#if defined (CY_USING_HAL)
const cyhal_resource_inst_t peri_0_div_16_1_obj =
{
.type = CYHAL_RSC_CLOCK,
.block_num = peri_0_div_16_1_HW,
.channel_num = peri_0_div_16_1_NUM,
};
#endif //defined (CY_USING_HAL)

void init_cycfg_clocks(void)
{
Cy_SysClk_PeriphDisableDivider(CY_SYSClk_DIV_16_BIT, 1U);
Cy_SysClk_PeriphSetDivider(CY_SYSClk_DIV_16_BIT, 1U, 108U);
Cy_SysClk_PeriphEnableDivider(CY_SYSClk_DIV_16_BIT, 1U);
}

void reserve_cycfg_clocks(void)
{
#if defined (CY_USING_HAL)
cyhal_hwmgr_reserve(&peri_0_div_16_1_obj);
#endif //defined (CY_USING_HAL)
}

```

CY8C624ABZI-S2D44

LBEE5KL1DX/CYW4343WKUBG

Peripherals

Pins

Analog-Routing

System

Peripheral-Clocks

DMA

Enter filter text...

| Resource | Name(s) | Personality |
|-----------|--|-------------|
| > Port 0 | | |
| > Port 1 | | |
| > Port 2 | | |
| > Port 3 | | |
| > Port 4 | | |
| > Port 5 | | |
| > Port 6 | | |
| > Port 7 | | |
| > Port 8 | | |
| > Port 9 | | |
| > Port 10 | | |
| > Port 11 | | |
| > Port 12 | | |
| > Port 13 | <div> <div><input type="checkbox"/></div> <div>P13[0] CYBSP_SDHC_IO0</div> </div> <div> <div><input type="checkbox"/></div> <div>P13[1] CYBSP_SDHC_IO1</div> </div> <div> <div><input type="checkbox"/></div> <div>P13[2] CYBSP_SDHC_IO2</div> </div> <div> <div><input type="checkbox"/></div> <div>P13[3] CYBSP_SDHC_IO3</div> </div> <div> <div><input type="checkbox"/></div> <div>P13[4] ioss_0_port_13_pin_4</div> </div> <div> <div><input type="checkbox"/></div> <div>P13[5] CYBSP_SDHC_DETECT</div> </div> <div> <div><input type="checkbox"/></div> <div>P13[6] ioss_0_port_13_pin_6</div> </div> <div> <div><input checked="" type="checkbox"/></div> <div>P13[7] CYBSP_LED4,CYBSP_USER_LED1,CYBSP_USER_LED</div> </div> | Pin-3.0 |
| > Port 14 | | |

P13[7] (CYBSP_LED4, CYBSP_USER_LED1, CYBSP_USER_LED) - Parameters

Enter filter text...

| Name | Value |
|--|-------------------------------------|
| Overview | |
| Configuration Help Open GPIO Documentation | |
| General | |
| Drive Mode | Resistive Pull-Up, Input buffer off |
| Initial Drive State | Low (0) |
| Input | |
| Threshold | CMOS |
| Interrupt Trigger Type | None |
| Output | |
| Slew Rate | Fast |
| Drive Strength | 1 / 2 |
| Internal Connection | |
| Analog | <unassigned> |
| Digital Input | <unassigned> |
| Digital Output | <unassigned> |
| Digital InOut | <unassigned> |
| Advanced | |
| Store Config in Flash | <input checked="" type="checkbox"/> |

```
/* NOTE: This is a preview only. It combines elements of the
 * cycfg_pins.c and cycfg_pins.h files located in the folder
 * C:/Users/sidra/Hello_world/Empty_App/bsps/TARGET_APP_CY8CPROTO-062-4343W/config/GeneratedSource.
 */
```

```
#include "cy_gpio.h"
#if defined (CY_USING_HAL)
#include "cyhal_hwmgr.h"
#endif //defined (CY_USING_HAL)

#define CYBSP_LED4_PORT GPIO_PRT13
#define CYBSP_USER_LED1_PORT CYBSP_LED4_PORT
#define CYBSP_USER_LED_PORT CYBSP_LED4_PORT
#define CYBSP_LED4_PORT_NUM 13U
#define CYBSP_USER_LED1_PORT_NUM CYBSP_LED4_PORT_NUM
#define CYBSP_USER_LED_PORT_NUM CYBSP_LED4_PORT_NUM
#define CYBSP_LED4_PIN 7U
#define CYBSP_USER_LED1_PIN CYBSP_LED4_PIN
#define CYBSP_USER_LED_PIN CYBSP_LED4_PIN
#define CYBSP_LED4_NUM 7U
#define CYBSP_USER_LED1_NUM CYBSP_LED4_NUM
#define CYBSP_USER_LED_NUM CYBSP_LED4_NUM
#define CYBSP_LED4_DRIVEMODE CY_GPIO_DM_PULLUP_IN_OFF
#define CYBSP_USER_LED1_DRIVEMODE CYBSP_LED4_DRIVEMODE
#define CYBSP_USER_LED_DRIVEMODE CYBSP_LED4_DRIVEMODE
#define CYBSP_LED4_INIT_DRIVESTATE 0
#define CYBSP_USER_LED1_INIT_DRIVESTATE CYBSP_LED4_INIT_DRIVESTATE
#define CYBSP_USER_LED_INIT_DRIVESTATE CYBSP_LED4_INIT_DRIVESTATE
#ifndef ioss_0_port_13_pin_7_HSIOM
#define ioss_0_port_13_pin_7_HSIOM HSIOM_SEL_GPIO
#endif
#define CYBSP_LED4_HSIOM ioss_0_port_13_pin_7_HSIOM
#define CYBSP_USER_LED1_HSIOM CYBSP_LED4_HSIOM
#define CYBSP_USER_LED_HSIOM CYBSP_LED4_HSIOM
#define CYBSP_LED4_IRQ ioss_interrupts_gpio_13_IRQn
#define CYBSP_USER_LED1_IRQ CYBSP_LED4_IRQ
#define CYBSP_USER_LED_IRQ CYBSP_LED4_IRQ
#if defined (CY_USING_HAL)
#define CYBSP_LED4_HAL_PORT_PIN P13_7
#define CYBSP_USER_LED1_HAL_PORT_PIN CYBSP_LED4_HAL_PORT_PIN
#define CYBSP_USER_LED_HAL_PORT_PIN CYBSP_LED4_HAL_PORT_PIN
#define CYBSP_LED4_P13_7
#define CYBSP_USER_LED1_CYBSP_LED4
#define CYBSP_USER_LED_CYBSP_LED4
#define CYBSP_LED4_HAL_IRQ CYHAL_GPIO_IRQ_NONE
#define CYBSP_USER_LED1_HAL_IRQ CYBSP_LED4_HAL_IRQ
#define CYBSP_USER_LED_HAL_IRQ CYBSP_LED4_HAL_IRQ
#define CYBSP_LED4_HAL_DIR CYHAL_GPIO_DIR_OUTPUT
#define CYBSP_USER_LED1_HAL_DIR CYBSP_LED4_HAL_DIR
#define CYBSP_USER_LED_HAL_DIR CYBSP_LED4_HAL_DIR
#define CYBSP_LED4_HAL_DRIVEMODE CYHAL_GPIO_DRIVE_PULLUP
#define CYBSP_USER_LED1_HAL_DRIVEMODE CYBSP_LED4_HAL_DRIVEMODE
#define CYBSP_USER_LED_HAL_DRIVEMODE CYBSP_LED4_HAL_DRIVEMODE
#endif //defined (CY_USING_HAL)
```

