AN99218

# Multifunction Serial Interface of FM MCU

**Author: Edison Zhang**
**Associated Part Family: FM0+, FM3, FM4**
**Associated Code Examples: None**
**Related Application Notes: None**

AN99218 explains the various modes of the multifunction serial (MFS) interface. The information presented is intended to help you get the MFS module up and running quickly for any FM MCU.

## Contents

1 Introduction....1
2 UART....2
2.1 Features....2
2.2 Data Format....3
2.3 Interrupt Factor and Timing....4
2.4 Low-Level API....7
2.5 Example Code....8
3 CSIO (SPI)....10
3.1 Features....10
3.2 Transfer Mode....11
3.3 Serial Timer....11
3.4 Chip Selection Function....12
3.5 Interrupt Factor and Timing....13
3.6 Low-Level API....16
3.7 Example Code....17
4 $I^2C$....19
4.1 Features....19
4.2 Protocol....20
4.3 Data Transfer Timing....23
4.4 Low-Level API....24
4.5 Example Code....25
5 LIN....29
5.1 Features....29
5.2 Data Format....30
5.3 Communication System....31
5.4 Operation Timing....31
5.5 Low-Level API....33
5.6 Example Code....34
6 Summary....36
Document History....37
Worldwide Sales and Design Support....38

## 1 Introduction

The serial communication interface (SCI) is the most common communication interface. It includes the universal asynchronous receiver transmitter (UART), Serial Peripheral Interface (SPI), inter-integrated circuit ($I^2C$), and local interconnect network (LIN). Most microcontroller manufacturers offer independent built-in peripherals for these interfaces. The FM family microcontroller includes a built-in MFS interface, which can be configured to UART, Clock Synchronous Serial Interface—CSIO (SPI), $I^2C$, and LIN with user settings, providing flexibility and convenience in the application.

This application note introduces the MFS modes and shows how to use MFS with the Peripheral Driver Library (PDL). It also shows the data format of each communication protocol and then explains interrupt timing. This basic information can help you gain a better understanding of how each MFS mode works. Finally, some examples that use the PDL are given to show how to implement the data transmission and reception of each mode.

# 2 UART

The UART is a general-purpose serial data communications interface for asynchronous communications (start/stop synchronization) with external devices.

When the MD bits' SMR register is set to b'000, the UART mode is configured.

| bit7 | bit6 | bit5 | Description |
|---|---|---|---|
| 0 | 0 | 0 | Operation mode 0 (asynchronous normal mode) |
| 0 | 0 | 1 | Operation mode 1 (asynchronous multiprocessor mode) |
| 0 | 1 | 0 | Operation mode 2 (clock sync mode) |
| 0 | 1 | 1 | Operation mode 3 (LIN communication mode) |
| 1 | 0 | 0 | Operation mode 4 ($I^2C$ mode) |
| Other than the above | | | Setting is prohibited. |

## 2.1 Features

- Full-duplex operation
- 15-bit baud rate selection[1]
- 5- to 9-bit data length selection
- Support for non-return-to-zero (NRZ) and inverted NRZ data format
- Support for MSB/LSB transfer direction
- Support for hardware flow control[2]
- Received error detection
  - Framing error
  - Overrun error
  - Parity error
- Interrupt requests
  - Received interrupt request by factor of reception completion, framing error, overrun error, or parity error
  - Transmit interrupt request by factor of transmit data empty, transmit bus idle
  - Transmit FIFO interrupt request by factor of transmit FIFO empty
- Support for DMA transferring
- Transmit/receive FIFO installed[3]

[1] The baud rate generator can also be sourced by an external clock.

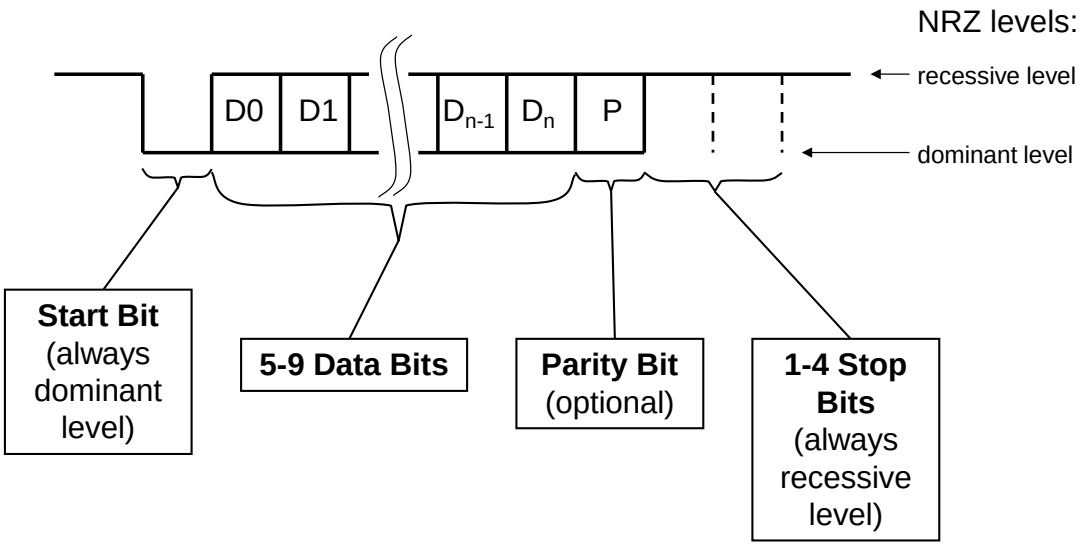
[2] Only some MFS channels have a hardware flow control function; see the datasheet of the product used.

[3] The FIFO capacity varies depending on the product type; see the datasheet of the product used.

## 2.2 Data Format

The UART frame starts with a start bit, followed by data bits, and ends with a stop bit, as shown in Figure 1. The data length can be selected from 5 bits to 9 bits by setting the L2, L1, L0 bits of the ESCR register. Parity Check is optional, depending on the setting of the PEN bit. If Parity Check is enabled, Even-Number Parity check or Odd-Number Selection parity can be selected by the P bit of the ESCR register. The Stop bit length can be selected by the SBL bit of the SMR register and the ESBL bit of the ESCR register.

Figure 1. UART Data Format




The ESCR register configures most data format settings. Figure 2 describes the bits of the ESCR register.

Figure 2. ESCR Register Description

| bit | 15 ... 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Field | (SSR) | FLWEN | ESBL | INV | PEN | P | L2 | L1 | L0 |
| Attribute | | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Initial value | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**[bit2:0] L2, L1, L0: Data length select bit**

| bit2 | bit1 | bit0 | Description |
|---|---|---|---|
| 0 | 0 | 0 | 8-bit length |
| 0 | 0 | 1 | 5-bit length |
| 0 | 1 | 0 | 6-bit length |
| 0 | 1 | 1 | 7-bit length |
| 1 | 0 | 0 | 9-bit length |

**[bit6] ESBL: Extension stop bit length**

| Bit | Description | |
|---|---|---|
| 0 | SMR:SBL=0 | 1 bit |
| | SMR:SBL=1 | 2 bits |
| 1 | SMR:SBL=0 | 3 bits |
| | SMR:SBL=1 | 4 bits |

**[bit3] P: Parity select bit**

| Bit | Description |
|---|---|
| 0 | Even-number parity |
| 1 | Odd-number parity |

**[bit4] PEN: Parity enable bit**

| Bit | Description |
|---|---|
| 0 | Disables parity. |
| 1 | Enables parity. |

## 2.3 Interrupt Factor and Timing

The timing diagrams in this section refer to the following bits and registers:

- TDRE (Transmit Data Register Empty) bit indicates whether the data in the Transmit Data Register is empty or not.
- TBI (Transmit Bus Idle) bit indicates whether the SOT line is idle or not.
- TIE (Transmit Interrupt Enable) bit is used to enable or disable transmit interrupt.
- TBIE (Transmit Bus Interrupt Enable) bit is used to enable or disable transmit interrupt.
- TDR (Transmit Data Register) is a buffer register for serial data transmission.
- RDRF (Received Data Register Full) bit is used to indicate whether the Received Data Register is full or not.
- FRE (Frame Error) bit is used to indicate a frame error, which occurs when the stop bit of the received data is 0.
- ORE (Overrun Error) bit is used to indicate an overrun error, which occurs when the next data is received before the received data is read.
- REC (Received Error Clear) bit is used to clear received errors.
- RIE (Receive Interrupt Enable) bit is used to enable or disable transmit interrupt.
- RDR (Received Data Register) is a buffer register for serial data reception.

Figure 3 is the UART data transmit timing diagram when FIFO is not used.

- When the TDR is empty (TDRE = 1), if Transmit Interrupt is enabled (TIE = 1), a Transmit Interrupt Request will be issued. Then transfer data can be written into the TDR, and the TDRE bit turns to 0.
- The TDR will be loaded into the transmit shift register first, and then data will be shifted out bit by bit to the SOT pin. After the first bit of transfer data shifts out to the SOT pin, the TDRE bit turns to 1. If TIE = 1, a Transmit Interrupt Request will be issued. Then the following data can be written into the TDR again.
- After all bits of the first data are shifted out to the SOT pin and the first bit of second data is shifted out to the SOT pin, the TDRE bit turns to 1 to indicate that the transmit shift register is empty.
- After all bits of the second data are shifted out to the SOT pin, a 2-byte transmission is completed, and the TBI bit turns to 1. If Transmit Bus Interrupt is enabled (TBIE = 1), a Transmit Bus Interrupt Request will be issued.

Figure 3. UART Transmission Timing Diagram

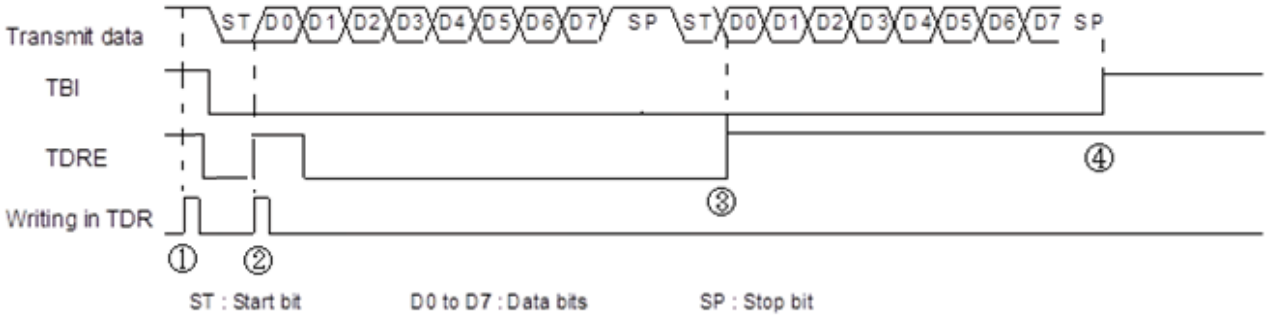


Figure 4 is the UART data receive timing diagram when FIFO is not used. When received data is stored in the RDR, the RDRF bit will be set to 1, and if Receive Interrupt is enabled (RIE = 1), a Receive Interrupt Request will be issued. After the data is read from the RDR, the RDRF bit will be cleared automatically. However, when the RDRF bit is set, if the frame error occurs (FRE = 1) or the overrun error occurs (ORE = 1), the received data is invalid and errors should be cleared by setting the REC bit to 1.

Figure 4. UART Receive Timing Diagram

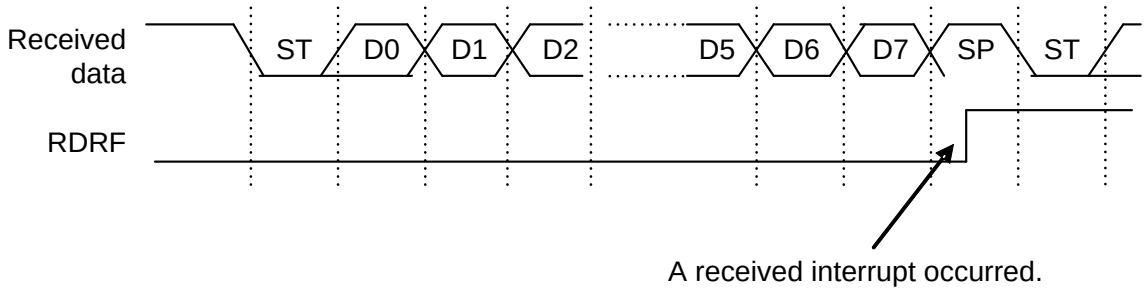



Figure 5 is the data transmit timing diagram when FIFO is enabled. It intends to transmit 5 bytes of data using transmit FIFO.

