AN99218

FM MCU 的多功能串行接口

作者： Edison Zhang

相关器件系列：FM0+、FM3、FM4

相关代码示例：无

相关应用笔记：无

AN99218 介绍的是多功能串行（MFS）接口的各种模式。所介绍的信息能够帮助您安排 MFS 模块并快速运行所有 FM MCU。

目录

1 简介 1
2 UART 2
2.1 特性 2
2.2 数据格式 3
2.3 中断因素和时序 4
2.4 底层 API 7
2.5 示例代码 8
3 CSIO（SPI） 10
3.1 特性 10
3.2 传输模式 11
3.3 串行定时器 11
3.4 芯片选择功能 12
3.5 中断因素和时序 13
3.6 底层 API 16
3.7 示例代码 17
4 I²C 19
4.1 特性 19
4.2 协议 20
4.3 数据传输时序 23
4.4 底层 API 24
4.5 示例代码 25
5 LIN 29
5.1 特性 29
5.2 数据格式 30
5.3 通信系统 31
5.4 操作时序参数 31
5.5 底层 API 33
5.6 示例代码 34
6 总结 36
文档修订记录 37
全球销售和设计支持 38
产品 38
PSoC®解决方案 38
赛普拉斯开发者社区 38
技术支持 38

1 简介

串行通信接口（SCI）是最常见的通信接口。它包括通用异步收发器（UART）、串行外设接口（SPI）、I2C 总线（I²C）和局部互联网络（LIN）。几乎所有微控制器制造商为这些接口提供了独立的内置外设。FM 系列微控制器包含一个内置的 MFS 接口，可以将其配置为带有用户设置的 UART、时钟同步串行接口 — CSIO （SPI）、I²C 和 LIN，以便为该应用提供更好的灵活性和便捷性。

本应用笔记介绍了 MFS 模块，以及如何将 MFS 与外设驱动程序库（PDL）一起使用。另外，它也显示了每种通信协议的数据格式，然后对中断时序进行了说明。该基本信息能够帮助您加深对 MFS 模式工作方式的了解。另外，本应用笔记还提供了使用 PDL 的一些示例，用于说明如何实现每个模式的数据传输和接收程序。

2 UART

UART 是通用的串行数据通信接口，用于同外部器件进行异步通信（启动/停止同步）。

将 SMR 寄存器中的 MD 位设置为 b'000 时，可以配置 UART 模式。

位 7	位 6	位 5	说明
0	0	0	工作模式 0（异步正常模式）
0	0	1	工作模式 1（异步多处理器模式）
0	1	0	工作模式 2（时钟同步模式）
0	1	1	工作模式 3（LIN 通信模式）
1	0	0	工作模式 4（I^2C 模式）
非上述位			设置被禁止

2.1 特性

- 支持全双工操作
- 15 位波特率选项 [1]
- 5 到 9 位数据长度选项
- 支持非归零（NRZ）和反向的 NRZ 数据格式
- 支持 MSB/LSB 传输方向
- 支持硬件流控制 [2]
- 支持接收错误检测
  - 帧错误
  - 溢出错误
  - 奇偶校验错误
- 中断请求
  - 发生接收完成、帧错误、溢出错误或奇偶校验错误事件时，将生成一个接收中断请求
  - 发送数据为空或发送总线闲置时，将生成一个发送中断请求
  - 发送 FIFO 为空时，将生成一个发送 FIFO 中断请求
- 支持 DMA 传输
- 集成发送/接收 FIFO[3]

[1] 波特率发生器也可以由外部时钟提供脉冲。

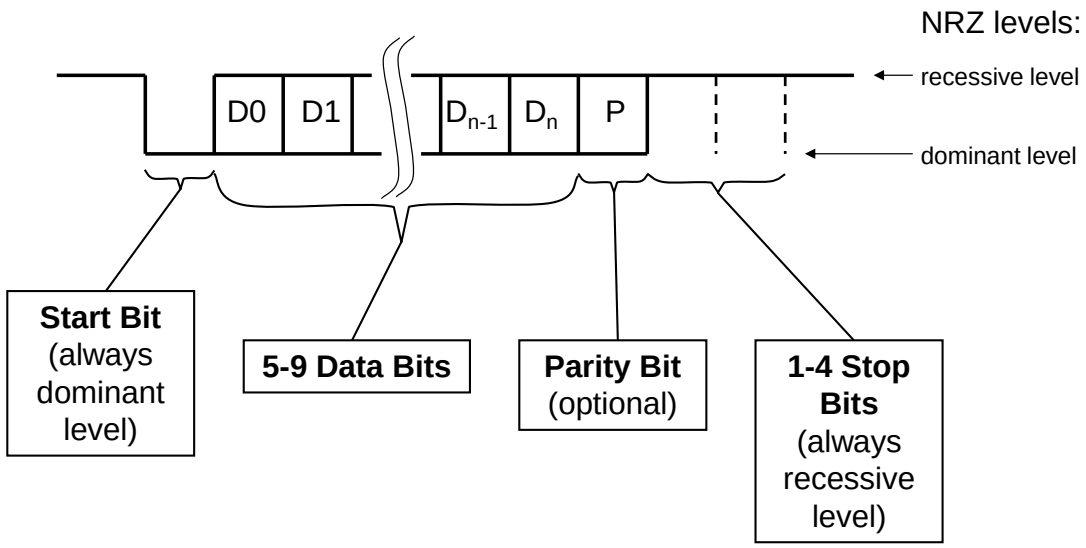
[2] 只有一些 MFS 通道拥有硬件流控制功能；请参考所用产品的数据手册。

[3] FIFO 容量根据产品类型而变化；请参考所用产品的数据手册。

2.2 数据格式

UART 帧开始于起始位，接着是数据位，最后是停止位，如图 1 所示。通过设置 ESCR 寄存器的 L2、L1 和 L0 位可以将数据长度选择为 5 位到 9 位。根据 PEN 位的设置情况，可以选择奇偶校验功能。如果使能奇偶校验功能，则通过使用 ESCR 寄存器的 P 位可以选择偶数奇偶校验或奇数奇偶校验。通过 SMR 寄存器的 SBL 位和 ESCR 寄存器的 ESBL 位，可以选择停止位的长度。

图 1. UART 数据格式



ESCR 寄存器配置了大多数数据格式设置。图 2 说明了 ESCR 寄存器的各个位。

图 2. ESCR 寄存器说明

位	15 ... 8	7	6	5	4	3	2	1	0
字段	（SSR）	FLWEN	ESBL	INV	PEN	P	L2	L1	L0
属性		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值		0	0	0	0	0	0	0	0

[位 6] ESBL：扩展停止位长度选择位

位	说明	
0	SMR:SBL=0	1 位
	SMR:SBL=1	2 位
1	SMR:SBL=0	3 位
	SMR:SBL=1	4 位

[位 2:0] L2、L1、L0：数据长度选择位

位 2	位 1	位 0	说明
0	0	0	8 位长度
0	0	1	5 位长度
0	1	0	6 位长度
0	1	1	7 位长度
1	0	0	9 位长度

[位 3] P：奇偶校验选择位

位	说明
0	偶数校验
1	奇数校验

[位 4] PEN：奇偶校验使能位

位	说明
0	禁用奇偶校验
1	使能奇偶校验

2.3 中断因素和时序

本节中的各个框图使用了以下各位和寄存器：

- TDRE（发送数据寄存器为空）位指出了发送数据寄存器中的数据是否为空。
- TBI（发送总线为闲置）位指出了 SOT 线是否被闲置。
- TIE（发送中断使能）位用于使能或禁用发送总线中断。
- TBIE（发送总线中断使能）位用于使能或禁用发送总线中断。
- TDR（发送数据寄存器）是用于发送串行数据的缓冲区寄存器。
- RDRF（接收数据寄存器为满）位用于指出接收数据寄存器是否已满。
- FRE（帧错误）位用于指出某个帧错误，接收数据的停止位为 0 时将发生该错误。
- ORE（溢出错误）位用于指出一个溢出错误，在读取接收数据前接收下一个数据时会发生这种错误。
- REC（清除接收错误）位用于清除已接收到的错误。
- RIE（接收中断使能）位用于使能或禁用发送总线中断。
- RDR（接收数据寄存器）是用于接收串行数据的缓冲区寄存器。

图 3 显示的是未使用 FIFO 时的 UART 数据发送时序框图。

- 当 TDR 为空（TDRE = 1）时，如果使能发送中断（TIE = 1），将生成一个发送中断请求。然后，可以将传输数据写入到 TDR 内，而且 TDRE 位为 0。
- TDR 先被加载到发送移位寄存器，然后数据中每一位都将依次被移出到 SOT 引脚上。传输数据的第一位被移出到 SOT 引脚后，TDRE 位为 1。如果 TIE = 1，将生成一个发送中断请求。然后可以再次将随后的数据写入到 TDR 内。
- 第一个数据的所有位通过 SOT 引脚移出，并且第二个数据的第一位通过 SOT 引脚移出后，TDRE 位将变为 1，以表示发送移位寄存器为空。
- 在第二个数据的所有位通过 SOT 引脚移出后，便完成 2 字节数据的发送，并且 TBI 位变为 1。如果使能发送总线中断（TBIE = 1），将生成一个发送总线中断请求。

图 3．UART 发送时序框图

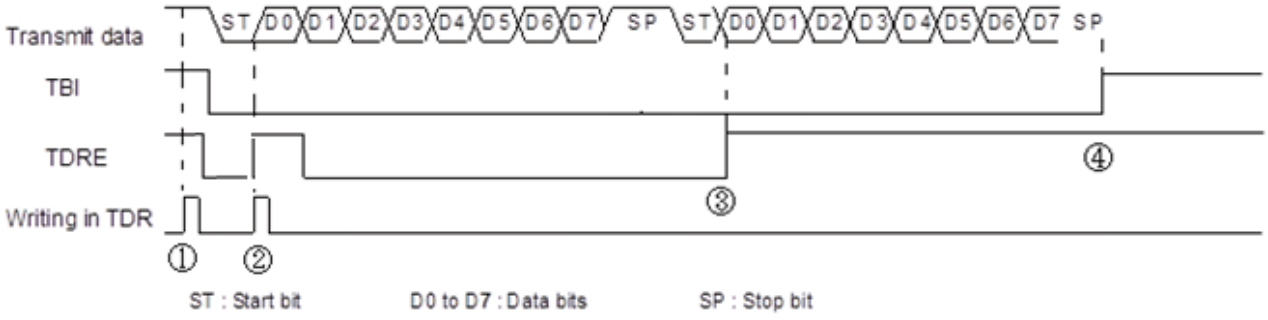


图 4 显示的是未使用 FIFO 时的 UART 数据接收时序框图。将接收到的数据保存在 RDR 内时，RDRF 位将被设置为 1。如果使能接收中断（RIE = 1），将生成一个接收中断请求。读取 RDR 中的数据后，会自动清除 RDRF 位。但是设置 RDRF 位时，如果发生帧错误（FRE = 1）或溢出错误（ORE = 1），则接收到的数据无效，并且需要将 REC 位设置为 1 来清除各个错误。