```
const cy_stc_gpio_pin_config_t CYBSP_LED4_config =
{
```

```

.outVal = 0,
.driveMode = CY_GPIO_DM_PULLUP_IN_OFF,
.hsiom = CYBSP_LED4_HSIOM,
.intEdge = CY_GPIO_INTR_DISABLE,
.intMask = 0UL,
.vtrip = CY_GPIO_VTRIP_CMOS,
.slewRate = CY_GPIO_SLEW_FAST,
.driveSel = CY_GPIO_DRIVE_1_2,
.vregEn = 0UL,
.ibufMode = 0UL,
.vtripSel = 0UL,
.vrefSel = 0UL,
.vohSel = 0UL,
};
#if defined (CY_USING_HAL)
const cyhal_resource_inst_t CYBSP_LED4_obj =
{
.type = CYHAL_RSC_GPIO,
.block_num = CYBSP_LED4_PORT_NUM,
.channel_num = CYBSP_LED4_PIN,
};
#endif //defined (CY_USING_HAL)

void init_cycfg_pins(void)
{
Cy_GPIO_Pin_Init(CYBSP_LED4_PORT, CYBSP_LED4_PIN, &CYBSP_LED4_config);
}

void reserve_cycfg_pins(void)
{
#if defined (CY_USING_HAL)
cyhal_hwmgr_reserve(&CYBSP_LED4_obj);
#endif //defined (CY_USING_HAL)
}

```

Code Understanding in main :-

```

__STATIC_INLINE void Cy_SCB_UART_Enable(CySCB_Type *base);

/*****
* Function Name: Cy_SCB_UART_Enable
*****/
*
* Enables the SCB block for the UART operation.
*
* \param base
* The pointer to the UART SCB instance.
*
*****/
__STATIC_INLINE void Cy_SCB_UART_Enable(CySCB_Type *base)
{
    SCB_CTRL(base) |= SCB_CTRL_ENABLED_Msk;
}

```

```
core_cm4.h  main.c  cy_scb_uart.h  cy_scb_uart.c  cyip_scb.h

#define SCB_CTRL_BYTE_MODE_Msk          0x800UL
#define SCB_CTRL_CMD_RESP_MODE_Pos      12UL
#define SCB_CTRL_CMD_RESP_MODE_Msk      0x1000UL
#define SCB_CTRL_ADDR_ACCEPT_Pos        16UL
#define SCB_CTRL_ADDR_ACCEPT_Msk        0x10000UL
#define SCB_CTRL_BLOCK_Pos              17UL
#define SCB_CTRL_BLOCK_Msk              0x20000UL
#define SCB_CTRL_MODE_Pos               24UL
#define SCB_CTRL_MODE_Msk               0x3000000UL
#define SCB_CTRL_ENABLED_Pos            31UL
#define SCB_CTRL_ENABLED_Msk            0x80000000UL
/* SCB.STATUS */
#define SCB_STATUS_EC_BUSY_Pos          0UL
#define SCB_STATUS_EC_BUSY_Msk          0x1UL
/* SCB.CMD RESP CTRL */
```

```

/*****
 * Function Name: Cy_SCB_UART_GetNumInRxFifo
 *****/
 * Returns the number of data elements in the UART RX FIFO.
 *
 * \param base
 * The pointer to the UART SCB instance.
 *
 * \return
 * The number of data elements in the RX FIFO.
 * The size of data element defined by the configured data width.
 *****/
STATIC_INLINE uint32_t Cy_SCB_UART_GetNumInRxFifo(CySCB_Type const *base)
{
    return Cy_SCB_GetNumInRxFifo(base);
}
```

```

/*****
 * Function Name: Cy_SCB_GetNumInRxFifo
 *****/
 * Returns the number of data elements currently in the RX FIFO.
 *
 * \param base
 * The pointer to the SCB instance.
 *
 * \return
 * The number of data elements in RX FIFO.
 *****/
STATIC_INLINE uint32_t Cy_SCB_GetNumInRxFifo(CySCB_Type const *base)
{
    return _FLD2VAL(SCB_RX_FIFO_STATUS_USED, SCB_RX_FIFO_STATUS(base));
}
```


| Bits | Name | Description |
|---------|----------|--|
| 30 : 24 | WR_PTR | FIFO write pointer: FIFO location at which a new data frame is written by the hardware. Default Value: 0 |
| 22 : 16 | RD_PTR | FIFO read pointer: FIFO location from which a data frame is read. Default Value: 0 |
| 15 | SR_VALID | Indicates whether the RX shift registers holds a (partial) valid data frame ('1') or not ('0'). The shift register can be considered the bottom of the RX FIFO (the data frame is not included in the USED field of the RX FIFO). The shift register is a working register and holds the data frame that is currently being received (when the protocol state machine is receiving a data frame). Default Value: 0 |
| 7 : 0 | USED | Amount of entries in the receiver FIFO. The value of this field ranges from 0 to FF_DATA_NR. Default Value: 0 |

```

/*****
* Function Name: Cy_SCB_UART_Init
*****/
*
* Initializes the SCB for UART operation.
*
* \param base
* The pointer to the UART SCB instance.
*
* \param config
* The pointer to configuration structure \ref cy_stc_scb_uart_config_t.
*
* \param context
* The pointer to the context structure \ref cy_stc_scb_uart_context_t allocated
* by the user. The structure is used during the UART operation for internal
* configuration and data retention. The user must not modify anything
* in this structure.
* If only UART \ref group_scb_uart_ll will be used pass NULL as pointer to
* context.
*
* \return
* \ref cy_en_scb_uart_status_t
*
* \note
* Ensure that the SCB block is disabled before calling this function.
*
*****/
cy_en_scb_uart_status_t Cy_SCB_UART_Init(CySCB_Type *base, cy_stc_scb_uart_config_t const *config,
cy_stc_scb_uart_context_t *context)
{
    if ((NULL == base) || (NULL == config))
    {
        return CY_SCB_UART_BAD_PARAM;
    }

    CY_ASSERT_L3(CY_SCB_UART_IS_MODE_VALID (config->uartMode));
    CY_ASSERT_L3(CY_SCB_UART_IS_STOP_BITS_VALID(config->stopBits));
    CY_ASSERT_L3(CY_SCB_UART_IS_PARITY_VALID (config->parity));
    CY_ASSERT_L3(CY_SCB_UART_IS_POLARITY_VALID (config->ctsPolarity));
    CY_ASSERT_L3(CY_SCB_UART_IS_POLARITY_VALID (config->rtsPolarity));

    CY_ASSERT_L2(CY_SCB_UART_IS_OVERSAMPLE_VALID (config->oversample, config->uartMode, config->
irdaEnableLowPowerReceiver));
    CY_ASSERT_L2(CY_SCB_UART_IS_DATA_WIDTH_VALID (config->dataWidth));
    CY_ASSERT_L2(CY_SCB_UART_IS_ADDRESS_VALID (config->receiverAddress));
    CY_ASSERT_L2(CY_SCB_UART_IS_ADDRESS_MASK_VALID(config->receiverAddressMask));

    CY_ASSERT_L2(CY_SCB_UART_IS_MUTLI_PROC_VALID (config->enableMutliProcessorMode, config->uartMode,
config->dataWidth, config->parity));

    CY_ASSERT_L2(CY_SCB_IS_INTR_VALID(config->rxFifoIntEnableMask, CY_SCB_UART_RX_INTR_MASK));
    CY_ASSERT_L2(CY_SCB_IS_INTR_VALID(config->txFifoIntEnableMask, CY_SCB_UART_TX_INTR_MASK));

    uint32_t ovs;

```

```

if ((CY_SCB_UART_IRDA == config->uartMode) && (!config->irdaEnableLowPowerReceiver))
{
    /* For Normal IrDA mode oversampling is always zero */
    ovs = 0UL;
}
else
{
    ovs = (config->oversample - 1UL);
}