1. When the FIFO Transmit Data Request bit is set to 1 (FDRQ = 1), if FIFO transmit interrupts are enabled (FTIE = 1), a transmit interrupt occurs.
2. Three bytes are written into the transmit FIFO, as the TDR is empty, so the first byte is transferred to the TDR. After that, the FDRQ bit should be cleared to 0 manually. At this time, the data count in the FIFO is 2, as displayed by the FBYTE register.
3. After the FIFO is empty and the first bit of data is shifted out from the shift register, the FDRQ bit turns to 1, which indicates the FIFO is empty. If FIFO transmit interrupts are enabled (FTIE = 1), a transmit interrupt occurs.
4. Two bytes are written into the transmit FIFO. After that, the FDRQ bit should be cleared to 0 manually. At this time, the data count in the FIFO is 2 again, as displayed by the FBYTE register.
5. After the FIFO is empty and the first bit of data is shifted out from the shift register, the FDRQ bit turns to 1, which indicates the FIFO is empty. If FIFO transmit interrupts are enabled (FTIE = 1), a transmit interrupt occurs.
6. After the TDR is empty and the first bit of data in the shift register is shifted out, the TDRE bit turns to 1.

   When the data bits in the shift register are all shifted out, the TBI bit is set to 1, which indicates all data transmission is completed.

Figure 5. UART Transmission Timing Diagram with FIFO Enabled

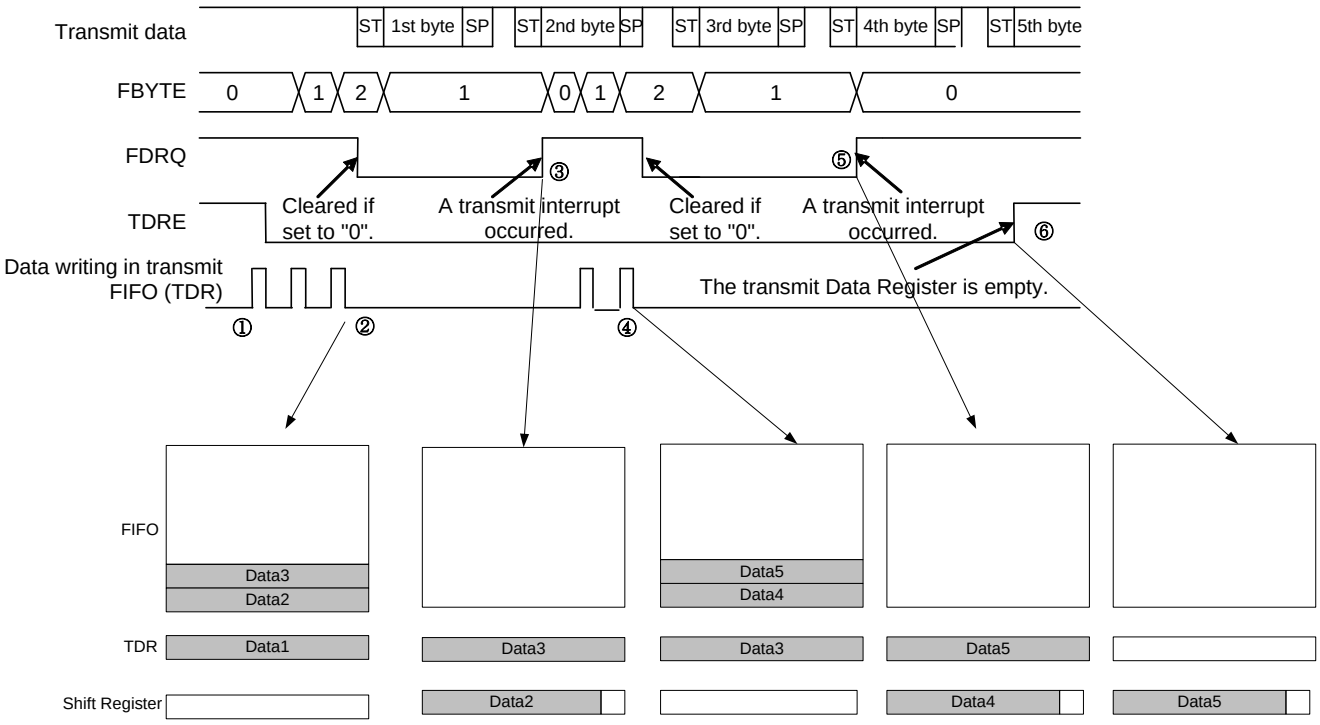
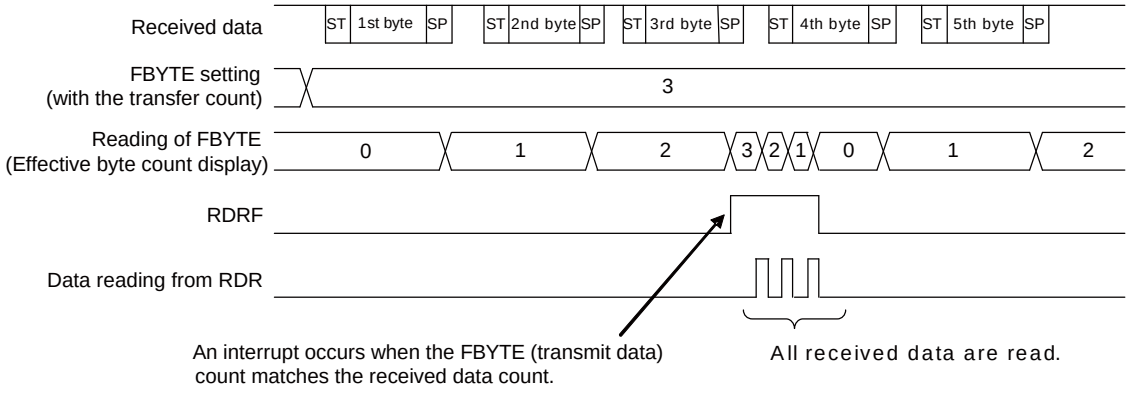



Figure 6 is the data receive timing diagram when FIFO is enabled.

1. The FIFO match count should be set in the FBYTE.
2. After data receive starts, the received data will be stored in the FIFO in sequence. When the data count in the FIFO matches FBYTE, the RDRF bit will be set to 1. If the RIE bit is set to 1, a receive interrupt will occur.
3. When only 1 byte is received with no following data, if Received FIFO idle detection is enabled (FRIIE = 1), and if the received idle state continues for more than 8 baud rate clocks, the RDRF bit will be set to 1.
4. When the data in the FIFO is all read out, then the RDRF bit will be cleared to 0 automatically.

   If the received data count exceeds the maximum capacity of the receive FIFO, an overrun error will occur.

Figure 6. UART Receive Timing Diagram with FIFO Enabled



## 2.4 Low-Level API

Following is the UART driver API of the PDL, which is placed in the `mfs.c/h` files.

- Mfs_Uart_Init()
- Mfs_Uart_DeInit()
- Mfs_Uart_EnableIrq()
- Mfs_Uart_DisableIrq()
- Mfs_Uart_SetBaudRate()
- Mfs_Uart_EnableFunc()
- Mfs_Uart_DisableFunc()
- Mfs_Uart_GetStatus()
- Mfs_Uart_ClrStatus()
- Mfs_Uart_SendData()
- Mfs_Uart_ReceiveData()
- Mfs_Uart_ResetFifo()
- Mfs_Uart_SetBaudRate()
- Mfs_Uart_SetFifoCount()
- Mfs_Uart_GetFifoCount()

`Mfs_Uart_Init()` is used to initialize an MFS instance to UART mode with parameter `Mfs_Uart_Init #pstcConfig` of type `#stc_mfs_uart_config_t`. This function sets only the basic UART configuration. `Mfs_Uart_DeInit()` is used to reset all MFS UART-related registers.

`Mfs_Uart_EnableIrq()` enables UART interrupt sources selected by enumeration type `#en_uart_irq_sel_t`. `Mfs_Uart_DisableIrq()` disables the UART interrupt sources selected by enumeration type `#en_uart_irq_sel_t`.

`Mfs_Uart_SetBaudRate()` provides a possibility to change the UART baud rate after the UART is initialized.

`Mfs_Uart_EnableFunc()` enables the UART function selected by the parameter `Mfs_Uart_EnableFunc#enFunc`, and `Mfs_Uart_DisableFunc()` disables the UART function.

`Mfs_Uart_GetStatus()` gets the status selected by `Mfs_Uart_GetStatus#enStatus`, and `Mfs_Uart_ClrStatus()` clears the UART status selected; some statuses can only be cleared by hardware automatically.

`Mfs_Uart_SendData()` writes data into the UART transfer buffer, and `Mfs_Uart_ReceiveData()` reads data from the UART receive buffer.

`Mfs_Uart_ResetFifo()` resets the UART hardware FIFO.

`Mfs_Uart_SetFifoCount()` provides a possibility to change the FIFO size after the UART is initialized.

`Mfs_Uart_GetFifoCount()` gets the current data count in the FIFO.

## 2.5 Example Code

Based on the low-level driver API, an example is given here to show how to transfer data via the UART using the interrupt flag polling method. It uses UART ch.0 to transfer 10 bytes and then receive 10 bytes.

Before using UART, configure the pin function of the UART as follows:

```
/* Initialize UART function I/O */
SetPinFunc_SIN0_0();
SetPinFunc_SOT0_0();
```

Then configure the UART configuration structure and initialize the UART channel.

- Baud rate: 115200
- Data length: 8 bits
- Parity check: No
- Stop bit length: 1 bit
- H/W flow control: No

```
stc_mfs_uart_config_t stcUartConfig;

/* Initialize UART TX and RX channel  */
stcUartConfig.enMode = UartNormal;
stcUartConfig.u32BaudRate = 115200;
stcUartConfig.enDataLength = UartEightBits;
stcUartConfig.enParity = UartParityNone;
stcUartConfig.enStopBit = UartOneStopBit;
stcUartConfig.enBitDirection = UartDataLsbFirst;
stcUartConfig.bInvertData = FALSE;
stcUartConfig.bHwFlow = FALSE;
stcUartConfig.bUseExtClk = FALSE;
stcUartConfig.pstcFifoConfig = NULL;

if (Ok != Mfs_Uart_Init(&UART0, &stcUartConfig))
{
    while(1); // Initialization error
}
```

The following code will send 10 bytes via SOT0_0.

```
uint8_t u8Cnt = 0;
uint8_t au8UartTxBuf[10] = {0x01,0x23,0x45,0x67,0x89,0xAB,0xCD,0xEF};

/* Enable TX function of UART0   */
Mfs_Uart_EnableFunc(&UART0, UartTx);

while(u8Cnt < 10)
{
    /* wait until TX buffer empty */
    while (TRUE != Mfs_Uart_GetStatus((&UART0, UartTxEmpty));
    /* Write data to transmit data register */
    Mfs_Uart_SendData(UartCh0, au8UartTxBuf[u8Cnt]);

    u8Cnt++;
}

/* wait until TX idle */
while (TRUE != Mfs_Uart_GetStatus((&UART0, UartTxIdle));


/* Disable TX function of UART0   */
Mfs_Uart_DisableFunc(&UART0, UartTx);
```

The following code will receive 10 bytes from SIN0_0.

```
uint8_t u8Cnt = 0;
uint8_t au8UartRxBuf[10] = {0};

/* Enable RX function of UART0   */
Mfs_Uart_EnableFunc(&UART0, UartRx);

while(u8Cnt < 10)
{
    /* wait until RX buffer full */
    while(TRUE != Mfs_Uart_GetStatus(&UART0, UartRxFull));
    /* Read data from receive data register */
    au8UartRxBuf[u8Cnt] = Mfs_Uart_ReceiveData(&UART0);
    u8Cnt++;
}

/* Disable TX function of UART0   */
Mfs_Uart_DisableFunc(&UART0, UartRx);
```

**Note:** In the PDL project, make sure that the definition of “PDL_PERIPHERAL_ENABLE_MFSx” is enabled before using an MFS channel.

# 3 CSIO (SPI)

CSIO is a general-purpose serial data communication interface (supporting the SPI) to allow synchronous communication with an external device. It also has the transmit/receive FIFO installed.

When the MD bits' SMR register is set to b'010, the CSIO mode is configured.

| bit7 | bit6 | bit5 | Description |
|---|---|---|---|
| 0 | 0 | 0 | Operation mode 0 (asynchronous normal mode) |
| 0 | 0 | 1 | Operation mode 1 (asynchronous multiprocessor mode) |
| 0 | 1 | 0 | Operation mode 2 (clock sync mode) |
| 0 | 1 | 1 | Operation mode 3 (LIN communication mode) |
| 1 | 0 | 0 | Operation mode 4 ($I^2C$ mode) |
| Other than the above | | | Setting is prohibited. |

## 3.1 Features

- Full-duplex operation
- Clock synchronization (without start/stop bit)
- Master/slave function
- SPI supported (for both master and slave modes)
- 15-bit baud rate selection[1]
- 5- to 16-bit data length selection
- Support for MSB/LSB transfer direction
- Support for chip selection function[2]
  - 4-channel control (single control, round-robin control)
  - Setup/hold/deselect time can be set to be changeable
  - Active level can be set for each channel
- Support for serial data transfer triggered by a serial timer[2]
- Received error detection
  - Overrun error
- Interrupt requests
  - Received interrupt request by factor of reception completion, overrun error
  - Transmit interrupt request by factor of transmit data empty, transmit bus idle
  - Transmit FIFO interrupt request by factor of transmit FIFO empty
  - Status interrupt request by factor of value of the Serial Timer Register (STMR) and the Serial Timer
  - Serial Timer Comparison Register (STMCR) match
- Transmit/receive FIFO installed[3]

[1] The baud rate generator can also be sourced by an external clock.