图 4. UART 接收时序图

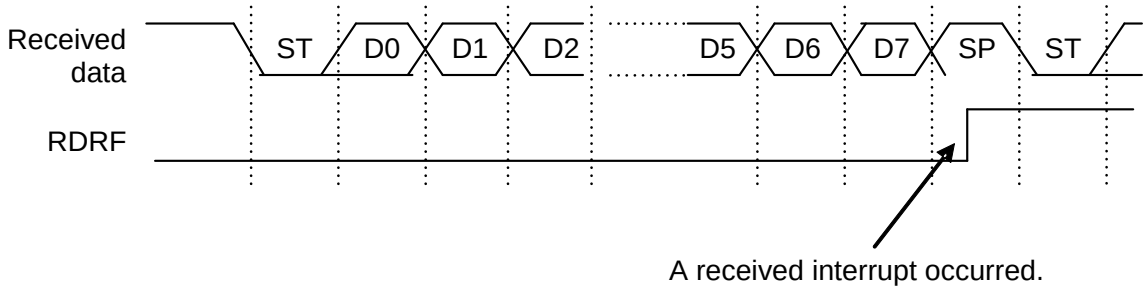


图 5 显示的是使能 FIFO 时的数据发送时序框图。它将使用发送 FIFO 来发送 5 字节数据。

1. 将 FIFO 发送数据请求位设置为 1（FDRQ = 1）时，如果使能 FIFO 发送中断（FTIE = 1），将发生一次发送中断。
2. 三个字节被写入到发送 FIFO 内，但由于 TDR 为空，因此第一个字节被传输到 TDR。然后，应该手动将 FDRQ 位清除为 0。这时，FIFO 中的数据计数为 2，如 FBYTE 寄存器所示。
3. 在 FIFO 为空而且数据的第一位被移出移位寄存器后，FDRQ 位将变为 1，以表示 FIFO 为空。如果使能 FIFO 发送中断（FTIE = 1），将发生发送中断。
4. 两个字节被写入到发送 FIFO 内。然后，应该手动将 FDRQ 位清除为 0。这时，FIFO 中的数据计数将再次变为 2，如 FBYTE 寄存器所示。
5. 在 FIFO 为空，并且数据的第一位被移出移位寄存器后，FDRQ 位将变为 1 以表示 FIFO 为空。如果使能 FIFO 发送中断（FTIE = 1），将发生一次发送中断。
6. 在 TDR 为空，并且数据的第一位被移出移位寄存器后，TDRE 位将变为 1。

   当移出移位寄存器中的所有数据位时，TBI 位被设置为 1，用于表示已经完成所有数据传输。

图 5. 使能 FIFO 时的 UART 发送时序框图

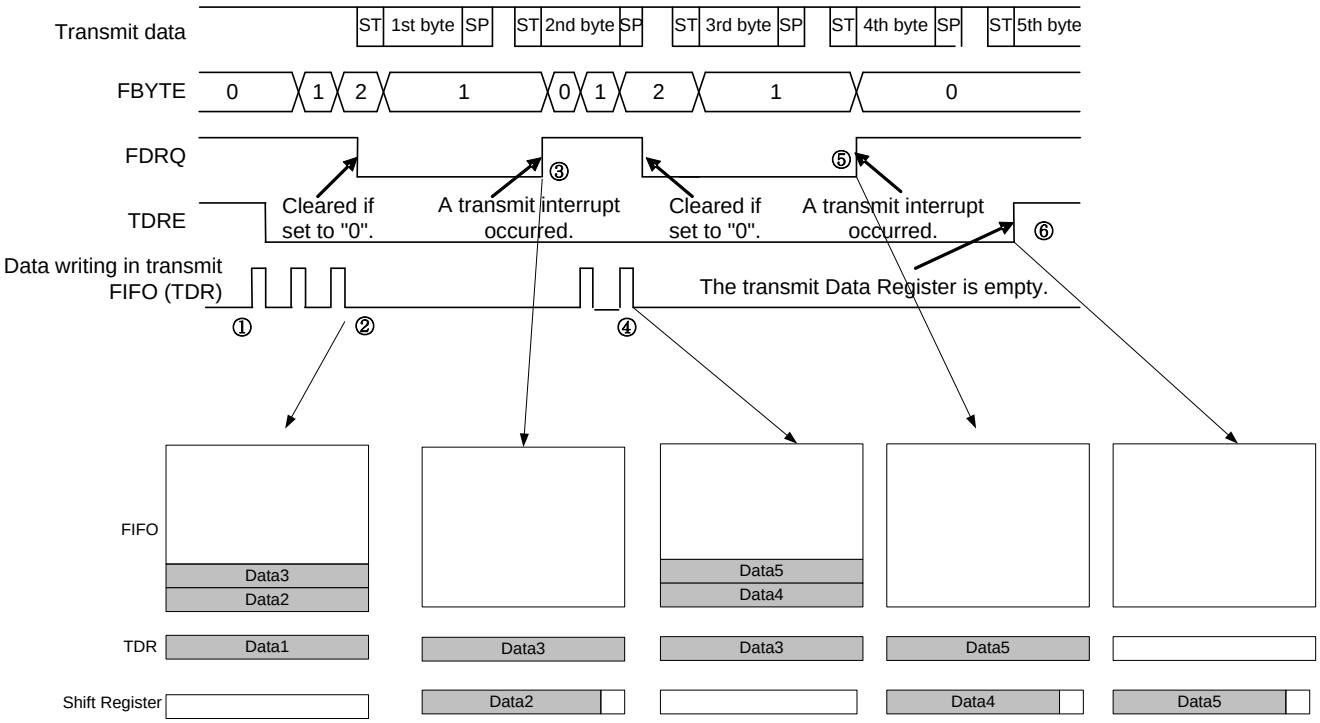
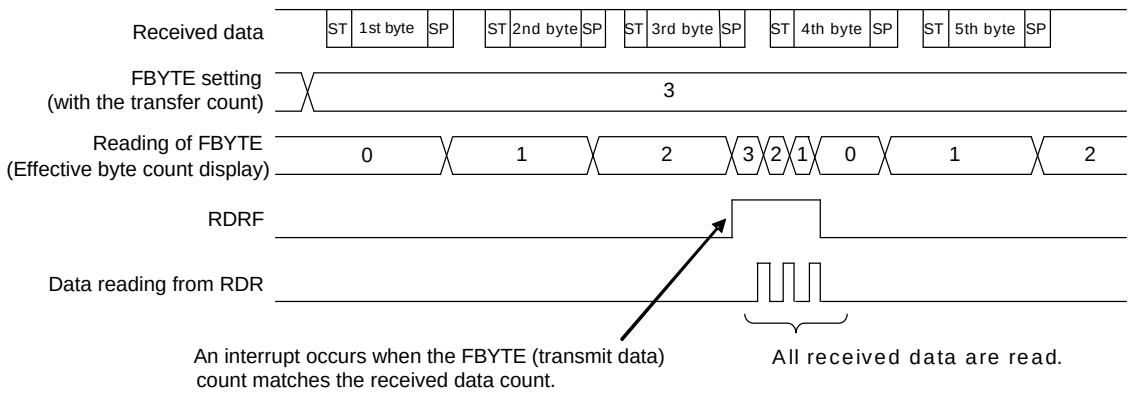


图 6 显示的是使能 FIFO 时的数据接收时序框图。

1. 应该在 FBYTE 寄存器中设置 FIFO 匹配计数。
2. 接收数据开始后，已接收到的数据将按顺序被保存在 FIFO 内。当 FIFO 中的数据计数与 FBYTE 相匹配时，RDRF 位将被设置为 1。如果 RIE 位被设置为 1，将发生接收中断。
3. 只接收到一个字节而没有其他数据时，如果使能了接收 FIFO 闲置检测（FRIIE = 1），并且该接收闲置状态持续多于 8 个波特率时钟，则 RDRF 位将被设置为 1。
4. 读取完 FIFO 中的所有数据后，RDRF 位将自动被清除为 0。

   如果接收数据计数超过接收 FIFO 的最大限制，将发生溢出错误。

图 6. 使能 FIFO 时的 UART 接收时序框图



2.4 底层 API

下面显示的是 PDL 的 UART 驱动程序 API，它被放置在 mfs.c/h 文件内。

- 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()用于初始化 UART 模式的 MFS 实例，它具有#stc_mfs_uart_config_t 类型的 Mfs_Uart_Init #pstcConfig 参数。该函数仅设置了基本的 UART 配置。Mfs_Uart_DeInit()用于复位与 MFS UART 相关的所有寄存器。

Mfs_Uart_EnableIrq()使能由枚举类型#en_uart_irq_sel_t 所选择的 UART 中断源。
Mfs_Uart_DisableIrq()禁用由枚举类型#en_uart_irq_sel_t 所选择的 UART 中断源。

Mfs_Uart_SetBaudRate()在初始化 UART 后，该函数允许更改 UART 波特率。

Mfs_Uart_EnableFunc()使能由参数 Mfs_Uart_EnableFunc#enFunc 所选择的 UART 功能，而 Mfs_Uart_DisableFunc()则禁用该 UART 功能。

Mfs_Uart_GetStatus()获取由 Mfs_Uart_GetStatus#enStatus 所选择的状态，而 Mfs_Uart_ClrStatus()清除已选的 UART 状态；某些状态只能通过硬件自动清除。

Mfs_Uart_SendData()将数据写入到 UART 传输缓冲区内，而 Mfs_Uart_ReceiveData()则读取 UART 接收缓冲区中的数据。

Mfs_Uart_ResetFifo()复位 UART 硬件 FIFO。

Mfs_Uart_SetFifoCount()在初始化 UART 后，该函数允许更改 FIFO 大小。

Mfs_Uart_GetFifoCount()获取 FIFO 中的当前数据计数。

2.5 示例代码

根据底层驱动程序 API，这里提供了一个示例用于说明如何使用中断标志轮询方法通过 UART 传输数据。它使用 UART 通道 0 来传输 10 个字节，然后接收 10 个字节。