/* Configure the UART interface */
#if((defined (CY_IP_MXSCB_VERSION) && (CY_IP_MXSCB_VERSION>=2)) || defined (CY_IP_MXS22SCB))
    SCB_CTRL(base) = _BOOL2FLD(SCB_CTRL_ADDR_ACCEPT, config->acceptAddrInFifo) |
        _VAL2FLD(SCB_CTRL_MEM_WIDTH, ((config->dataWidth <= CY_SCB_BYTE_WIDTH)? 0UL:1UL)) |
        _VAL2FLD(SCB_CTRL_OVS, ovs) |
        _VAL2FLD(SCB_CTRL_MODE, CY_SCB_CTRL_MODE_UART);
#elif((defined (CY_IP_MXSCB_VERSION) && CY_IP_MXSCB_VERSION==1))
    SCB_CTRL(base) = _BOOL2FLD(SCB_CTRL_ADDR_ACCEPT, config->acceptAddrInFifo) |
        _BOOL2FLD(SCB_CTRL_BYTE_MODE, (config->dataWidth <= CY_SCB_BYTE_WIDTH)) |
        _VAL2FLD(SCB_CTRL_OVS, ovs) |
        _VAL2FLD(SCB_CTRL_MODE, CY_SCB_CTRL_MODE_UART);
#endif /* CY_IP_MXSCB_VERSION */
/* Configure SCB_CTRL.BYTE_MODE then verify levels */
CY_ASSERT_L2(CY_SCB_IS_TRIGGER_LEVEL_VALID(base, config->rxFifoTriggerLevel));
CY_ASSERT_L2(CY_SCB_IS_TRIGGER_LEVEL_VALID(base, config->txFifoTriggerLevel));
CY_ASSERT_L2(CY_SCB_IS_TRIGGER_LEVEL_VALID(base, config->rtsRxFifoLevel));

SCB_UART_CTRL(base) = _VAL2FLD(SCB_UART_CTRL_MODE, (uint32_t) config->uartMode);

/* Configure the RX direction */
SCB_UART_RX_CTRL(base) = _BOOL2FLD(SCB_UART_RX_CTRL_POLARITY, config->irdaInvertRx) |
    _BOOL2FLD(SCB_UART_RX_CTRL_MP_MODE, config->enableMutliProcessorMode) |
    _BOOL2FLD(SCB_UART_RX_CTRL_DROP_ON_PARITY_ERROR, config->dropOnParityError) |
    _BOOL2FLD(SCB_UART_RX_CTRL_DROP_ON_FRAME_ERROR, config->dropOnFrameError) |
    _VAL2FLD(SCB_UART_RX_CTRL_BREAK_WIDTH, (config->breakWidth - 1UL)) |
    _VAL2FLD(SCB_UART_RX_CTRL_STOP_BITS, ((uint32_t) config->stopBits) - 1UL) |
    _VAL2FLD(CY_SCB_UART_RX_CTRL_SET_PARITY, (uint32_t) config->parity);
#if((defined (CY_IP_MXSCB_VERSION) && (CY_IP_MXSCB_VERSION>=2)) || defined (CY_IP_MXS22SCB))
    SCB_UART_RX_CTRL(base) |= _BOOL2FLD(SCB_UART_RX_CTRL_BREAK_LEVEL, config->breakLevel);
#endif /* CY_IP_MXSCB_VERSION */

#if ((defined(CY_IP_MXSCB_VERSION)) && (CY_IP_MXSCB_VERSION >= 4))
    SCB_UART_RX_CTRL(base) |= _BOOL2FLD(SCB_UART_RX_CTRL_HDXEN, config->halfDuplexMode);
#endif /* ((defined(CY_IP_MXSCB_VERSION)) && (CY_IP_MXSCB_VERSION >= 4)) */

SCB_RX_CTRL(base) = _BOOL2FLD(SCB_RX_CTRL_MSB_FIRST, config->enableMsbFirst) |
    _BOOL2FLD(SCB_RX_CTRL_MEDIAN, ((config->enableInputFilter) || \
        (config->uartMode == CY_SCB_UART_IRDA))) |
    _VAL2FLD(SCB_RX_CTRL_DATA_WIDTH, (config->dataWidth - 1UL));

SCB_RX_MATCH(base) = _VAL2FLD(SCB_RX_MATCH_ADDR, config->receiverAddress) |
    _VAL2FLD(SCB_RX_MATCH_MASK, config->receiverAddressMask);

/* Configure SCB_CTRL.RX_CTRL then verify break width */
CY_ASSERT_L2(CY_SCB_UART_IS_RX_BREAK_WIDTH_VALID(base, config->breakWidth));

/* Configure the TX direction */
SCB_UART_TX_CTRL(base) = _BOOL2FLD(SCB_UART_TX_CTRL_RETRY_ON_NACK, ((config->smartCardRetryOnNack) &&
    (config->uartMode ==
CY_SCB_UART_SMARTCARD))) |
    _VAL2FLD(SCB_UART_TX_CTRL_STOP_BITS, ((uint32_t) config->stopBits) - 1UL) |
    _VAL2FLD(CY_SCB_UART_TX_CTRL_SET_PARITY, (uint32_t) config->parity);

SCB_TX_CTRL(base) = _BOOL2FLD(SCB_TX_CTRL_MSB_FIRST, config->enableMsbFirst) |
    _VAL2FLD(SCB_TX_CTRL_DATA_WIDTH, (config->dataWidth - 1UL)) |
    _BOOL2FLD(SCB_TX_CTRL_OPEN_DRAIN, (config->uartMode == CY_SCB_UART_SMARTCARD));

```

```

SCB_RX_FIFO_CTRL(base) = _VAL2FLD(SCB_RX_FIFO_CTRL_TRIGGER_LEVEL, config->rxFifoTriggerLevel);

/* Configure the flow control */
SCB_UART_FLOW_CTRL(base) = _BOOL2FLD(SCB_UART_FLOW_CTRL_CTS_ENABLED, config->enableCts) |
    _BOOL2FLD(SCB_UART_FLOW_CTRL_CTS_POLARITY, (CY_SCB_UART_ACTIVE_HIGH == config-
>ctsPolarity)) |
    _BOOL2FLD(SCB_UART_FLOW_CTRL_RTS_POLARITY, (CY_SCB_UART_ACTIVE_HIGH == config-
>rtsPolarity)) |
    _VAL2FLD(SCB_UART_FLOW_CTRL_TRIGGER_LEVEL, config->rtsRxFifoLevel);

SCB_TX_FIFO_CTRL(base) = _VAL2FLD(SCB_TX_FIFO_CTRL_TRIGGER_LEVEL, config->txFifoTriggerLevel);

/* Set up interrupt sources */
SCB_INTR_RX_MASK(base) = (config->rxFifoIntEnableMask & CY_SCB_UART_RX_INTR_MASK);
SCB_INTR_TX_MASK(base) = (config->txFifoIntEnableMask & CY_SCB_UART_TX_INTR_MASK);

/* Initialize context */
if (NULL != context)
{
    context->rxStatus = 0UL;
    context->txStatus = 0UL;

    context->rxRingBuf = NULL;
    context->rxRingBufSize = 0UL;

    context->rxBufIdx = 0UL;
    context->txLeftToTransmit = 0UL;

    context->cbEvents = NULL;
    context->irdaEnableLowPowerReceiver = config->irdaEnableLowPowerReceiver;

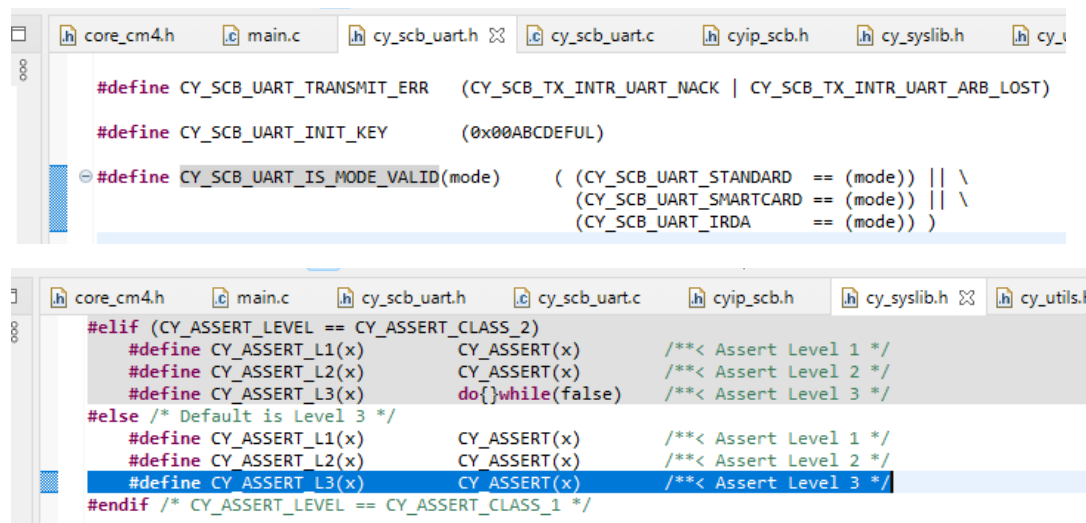
#ifdef NDEBUG
    /* Put an initialization key into the initKey variable to verify
     * context initialization in the transfer API.
     */
    context->initKey = CY_SCB_UART_INIT_KEY;
#endif /* !NDEBUG */
}