[2] Only some FM products have the chip selection and serial timer function; see the datasheet of the product used.

[3] The FIFO capacity varies depending on the product type; see the datasheet of the product used.

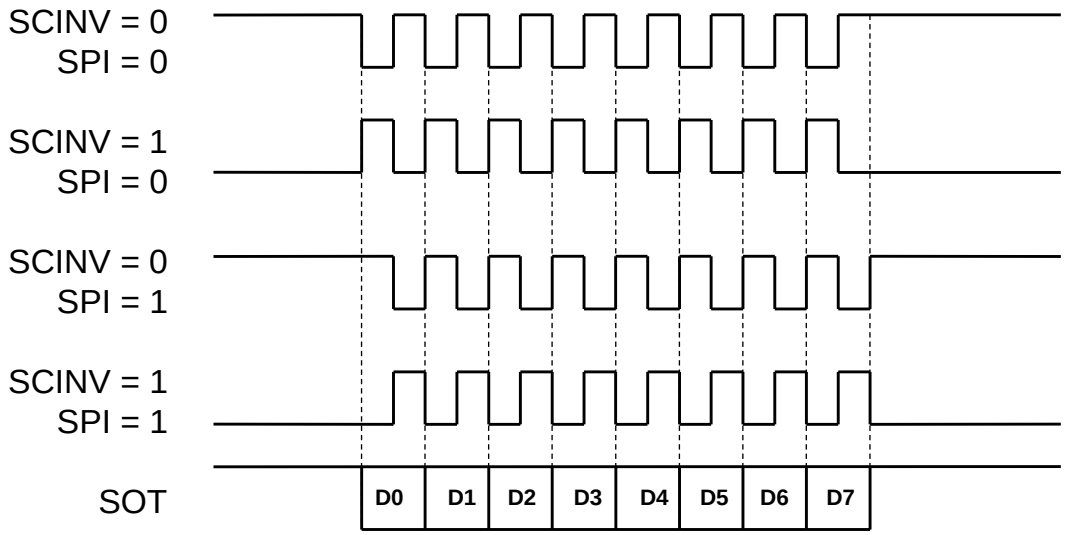
## 3.2 Transfer Mode

Four transfer modes are available for CSIO communication, which can be configured as SCINV and SPI bits. These modes cover all synchronization communication timing in the application.

| Item | Mode 0 (SCINV = 0, SPI = 0) | Mode 1 (SCINV = 1, SPI = 0) | Mode 2 (SCINV = 0, SPI = 1) | Mode 3 (SCINV = 1, SPI = 1) |
|---|---|---|---|---|
| Serial clock (SCK) signal mark level | "HIGH" | "LOW" | "HIGH" | "LOW" |
| Transmit data output timing | SCK signal falling edge | SCK signal rising edge | SCK signal rising edge | SCK signal falling edge |
| Received data sampling | SCK signal rising edge | SCK signal falling edge | SCK signal falling edge | SCK signal rising edge |
| Data length | 5 to 16 bits | 5 to 16 bits | 5 to 16 bits | 5 to 16 bits |

Modes 0 and 1 are called “CSIO mode,” and modes 2 and 3 are called “SPI mode.” Figure 7 shows the timing diagram of these transfer modes.

Figure 7. CSIO and SPI Transfer Modes



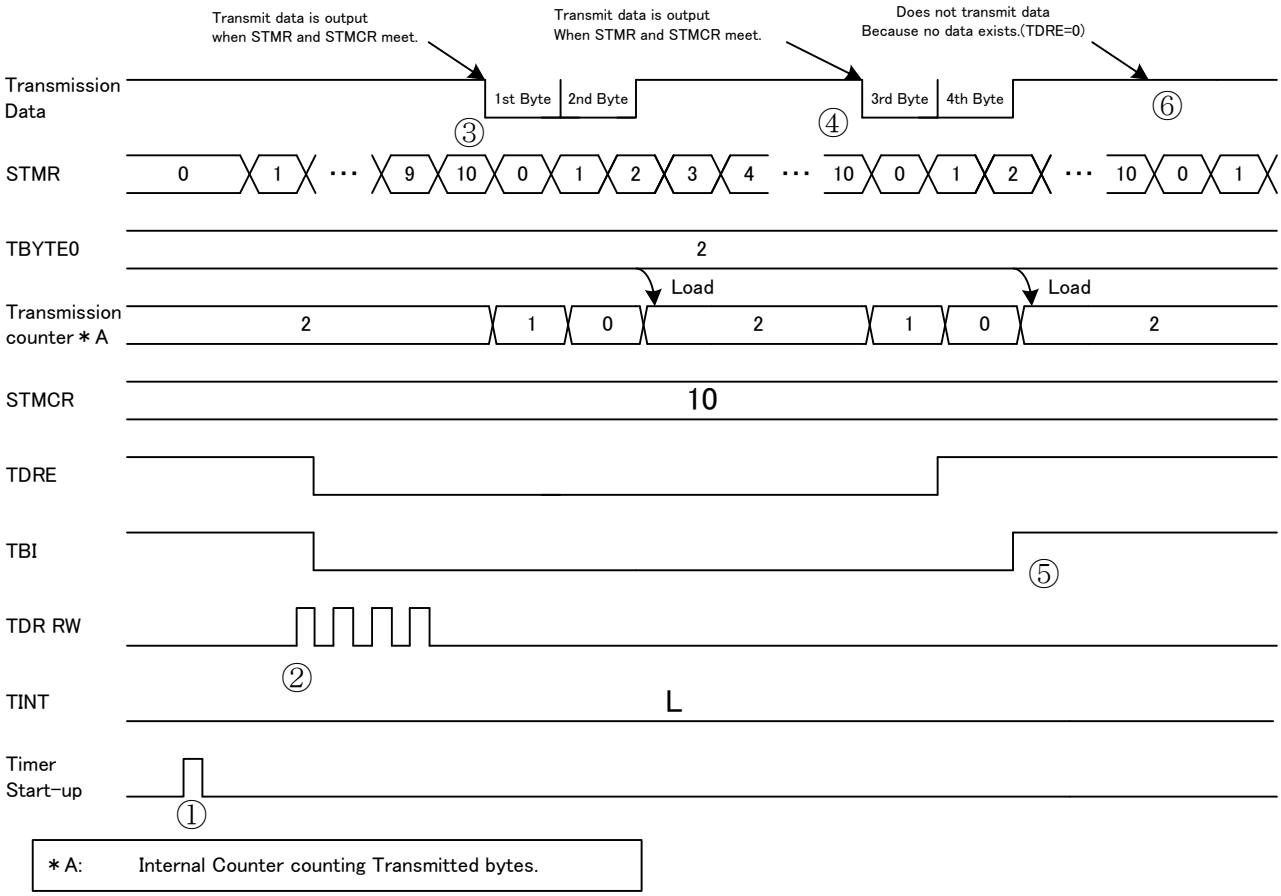

## 3.3 Serial Timer

The CSIO module integrates a serial timer that can trigger the CSIO data transfer according to a certain interval. Figure 8 shows an example of using a serial timer to trigger the CSIO data transfer.

1. Before using the serial timer, set the count comparison value in the STMCR and transfer byte count in the FBYTE0 register. Then start the serial timer.
2. Four bytes of data are written into the transmit FIFO, but data is not transferred immediately.
3. The timer counts from 0 according to the timer clock. When the current timer count (reflected by the STMR) matches the value of the STMCR, 2 bytes are transferred (because TBYTE = 2), and the count of the timer resets to 0.
4. The timer counts from 0 according to the timer clock. Again, when the current timer count (reflected by the STMR) matches the value of the STMCR, 2 bytes are transferred (because TBYTE = 2), and the count of the timer resets to 0.
5. After all four data transmissions are completed, the TBI bit is set to 1.

6. When the current timer count (reflected by the STMR) matches the value of the STMCR again, no data will be transferred because there is no data in the TDR or transmit FIFO.

Figure 8. Serial Timer Operation Timing




**Notes:**

1. Only some FM products have the serial timer function; see the datasheet of the product used.
2. For a product with the CSIO serial timer, the CSIO serial timer can work only in CSIO master mode.

## 3.4 Chip Selection Function

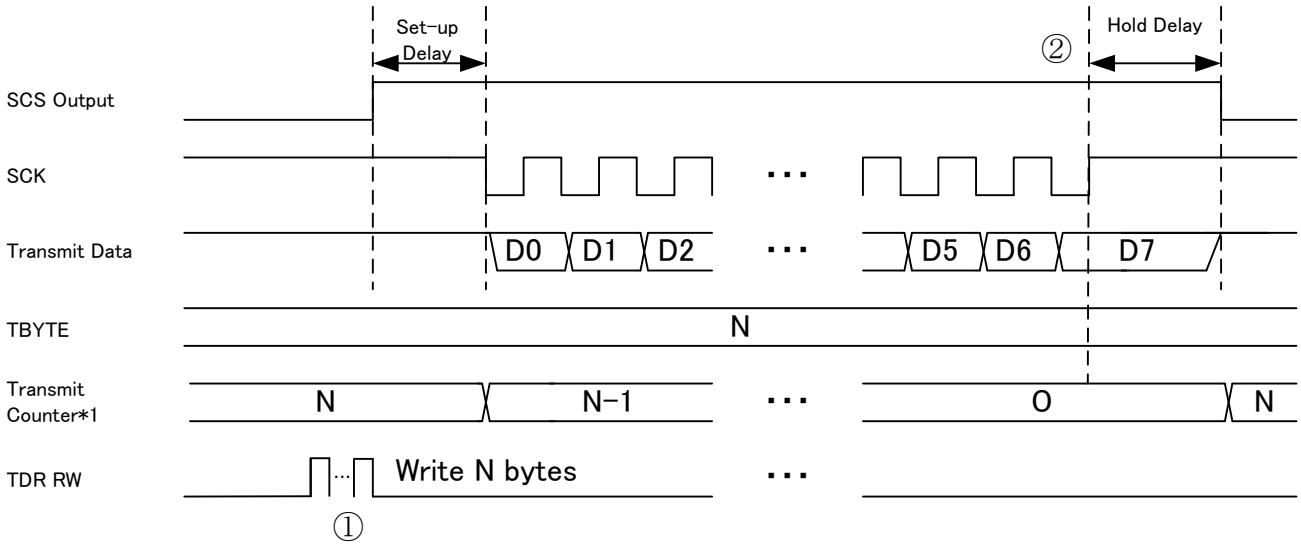
The CSIO module has the chip selection pin (SCS pin) to control whether the data transfer is available or not. Both master mode and slave mode support the chip selection function, but only the SCS0 pin can be used as a chip selection pin in slave mode.

Figure 9 is the timing diagram to transfer N bytes using the chip selection pin in master transfer mode 0 (MS = 0, SPI = 0, SCINV = 0).

1. Before starting data transmission, the data transfer count should be specified in TBYTE. Then select the chip selection pin by setting the SST and SED bits in the Serial Chip Select Control Status Register (SCSCR). Enable the chip selection function by setting the corresponding CSEN bit to 1, and enable data transmission by setting the TEX bit to 1. Finally, N bytes can be written into the transmit FIFO, and data starts transmitting after the elapse of the Set-Up Delay time.
2. When N bytes of transmission are completed, SCK turns to high and SCS turns to high after a Hold Delay time.

   The inactive level of the SCS pin can be set by the CSLVL bit in the SCSCR, but only SCS0 has this function. The Set-Up Delay and Hold Delay time can be set by the Serial Chip Select Timing Register (SCSTR)

Figure 9. CSIO Data Transfer Using Chip Selection Function




*1:Internal Counter counting transmit bytes.

**Note:** Only some FM products have the chip selection function; see the datasheet of the product used.

## 3.5 Interrupt Factor and Timing

The timing diagrams in this section refer to the following bits and registers:

- TDRE (Transmit Data Register Empty) bit indicates whether the data in the Transmit Data Register is empty or not.
- TBI (Transmit Bus Idle) bit indicates whether the SOT line is idle or not.
- TIE (Transmit Interrupt Enable) bit is used to enable or disable transmit interrupt.
- TBIE (Transmit Bus Interrupt Enable) bit is used to enable or disable transmit interrupt.
- TDR (Transmit Data Register) is a buffer register for serial data transmission.
- RDRF (Received Data Register Full) bit is used to indicate whether the Received Data Register is full or not.
- ORE (Overrun Error) bit is used to indicate an overrun error, which occurs when the next data is received before the received data is read.
- REC (Received Error Clear) bit is used to clear received errors.
- RIE (Receive Interrupt Enable) bit is used to enable or disable transmit interrupt.
- RDR (Received Data Register) is a buffer register for serial data reception.