在使用 UART 前，请根据下面的内容配置 UART 的引脚功能：

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

然后配置 UART 配置结构，并初始化 UART 通道。

- 波特率：115200
- 数据长度：8 位
- 奇偶校验：无
- 停止位长度：1 位
- H/W 流控制：无

```
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
}
```

以下代码将通过 SOT0_0 发送 10 个字节。

```
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);
```

以下代码会在 SIN0_0 端接收 10 个字节。

```
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);
```

注释： 在 PDL 项目中，请确保在使用 MFS 通道前已经使能了“PDL_PERIPHERAL_ENABLE_MFSx”的定义。

3 CSIO（SPI）

CSIO 是通用的串行数据通信接口（支持 SPI），通过它可以与外部器件进行同步通信。它也集成了发送/接收 FIFO。

将 SMR 寄存器中的 MD 位设置为 b'010 时，可以配置 CSIO 模式。

位 7	位 6	位 5	说明
0	0	0	工作模式 0（异步正常模式）
0	0	1	工作模式 1（异步多处理器模式）
0	1	0	工作模式 2（时钟同步模式）
0	1	1	工作模式 3（LIN 通信模式）
1	0	0	工作模式 4（I^2C 模式）
非上述位			设置被禁止

3.1 特性

- 支持全双工操作
- 时钟同步（无起始/停止位）
- 主设备/从设备功能
- 在主设备和从设备模式下支持 SPI
- 15 位波特率选项 [1]
- 5 到 16 位数据长度选项
- 支持 MSB/LSB 传输方向
- 支持芯片选择功能 [2]
  - 4 通道控制（单控制、循环控制）
  - 可以更改设置/保持/取消选择时间
  - 可以将各个通道设置为有效电平
- 支持由串行定时器触发的串行数据传输 [2]
- 支持接收错误检测
  - 溢出错误
- 中断请求
  - 发生接收完成或溢出错误事件时，将生成一个接收中断请求
  - 发送数据为空或发送总线闲置时，将生成一个发送中断请求
  - 发送 FIFO 为空时，将生成一个发送 FIFO 中断请求
  - 当串行定时器寄存器（STMR）达到某一数值时，串行定时器将生成状态中断请求
  - 串行定时器比较寄存器（STMCR）与定时器的计数器值相匹配时，将生成中断请求
- 集成了发送/接收 FIFO[3]

[1] 波特率发生器也可以由外部时钟提供脉冲。

[2] 只有一些 FM 产品才有芯片选择和串行定时器功能；请参考所用产品的数据手册。

[3] FIFO 容量会根据产品类型而变化；请参考所用产品的数据手册。

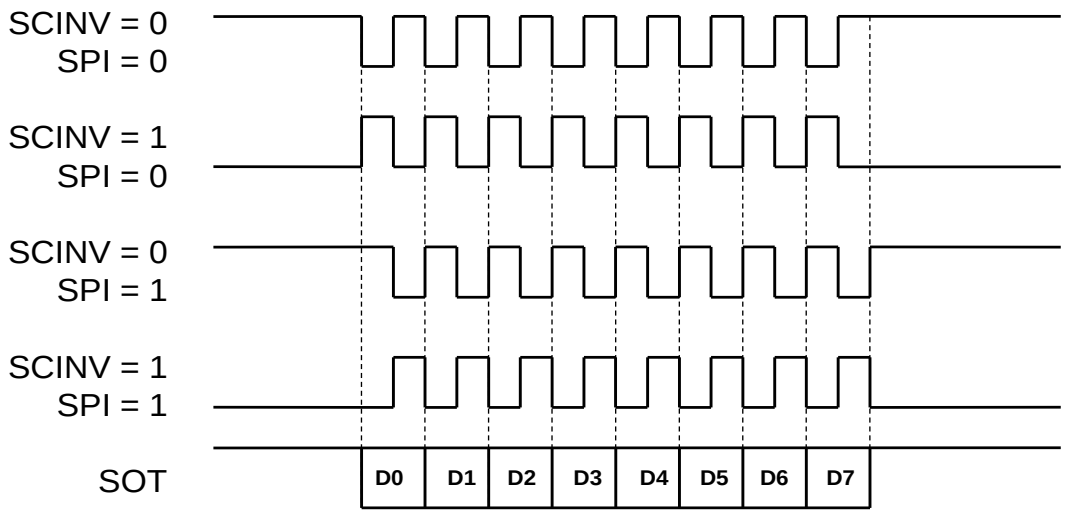
3.2 传输模式

存在四种传输模式适用于 CSIO 通信，通过 SCINV 和 SPI 位可以配置该通信。这些模式包括应用程序中的所有同步通信时序。

项目	模式 0（SCINV = 0、SPI = 0）	模式 1（SCINV = 1、SPI = 0）	模式 2（SCINV = 0、SPI = 1）	模式 3（SCINV = 1、SPI = 1）
串行时钟（SCK）的信号标志电平	“高”	“低”	“高”	“低”
发送数据输出时序	SCK 信号下降沿	SCK 信号上升沿	SCK 信号上升沿	SCK 信号下降沿
接收数据采样	SCK 信号上升沿	SCK 信号下降沿	SCK 信号下降沿	SCK 信号上升沿
数据长度	5 到 16 位	5 到 16 位	5 到 16 位	5 到 16 位

模式 0 和模式 1 被称为“CSIO 模式”，模式 2 和模式 3 被称为“SPI 模式”。图 7 显示的是这些传输模式的时序框图。

图 7. CSIO 和 SPI 传输模式



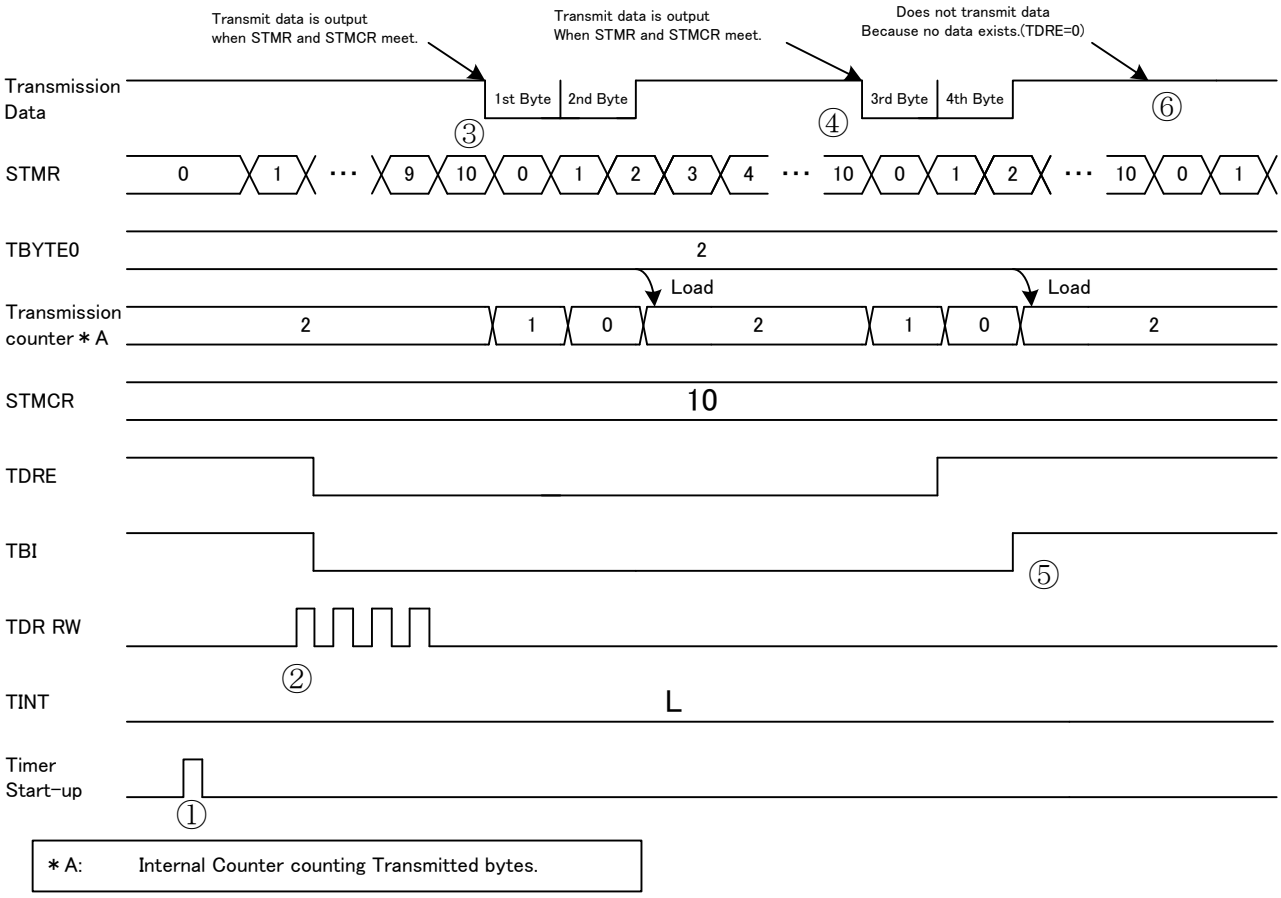
3.3 串行定时器

CSIO 模块集成了一个串行定时器，该定时器可以按照一定间隔触发 CSIO 数据传输。图 8 显示了一个使用串行定时器触发 CSIO 数据传输的示例。

1. 使用串行定时器前，应先设置 STMCR 中的计数比较值并传输 FBYTE0 寄存器中的字节计数值。然后再启动串行定时器。
2. 四字节的数据将被写入到发送 FIFO 内，但没有立即传输数据。
3. 定时器会根据定时器时钟从 0 开始计数。如果当前定时器计数值（由 STMR 反映）与 STMCR 的值相匹配，那么两个字节将被传输（因为 TBYTE = 2），定时器的计数值将被复位为 0。
4. 定时器会根据定时器时钟从 0 开始计数。同样，如果当前定时器计数值（由 STMR 反映）与 STMCR 的值相匹配，那么两个字节将被传输（因为 TBYTE = 2），定时器的计数值将被复位为 0。
5. 完成传输所有四个数据后，TBI 位将被设置为 1。

6. 如果当前定时器计数值（由 STMR 反映）再次与 STMCR 的值匹配，将不会传输任何数据，因为在 TDR 或发送 FIFO 中不存在任何数据。

图 8. 串行定时器操作时序



注意：

1. 只有部分 FM 产品才具备串行定时器功能；请参考所用产品的数据手册。
2. 对于具备 CSIO 串行定时器的产品，CSIO 串行定时器仅在 CSIO 主设备模式下运行。