return CY_SCB_UART_SUCCESS;
}

```

Decoding each lines: -

```
CY_ASSERT_L3(CY_SCB_UART_IS_MODE_VALID (config->uartMode));
```



The screenshot shows an IDE with two windows. The top window displays the contents of `cy_scb_uart.h`, showing several `#define` macros. The bottom window displays the contents of `cy_scb_uart.c`, showing `#if` and `#define` blocks for assertion levels.

Top Window: `cy_scb_uart.h`

```

#define CY_SCB_UART_TRANSMIT_ERR (CY_SCB_TX_INTR_UART_NACK | CY_SCB_TX_INTR_UART_ARB_LOST)

#define CY_SCB_UART_INIT_KEY (0x00ABCDEFUL)

#define CY_SCB_UART_IS_MODE_VALID(mode) ( (CY_SCB_UART_STANDARD == (mode)) || \
    (CY_SCB_UART_SMARTCARD == (mode)) || \
    (CY_SCB_UART_IRDA == (mode)) )

```

Bottom Window: `cy_scb_uart.c`

```

#ifdef CY_ASSERT_LEVEL
    #if (CY_ASSERT_LEVEL == CY_ASSERT_CLASS_2)
        #define CY_ASSERT_L1(x) CY_ASSERT(x) /*< Assert Level 1 */
        #define CY_ASSERT_L2(x) CY_ASSERT(x) /*< Assert Level 2 */
        #define CY_ASSERT_L3(x) do{}while(false) /*< Assert Level 3 */
    #else /* Default is Level 3 */
        #define CY_ASSERT_L1(x) CY_ASSERT(x) /*< Assert Level 1 */
        #define CY_ASSERT_L2(x) CY_ASSERT(x) /*< Assert Level 2 */
        #define CY_ASSERT_L3(x) CY_ASSERT(x) /*< Assert Level 3 */
    #endif /* CY_ASSERT_LEVEL == CY_ASSERT_CLASS_1 */

```

```
core_cm4.h | main.c | cy_scb_uart.h | cy_scb_uart.c | cyip_scb.h | cy_syslib.h | cy_utils.h |
- #if defined(NDEBUG) || defined(CY_NO_ASSERT)
- #define CY_ASSERT(x) do { \
-     } while(false)
- #else
- #define CY_ASSERT(x) do { \
-     if(!(x)) \
-     { \
-         CY_HALT(); \
-     } \
-     } while(false)
- #endif // defined(NDEBUG)

core_cm4.h | main.c | cy_scb_uart.h | cy_scb_uart.c | cyip_scb.h | cy_syslib.h | cy_utils.h |
- /** Simple macro to suppress the unused parameter warning by casting to void. */
- #define CY_UNUSED_PARAMETER(x) ( (void)(x) )
- /** Halt the processor in the debug state
-  */
- static inline void CY_HALT(void)
- {
-     __asm("    bkpt    1");
- }


```

CY_ASSERT_L2(CY_SCB_IS_INTR_VALID(config->rxFifoIntEnableMask, CY_SCB_UART_RX_INTR_MASK));

CY_SCB_IS_INTR_VALID is used for checking the receive interrupt is valid

```
core_cm4.h | main.c | cy_scb_uart.h | cy_scb_uart.c | cyip_scb.h | cy_syslib.h | cy_utils.h | cy_scb_common.h |
909
910 //*****
911 * Internal Constants
912 //*****
913
914 /** \cond INTERNAL */
915 #define CY_SCB_UART_TX_INTR_MASK (CY_SCB_UART_TX_TRIGGER | CY_SCB_UART_TX_NOT_FULL | CY_SCB_UART_TX_EMPTY | \
916     CY_SCB_UART_TX_OVERFLOW | CY_SCB_UART_TX_UNDERFLOW | CY_SCB_UART_TX_DONE | \
917     CY_SCB_UART_TX_NACK | CY_SCB_UART_TX_ARB_LOST)
918
919 #define CY_SCB_UART_RX_INTR_MASK (CY_SCB_UART_RX_TRIGGER | CY_SCB_UART_RX_NOT_EMPTY | CY_SCB_UART_RX_FULL | \
920     CY_SCB_UART_RX_OVERFLOW | CY_SCB_UART_RX_UNDERFLOW | CY_SCB_UART_RX_ERR_FRAME | \
921     CY_SCB_UART_RX_ERR_PARITY | CY_SCB_UART_RX_BREAK_DETECT)
922
923

```

```

core_cm4.h  main.c  cy_scb_uart.h  cy_scb_uart.c  cyip_scb.h  cy_syslib.h  cy_utils.h
724 /** \} group_scb_uart_macros_irda_lp_ovs */
725
726 /**
727 * \defgroup group_scb_uart_macros_rx_fifo_status UART RX FIFO status.
728 * \{
729 * Macros to check UART RX FIFO status returned by \ref Cy_SCB_UART_GetRxFifoStatus
730 * function or assign mask for \ref Cy_SCB_UART_ClearRxFifoStatus function.
731 * Each UART RX FIFO status is encoded in a separate bit, therefore multiple
732 * bits may be set to indicate the current status.
733 */
734
735 /** The number of entries in the RX FIFO is more than the RX FIFO trigger level
736 * value
737 */
738 #define CY_SCB_UART_RX_TRIGGER (SCB_INTR_RX_TRIGGER_Msk)
739
740 /** The RX FIFO is not empty, there is data to read */
741 #define CY_SCB_UART_RX_NOT_EMPTY (SCB_INTR_RX_NOT_EMPTY_Msk)
742
743 /**
744 * The RX FIFO is full, there is no more space for additional data, and
745 * any additional data will be dropped
746 */
747 #define CY_SCB_UART_RX_FULL (SCB_INTR_RX_FULL_Msk)
748
749 /**
750 * The RX FIFO was full and there was an attempt to write to it.
751 * That additional data was dropped.
752 */
753 #define CY_SCB_UART_RX_OVERFLOW (SCB_INTR_RX_OVERFLOW_Msk)
754
755 /** An attempt to read from an empty RX FIFO */
756 #define CY_SCB_UART_RX_UNDERFLOW (SCB_INTR_RX_UNDERFLOW_Msk)
757
758 /** The RX FIFO detected a frame error, either a stop or stop-bit error */
759 #define CY_SCB_UART_RX_ERR_FRAME (SCB_INTR_RX_FRAME_ERROR_Msk)
760
761 /** The RX FIFO detected a parity error */
762 #define CY_SCB_UART_RX_ERR_PARITY (SCB_INTR_RX_PARITY_ERROR_Msk)
763

```