Figure 10 is the CSIO data transmit timing diagram when the FIFO is not used.

1. When the TDR is empty (TDRE = 1), if a Transmit Interrupt is enabled (TIE = 1), a Transmit Interrupt Request will be issued. Then transfer data can be written into the TDR, and the TDRE bit turns to 0.
2. The TDR will be loaded into the transmit shift register first, and then data will be shifted out bit by bit to the SOT pin. After the data in the TDR loads into the transmit shift register, the TDRE bit turns to 1. If TIE = 1, a Transmit Interrupt Request will be issued. Then the following data can be written into the TDR again.
3. All bits of the first data are shifted out to the SOT pin, and the next data loads into the transmit shift register from the TDR again. Then the TDRE bit turns to 1 to indicate that the transmit shift register is empty.

4. After all bits of the second data are shifted out to the SOT pin, 2 bytes of transmission are completed, and the TBI bit turns to 1. If Transmit Bus Interrupt is enabled (TBIE = 1), a Transmit Bus Interrupt Request will be issued.

Figure 10. CSIO Data Transmission Timing Diagram

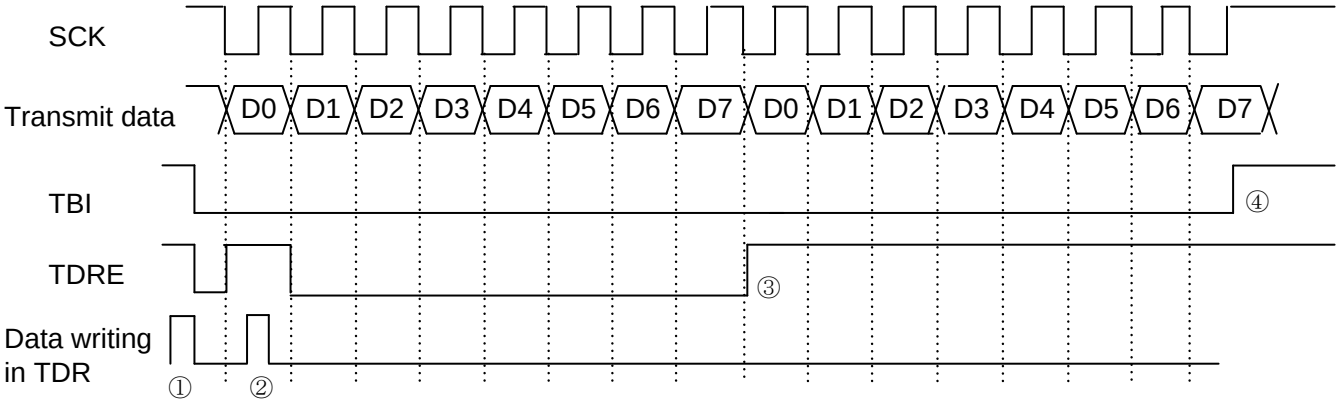
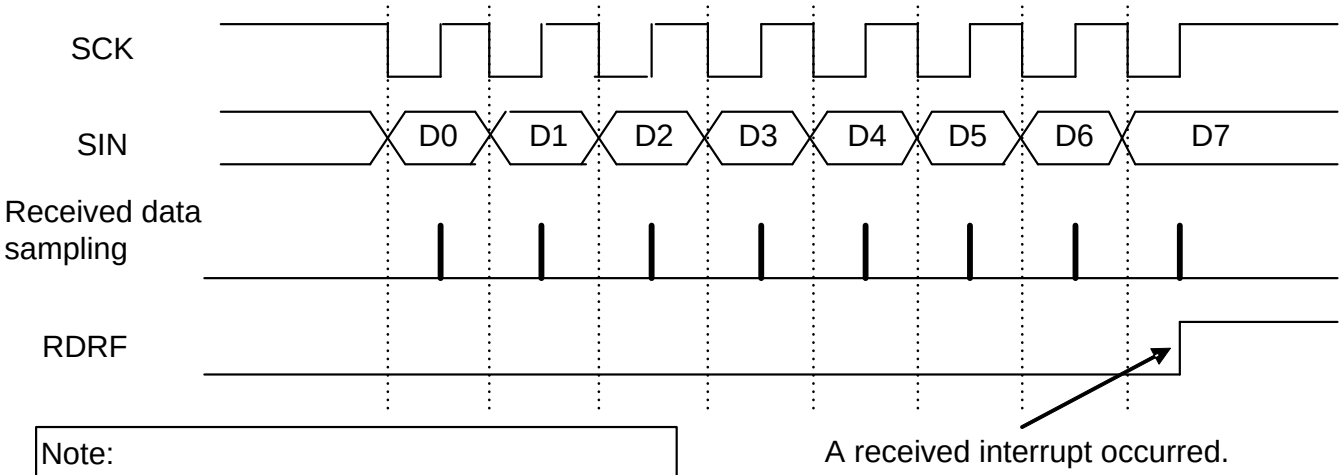



Figure 11 is the CSIO data receive timing diagram when FIFO is not used. When received data is stored in the RDR, the RDRF bit will be set to 1, and if Receive Interrupt is enabled (RIE = 1), a Receive Interrupt Request will be issued. After data is read from the RDR, the RDRF bit will be cleared automatically. However, when the RDRF bit is set, if the overrun error occurs (ORE = 1), the received data is invalid, and errors should be cleared by setting REC to 1.

Figure 11. CSIO Data Receive Timing Diagram




Note:
This figure shows the signal timing under the following conditions.
SCR: MS=1, SPI=0
ESCR: L2 to L0=0b000
SMR:SCINV=0, BDS=0, SCKE=0, SOE=0

Figure 12 is a timing diagram of data transmission when FIFO is enabled. It intends to transmit 4 bytes of data using transmit FIFO.

1. When FDRQ = 1, if FIFO transmit interrupts are enabled (FTIE = 1), a transmit interrupt occurs.
2. Three bytes are written into transmit FIFO, as the TDR is empty, so the first byte is transferred to the TDR. After that, FDRQ should be cleared to 0 manually. At this time, the data count in the FIFO is 2, as displayed by the FBYTE register.

3. After the FIFO is empty, the FDRQ bit turns to 1, which indicates the FIFO is empty. If FIFO transmit interrupts are enabled (FTIE = 1), a transmit interrupt occurs.
4. One byte is written into transmit FIFO. After that, FDRQ should be cleared to 0 manually. At this time, the data count in the FIFO is 1, as displayed by FBYTE register.
5. After the FIFO is empty, the FDRQ bit turns to 1, which indicates the FIFO is empty. If FIFO transmit interrupts are enabled (FTIE = 1), a transmit interrupt occurs.
6. After the TDR is empty, the TDRE bit turns to 1.

When the data bits in the shift register are all shifted out, the TBI bit is set to 1, which indicates all data transmission is completed.

Figure 12. CSIO Data Transmission Timing Diagram with FIFO Enabled

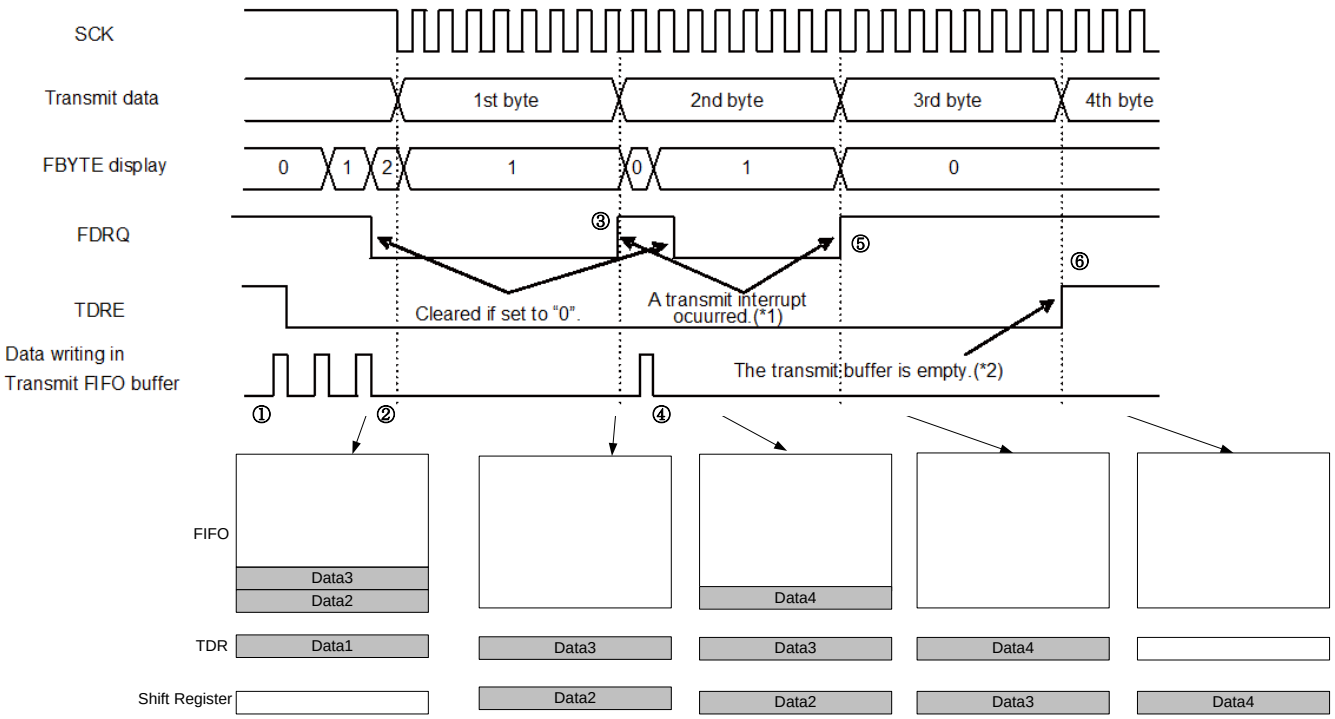
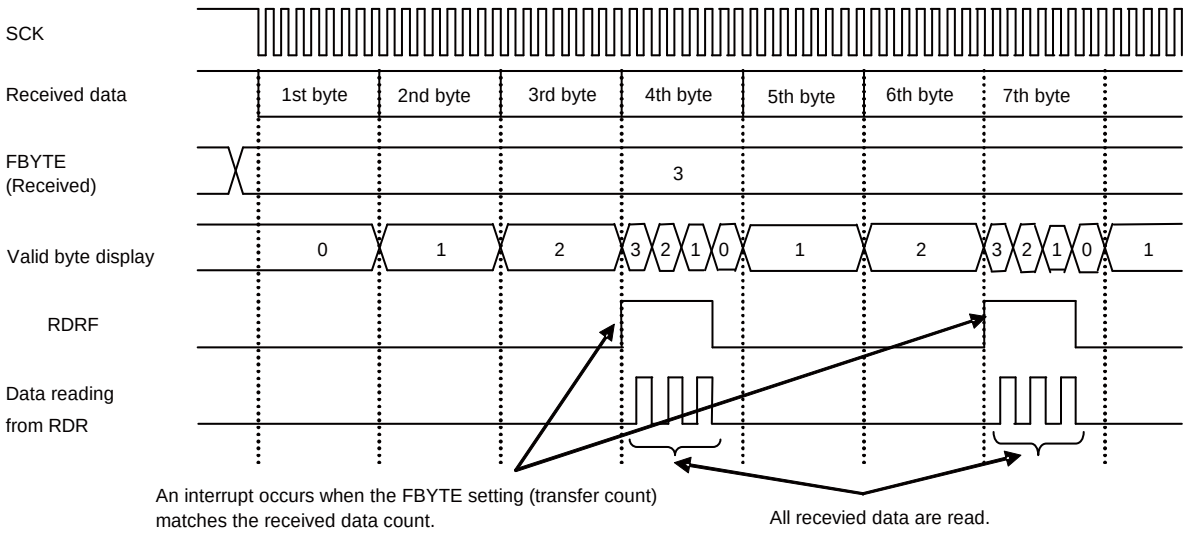



Figure 13 is the timing diagram of data receive when FIFO is enabled.

1. The FIFO match count should be set in the FBYTE.
2. After data receive starts, the received data will be stored in the FIFO in sequence. When the data count in the FIFO matches FBYTE, the RDRF bit will be set to 1. If RIE is set to 1, a receive interrupt will occur.
3. When only 1 byte is received, with no following data, if Received FIFO idle detection is enabled (FRIIE = 1), and if the received idle state continues for more than 8 baud rate clocks, the RDRF bit will be set to 1.
4. When the reads in the FIFO are all read out, then the RDRF bit will be cleared to 0 automatically.

   If the received data count exceeds the maximum capacity of the receive FIFO, an overrun error will occur.