3.4 芯片选择功能

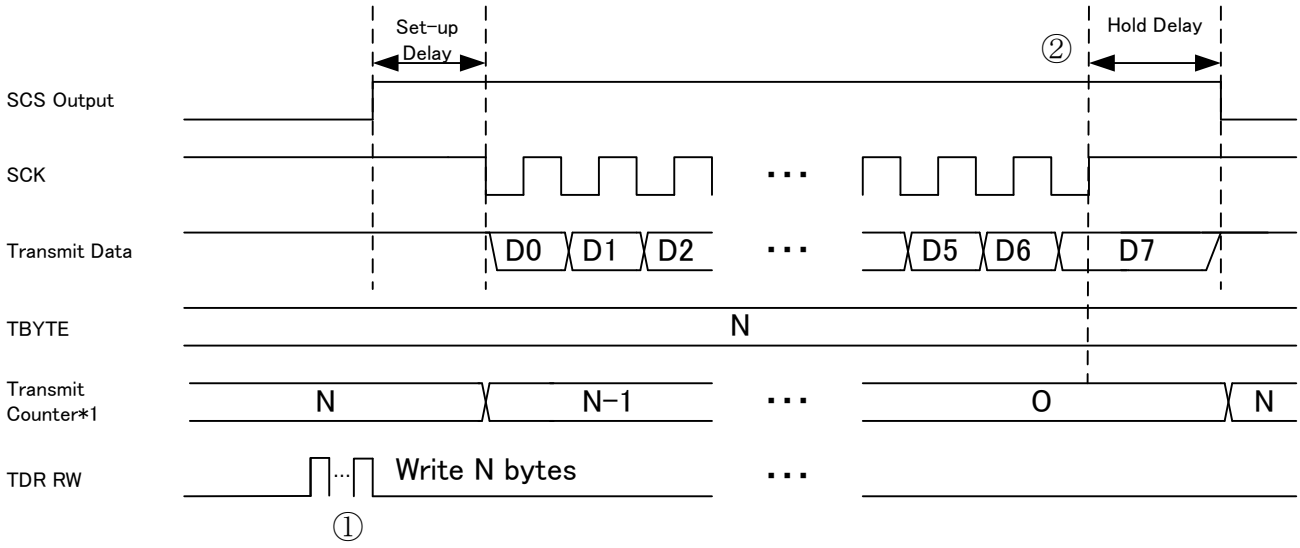
CSIO 模块具有芯片选择引脚（SCS 引脚），用于控制是否能够进行数据传输。主设备模式和从设备模式均支持芯片选择功能，但只有 SCS0 引脚在从设备模式下能够作为芯片选择引脚使用。

图 9 显示的是在主设备传输模式 0（MS = 0、SPI = 0、SCINV = 0）下使用芯片选择引脚传输 N 个字节的时序框图。

1. 启动数据传输前，应该在 TBYTE 中指定数据传输计数值大小。然后，通过设置串行芯片选择控制状态寄存器（SCSCR）中的 SST 和 SED 位，选择芯片选择引脚。通过将相应的 CSEN 位设置为 1，使能芯片选择功能。通过将 TEX 位设置为 1，使能数据传输。最后，可以将 N 个字节写入到发送 FIFO 内，经过设置延迟时间后开始传输数据。
2. 如果传输完 N 个字节，再经过一个保持延迟时间后，SCK 和 SCS 会变为高电平。

   可以通过更改 SCSCR 中 CSLVL 位的值来设置无效状态下 SCS 引脚的电平，但只有 SCS0 才有 CSLVL 位。设置延迟和保持延迟时间可通过串行芯片选择时序寄存器（SCSTR）设置

图 9. 使用芯片选择功能进行 CSIO 数据传输



*1:Internal Counter counting transmit bytes.

注释： 只有部分 FM 产品才有芯片选择功能；请参考所用产品的数据手册。

3.5 中断因素和时序

本节中的各个框图使用了以下各位和寄存器：

- TDRE（发送数据寄存器为空）位指出了发送数据寄存器中的数据是否为空。
- TBI（发送总线为闲置）位指出了 SOT 线是否被闲置。
- TIE（发送中断使能）位用于使能或禁用发送总线中断。
- TBIE（发送总线中断使能）位用于使能或禁用发送总线中断。
- TDR（发送数据寄存器）是用于发送串行数据的缓冲区寄存器。
- RDRF（接收数据寄存器为满）位用于指出接收数据寄存器是否已满。
- ORE（溢出错误）位用于指出一个溢出错误，在读取接收数据前接收下一个数据时会发生这种错误。
- REC（清除接收错误）位用于清除已接收到的错误。
- RIE（接收中断使能）位用于使能或禁用发送总线中断。
- RDR（接收数据寄存器）是用于接收串行数据的缓冲区寄存器。

图 10 显示的是未使用 FIFO 时的 CSIO 数据发送时序框图。

1. 当 TDR 为空（TDRE = 1）时，如果使能发送中断（TIE = 1），将生成一个发送中断请求。然后，可以将传输数据写入到 TDR 内，而且 TDRE 位为 0。
2. TDR 先被加载到发送移位寄存器，然后数据中每一位都将依次被移出到 SOT 引脚上。将 TDR 中的数据加载到发送移位寄存器内后，TDRE 位将为 1。如果 TIE = 1，则会生成一个发送中断请求。然后可以再次将随后的数据写入到 TDR 内。
3. 第一个数据的所有位通过 SOT 引脚移出，下一个数据将再次从 TDR 加载到发送移位寄存器内。然后，TDRE 位将变为 1，用于指示发送移位寄存器为空。

4. 在第二个数据的所有位被移出到 SOT 引脚后，便完成了 2 字节数据的发送，并且 TBI 位变为 1。如果使能发送总线中断（TBIE = 1），将生成一个发送总线中断请求。

图 10. CSIO 数据传输的时序框图

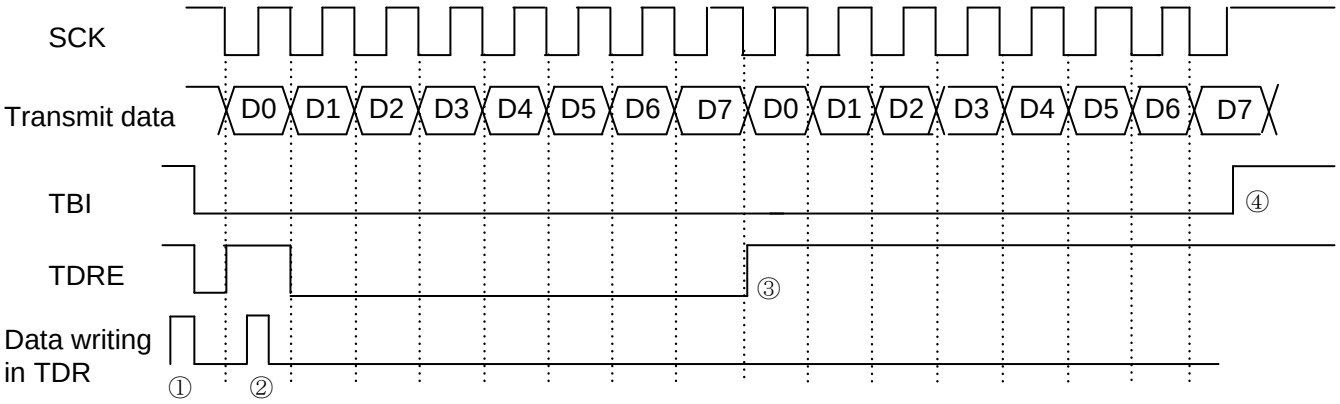


图 11 显示的是未使用 FIFO 时的 CSIO 数据接收时序框图。将接收到的数据保存在 RDR 内时，RDRF 位将被设置为 1。如果使能接收中断（RIE = 1），将生成一个接收中断请求。读取 RDR 中的数据后，会自动清除 RDRF 位。但是设置 RDRF 位时，如果发生溢出错误（ORE = 1），则接收到的数据无效，并且需要将 REC 位设置为 1 来清除各错误。

图 11. CSIO 数据接收时序框图

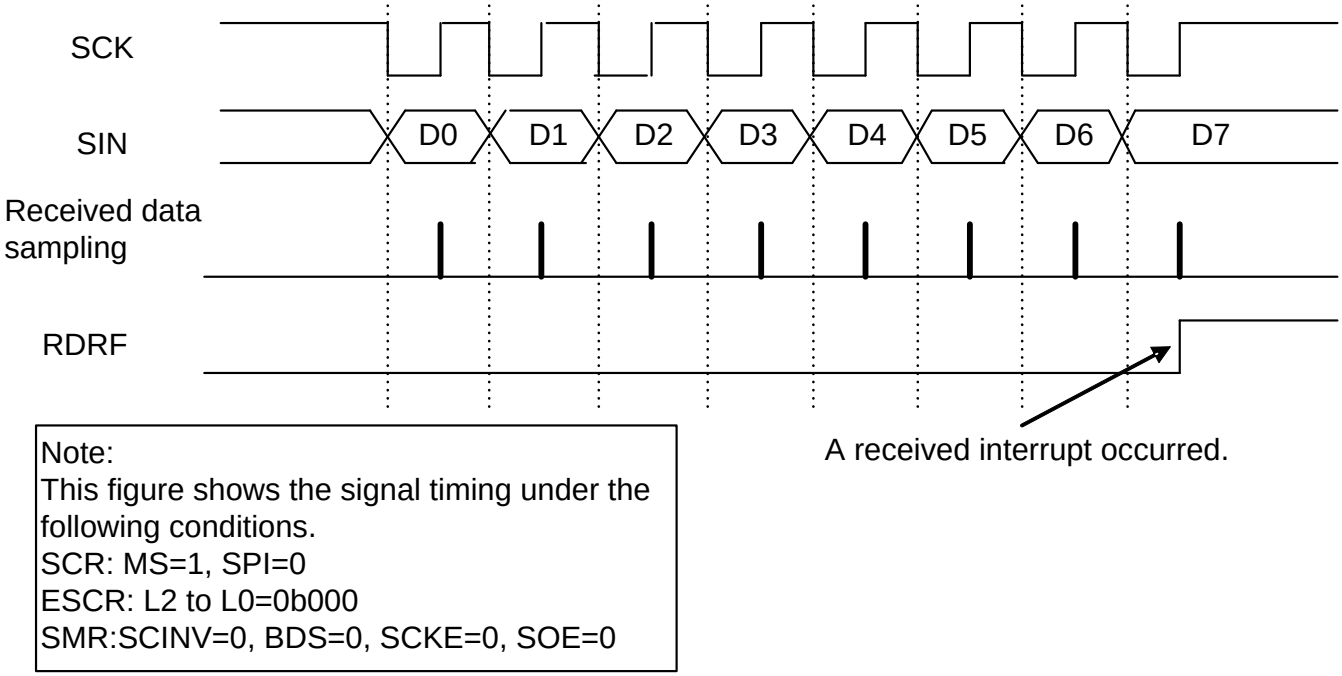


图 12 显示的是使能 FIFO 时的数据发送时序框图。它将使用发送 FIFO 来发送 4 字节数据。

1. FDRQ = 1 时，如果使能了 FIFO 发送中断（FTIE = 1），将发生一个发送中断。
2. 三个字节被写入到发送 FIFO 内，但由于 TDR 为空，因此第一个字节将被传输到 TDR 内。然后，应该手动将 FDRQ 位清除为 0。这时，FIFO 中的数据计数为 2，如 FBYTE 寄存器所示。
3. 如果 FIFO 为空，那么 FDRQ 位将变为 1，从而表示 FIFO 的状态（空）。如果使能 FIFO 发送中断（FTIE = 1），将发生一次发送中断。