```

core_cm4.h  main.c  cy_scb_uart.h  cy_scb_uart.c  cyip_scb.h
661 #define SCB_INTR_TX_MASKED_UART_DONE_Pos 9UL
662 #define SCB_INTR_TX_MASKED_UART_DONE_Msk 0x200UL
663 #define SCB_INTR_TX_MASKED_UART_ARB_LOST_Pos 10UL
664 #define SCB_INTR_TX_MASKED_UART_ARB_LOST_Msk 0x400UL
665 /* SCB_INTR_RX */
666 #define SCB_INTR_RX_TRIGGER_Pos 0UL
667 #define SCB_INTR_RX_TRIGGER_Msk 0x1UL
668 #define SCB_INTR_RX_NOT_EMPTY_Pos 2UL
669 #define SCB_INTR_RX_NOT_EMPTY_Msk 0x4UL
670 #define SCB_INTR_RX_FULL_Pos 3UL
671 #define SCB_INTR_RX_FULL_Msk 0x8UL
672 #define SCB_INTR_RX_OVERFLOW_Pos 5UL
673 #define SCB_INTR_RX_OVERFLOW_Msk 0x20UL
674 #define SCB_INTR_RX_UNDERFLOW_Pos 6UL
675 #define SCB_INTR_RX_UNDERFLOW_Msk 0x40UL
676 #define SCB_INTR_RX_BLOCKED_Pos 7UL
677 #define SCB_INTR_RX_BLOCKED_Msk 0x80UL
678 #define SCB_INTR_RX_FRAME_ERROR_Pos 8UL
679 #define SCB_INTR_RX_FRAME_ERROR_Msk 0x100UL
680 #define SCB_INTR_RX_PARITY_ERROR_Pos 9UL
681 #define SCB_INTR_RX_PARITY_ERROR_Msk 0x200UL
682 #define SCB_INTR_RX_BAUD_DETECT_Pos 10UL
683 #define SCB_INTR_RX_BAUD_DETECT_Msk 0x400UL
684 #define SCB_INTR_RX_BREAK_DETECT_Pos 11UL
685 #define SCB_INTR_RX_BREAK_DETECT_Msk 0x800UL
686 /* SCB_INTR_RX_SET */
687 #define SCB_INTR_RX_SET_TRIGGER_Pos 0UL
688 #define SCB_INTR_RX_SET_TRIGGER_Msk 0x1UL
689 #define SCB_INTR_RX_SET_NOT_EMPTY_Pos 2UL
690 #define SCB_INTR_RX_SET_NOT_EMPTY_Msk 0x4UL
691 #define SCB_INTR_RX_SET_FULL_Pos 3UL
692 #define SCB_INTR_RX_SET_FULL_Msk 0x8UL
693 #define SCB_INTR_RX_SET_OVERFLOW_Pos 5UL
694 #define SCB_INTR_RX_SET_OVERFLOW_Msk 0x20UL

```

CY_SCB_IS_INTR_VALID is used for checking the receive interrupt is valid

```

do {
    \
    if( !( ( 0UL == ((config->rxFifoIntEnableMask) & ((uint32_t)~( (0x1UL) | (0x4UL) | (0x8UL) | \
    (0x20UL) | (0x40UL) | (0x100UL) | \
    (0x200UL) | (0x800UL)) ))) ) ) ) \
    {
        \
    }

```

```

        CY_HALT(); \
    } \
} while(false)
do { \
    if( !( ( OUL == ((config->rxFifoIntEnableMask) & ((uint32_t) ~((0xb6dUL)) ) ) ) ) ) \
    { \
        CY_HALT(); \
    } \
} while(false)

```

-> (OUL == ((config->rxFifoIntEnableMask) & (0xffff f492)))

-> (OUL == ((4) & (0xffff f492)))

->(OUL == (OUL))

0000 0000 0000 0001

0000 0000 0000 0100

0000 0000 0000 1000

0000 0000 0010 0000

0000 0000 0100 0000

0000 0001 0000 0000

0000 0010 0000 0000

0000 1000 0000 0000

0000 1011 0110 1101(final value) = 0xb6d

1111 0100 1001 0010 = 0xffff f492

```

SCB_CTRL(base) = _BOOL2FLD(SCB_CTRL_ADDR_ACCEPT, config->acceptAddrInFifo) |
                 _BOOL2FLD(SCB_CTRL_BYTE_MODE, (config->dataWidth <= CY_SCB_BYTE_WIDTH)) |
                 _VAL2FLD(SCB_CTRL_OVS, ovs) |
                 _VAL2FLD(SCB_CTRL_MODE, CY_SCB_CTRL_MODE_UART);

```

-> SCB_CTRL(base)= 0UL | 0x800UL | 0x7UL | 0x2000000 = 0x2000807

->0000 0010 0000 0000 0000 1000 0000 0111

```

_BOOL2FLD(SCB_CTRL_ADDR_ACCEPT, config->acceptAddrInFifo) (((config->acceptAddrInFifo) != false) ?
(0x10000UL) : 0UL)

```

```

core_cm4.h  main.c  cy_scb_uart.h  cy_scb_uart.c  cyp_scb.h  cy_syslib.h  cy_utils.h  cy
363
364
365- /*****
366  * Macro Name: _BOOL2FLD
367- *****/
368  *
369  * Returns a field mask if the value is not false.
370  * Returns 0, if the value is false.
371  *
372  *****/
373 #define _BOOL2FLD(field, value) (((value) != false) ? (field ## _Msk) : 0UL)
374

```

```

core_cm4.h  main.c  cy_scb_uart.h  cy_scb_uart.c  cyip_scb.h  cy_syslib.h  cy_utils.h  cy_scb_
1510 #define CoreDebug_DEMCR_VC_CORERESET_Pos    0U                               /*!< CoreDebug DE
1511 #define CoreDebug_DEMCR_VC_CORERESET_Msk     (1UL /*<< CoreDebug_DEMCR_VC_CORERESET_Pos*/) /*!< CoreDebug DE
1512
1513 /*@} end of group CMSIS_CoreDebug */
1514
1515
1516 /**
1517 \ingroup CMSIS_core_register
1518 \defgroup CMSIS_core_bitfield Core register bit field macros
1519 \brief Macros for use with bit field definitions (xxx_Pos, xxx_Msk).
1520 @{
1521 */
1522
1523 /**
1524 \brief Mask and shift a bit field value for use in a register bit range.
1525 \param[in] field Name of the register bit field.
1526 \param[in] value Value of the bit field. This parameter is interpreted as an uint32_t type.
1527 \return Masked and shifted value.
1528 */
1529 #define VAL2FLD(field, value) (((uint32_t)(value) << field ## _Pos) & field ## _Msk)
1530

```

_VAL2FLD(SCB_UART_CTRL_MODE, (uint32_t) config->uartMode) (((uint32_t) ((uint32_t) config->uartMode) << 24UL) & 0x3000000UL)

_VAL2FLD(SCB_CTRL_MEM_WIDTH, ((config->dataWidth <= CY_SCB_BYTE_WIDTH)? 0UL:1UL))

```

core_cm4.h  cy_scb_uart.h  cy_scb_uart.c  cyip_scb.h  cy_utils.h  cy_scb_common.h  » 34
772
773 /* SPI clock modes: CPHA and CPOL */
774 #define CY_SCB_SPI_CTRL_CLK_MODE_Pos    SCB_SPI_CTRL_CPHA_Pos
775 #define CY_SCB_SPI_CTRL_CLK_MODE_Msk   (SCB_SPI_CTRL_CPHA_Msk | SCB_SPI_CTRL_CPOL_Msk)
776
777 /* UART parity and parity enable combination */
778 #define CY_SCB_UART_RX_CTRL_SET_PARITY_Msk (SCB_UART_RX_CTRL_PARITY_ENABLED_Msk | \
779                                             SCB_UART_RX_CTRL_PARITY_Msk)
780 #define CY_SCB_UART_RX_CTRL_SET_PARITY_Pos SCB_UART_RX_CTRL_PARITY_Pos
781
782 #define CY_SCB_UART_TX_CTRL_SET_PARITY_Msk (SCB_UART_TX_CTRL_PARITY_ENABLED_Msk | \
783                                             SCB_UART_TX_CTRL_PARITY_Msk)
784 #define CY_SCB_UART_TX_CTRL_SET_PARITY_Pos SCB_UART_TX_CTRL_PARITY_Pos
785
786 /* Max number of bits for byte mode */
787 #define CY_SCB_BYTE_WIDTH (8UL)
788

```