Figure 13. CSIO Data Receive Timing Diagram with FIFO Enabled




## 3.6 Low-Level API

Following is the CSIO driver API of the PDL, which is placed in the `mfs.c/h` files.

- Mfs_Csio_Init()
- Mfs_Csio_DeInit()
- Mfs_Csio_EnableIrq()
- Mfs_Csio_DisableIrq()
- Mfs_Csio_SetBaudRate()
- Mfs_Csio_SetTimerCompareValue()
- Mfs_Csio_SetCsTransferByteCount()
- Mfs_Csio_SetCsHoldStatus()
- Mfs_Csio_SetTimerTransferByteCount()
- Mfs_Csio_EnableFunc()
- Mfs_Csio_DisableFunc()
- Mfs_Csio_GetStatus()
- Mfs_Csio_ClrStatus()
- Mfs_Csio_SendData()
- Mfs_Csio_ReceiveData()
- Mfs_Csio_ResetFifo()
- Mfs_Csio_SetFifoCount()
- Mfs_Csio_GetFifoCount()

`Mfs_Csio_Init()` is used to initialize an MFS instance to CSIO mode with parameter pstcConfig of type `#stc_mfs_csio_config_t`. `Mfs_Csio_DeInit()` is used to reset all MFS CSIO-related registers.

`Mfs_Csio_EnableIrq()` enables CSIO interrupt sources selected by enumeration type `#en_csio_irq_sel_t`. `Mfs_Csio_DisableIrq()` disables the CSIO interrupt sources selected by enumeration type `#en_csio_irq_sel_t`.

`Mfs_Csio_SetBaudRate()` provides a possibility to change the CSIO baud rate after CSIO is initialized.

`Mfs_Csio_SetTimerCompareValue()` can change the compare value of the CSIO serial timer.

`Mfs_Csio_SetCsTransferByteCount()` can change the transfer byte count of the Chip Selection pin selected.

`Mfs_Csio_SetTimerTransferByteCount()` sets the transfer byte count of the transfer process triggered by the serial timer.

`Mfs_Csio_SetCsHoldStatus()` sets the hold status of the CS pin after one transfer is finished.

`Mfs_Csio_EnableFunc()` enables the CSIO function selected by the parameter `Mfs_Csio_EnableFunc#enFunc`, and `Mfs_Csio_DisableFunc()` disables the CSIO function.

`Mfs_Csio_GetStatus()` gets the status selected by `Mfs_Csio_GetStatus#enStatus`, and `Mfs_Csio_ClrStatus()` clears the CSIO status selected. Some statuses can only be cleared by hardware automatically.

`Mfs_Csio_SendData()` writes a byte of data into the CSIO transfer buffer, and `Mfs_Csio_ReceiveData()` reads a byte of data from the CSIO receive buffer. The bit length of data is configured in `Mfs_Csio_Init()`.

`Mfs_Csio_ResetFifo()` resets CSIO hardware FIFO.

`Mfs_Csio_SetFifoCount()` provides a possibility to change the FIFO size after CSIO is initialized. `Mfs_Csio_GetFifoCount()` gets the current data count in the FIFO.

## 3.7 Example Code

Based on the low-level driver API, an example is given here to show how to transfer data via CSIO using the interrupt flag polling method. It uses CSIO ch.0 to transfer 10 bytes and then receive 10 bytes.

Before using CSIO, configure the pin function of CSIO as follows:

```
/* Initialize CSIO function I/O */
SetPinFunc_SIN0_0();
SetPinFunc_SOT0_0();
SetPinFunc_SCK0_0();
```

Then configure the CSIO configuration structure and initialize the CSIO channel.

- Mode: Master mode
- Baud rate: 100 kbps
- Data length: 8 bits
- Direction: MSB first

```
stc_mfs_csio_config_t stcCsioConfig;

/* Clear configuration structure */
PDL_ZERO_STRUCT(stcCsioConfig);

/* Initialize CSIO master  */
stcCsioConfig.enMsMode = CsioMaster;
stcCsioConfig.enActMode = CsioActNormalMode;
stcCsioConfig.bInvertClk = FALSE;
stcCsioConfig.u32BaudRate = 100000;
stcCsioConfig.enDataLength = CsioEightBits;
stcCsioConfig.enBitDirection = CsioDataMsbFirst;
stcCsioConfig.enSyncWaitTime = CsioSyncWaitZero;
stcCsioConfig.pstcFifoConfig = NULL;

if (Ok != Mfs_Csio_Init(&CSIO0, &stcCsio0Config))
{
    While(1);
}
```

The following code can send 10 CSIO bytes via the SOT pin.

```
uint8_t u8Cnt = 0;
uint8_t au8TxBuf[10] = {0x01,0x23,0x45,0x67,0x89,0xAB,0xCD,0xEF};

/* Enable TX function of CSIO0   */
Mfs_Csio_EnableFunc(&CSIO0, CsioTx);

while(u8Cnt < 10)
{
    /* Wait until transmit data register empty */
    while (TRUE != Mfs_Csio_GetStatus(&CSIO0,CsioTxEmpty));
    /* Write data to transmit data register */
    Mfs_Csio_SendData(&CSIO0, au8TxBuf[u8Cnt], TRUE);
    u8Cnt++;
}

/* Wait until master TX bus idle */
while (TRUE != Mfs_Csio_GetStatus(&CSIO0, CsioTxIdle));

/* Disable TX function of CSIO0   */
Mfs_Csio_DisableFunc(&CSIO0, CsioTx);
```

The following code can receive 10 CSIO bytes via the SIN pin. It should be noted that in the synchronization communication, the clock line is always controlled by the master, even if the master is receiving data from the slave. When the master needs to receive data from the slave, the dummy data should be sent to generate the clock.

```
uint8_t u8Cnt = 0;
uint8_t au8RxBuf[10];

/* Enable TX and RX function of CSIO0   */
Mfs_Csio_EnableFunc(&CSIO0, CsioTx);
Mfs_Csio_EnableFunc(&CSIO0, CsioRx);

while(u8Cnt < 10)
{
    /* write a dummy data */
    Mfs_Csio_SendData(&CSIO0, 0x00u, FALSE);

    /* wait until receive data register full */
    while(TRUE != Mfs_Csio_GetStatus(&CSIO0, CsioRxFull));
    /* Read data from receive data register */
    au8RxBuf[u8Cnt] = Mfs_Csio_ReceiveData(&CSIO0);
    u8Cnt++;
}

/* Wait until master TX bus idle */
while (TRUE != Mfs_Csio_GetStatus(&CSIO0, CsioTxIdle));

/* Disable RX function of CSIO0   */
Mfs_Csio_DisableFunc(&CSIO0, CsioTx);
Mfs_Csio_DisableFunc(&CSIO0, CsioRx);
```

# 4 I²C

The I²C interface (I²C communications control interface) supports the I²C bus and operates as a master/slave device on the I²C bus.

When the MD bits' SMR register is set to b'100, I²C mode is configured.

| bit7 | bit6 | bit5 | Description |
|---|---|---|---|
| 0 | 0 | 0 | Operation mode 0 (asynchronous normal mode) |
| 0 | 0 | 1 | Operation mode 1 (asynchronous multiprocessor mode) |
| 0 | 1 | 0 | Operation mode 2 (clock sync mode) |
| 0 | 1 | 1 | Operation mode 3 (LIN communication mode) |
| 1 | 0 | 0 | Operation mode 4 (I²C mode) |
| Other than the above | | | Setting is prohibited. |

## 4.1 Features

- Master/slave function
- 15-bit baud rate selection
- 8-bit data length
- Bus arbitration function
- Transmission direction detection function in slave mode
- Function to generate and detect iteration start condition

- Bus error detection function
- 7-bit addressing as slave
- Generation of an interrupt enabled during transmission or a bus error
- Removal of noise from 2 to 32 clocks in the bus clock for serial clock/serial data input[1]
- Support for DMA transferring
- Interrupt requests
  - Received interrupt request by factor of reception completion, overrun error
  - Transmit interrupt request by factor of transmit data empty, transmit bus idle
  - Transmit FIFO interrupt request by factor of transmit FIFO empty
  - Status interrupt request by factor of data transmission/reception completion, detection of bus error and NACK
  - Detection of stop condition and iteration start
- Transmit/receive FIFO installed[2]

[1] Only some FM products have a noise filter; see the datasheet of the product used.

[2] The FIFO capacity varies depending on the product type; see the datasheet of the product used.

## 4.2 Protocol

The $I^2C$ frame always consists of a start condition, 7-bit slave address + 1-bit R/W, data, and stop condition, as shown in Figure 14.

When the $I^2C$ master sends data to the $I^2C$ slave, set the R/W bit to 0. After sending the start condition signal and first byte, the master will receive an ACK bit from the slave. The master then continues to send data to the slave and will receive an ACK bit from the slave at the end of each byte in a normal condition. After the $I^2C$ master sends all the data, it sends a stop condition signal to the slave to indicate that the data transfer is complete.

When the $I^2C$ master receives data from the $I^2C$ slave, set the R/W bit to 1. After sending the start condition signal and first byte, the master will receive an ACK bit from the slave. The master can then read data from the $I^2C$ slave and should send a 1-bit ACK to the slave after receiving each byte. Note that master should send a 1-bit NACK to the slave after receiving the last data, which informs the slave that it is the last data that the master wants to receive. Then the master sends the stop condition signal to complete the $I^2C$ data receive process.

Figure 14. I²C Data Format

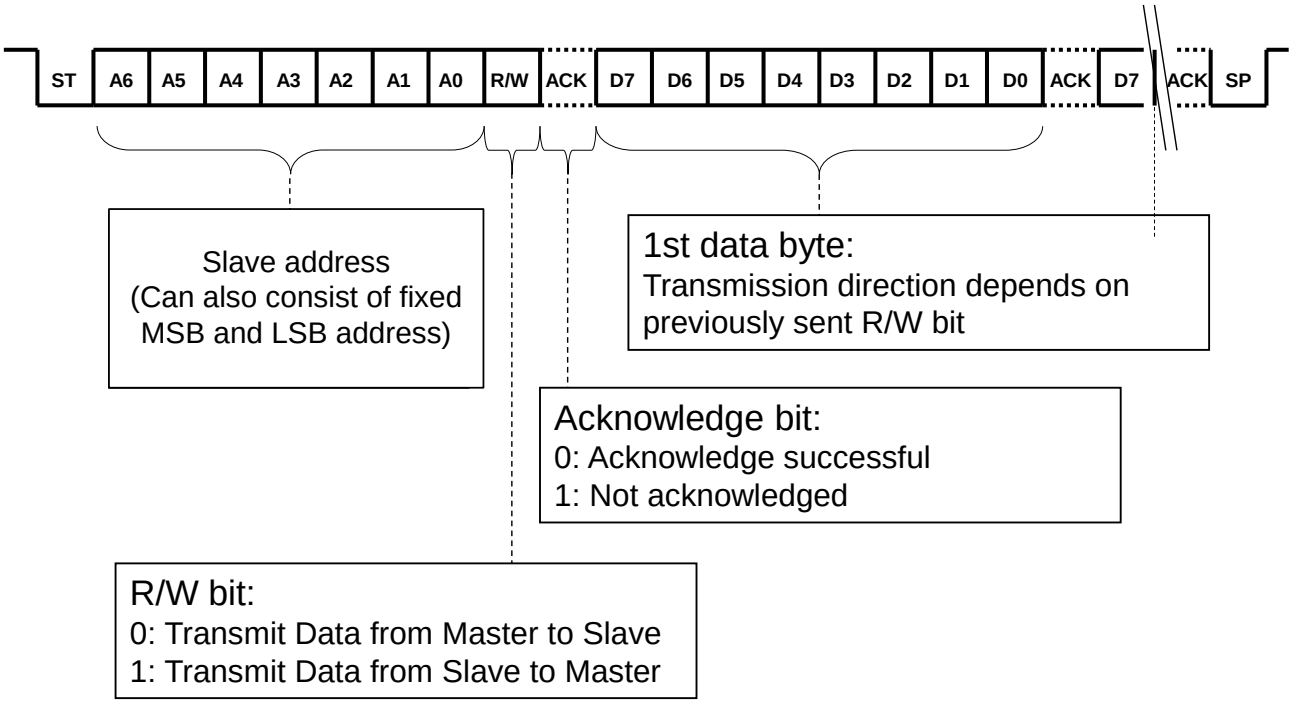



Figure 15 is the timing diagram of the I²C start and stop conditions.

**Start condition:**

- When the SCL (SCK) line is high, if it has a falling edge on the SDA (SOT) line, it indicates the start of the I²C frame.
- In the I²C module of the FM MCU, when MSS = 0 and ACT = 0, setting MSS to 1 will generate the I²C start condition. Then the ACT bit will also be set to 1 automatically to indicate that it has entered master mode operation.