4. 一个字节将被写入到发送 FIFO 内。然后，应手动将 FDRQ 位清除为 0。这时，FIFO 中的数据计数为 1，如 FBYTE 寄存器所示。
5. 如果 FIFO 为空，那么 FDRQ 位将变为 1，从而表示 FIFO 的状态为空。如果使能 FIFO 发送中断（FTIE = 1），将发生一次发送中断。
6. 如果 TDR 为空，则 TDRE 位将变为 1。

当移出移位寄存器中的所有数据位时，TBI 位被设置为 1，用于表示已完成所有数据传输。

图 12. 使能 FIFO 时的 CSIO 数据发送时序框图

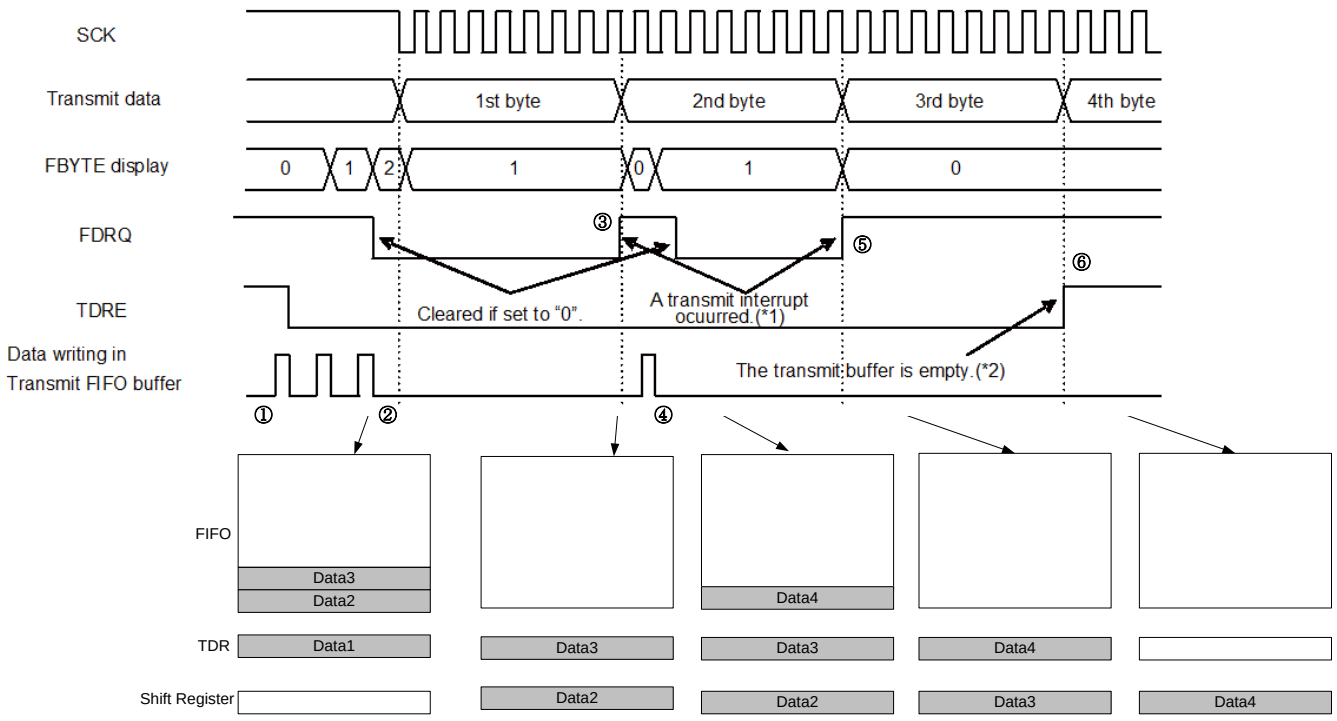
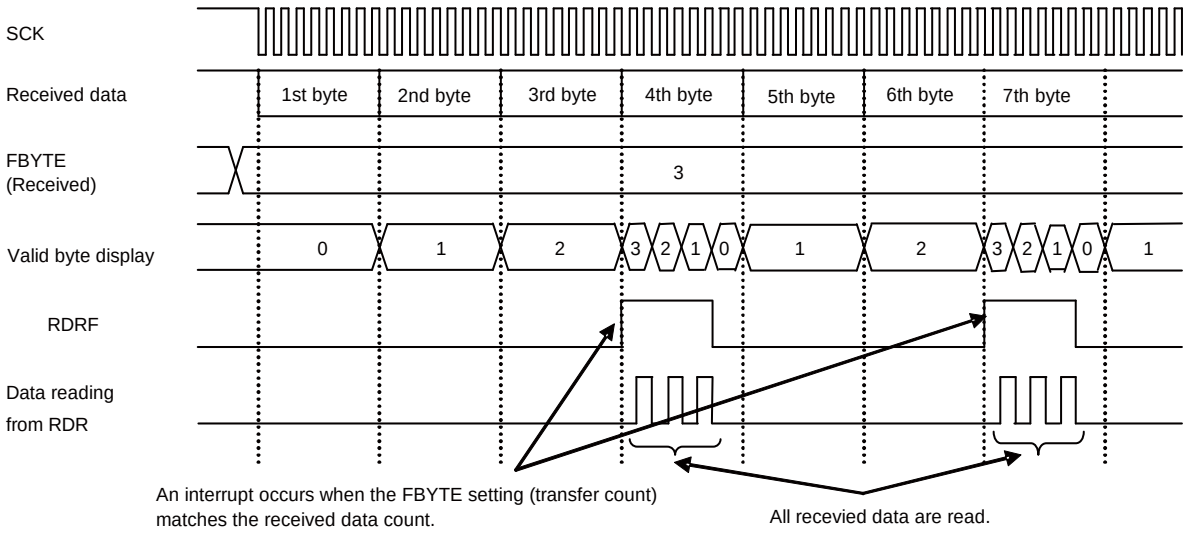


图 13 显示的是使能 FIFO 时的数据接收时序框图。

1. 应该在 FBYTE 寄存器中设置 FIFO 匹配计数。
2. 接收数据开始后，已接收到的数据将按顺序被保存在 FIFO 内。当 FIFO 中的数据计数值与 FBYTE 相匹配时，RDRF 位将被设置为 1。如果 RIE 位被设置为 1，将发生一次接收中断。
3. 只接收到一个字节而没有其他数据时，如果使能了接收 FIFO 闲置检测（FRIIE = 1），并且该接收闲置状态持续多于 8 个波特率时钟，则 RDRF 位将被设置为 1。
4. 读取完 FIFO 中的所有数据后，RDRF 位将自动被清除为 0。

   如果接收数据计数超过接收 FIFO 的最大容量，将发生溢出错误。

图 13. 使能 FIFO 时的 CSIO 数据接收时序框图



3.6 底层 API

下面显示的是 PDL 的 CSIO 驱动程序 API，它被放置在 `mfs.c/h` 文件内。

- 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() 使用#stc_mfs_csio_config_t 类型的 pstcConfig 参数为 MFS 实例初始化 CSIO 模式。Mfs_Csio_DeInit() 用于复位所有与 MFS CSIO 相关的寄存器。

Mfs_Csio_EnableIrq() 使能了由枚举类型#en_csio_irq_sel_t 所选择的 CSIO 中断源。
Mfs_Csio_DisableIrq() 禁用了由枚举类型#en_csio_irq_sel_t 所选择的 CSIO 中断源。

Mfs_Csio_SetBaudRate() 在初始化 CSIO 后能够更改 CSIO 波特率。

Mfs_Csio_SetTimerCompareValue() 能够更改 CSIO 串行定时器的比较值。

Mfs_Csio_SetCsTransferByteCount() 能够更改选定的芯片选择引脚的传输字节计数值。

Mfs_Csio_SetTimerTransferByteCount() 设置了传输过程（由串行定时器触发）的传输字节计数。

Mfs_Csio_SetCsHoldStatus() 设置了结束一次传输后 CS 引脚的保持状态。

Mfs_Csio_EnableFunc() 使能了由 Mfs_Csio_EnableFunc#enFunc 参数所选中的 CSIO 功能，
Mfs_Csio_DisableFunc() 则禁用了 CSIO 功能。

Mfs_Csio_GetStatus() 用于读取由 Mfs_Csio_GetStatus#enStatus 所选择的状态，
Mfs_Csio_ClrStatus() 则用于清除所选的 CSIO 状态。某些状态只能由硬件自动清除。

Mfs_Csio_SendData() 将一个字节数据写入到 CSIO 传输缓冲区内；Mfs_Csio_ReceiveData() 则读取 CSIO 接收缓冲区中一个字节的数据。在 Mfs_Csio_Init() 中配置数据的字节长度。

Mfs_Csio_ResetFifo() 复位了 CSIO 硬件 FIFO。

Mfs_Csio_SetFifoCount() 在初始化 CSIO 后能够更改 FIFO 的大小。Mfs_Csio_GetFifoCount() 用于读取 FIFO 中的当前数据计数值。

3.7 示例代码

根据底层驱动程序 API，这里提供了一个示例用于说明如何使用中断标志轮询方法通过 CSIO 传输数据。它使用 CSIO 通道 0 来传输 10 个字节，然后接收 10 个字节。