```

core_cm4.h  cy_scb_uart.h  cy_scb_uart.c  cy_utils.h  cy_scb_common.h  core_cm0.h
1514
1515
1516 /**
1517  \ingroup CMSIS_core_register
1518  \defgroup CMSIS_core_bitfield Core register bit field macros
1519  \brief Macros for use with bit field definitions (xxx_Pos, xxx_Msk).
1520  @{
1521  */
1522
1523 /**
1524  \brief Mask and shift a bit field value for use in a register bit range.
1525  \param[in] field Name of the register bit field.
1526  \param[in] value Value of the bit field. This parameter is interpreted as an uint32_t type.
1527  \return Masked and shifted value.
1528  */
1529 #define VAL2FLD(field, value) (((uint32_t)(value) << field ## _Pos) & field ## _Msk)
1530
1531

```

```

/* Configure SCB_CTRL.BYTE_MODE then verify levels */
CY_ASSERT_L2(CY_SCB_IS_TRIGGER_LEVEL_VALID(base, config->rxFifoTriggerLevel));
CY_ASSERT_L2(CY_SCB_IS_TRIGGER_LEVEL_VALID(base, config->txFifoTriggerLevel));
CY_ASSERT_L2(CY_SCB_IS_TRIGGER_LEVEL_VALID(base, config->rtsRxFifoLevel));

#define CY_SCB_IS_TRIGGER_LEVEL_VALID(base, level) ((level) < Cy_SCB_GetFifoSize(base))

/** \cond INTERNAL */
/*****
 * Function Name: Cy_SCB_GetFifoSize
 *****/
*
* Returns the RX and TX FIFO depth.
*
* \param base
* The pointer to the SCB instance.
*
* \return
* FIFO depth.
*
*****/
STATIC_INLINE uint32_t Cy_SCB_GetFifoSize(CySCB_Type const *base)
{
    #if((defined (CY_IP_MXSCB_VERSION) && (CY_IP_MXSCB_VERSION>=2)) || defined (CY_IP_MXS22SCB))
        {return (((uint32_t)(CY_SCB_FIFO_SIZE)) >> _FLD2VAL(SCB_CTRL_MEM_WIDTH, SCB_CTRL(base)));}
    #elif(defined (CY_IP_MXSCB_VERSION) && (CY_IP_MXSCB_VERSION==1))
        {return (_FLD2BOOL(SCB_CTRL_BYTE_MODE, SCB_CTRL(base)) ? (CY_SCB_FIFO_SIZE) : (CY_SCB_FIFO_SIZE / 2UL));}
    #else
        return 0;
    #endif /* ((CY_IP_MXSCB_VERSION>=2) || defined (CY_IP_MXS22SCB)) */
}

do {
    \
    if(!(((config->rxFifoTriggerLevel) < Cy_SCB_GetFifoSize(base)))) \
    { \
        CY_HALT(); \
    } \
} while(false)

```


2. Configure the UART interface to operate as a Standard protocol by writing '00' to the MODE field (bits [25:24]) of the SCB_UART_CTRL register.

24.1.4 SCB0_UART_CTRL

UART control

Address: 0x40600040

Retention: Retained

| Bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------|--------------|----|----|----|----|----|--------------|----------|
| SW Access | None | | | | | | | |
| HW Access | None | | | | | | | |
| Name | None [7:0] | | | | | | | |
| Bits | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| SW Access | None | | | | | | | |
| HW Access | None | | | | | | | |
| Name | None [15:8] | | | | | | | |
| Bits | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| SW Access | None | | | | | | | RW |
| HW Access | None | | | | | | | R |
| Name | None [23:17] | | | | | | | LOOPBACK |
| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| SW Access | None | | | | | | RW | |
| HW Access | None | | | | | | R | |
| Name | None [31:26] | | | | | | MODE [25:24] | |

Bits Name Description

25 : 24 MODE Submode of UART operation (3: Reserved)
Default Value: 3

0x0: UART_STD :
Standard UART submode.

0x1: UART_SMARTCARD :
SmartCard (ISO7816) submode. Support for negative acknowledgement (NACK) on the receiver side and retransmission on the transmitter side.

0x2: UART_IRDA :
Infrared Data Association (IrDA) submode. Return to Zero modulation scheme.

16 LOOPBACK Local loopback control (does NOT affect the information on the pins). When '0', the Transmitter TX line "uart_tx_out" is connected to the TX pin and the receiver RX line "uart_rx_in" is connected to the RX pin. When '1', the transmitter TX line "uart_tx_out"

is connected to the receiver RX line "uart_rx_in". A similar connections scheme is followed for "uart_rts_out" and "uart_cts_in".

This allows a SCB UART transmitter to communicate with its receiver counterpart.

Default Value: 0

```
Cy_SCB_UART_PutString(CYBSP_UART_HW, "\x1b[2]\x1b[;H");
Cy_SCB_UART_PutString(CYBSP_UART_HW, "*****\r\n ");
Cy_SCB_UART_PutString(CYBSP_UART_HW, "PSoC6 MCU UART tx and rx \r\n ");
Cy_SCB_UART_PutString(CYBSP_UART_HW, "*****\r\n \n");
Cy_SCB_UART_PutString(CYBSP_UART_HW, ">> start typing \r\n\n ");

/*****
 * Function Name: Cy_SCB_UART_PutString
 *****/
*
* Places a NULL terminated string in the UART TX FIFO.
* This function blocks until the entire string is placed in the TX FIFO.
*
* \param base
* The pointer to the UART SCB instance.
*
* \param string
* The pointer to the null terminated string array.
*
*****/
STATIC_INLINE void Cy_SCB_UART_PutString(CySCB_Type *base, char_t const string[])
{
    CY_ASSERT_L1(CY_SCB_IS_BUFFER_VALID(string, 1UL));

    Cy_SCB_WriteString(base, string);
}

/*****
 * Function Name: Cy_SCB_WriteString
 *****/
*
* Places a NULL terminated string in the transmit FIFO.
* This function blocks until the entire string is placed in the transmit FIFO.
*
* \param base
* The pointer to the SCB instance.
*
* \param string
* The pointer to the null terminated string array.
*
*****/
void Cy_SCB_WriteString(CySCB_Type *base, char_t const string[])
{
    uint32_t idx = 0UL;
    uint32_t fifoSize = Cy_SCB_GetFifoSize(base);

    /* Put data from TX FIFO. Stop when string is terminated */
    while (((char_t) 0) != string[idx])
    {
        /* Wait for free space to be available */
        while (fifoSize == Cy_SCB_GetNumInTxFifo(base))
        {
        }

        Cy_SCB_WriteTxFifo(base, (uint32_t) string[idx]);
        ++idx;
    }
}
```

```

/*****
* Function Name: Cy_SCB_WriteTxFifo
*****/
*
* Writes data directly into the TX FIFO.
* This function does not check whether the TX FIFO is not full before writing
* into it.
*
* \param base
* The pointer to the SCB instance.
*
* \param data
* Data to write to the TX FIFO.
*
*****/
STATIC_FORCEINLINE void Cy_SCB_WriteTxFifo(CySCB_Type* base, uint32_t data)
{
    SCB_TX_FIFO_WR(base) = data;
}

#define SCB_TX_FIFO_WR(base) (((CySCB_V1_Type*) (base))->TX_FIFO_WR)

#ifndef _CYIP_SCB_H_
#define _CYIP_SCB_H_

#include "cyip_headers.h"