**Stop condition:**

- When the SCL (SCK) line is high, if it has a rising edge on the SDA (SOT) line, it indicates the stop of the I²C frame.
- In the I²C module of the FM MCU, when MSS = 1 and ACT = 1, setting MSS to 0 will generate the I²C stop condition. Then the ACT bit will also be set to 0 automatically to indicate that it has entered stop mode operation.

Figure 15. I²C Start and Stop Condition Timing Diagram

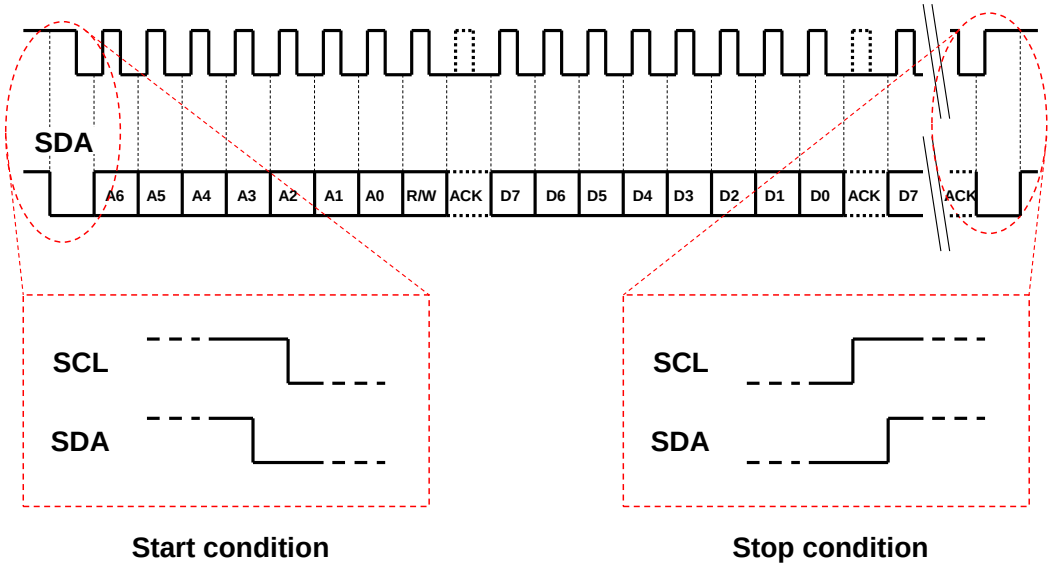


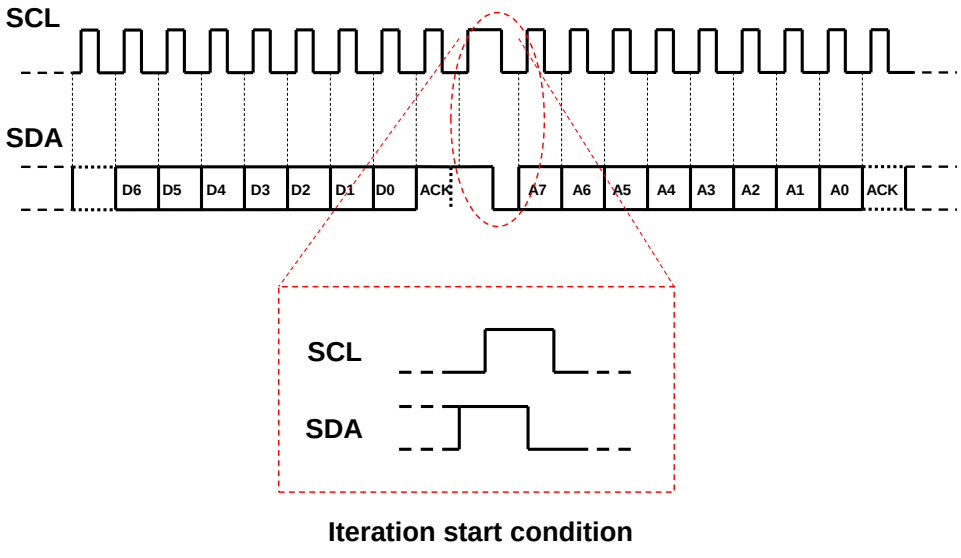

Figure 16 is the timing diagram of the I²C iteration (repeated) start condition.

**Iteration (repeated) start condition:**

When I²C communication has started, if the master generates the start condition again, it is called an “iteration start condition” or “repeated start condition.” This signal may be used when reading the data from an I²C EEPROM.

In the I²C module of the FM MCU, when MSS = 1 and ACT = 1, setting the MSS to 1 will generate a repeated start condition.

Figure 16. I²C Repeated Start Condition Timing Diagram



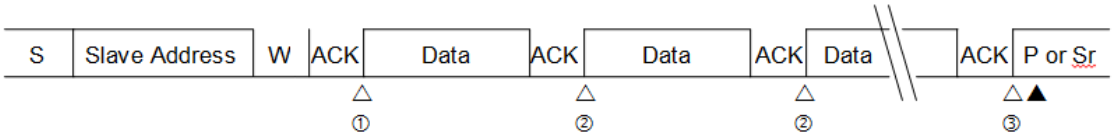
The following is a description of the MSS and ACT bits in the IBCR register:

| MSS Bit | ACT Bit | State |
|---|---|---|
| 0 | 0 | Idle |
| 0 | 1 | The slave address matching or ACK is responded to the reserved address, and slave mode is in operation. |
| 1 | 0 | The master mode operation waits. |
| 1 | 1 | During master mode operation |

## 4.3 Data Transfer Timing

Figure 17 is the timing diagram to write some data via I$^2$C when FIFO is not enabled.

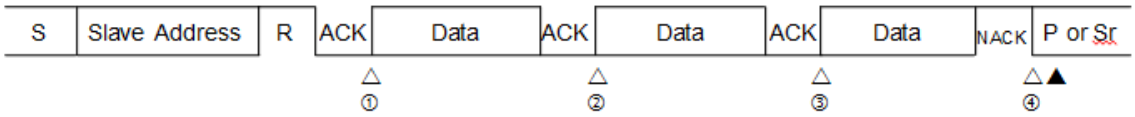
Figure 17. I$^2$C Data Transfer Diagram




S: Start condition
W: Data direction bit (write direction)
P: Stop condition
Sr: Iteration start condition
△: Interrupt by INTE="1"
▲: Interrupt by CNDE="1"
① An interrupt occurs when the slave address is sent, the direction bit is sent, and an ACK is received.
- The send data is written in the TDR register, and the INT bit is set to "0".
② An interrupt occurs when a single byte is sent and an ACK is received.
- The send data is written in the TDR register, and the INT bit is set to "0".
③ An interrupt occurs when a single byte is sent and an ACK is received.
- MSS bit is set to "0", or MSS and SCC bits are set to "1".
*) If an interrupt flag (INT) is set, the TDRE bit is set to "1".

Figure 18 is the timing diagram to read some data via I²C when FIFO is not enabled.

Figure 18. I²C Data Receive Diagram




△: Interrupt by INTE="1"
▲: Interrupt by CNDE="1"
① An interrupt occurs when the slave address is sent, the direction bit is sent, and an ACK is received.
- If the INT bit is set to "0", the interrupt flag is cleared to "0".
② An interrupt occurs when a single byte is received and an ACK is sent.
- After the received data has been read, the INT bit is set to "0".
③ An interrupt occurs when a single byte is received and an ACK is sent.
- After the received data has been read, both ACKE and INT bits are set to "0".
④ An interrupt occurs when a single byte is received and an ACK is sent.
- MSS bit is set to "0", or both MSS and SCC bits are set to "1".
*) If an interrupt flag (INT) is set, the TDRE bit is set to "1".

## 4.4 Low-Level API

Following is the I²C driver API of the PDL, which is placed in the `mfs.c/h` files.

- Mfs_I2c_Init()
- Mfs_I2c_DeInit()
- Mfs_I2c_EnableIrq()
- MfI2c_DisableIrq()
- Mfs_I2c_GenerateStart()
- Mfs_I2c_GenerateRestart()
- Mfs_I2c_GenerateStop()
- Mfs_I2c_SetBaudRate()
- Mfs_I2c_SendData()
- Mfs_I2c_ReceiveData()
- Mfs_I2c_ConfigAck()
- Mfs_I2c_GetAck()
- Mfs_I2c_GetStatus()
- Mfs_I2c_ClrStatus()
- Mfs_I2c_GetDataDir()
- Mfs_I2c_ResetFifo()
- Mfs_I2c_SetFifoCount()
- Mfs_I2c_GetFifoCount()

`Mfs_I2c_Init()` is used to initialize an MFS instance to I²C mode with parameter `pstcConfig` of type `#stc_mfs_i2c_config_t`. This function sets only the basic I²C configuration. `Mfs_I2c_DeInit()` is used to reset all MFS I²C-related registers.

`Mfs_I2c_EnableIrq()` enables I²C interrupt sources selected by enumeration type `#en_i2c_irq_sel_t`, and `Mfs_I2c_DisableIrq()` disables the I²C interrupt sources selected by enumeration type `#en_i2c_irq_sel_t`.

`Mfs_I2c_SetBaudRate()` provides a possibility to change the I²C baud rate after I²C is initialized.

`Mfs_I2c_SendData()` writes a byte of data into the I²C transfer buffer, and `Mfs_I2c_ReceiveData()` reads a byte of data from the I²C receive buffer.

`Mfs_I2c_GenerateStart()` generates an I²C start signal. `Mfs_I2c_GenerateRestart()` generates an I²C restart signal. `Mfs_I2c_GenerateStop()` generates an I²C stop signal.

`Mfs_I2c_ConfigAck()` configures the ACK signal when receiving data. `Mfs_I2c_GetAck()` gets the ACK signal status after receiving an ACK.

`Mfs_I2c_GetStatus()` gets the status selected by `Mfs_I2c_GetStatus#enStatus`, and `Mfs_I2c_ClrStatus()` clears the I²C status selected. Some statuses can only be cleared by hardware automatically.

`Mfs_I2c_GetDataDir()` gets the data direction of I²C in the slave mode.

`Mfs_I2c_ResetFifo()` resets the I²C hardware FIFO. `Mfs_I2c_SetFifoCount()` provides a possibility to change the FIFO size after I²C is initialized.

`Mfs_I2c_GetFifoCount()` gets the current data count in the FIFO.

## 4.5 Example Code

Based on the low-level driver API, an example is given here to show how to transfer data via I²C using the interrupt flag polling method. It uses I²C ch.0 to transfer 10 bytes and then receive 10 bytes.

Before using I²C, configure the pin function of I²C as follows:

```
/* Initialize I2C function I/O */
SetPinFunc_SOT0_0();
SetPinFunc_SCK0_0();
```

Then configure the I²C configuration structure and initialize the I²C channel.

- Mode: Master mode
- Baud rate: 100 kbps

```
stc_mfs_i2c_config_t stcI2c0Config;

/* Configure I2C structure */
stcI2c0Config.enMsMode = I2cMaster;
stcI2c0Config.u32BaudRate = 100000u;
stcI2c0Config.bWaitSelection = FALSE;
stcI2c0Config.bDmaEnable = FALSE;
stcI2c0Config.pstcFifoConfig = NULL;

if (Ok != Mfs_I2c_Init(&I2C0, &stcI2c0Config))
{
    While(1);
}
```

The following code will send 10-byte data via I²C; assume that the device address is 0x50.

```
/*****************************************************************/
/*      Generate start condition                                 */
/*****************************************************************/
/* Prepare I2C device address */
Mfs_I2c_SendData(&I2C0, (0x50<<1));

/* Generate I2C start signal */
if(Ok != Mfs_I2c_GenerateStart(&I2C0))
{
    while(1); /* Timeout or other error */
}

while(1)
{
    if(TRUE != Mfs_I2c_GetStatus(&I2C0, I2cRxTxIrq))
    {
        break;
    }
}

if(I2cNAck == Mfs_I2c_GetAck((&I2C0))
{
    while(1);   /* NACK */
}

if(TRUE == Mfs_I2c_GetStatus((&I2C0, I2cBusErr))
{
    while(1); /* Bus error occurs? */
}

/*****************************************************************/
/*      Send data                                                */
/*****************************************************************/

for(uint8_t i=0;i<10;i++)
{
    /* Transmit the data */
    Mfs_I2c_SendData(&I2C0, pTxData[i]);
    Mfs_I2c_ClrStatus(&I2C0, I2cRxTxIrq);
    /* Wait for end of transmission */
    while(1)
    {
        if(TRUE == Mfs_I2c_GetStatus(&I2C0, I2cRxTxIrq))
        {
            break;
        }
    }

    while(1)
    {
        if(TRUE == Mfs_I2c_GetStatus(&I2C0, I2cTxEmpty))
        {
            break;
        }
    }

    if(I2cNAck == Mfs_I2c_GetAck(&I2C0))
    {
        while(1);   /* NACK */
    }
```

```
    if(TRUE == Mfs_I2c_GetStatus(&I2C0, I2cBusErr))
    {
        while(1); /* Bus error occurs? */
    }
}
/*****************************************************************/
/*      Generate stop condition                                  */
/*****************************************************************/
/* Generate I2C start signal */
if(Ok != Mfs_I2c_GenerateStop(&I2C0))
{
    while(1); /* Timeout or other error */
}
/* Clear Stop condition flag */
while(1)
{
    if(TRUE == Mfs_I2c_GetStatus(&I2C0, I2cStopDetect))
    {
        break;
    }
}
Mfs_I2c_ClrStatus(&I2C0, I2cStopDetect);
Mfs_I2c_ClrStatus(&I2C0, I2cRxTxIrq);
```