在使用 CSIO 前，请根据下面内容配置 CSIO 的引脚功能：

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

然后配置 CSIO 配置结构，并初始化 CSIO 通道。

- 模式：主设备模式
- 波特率：100 kbps
- 数据长度：8 位
- 方向：MSB 优先

```
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);
}
```

下面代码能够通过 SOT 引脚发送 10 个 CSIO 字节。

```
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);
```

下面代码能够通过 SIN 引脚接收 10 个 CSIO 字节。请注意，在同步通信中，即使主设备正在接收从设备发送的数据，时钟线也始终受主设备的控制。如果主设备需要接收从设备所发送的数据，那么应该发送模拟数据来生成时钟。

```
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 I^2C

I^2C 接口（I^2C 通信控制接口）支持 I^2C 总线并能够作为 I^2C 总线上的主设备/从设备器件运行。

将 SMR 寄存器中的 MD 位设置为 b'100 时，可以配置 I^2C 模式。

位 7	位 6	位 5	说明
0	0	0	工作模式 0（异步通用模式）
0	0	1	工作模式 1（异步多处理器模式）
0	1	0	工作模式 2（时钟同步模式）
0	1	1	工作模式 3（LIN 通信模式）
1	0	0	工作模式 4（I^2C 模式）
非上述位			设置被禁止

4.1 特性

- 主设备/从设备功能
- 15 位波特率选项
- 8 位数据长度
- 总线仲裁功能
- 从设备模式下的传送方向检测功能

- 生成和检测迭代启动条件功能
- 总线错误检测功能
- 从设备的 7 位地址
- 执行数据传送或检测总线错误时的中断生成
- 滤除串行时钟/串行数据输入总线时钟中频率为时钟 2 到 32 倍的噪声 [1]
- 支持 DMA 传输
- 中断请求
  - 发生接收完成或溢出错误事件时，将生成一个接收中断请求
  - 发送数据为空或发送总线闲置时，将生成一个发送中断请求
  - 发送 FIFO 为空时，将生成一个发送 FIFO 中断请求
  - 完成发送/接收数据、检测总线错误和 NACK 信号时，将生成状态中断请求
  - 检测停止条件和迭代启动条件
- 集成了发送/接收 FIFO[2]

[1] 只有部分 FM 产品具有噪声滤波器；请参考所用产品的数据手册。

[2] FIFO 容量根据产品类型而变化；请参考所用产品的数据手册。

4.2 协议

I^2C 帧始终包含一个启动条件、7 位从设备地址 + 1 位 R/W 位、数据和停止条件，具体内容请参考图 14。

如果 I^2C 主设备将数据发送到 I^2C 从设备内，那么会将 R/W 为设置为 0。发送启动条件信号和第一个字节后，主设备将收到由从设备发送的 ACK 位。然后，主设备会持续将数据发送给从设备，并在正常条件下每次完成发送字节时，主设备都将收到从设备的 ACK 位。I^2C 主设备发送完所有数据后，它会将一个停止条件信号发送到给设备，以表示数据传输已经完成。

如果 I^2C 主设备收到由 I^2C 从设备所发送的数据，那么会将 R/W 位设置为 1。发送启动条件信号和第一个字节后，主设备将收到由从设备发送的 ACK 位。然后，主设备可以读取来自 I^2C 从设备的数据，每次收到字节后，应该将一个 1 位的 ACK 发送给从设备。请注意，收到最后的数据后，主设备应将一个 1 位的 NACK 发送给从设备，用于通知从设备该数据是主设备要接收的最后一个数据。然后，主设备将发送停止条件信号，从而完成 I^2C 数据接收过程。

图 14. I²C 数据格式

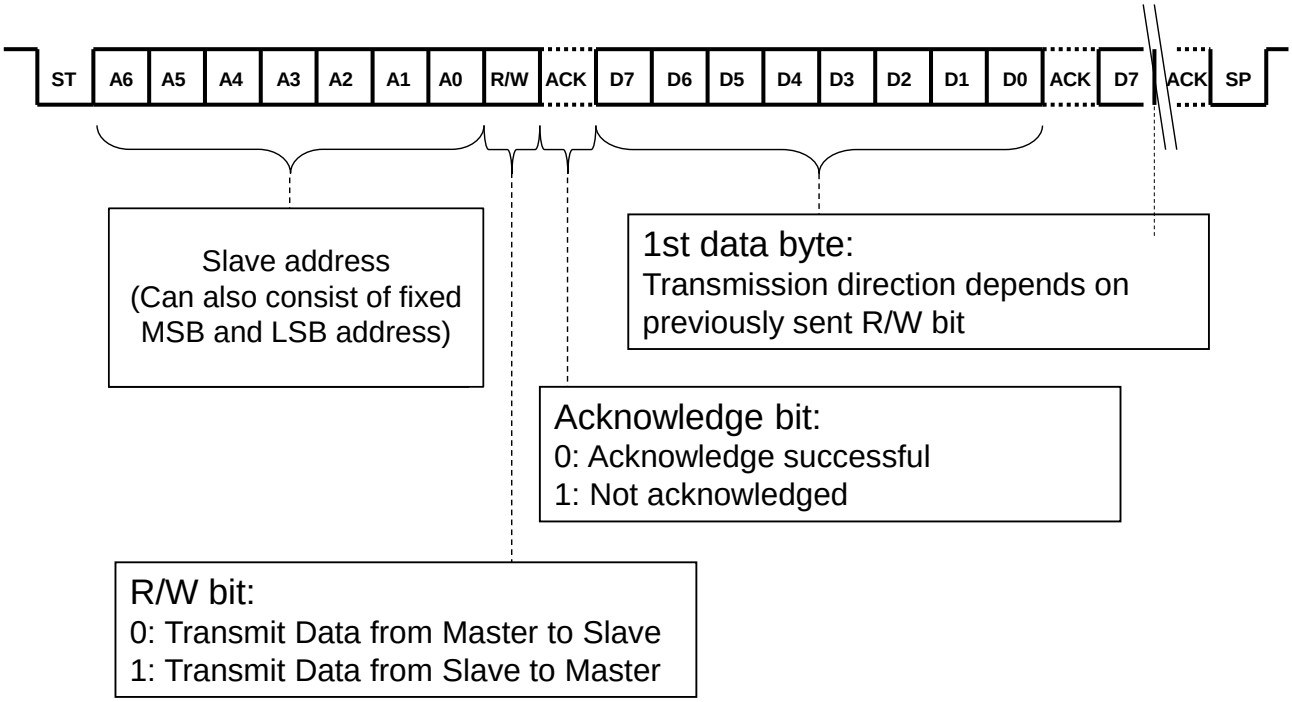


图 15 指的是 I²C 启动和停止条件的时序框图。

启动条件

- SCL（SCK）线为高电平时，如果在 SDA（SOT）线上存在一个下降沿，则表示 I²C 帧的开始。
- 在 FM MCU 的 I²C 模块中，如果 MSS = 0 和 ACT = 0，将 MSS 设置为 1 会生成 I²C 启动条件。然后，ACT 也自动被设置为 1，以表示已经进入主设备模式。

停止条件

- SCL（SCK）线为高电平时，如果在 SDA（SOT）线上存在一个上升沿，则表示 I²C 帧已经停止。
- 在 FM MCU 的 I²C 模块中，如果 MSS = 1 和 ACT = 1，将 MSS 设置为 0 会生成 I²C 停止条件。然后，ACT 也被自动设置为 0，用于表示已经进入停止模式。

图 15. I²C 启动和停止条件的时序框图

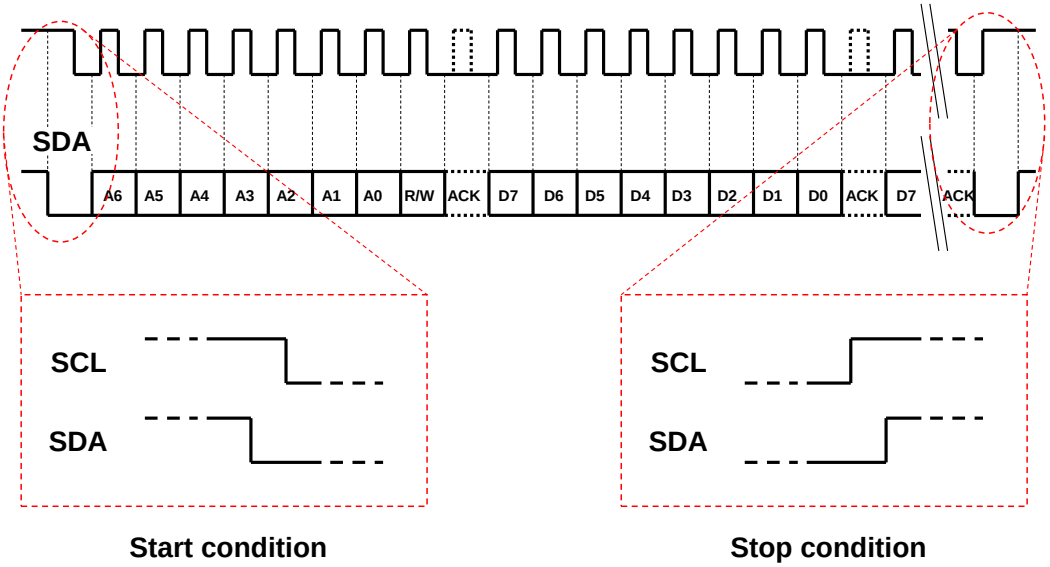


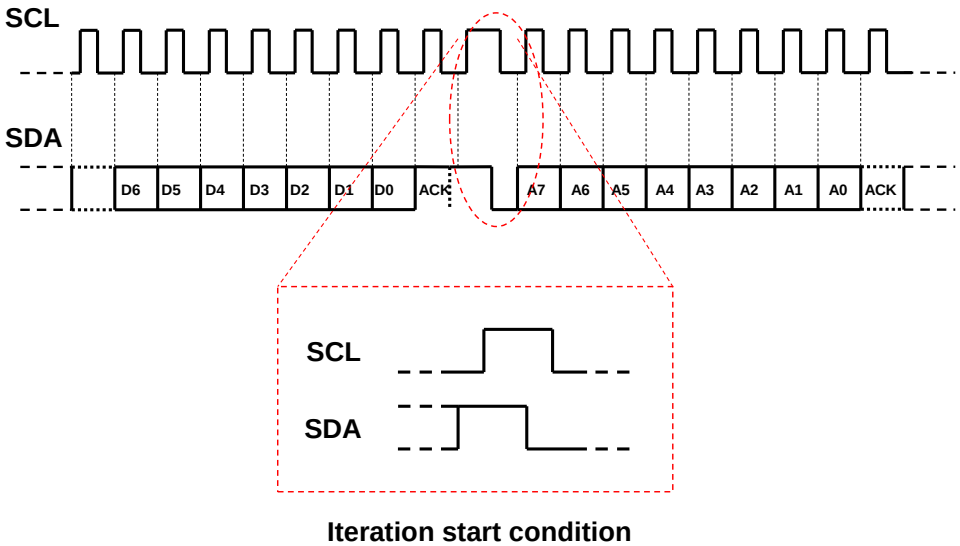
图 16 显示的是 I²C 迭代（重复）启动条件的时序框图。

迭代（重复）启动条件：

启动 I²C 通信时，如果主设备再次生成启动条件，可以称为“迭代启动条件”或“重复启动条件”。读取 I²C EEPROM 的数据时，可能使用该信号。

在 FM MCU 的 I²C 模块中，如果 MSS = 1 和 ACT = 1，则将 MSS 设置为 1 可生成重复启动条件。

图 16. I²C 重复启动条件的时序框图



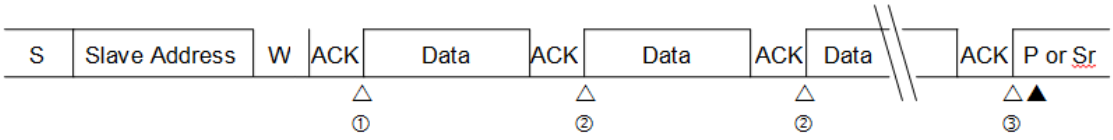
下面介绍的是 IBCR 寄存器中的 MSS 和 ACT 位：

MSS 位	ACT 位	状态
0	0	闲置
0	1	在从设备模式下，从设备地址与预留地址相匹配或从预留地址发送 ACK 信号。
1	0	当总线处于闲置（等待）状态时，I2C 器件将运行于主设备模式。
1	1	在主设备模式操作期间

4.3 数据传输时序

图 17 显示的是未使能 FIFO 时通过 I^2C 写入某些数据的时序框图。

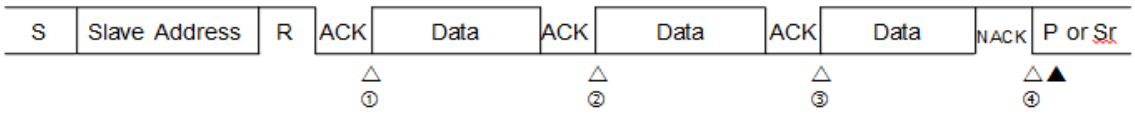
图 17. I^2C 数据传输框图



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".

图 18 显示的是未使能 FIFO 时通过 I²C 读取某些数据的时序框图。

图 18. I²C 数据接收框图



△: 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 底层 API

下面显示的是 PDL 的 I²C 驱动程序 API，它被放置在 `mfs.c/h` 文件内。

- 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()使用#stc_mfs_i2c_config_t 类型的 pstcConfig 参数为 MFS 实例初始化 I²C 模式。该函数只设置基本的 I²C 配置。Mfs_I2c_DeInit()用于复位所有与 MFS I²C 相关的寄存器。

Mfs_I2c_EnableIrq()使能了由枚举类型#en_i2c_irq_sel_t 所选定的 I²C 中断源；Mfs_I2c_DisableIrq()则禁用了由#en_i2c_irq_sel_t 枚举类型选定的 I²C 中断源。

Mfs_I2c_SetBaudRate()在初始化 I²C 波特率后能够更改 I²C。

Mfs_I2c_SendData()将一个字节数据写入到 I²C 发送缓冲区内；Mfs_I2c_ReceiveData()则从 I²C 接收缓冲区中读取一个字节数据。

Mfs_I2c_GenerateStart()生成了 I²C 启动信号。Mfs_I2c_GenerateRestart()则生成一个 I²C 重新启动信号。Mfs_I2c_GenerateStop()生成了 I²C 停止信号。

收到数据时，Mfs_I2c_ConfigAck()将配置 ACK 信号。收到 ACK 信号后，Mfs_I2c_GetAck()将读取 ACK 信号的状态。

Mfs_I2c_GetStatus()用于读取由 Mfs_I2c_GetStatus#enStatus 所选定的状态；Mfs_I2c_ClrStatus()则清除所选的 I²C 状态。某些状态只能由硬件自动清除。

Mfs_I2c_GetDataDir()用于读取在从设备模式下 I²C 的数据方向。

Mfs_I2c_ResetFifo()用于复位 I²C 硬件 FIFO。Mfs_I2c_SetFifoCount()在初始化 I²C 后能够更改 FIFO 大小。

Mfs_I2c_GetFifoCount()用于读取 FIFO 中当前数据计数值。

4.5 示例代码

根据底层驱动程序 API，这里提供的一个示例用于说明如何使用中断标志轮询方法通过 I²C 传输数据。它使用 I²C 通道 0 来传输 10 个字节，然后接收 10 个字节。

在使用 I²C 前，请根据下面的内容配置 I²C 的引脚功能：

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

然后配置 I²C 配置结构，并初始化 I²C 通道。

- 模式：主设备模式
- 波特率：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);
}
```