/*****
*
* SCB
*****/

#define SCB_SECTION_SIZE 0x00010000UL

/**
 * \brief Serial Communications Block (SPI/UART/I2C) (CySCB)
 */
typedef struct {
    __IOM uint32_t CTRL; /*!< 0x00000000 Generic control */
    __IM uint32_t STATUS; /*!< 0x00000004 Generic status */
    __IOM uint32_t CMD_RESP_CTRL; /*!< 0x00000008 Command/response control */
    __IM uint32_t CMD_RESP_STATUS; /*!< 0x0000000C Command/response status */
    __IM uint32_t RESERVED[4];
    __IOM uint32_t SPI_CTRL; /*!< 0x00000020 SPI control */
    __IM uint32_t SPI_STATUS; /*!< 0x00000024 SPI status */
    __IM uint32_t RESERVED1[6];
    __IOM uint32_t UART_CTRL; /*!< 0x00000040 UART control */
    __IOM uint32_t UART_TX_CTRL; /*!< 0x00000044 UART transmitter control */
    __IOM uint32_t UART_RX_CTRL; /*!< 0x00000048 UART receiver control */
    __IM uint32_t UART_RX_STATUS; /*!< 0x0000004C UART receiver status */
    __IOM uint32_t UART_FLOW_CTRL; /*!< 0x00000050 UART flow control */
    __IM uint32_t RESERVED2[3];
    __IOM uint32_t I2C_CTRL; /*!< 0x00000060 I2C control */
    __IM uint32_t I2C_STATUS; /*!< 0x00000064 I2C status */
    __IOM uint32_t I2C_M_CMD; /*!< 0x00000068 I2C master command */
    __IOM uint32_t I2C_S_CMD; /*!< 0x0000006C I2C slave command */
    __IOM uint32_t I2C_CFG; /*!< 0x00000070 I2C configuration */
    __IM uint32_t RESERVED3[99];
    __IOM uint32_t TX_CTRL; /*!< 0x00000200 Transmitter control */
    __IOM uint32_t TX_FIFO_CTRL; /*!< 0x00000204 Transmitter FIFO control */
    __IM uint32_t TX_FIFO_STATUS; /*!< 0x00000208 Transmitter FIFO status */
    __IM uint32_t RESERVED4[13];
    __IOM uint32_t TX_FIFO_WR; /*!< 0x00000240 Transmitter FIFO write */
    __IM uint32_t RESERVED5[47];
    __IOM uint32_t RX_CTRL; /*!< 0x00000300 Receiver control */
    __IOM uint32_t RX_FIFO_CTRL; /*!< 0x00000304 Receiver FIFO control */
    __IM uint32_t RX_FIFO_STATUS; /*!< 0x00000308 Receiver FIFO status */
    __IM uint32_t RESERVED6;
}

```

```

__IOM uint32_t RX_MATCH; /*!< 0x00000310 Slave address and mask */
__IM uint32_t RESERVED7[11];
__IM uint32_t RX_FIFO_RD; /*!< 0x00000340 Receiver FIFO read */
__IM uint32_t RX_FIFO_RD_SILENT; /*!< 0x00000344 Receiver FIFO read silent */
__IM uint32_t RESERVED8[46];
__IOM uint32_t EZ_DATA[512]; /*!< 0x00000400 Memory buffer */
__IM uint32_t RESERVED9[128];
__IM uint32_t INTR_CAUSE; /*!< 0x00000E00 Active clocked interrupt signal */
__IM uint32_t RESERVED10[31];
__IOM uint32_t INTR_I2C_EC; /*!< 0x00000E80 Externally clocked I2C interrupt request */
*/
__IM uint32_t RESERVED11;
__IOM uint32_t INTR_I2C_EC_MASK; /*!< 0x00000E88 Externally clocked I2C interrupt mask */
__IM uint32_t INTR_I2C_EC_MASKED; /*!< 0x00000E8C Externally clocked I2C interrupt masked */
__IM uint32_t RESERVED12[12];
__IOM uint32_t INTR_SPI_EC; /*!< 0x00000EC0 Externally clocked SPI interrupt request */
*/
__IM uint32_t RESERVED13;
__IOM uint32_t INTR_SPI_EC_MASK; /*!< 0x00000EC8 Externally clocked SPI interrupt mask */
__IM uint32_t INTR_SPI_EC_MASKED; /*!< 0x00000ECC Externally clocked SPI interrupt masked */
__IM uint32_t RESERVED14[12];
__IOM uint32_t INTR_M; /*!< 0x00000F00 Master interrupt request */
__IOM uint32_t INTR_M_SET; /*!< 0x00000F04 Master interrupt set request */
__IOM uint32_t INTR_M_MASK; /*!< 0x00000F08 Master interrupt mask */
__IM uint32_t INTR_M_MASKED; /*!< 0x00000F0C Master interrupt masked request */
__IM uint32_t RESERVED15[12];
__IOM uint32_t INTR_S; /*!< 0x00000F40 Slave interrupt request */
__IOM uint32_t INTR_S_SET; /*!< 0x00000F44 Slave interrupt set request */
__IOM uint32_t INTR_S_MASK; /*!< 0x00000F48 Slave interrupt mask */
__IM uint32_t INTR_S_MASKED; /*!< 0x00000F4C Slave interrupt masked request */
__IM uint32_t RESERVED16[12];
__IOM uint32_t INTR_TX; /*!< 0x00000F80 Transmitter interrupt request */
__IOM uint32_t INTR_TX_SET; /*!< 0x00000F84 Transmitter interrupt set request */
__IOM uint32_t INTR_TX_MASK; /*!< 0x00000F88 Transmitter interrupt mask */
__IM uint32_t INTR_TX_MASKED; /*!< 0x00000F8C Transmitter interrupt masked request */
__IM uint32_t RESERVED17[12];
__IOM uint32_t INTR_RX; /*!< 0x00000FC0 Receiver interrupt request */
__IOM uint32_t INTR_RX_SET; /*!< 0x00000FC4 Receiver interrupt set request */
__IOM uint32_t INTR_RX_MASK; /*!< 0x00000FC8 Receiver interrupt mask */
__IM uint32_t INTR_RX_MASKED; /*!< 0x00000FCC Receiver interrupt masked request */
} CySCB_V1_Type; /*!< Size = 4048 (0xFD0) */

```

```

core_cm4.h main.c cy_scb_uart.h cyip_scb.h cy_scb_common.h cy_device.h cy_scb_common.c
211 #endif
212
213 /* IO definitions (access restrictions to peripheral registers) */
214 /**
215  \defgroup CMSIS_glob_defs CMSIS Global Defines
216
217  <strong>IO Type Qualifiers</strong> are used
218  \li to specify the access to peripheral variables.
219  \li for automatic generation of peripheral register debug information.
220  */
221 #ifndef __cplusplus
222 #define __I volatile /*!< Defines 'read only' permissions */
223 #else
224 #define __I volatile const /*!< Defines 'read only' permissions */
225 #endif
226 #define __O volatile /*!< Defines 'write only' permissions */
227 #define __IO volatile /*!< Defines 'read / write' permissions */
228
229 /* following defines should be used for structure members */
230 #define __IM volatile const /*! Defines 'read only' structure member permissions */
231 #define __OM volatile /*! Defines 'write only' structure member permissions */
232 #define __IOM volatile /*! Defines 'read / write' structure member permissions */
233
234 /*@} end of group Cortex_M4 */
---
```

24.1.17 SCB0_TX_FIFO_WR

Transmitter FIFO write

Address: 0x40600240

Retention: Not Retained

| Bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----------|------------|---|---|---|---|---|---|---|
| SW Access | W | | | | | | | |
| HW Access | R | | | | | | | |
| Name | DATA [7:0] | | | | | | | |

| Bits | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
|-----------|-------------|----|----|----|----|----|---|---|
| SW Access | W | | | | | | | |
| HW Access | R | | | | | | | |
| Name | DATA [15:8] | | | | | | | |

| Bits | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|-----------|--------------|----|----|----|----|----|----|----|
| SW Access | None | | | | | | | |
| HW Access | None | | | | | | | |
| Name | None [23:16] | | | | | | | |

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
|-----------|--------------|----|----|----|----|----|----|----|
| SW Access | None | | | | | | | |
| HW Access | None | | | | | | | |
| Name | None [31:24] | | | | | | | |

| Bits | Name | Description |
|--------|------|--|
| 15 : 0 | DATA | Data frame written into the transmitter FIFO. Behavior is similar to that of a PUSH operation. Note that when CTRL.BYTE_MODE is '1', only DATA[7:0] are used. A write to a full TX FIFO sets INTR_TX.OVERFLOW to '1'. Default Value: 0 |

Explanation: -

The SCB_TX_FIFO_WR macro is used to write data into the Transmitter FIFO of the SCB (Serial Communication Block) in Cypress's PSoC or similar microcontroller architecture. Let's break down what this means and how it relates to the provided screenshot.

Understanding the SCB_TX_FIFO_WR Macro

The macro definition:

```
#define SCB_TX_FIFO_WR(base) (((CySCB_V1_Type*) (base))->TX_FIFO_WR)
```

- **base:** This is the base address of the SCB instance you are working with.
- **CySCB_V1_Type:** This is a structure that represents the SCB hardware registers. By casting the base address to this structure type, the macro accesses the TX_FIFO_WR register.
- **TX_FIFO_WR:** This is the register used to write data into the Transmitter FIFO. The FIFO (First In, First Out) buffer is used to queue data for transmission.

Register Details (SCB_TX_FIFO_WR)

The screenshot provides details of the SCB_TX_FIFO_WR register:

- **Bits [15:0] (DATA):** This field is used to write a data frame into the transmitter FIFO. If CTRL.BYTE_MODE is set to 1, only the lower 8 bits (DATA[7:0]) are used.
- **Writing to the Register:**
 - When you write to this register, the data is pushed into the FIFO queue for transmission.
 - If the FIFO is full and you try to write more data, the INTR_TX_OVERFLOW interrupt flag will be set to 1.

Key Points:

- **FIFO (First In, First Out):** The data is sent out in the order it was written to the FIFO.
- **Overflow Handling:** Ensure that the FIFO is not full before writing, or handle the INTR_TX_OVERFLOW interrupt if overflow occurs.
- **Byte Mode:** When BYTE_MODE is enabled, only 8 bits are used from the DATA field.

By using the SCB_TX_FIFO_WR macro, you can efficiently write data to the UART transmitter FIFO, leveraging the register's capabilities as described in the architecture manual.

Cy_SCB_UART_GetNumInRxFifo(CYBSP_UART_HW)

```
/* *****
 * Function Name: Cy_SCB_UART_GetNumInRxFifo
 * *****//**
 *
 * Returns the number of data elements in the UART RX FIFO.
 *
 * \param base
 * The pointer to the UART SCB instance.
 *
 * \return
 * The number of data elements in the RX FIFO.
 * The size of data element defined by the configured data width.
 *
 * *****/
STATIC_INLINE uint32_t Cy_SCB_UART_GetNumInRxFifo(CySCB_Type const *base)
{
    return Cy_SCB_GetNumInRxFifo(base);
}
```

#include "cy_scb_common.h"

```
/* *****
 * Function Name: Cy_SCB_GetNumInRxFifo
 * *****//**
 *
 * Returns the number of data elements currently in the RX FIFO.
 *
 * \param base
 * The pointer to the SCB instance.
 *
 * \return
 * The number of data elements in RX FIFO.
 *
 * *****/
STATIC_INLINE uint32_t Cy_SCB_GetNumInRxFifo(CySCB_Type const *base)
{
    return _FLD2VAL(SCB_RX_FIFO_STATUS_USED, SCB_RX_FIFO_STATUS(base));
}
```

In cy_device.h

```
#define SCB_RX_FIFO_STATUS(base) (((CySCB_V1_Type*) (base))->RX_FIFO_STATUS)
```

```
((uint32_t) ( ((CySCB_V1_Type*) (base))->RX_FIFO_STATUS)) & 0x1FFUL >> 0UL)
```

->

```
/**
 * \brief Mask and shift a register value to extract a bit field value.
 * \param[in] field Name of the register bit field.
 * \param[in] value Value of register. This parameter is interpreted as an uint32_t type.
 * \return Masked and shifted bit field value.
 */
#define _FLD2VAL(field, value) (((uint32_t)(value) & field ## _Msk) >> field ## _Pos)
```

In cyip_scb.h and inside that its within CySCB_V1_Type structure

```
_IM uint32_t RX_FIFO_STATUS; /*!< 0x0000308 Receiver FIFO status */
```

```
Cy_SCB_UART_Get(CYBSP_UART_HW);
```

```
/* *****
 * Function Name: Cy_SCB_UART_Get
 * *****//**
 *
 * Reads a single data element from the UART RX FIFO.
 * This function does not check whether the RX FIFO has data before reading it.
 * If the RX FIFO is empty, the function returns \ref CY_SCB_UART_RX_NO_DATA.
 *
 * \param base
 * The pointer to the UART SCB instance.
 *
 * \return
 * Data from the RX FIFO.
 * The data element size is defined by the configured data width.
 *
 * *****/
STATIC_INLINE uint32_t Cy_SCB_UART_Get(CySCB_Type const *base)
{
    return Cy_SCB_ReadRxFifo(base);
}

/* *****
 * Function Name: Cy_SCB_ReadRxFifo
 * *****//**
 *
 * Reads a data element directly out of the RX FIFO.
 * This function does not check whether the RX FIFO has data before reading it.
 *
 * \param base
 * The pointer to the SCB instance.
 *
 * \return
 * Data from RX FIFO.
 *
 * *****/
STATIC_FORCEINLINE uint32_t Cy_SCB_ReadRxFifo(CySCB_Type const *base)
{
    return (SCB_RX_FIFO_RD(base));
}

#define SCB_RX_FIFO_RD(base) (((CySCB_V1_Type*) (base))->RX_FIFO_RD)
```

```
Cy_SCB_UART_Put(CYBSP_UART_HW, read_data)
```

```
/* *****
 * Function Name: Cy_SCB_UART_Put
 * *****//**
 *
 * Places a single data element in the UART TX FIFO.
 * This function does not block and returns how many data elements were placed
 * in the TX FIFO.
 *
 * \param base
 * The pointer to the UART SCB instance.
 *
 * \param data
 * Data to put in the TX FIFO.
 * The element size is defined by the data type, which depends on the configured
 * data width.
 *
 * \return
 * The number of data elements placed in the TX FIFO: 0 or 1.
```

```

*
*****
STATIC_INLINE uint32_t Cy_SCB_UART_Put(CySCB_Type *base, uint32_t data)
{
    return Cy_SCB_Write(base, data);
}

```

```

/*****
* Function Name: Cy_SCB_Write
*****//**
*
* Places a single data element in the SCB transmit FIFO.
* This function does not block. It returns how many data elements are placed
* in the transmit FIFO.
*
* \param base
* The pointer to the SCB instance.
*
* \param data
* Data to put in the transmit FIFO.
* The item size is defined by the data type, which depends on the configured
* data width.
*
* \return
* The number of data elements placed in the transmit FIFO: 0 or 1.
*
*****/

```

```

uint32_t Cy_SCB_Write(CySCB_Type *base, uint32_t data)
{
    uint32_t numCopied = 0UL;

    if (Cy_SCB_GetFifoSize(base) != Cy_SCB_GetNumInTxFifo(base))
    {
        Cy_SCB_WriteTxFifo(base, data);

        numCopied = 1UL;
    }

    return (numCopied);
}

```

```

/* Transmit header to the terminal */
/* \x1b[2J\x1b[H - ANSI ESC sequence for clear screen */
Cy_SCB_UART_PutString(CYBSP_UART_HW, "\x1b[2J\x1b[H");
Cy_SCB_UART_PutString(CYBSP_UART_HW, "*****\r\n ");
Cy_SCB_UART_PutString(CYBSP_UART_HW, "PSoC6 MCU UART tx and rx \r\n ");
Cy_SCB_UART_PutString(CYBSP_UART_HW, "*****\r\n \n");
Cy_SCB_UART_PutString(CYBSP_UART_HW, ">> start typing \r\n\r\n ");

for (;;)
{
    if (0UL != Cy_SCB_UART_GetNumInRxFifo( CYBSP_UART_HW ))
    {
        read_data= Cy_SCB_UART_Get(CYBSP_UART_HW);
        while( 0UL == Cy_SCB_UART_Put(CYBSP_UART_HW, read_data))
        {
        }
    }
}

```