The following code will read 10-byte data via $I^2C$; assume that the device address is 0x50.

```
/*****************************************************************/
/*      Generate start condition                                 */
/*****************************************************************/
/* Prepare I2C device address */
Mfs_I2c_SendData(&I2C0, (0x50<<1));

/* Generate I2C start signal */
if(Ok != Mfs_I2c_GenerateStart(&I2C0))
{
    while(1); /* Timeout or other error */
}

while(1)
{
    if(TRUE != Mfs_I2c_GetStatus(&I2C0, I2cRxTxIrq))
    {
        break;
    }
}

if(I2cNAck == Mfs_I2c_GetAck((&I2C0))
{
    while(1);   /* NACK */
}

if(TRUE == Mfs_I2c_GetStatus((&I2C0, I2cBusErr))
{
    while(1); /* Bus error occurs? */
}

/*****************************************************************/
/*      Read data                                                */
```

```
        /*****************************************************************/
        uint8_t i;

        /* Clear interrupt flag generated by device address send */
        Mfs_I2c_ClrStatus(&I2C0, I2cRxTxIrq);

        if(I2cNAck == Mfs_I2c_GetAck(&I2C0))
        {
            while(1);   /* NACK */
        }

        while(i < u8Size)
        {
            /* Wait for the receive data */
            while(1)
            {
                if(TRUE == Mfs_I2c_GetStatus(&I2C0, I2cRxTxIrq))
                {
                    break;
                }
            }

            if(i == u8Size-1)
            {
                Mfs_I2c_ConfigAck(&I2C0, I2cNAck); /* Last byte send a NACK */
            }
            else
            {
                Mfs_I2c_ConfigAck(&I2C0, I2cAck);
            }

            /* Clear interrupt flag and receive data to RX buffer */
            Mfs_I2c_ClrStatus(&I2C0, I2cRxTxIrq);

            /* Wait for the receive data */
            while(1)
            {
                if(TRUE == Mfs_I2c_GetStatus(&I2C0, I2cRxFull))
                {
                    break;
                }
            }

            if(TRUE == Mfs_I2c_GetStatus(&I2C0, I2cBusErr))
            {
                while(1);   /* Bus error occurs? */
            }

            if(TRUE == Mfs_I2c_GetStatus(&I2C0, I2cOverrunError))
            {
                while(1;  /* Overrun error occurs? */
            }

            pRxData[i++] = Mfs_I2c_ReceiveData(&I2C0);
        }
        /*****************************************************************/
        /*      Generate stop condition                                  */
        /*****************************************************************/
        /* Generate I2C start signal */
        if(Ok != Mfs_I2c_GenerateStop(&I2C0))
        {
```

```
            while(1); /* Timeout or other error */
        }
        /* Clear Stop condition flag */
        while(1)
        {
            if(TRUE == Mfs_I2c_GetStatus(&I2C0, I2cStopDetect))
            {
                break;
            }
        }
        Mfs_I2c_ClrStatus(&I2C0, I2cStopDetect);
        Mfs_I2c_ClrStatus(&I2C0, I2cRxTxIrq);
```

# 5 LIN

The LIN interface (LIN communication control interface ver. 2.1) supports functions complying with the LIN bus.

When the MD bits' SMR register is set to b'011, the LIN mode is configured.

| bit7 | bit6 | bit5 | Description |
|---|---|---|---|
| 0 | 0 | 0 | Operation mode 0 (asynchronous normal mode) |
| 0 | 0 | 1 | Operation mode 1 (asynchronous multiprocessor mode) |
| 0 | 1 | 0 | Operation mode 2 (clock sync mode) |
| 0 | 1 | 1 | Operation mode 3 (LIN communication mode) |
| 1 | 0 | 0 | Operation mode 4 ($I^2C$ mode) |
| Other than the above | | | Setting is prohibited. |

## 5.1 Features

- Support for LIN Protocol Revision 2.1
- Master/slave device operation
- Full-duplex operation
- LIN break field generation (with variable bit length ranging from 13 to 16 bits)[1]
- LIN break delimiter generation (with variable data length ranging from 1 to 4 bits)[1]
- LIN break field detection[1]
- Detection of LIN sync field start/stop edges connected to input capture[1]
- 15-bit baud rate selection[2]
- 8-bit data length
- Received error detection
  - Framing error
  - Overrun error
- Interrupt requests
  - Received interrupt request by factor of reception completion, framing error, overrun error, or parity error
  - Transmit interrupt request by factor of transmit data empty, transmit bus idle
  - Transmit FIFO interrupt request by factor of transmit FIFO empty

- Support for DMA transferring
- Transmit/receive FIFO installed[3]

[1] Only some products are equipped with the LIN module; see the datasheet of the product used.

[2] The baud rate generator can also be sourced by an external clock.

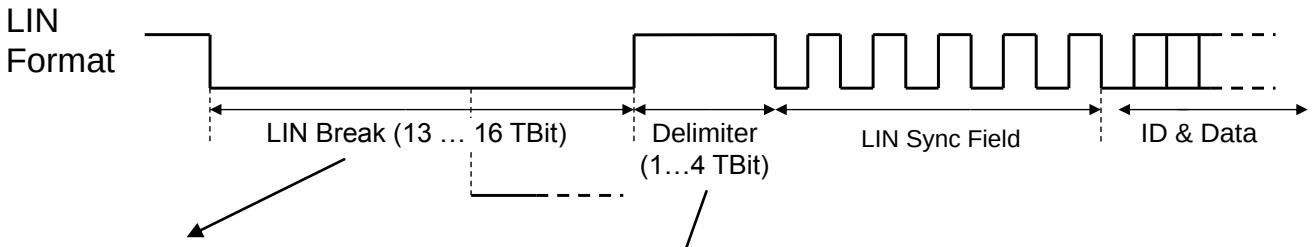
[3] The FIFO capacity size varies depending on the product type; see the datasheet of the product used.

## 5.2 Data Format

The LIN frame consists of the LIN break, Delimiter, LIN sync field, and ID and data fields, as shown in Figure 19.

In the LIN master mode, the LIN break length can be set by the LBL0 and LBL1 bits, and the LIN delimiter length can be set by the DEL0 and DEL1 bits in the ESCR register.

Figure 19. LIN Data Format




**[bit3:2] LBL1/LBL0: LIN break field length select bits**

| bit3 | bit2 | Description |
|---|---|---|
| 0 | 0 | 13-bit length |
| 0 | 1 | 14-bit length |
| 1 | 0 | 15-bit length |
| 1 | 1 | 16-bit length |

**[bit1:0] DEL1/DEL0: LIN break delimiter length select bits**

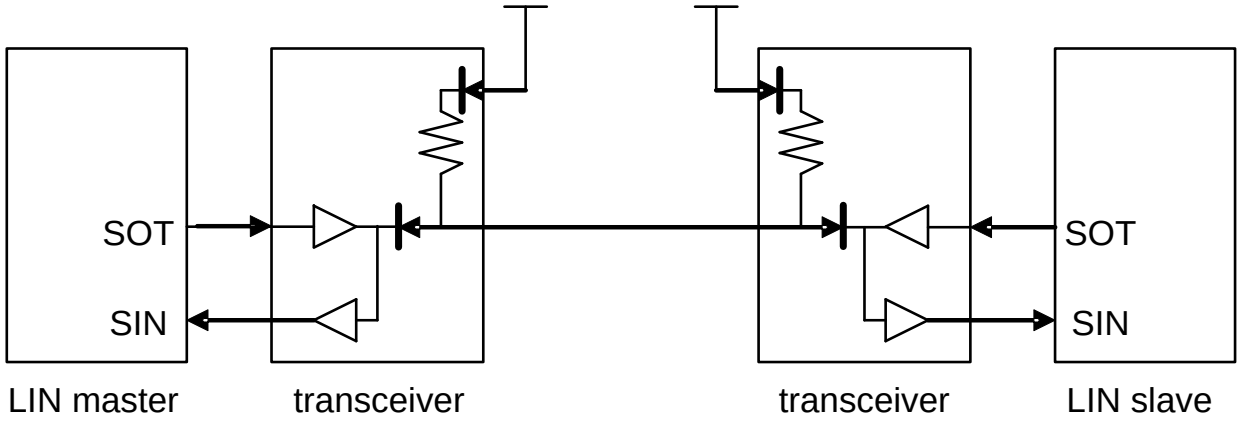
| bit1 | bit0 | Description |
|---|---|---|
| 0 | 0 | 1-bit length |
| 0 | 1 | 2-bit length |
| 1 | 0 | 3-bit length |
| 1 | 1 | 4-bit length |

In the LIN slave mode, the LIN break can be detected after a low level lasting 11 bit times. The baud rate can be adjusted by capturing the LIN sync field with the internal Input Capture Unit (ICU).

## 5.3 Communication System

Figure 20 shows a communication system consisting of one LIN master and one LIN slave via a LIN transceiver.

Figure 20. LIN Communication System




## 5.4 Operation Timing

Figure 21 is the timing diagram of data transmission in the LIN master mode.

1. Setting the LIN break field (LBR) bit to 1 causes a break to be sent by the master, and then 0x55 (LIN sync field) can be written into the TDR. It can be sent after the LIN break.
2. The first bit of the LIN sync field (0x55) is shifted out to the SOT pin. The TDRE bit will be set to 1, and a transmit interrupt will occur if the TIE bit is set to 1. Then the ID byte can be written to the TDR.
3. The LIN master can receive whatever it sends. At this point, 0x55 is received, and a comparison with the original data can be done to check if the data transmit is normal.
4. When the first bit of the ID is sent out, the TDRE bit turns to 1, and transmit interrupt will occur if the TIE bit is set to 1. Then the first data can be sent.

The data transmission process is the same with ID field transmission.

Figure 21. LIN Master Data Transmission Timing Diagram

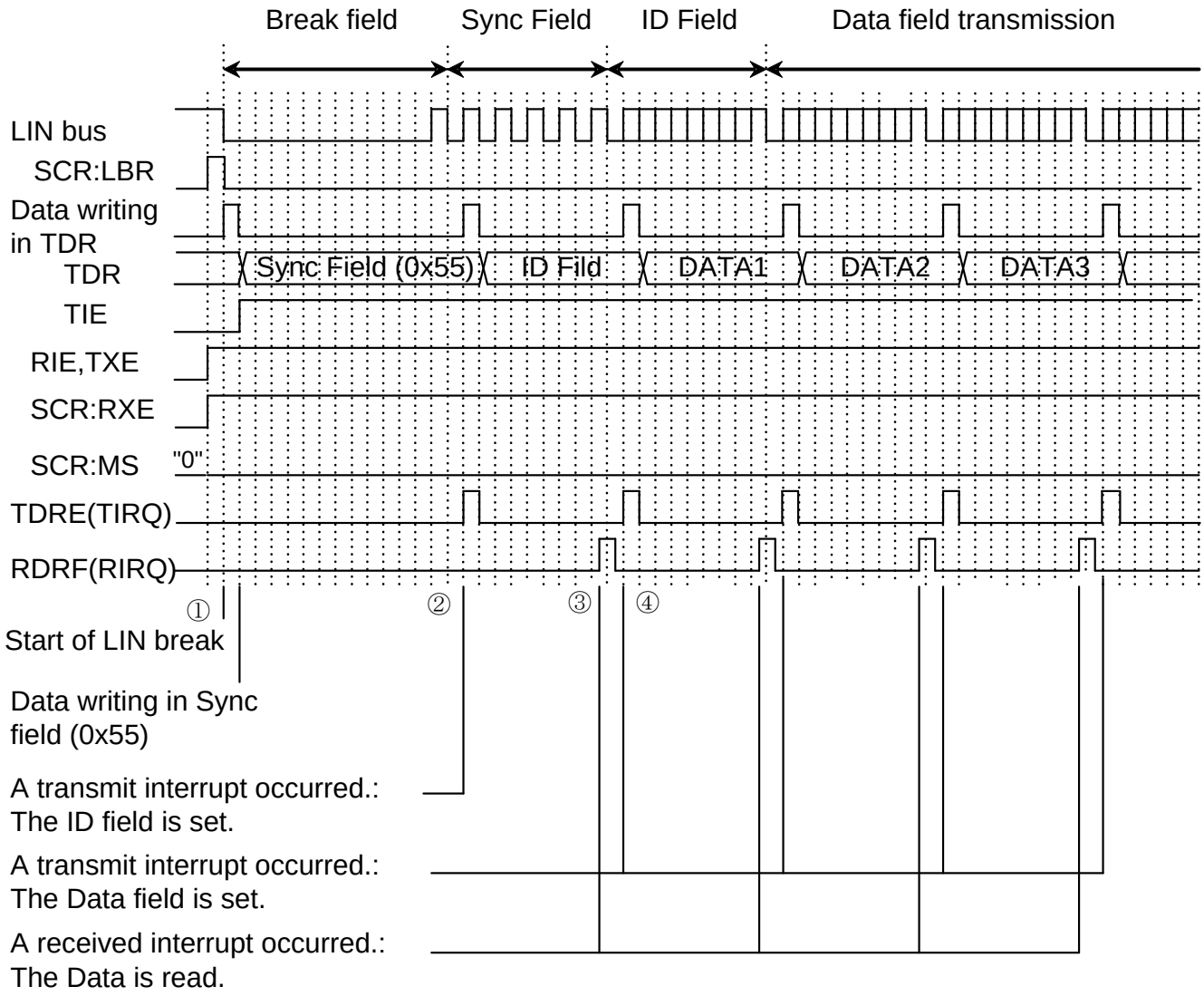



Figure 22 is the timing diagram of data transmission in the LIN slave mode.