以下代码将通过 I²C 发送 10 字节的数据。这里假设器件地址为 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);
```

以下代码将通过 I^2C 读取 10 字节的数据。这里假设器件地址为 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

LIN 接口（LIN 通信控制接口版本 2.1）支持多项功能，从而能够兼容 LIN 总线。

将 SMR 寄存器中的 MD 位设置为 b'011 时，可以配置 LIN 模式。

位 7	位 6	位 5	说明
0	0	0	工作模式 0（异步通用模式）
0	0	1	工作模式 1（异步多处理器模式）
0	1	0	工作模式 2（时钟同步模式）
0	1	1	工作模式 3（LIN 通信模式）
1	0	0	工作模式 4（I^2C 模式）
并非上述位			设置被禁止

5.1 特性

- 支持 LIN 协议版本 2.1
- 支持主设备/从设备的操作
- 支持全双工操作
- 生成 LIN 间隔场（长度为 13 到 16 位）[1]
- 生成 LIN 间隔符（长度为 1 到 4 位）[1]
- 支持 LIN 间隔场检测功能 [1]
- 支持通过捕获输入的起始/结束边沿来检测 LIN 同步场 [1]
- 15 位波特率选项 [2]
- 8 位数据长度
- 支持所接收错误的检测功能
  - 帧错误检测
  - 溢出错误检测

- 中断请求
  - 发生接收完成、帧错误、溢出错误或奇偶校验错误事件时，将生成接收中断请求
  - 发送数据为空或发送总线闲置时，将生成一个发送中断请求
  - 发送 FIFO 为空时，将生成一个发送 FIFO 中断请求
- 支持 DMA 传输
- 集成了发送/接收 FIFO[3]

[1] 只有部分产品具有 LIN 模块；请参考所用产品的数据手册了解详情。

[2] 波特率发生器也可以由外部时钟提供脉冲。

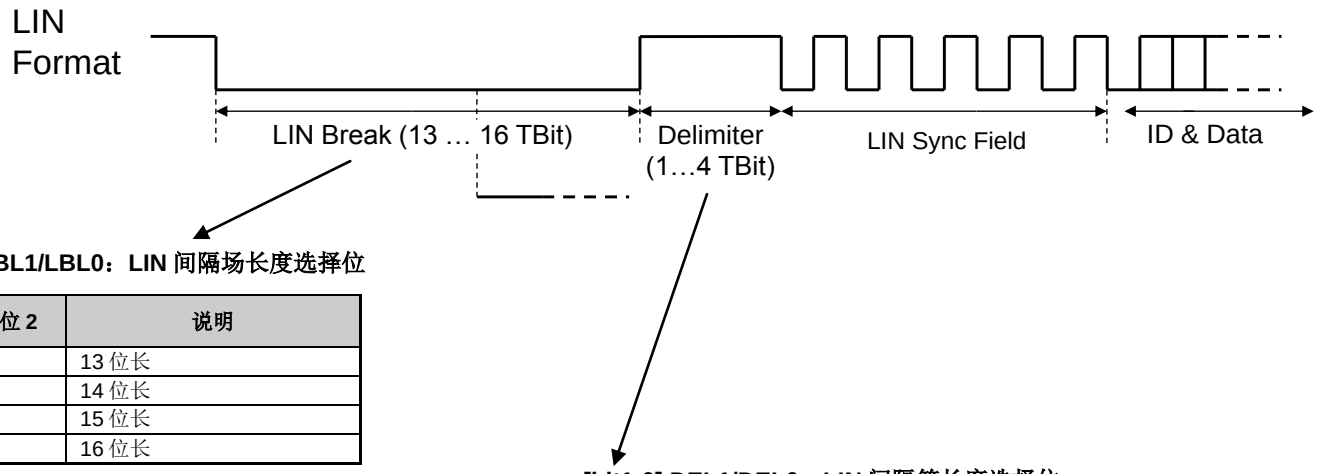
[3] FIFO 容量会因产品类型不同而变化；请参考所用产品的数据手册。

5.2 数据格式

LIN 帧由 LIN 间隔场、LIN 间隔符、LIN 同步场以及 ID 和数据字段组成，如图 19 所示。

在 LIN 主设备模式下，LIN 间隔场的长度可以通过 ESCR 寄存器中的 LBL0 和 LBL1 位设置，LIN 间隔符则可以通过 DEL0 和 DEL1 位设置。

图 19. LIN 数据格式



[bit3:2] LBL1/LBL0：LIN 间隔场长度选择位

位 3	位 2	说明
0	0	13 位长
0	1	14 位长