1. The LIN break can be detected after a low level lasting 11 bit times. If the LIN Break Interrupt Enable (LBIE) bit is set to 1, a LIN break interrupt will occur. Then the ICU that connects the SYNC signal should be initialized and enabled. Refer to the “GPIO” chapter of the peripheral manual to see the ICU channel corresponding to the particular LIN channel.
2. Upon the detection of the first falling edge in the sync field, the ICU interrupt will be triggered and the ICU data register value can be stored into a variable *a*.
3. When the ICU interrupt occurs again, the ICU data register value can be stored into a variable *b*.

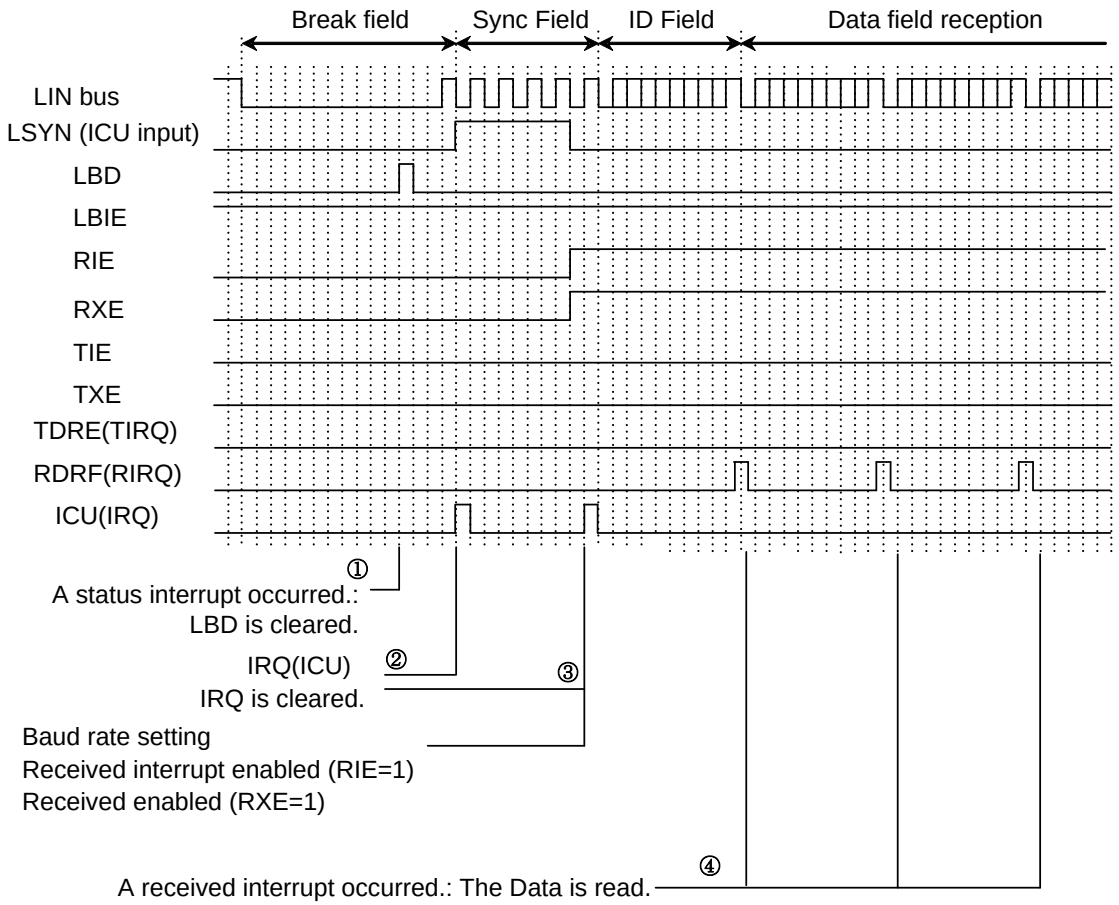
   The exact baud rate of the LIN master can be calculated with the following formula (if FRT does not overflow and it is the same clock for MFS and FRT). The new baud rate should be set to the BGR, and the ICU function should be disabled. Then receive can be enabled (RXE = 1), and receive interrupt can be enabled (RIE = 1).

```
BGR value = (b - a)/8 - 1
a:    The ICU data register value after the first interrupt
b:    The ICU data register value after the second interrupt
```

4. When the receive data register is full (RDRF = 1), if a receive interrupt is enabled, a receive interrupt will occur, and the ID field can be read from the RDR.

   The data receive process is the same with the ID field.

Figure 22. LIN Slave Data Transmission Timing Diagram




## 5.5 Low-Level API

Following is the LIN driver API of the PDL, which is placed in the `mfs.c/h` files.

- Mfs_Lin_Init()
- Mfs_Lin_DeInit()
- Mfs_Lin_EnableIrq()
- Mfs_Lin_DisableIrq()
- MfsLinIrqHandlerStatus()
- Mfs_Lin_SetBaudRate()
- Mfs_Lin_GenerateBreakField()
- Mfs_Lin_EnableFunc()
- Mfs_Lin_DisableFunc()
- Mfs_Lin_GetStatus()

- Mfs_Lin_ClrStatus()
- Mfs_Lin_SendData()
- Mfs_Lin_ReceiveData()
- Mfs_Lin_ResetFifo()
- Mfs_Lin_SetFifoCount()
- Mfs_Lin_GetFifoCount()

`Mfs_Lin_Init()` is used to initialize an MFS instance to LIN mode with its dedicated LIN configuration `#stc_mfs_lin_config_t`. This function sets only the basic LIN configuration. `Mfs_Lin_DeInit()` is used to reset all MFS LIN-related registers.

`Mfs_Lin_EnableIrq()` enables the LIN interrupt sources selected by interrupt type `#en_lin_irq_sel_t`, and `Mfs_Lin_DisableIrq()` disables the LIN interrupt sources selected by interrupt type `#en_lin_irq_sel_t`.

`Mfs_Lin_SetBaudRate()` provides a possibility to change the LIN baud rate after the LIN is initialized.

`Mfs_Lin_GenerateBreakField()` generates a LIN break field, which can also be detected by itself.

`Mfs_Lin_EnableFunc()` enables the LIN function selected by the parameter `Mfs_Lin_EnableFunc#enFunc`, and `Mfs_Lin_DisableFunc()` disables the LIN function.

`Mfs_Lin_GetStatus()` gets the status selected by `Mfs_Lin_GetStatus#enStatus`, and `Mfs_Lin_ClrStatus()` clears the LIN status selected. Some statuses can only be cleared by hardware automatically.

`Mfs_Lin_SendData()` writes a byte of data into the LIN transfer buffer, and `Mfs_Lin_ReceiveData()` reads a byte of data from the LIN receive buffer.

`Mfs_Lin_ResetFifo()` resets the LIN hardware FIFO.

`Mfs_Lin_SetFifoCount()` provides a possibility to change the FIFO size after the LIN is initialized.

`Mfs_Lin_GetFifoCount()` gets the current data count in the FIFO.

## 5.6 Example Code

Based on the low-level driver API, an example is given here to show how to transfer data via the LIN using the interrupt flag polling method. It uses LIN ch.0 as the LIN master to transfer 10 bytes.

Before using LIN, configure the pin function of LIN as follows:

```
/* Initialize LIN function I/O */
SetPinFunc_SOT0_0();
SetPinFunc_SIN0_0();
```

Then configure the LIN configuration structure and initialize the LIN channel.

- Mode: Master mode
- Baud Rate: 9600 bps
- Break Length: 13 bits
- Delimiter Length: 1 bit
- Stop Bit Length: 1 bit

```
stc_mfs_lin_config_t stcLinConfig;
uint32_t u32i;
uint8_t u8RdData;

/* Initialize LIN  */
stcLinConfig.enMsMode = LinMasterMode;
stcLinConfig.u32BaudRate = 9600;
stcLinConfig.enBreakLength = LinBreakLength13;
stcLinConfig.enDelimiterLength = LinDelimiterLength1;
stcLinConfig.enStopBits = LinOneStopBit;
stcLinConfig.pstcFifoConfig = NULL;

if (Ok != Mfs_Lin_Init(&LIN0, &stcLinConfig))
{
    while(1); /* Initialization error */
}
```

The following code will send 10 bytes with the ID field set to 0x3A.

```
/*****************************************************************/
/*   Send LIN break                                              */
/*****************************************************************/
/* Generate LIN break field */
Mfs_Lin_GenerateBreakField(&LIN0);
while(Mfs_Lin_GetStatus(&LIN0, LinBreakFlag) != TRUE);
Mfs_Lin_ClrStatus(&LIN0, LinBreakFlag);

/* Enable TX and RX function of LIN   */
Mfs_Lin_EnableFunc(&LIN0, LinTx);
Mfs_Lin_EnableFunc(&LIN0, LinRx);



/*****************************************************************/
/*   Send Sync filed                                             */
/*****************************************************************/
while(Mfs_Lin_GetStatus(&LIN0, LinTxEmpty) != TRUE); // Wait until TDR empty
Mfs_Lin_SendData(&LIN0, 0x55);
while(Mfs_Lin_GetStatus(&LIN0, LinRxFull) != TRUE);  // Wait until RDR full
u8RdData = Mfs_Lin_ReceiveData(&LIN0);
if(u8RdData != 0x55)
{
    while(1); /* Send data error */
}

/*****************************************************************/
/*   Send ID filed                                               */
/*****************************************************************/
while(Mfs_Lin_GetStatus(&LIN0, LinTxEmpty) != TRUE); // Wait until TDR empty
Mfs_Lin_SendData(&LIN0, 0x3A);
while(Mfs_Lin_GetStatus(&LIN0, LinRxFull) != TRUE);  // Wait until RDR full
u8RdData = Mfs_Lin_ReceiveData(&LIN0);
if(u8RdData != 0x3A)
{
    while(1); /* Send data error */
}
```

```
/*****************************************************************/
/*   Send Data filed                                             */
/*****************************************************************/
for(u32i=0; u32i<10; u32i++)
{
    while(Mfs_Lin_GetStatus(&LIN0, LinTxEmpty) != TRUE); // Wait until TDR empty
    Mfs_Lin_SendData(&LIN0, pData[u32i]);
    while(Mfs_Lin_GetStatus(&LIN0, LinRxFull) != TRUE); // Wait until RDR full
    u8RdData = Mfs_Lin_ReceiveData(&LIN0);
    if(u8RdData != pData[u32i])
    {
        While(1);
    }
}

while(Mfs_Lin_GetStatus(LinCh1, LinTxIdle) != TRUE); // Wait until TX bus idle
```

# 6 Summary

The MFS interface is highly flexible and can be applied to various types of serial communication.

## Document History

Document Title: AN99218 - Multifunction Serial Interface of FM MCU

Document Number: 001-99218

| Revision | ECN | Orig. of Change | Submission Date | Description of Change |
|---|---|---|---|---|
| ** | 4918086 | CHZH | 09/30/2015 | New application note |
| *A | 5700411 | AESATMP9 | 04/19/2017 | Updated logo and copyright. |

## Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

## Products

| | |
|---|---|
| ARM® Cortex® Microcontrollers | cypress.com/arm |
| Automotive | cypress.com/automotive |
| Clocks & Buffers | cypress.com/clocks |
| Interface | cypress.com/interface |
| Internet of Things | cypress.com/iot |
| Memory | cypress.com/memory |
| Microcontrollers | cypress.com/mcu |
| PSoC | cypress.com/psoc |
| Power Management ICs | cypress.com/pmic |
| Touch Sensing | cypress.com/touch |
| USB Controllers | cypress.com/usb |
| Wireless Connectivity | cypress.com/wireless |

## PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

## Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

## Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2015-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.