1	0	15 位长
1	1	16 位长

[bit1:0] DEL1/DEL0：LIN 间隔符长度选择位

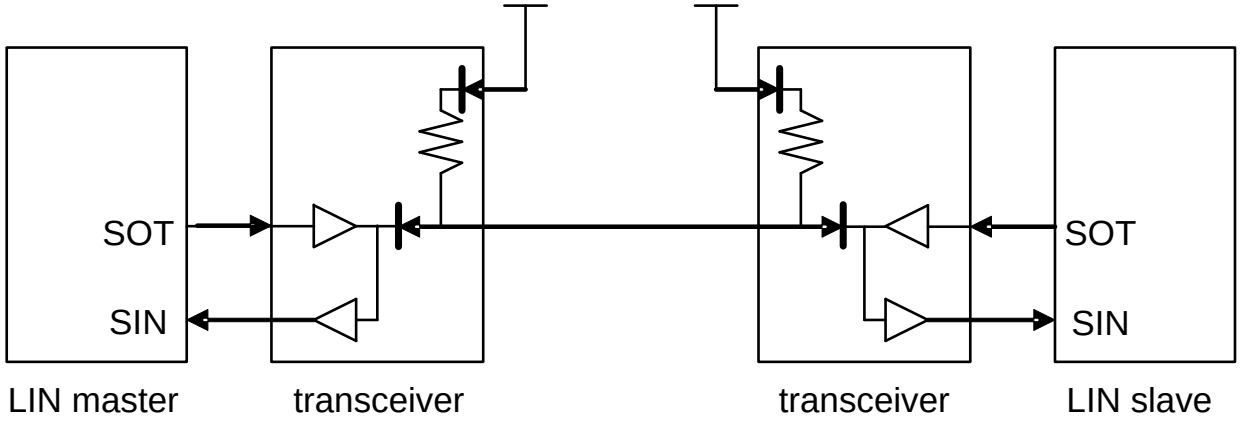
位 1	位 0	说明
0	0	1 位长
0	1	2 位长
1	0	3 位长
1	1	4 位长

在 LIN 从设备模式下，经过信号低电平的 11 位时间后可以检测到 LIN 间隔场。通过使用内部输入捕获单元（ICU）捕获 LIN 同步场，从而可以调整波特率。

5.3 通信系统

图 20 显示的是一个通信系统，它包括一个 LIN 主设备和一个 LIN 从设备（它们通过 LIN 收发器相连接）。

图 20. LIN 通信系统



5.4 操作时序参数

图 21 显示的是 LIN 主设备模式下的数据发送时序图。

1. 将 LIN 间隔场（LBR）位设置为 1 时，主设备会发送一个间隔场；然后 0x55（LIN 同步场）会被写入到 TDR 内，并在 LIN 间隔场结束后被发送。
2. LIN 同步场（0x55）的第一位通过 SOT 引脚被移出。TDRE 位被设置为 1，此时如果 TIE 位被设置为 1，则会生成一个发送中断。然后，ID 字节被写入到 TDR 内。
3. LIN 主设备可以接收它所发送的所有信息。此时会接收到 0x55，并对原始数据进行比较，从而检查所发送的数据是否正常。
4. ID 字段的第一位被发送时，TDRE 位会返回 1，如果 TIE 位被设为 1，将生成一个发送中断，然后可以传送第一个数据。

数据传输过程和 ID 字段传输是相同的。

图 21. LIN 主设备模式下数据传输的时序图

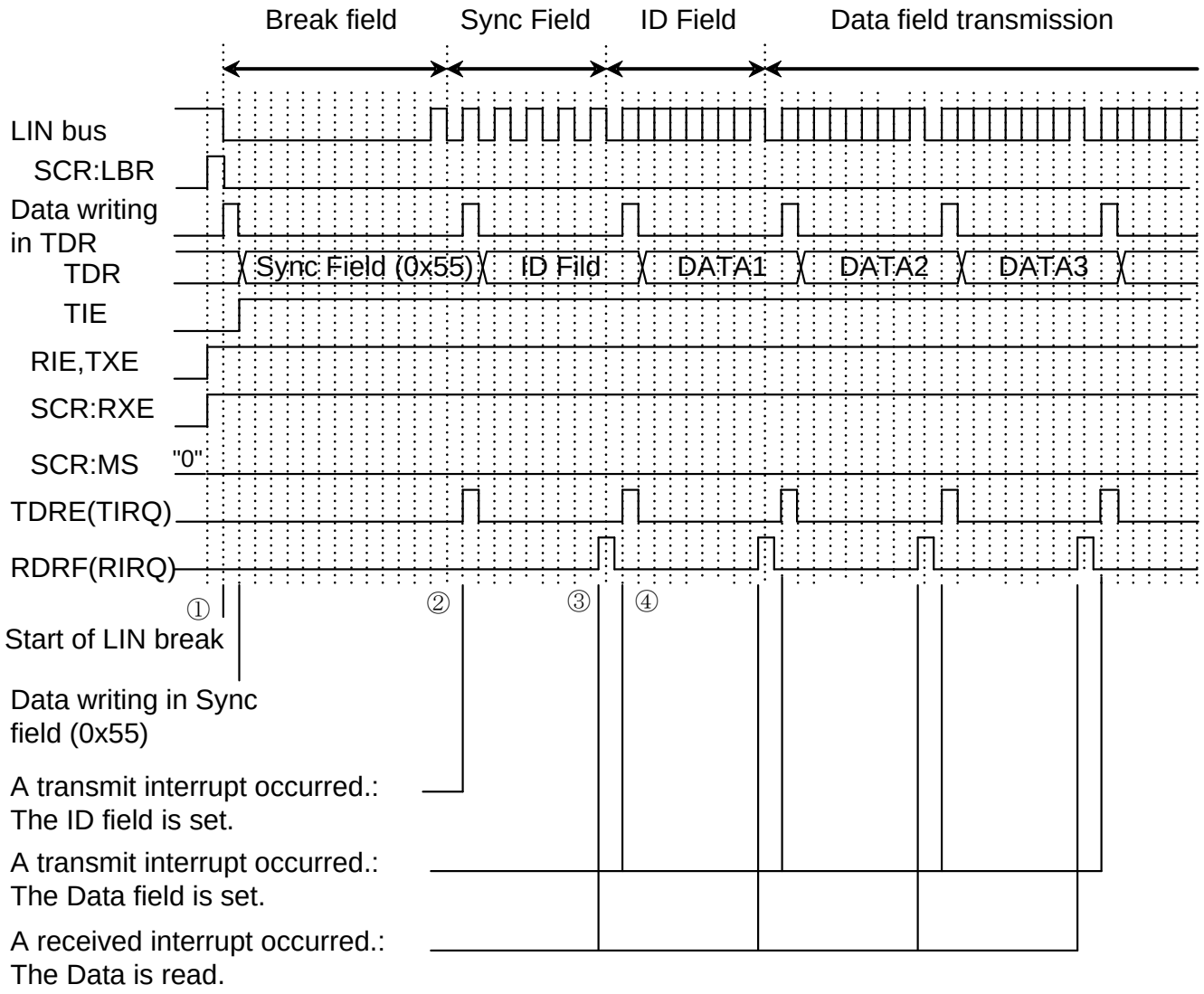


图 22 显示的是 LIN 从设备模式下数据发送的时序图。

1. 经过信号低电平的 11 位时间后，可以检测到 LIN 间隔场。如果将 LIN 间隔中断使能（LBIE）位设为 1，将生成一个 LIN 间隔中断。然后应该初始化并使能连接 SYNC 信号的 ICU。请参考外设手册中“GPIO”章节的内容，了解同特定 LIN 通道相对应的 ICU 通道。
2. 在同步场中检测到第一个下降沿时，将触发 ICU 中断，并且 ICU 数据寄存器的值被存储在变量 *a* 中。
3. 再次发生 ICU 中断时，ICU 数据寄存器的值可被存储在变量 *b* 中。

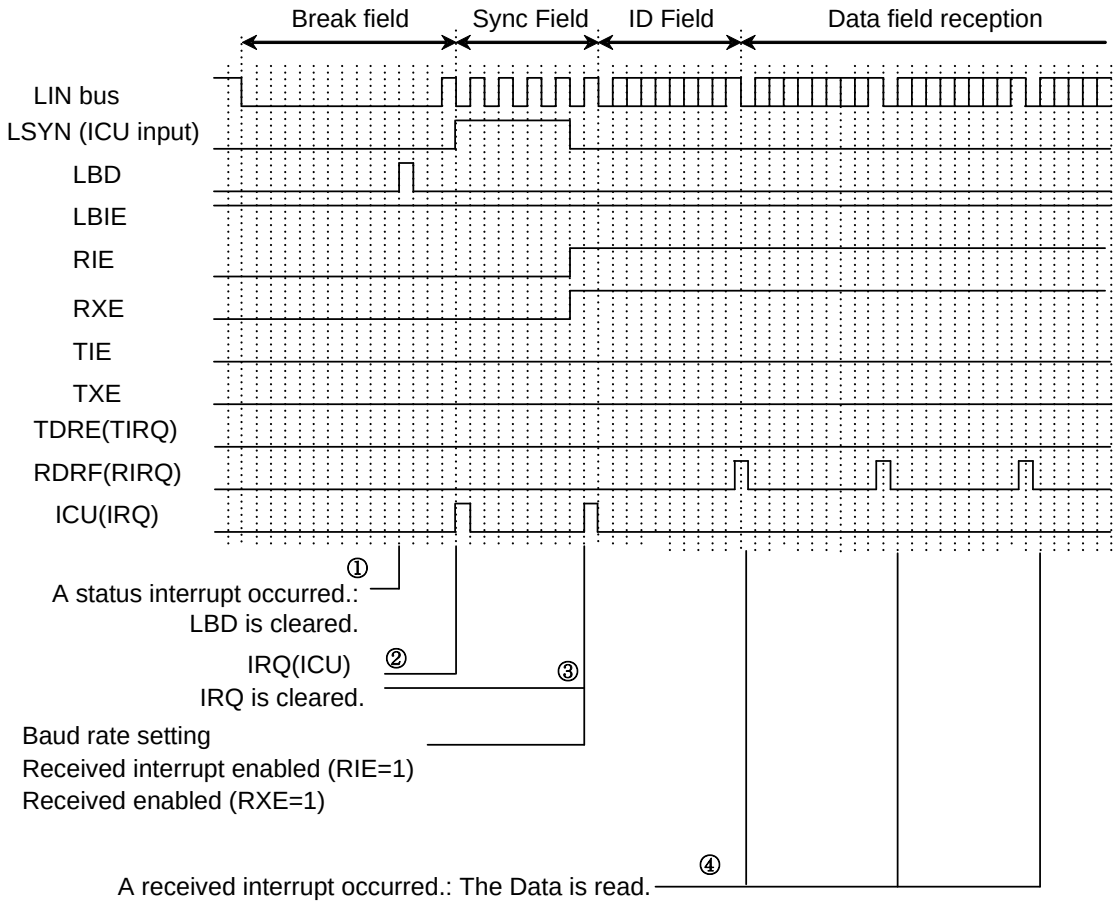
   如果 FRT 没有溢出，并且 MFS 和 FRT 使用同一个时钟，那么 LIN 主设备的准确波特率可通过以下公式计算得出。应将新的波特率设为 BGR，并禁用 ICU 功能。此时，可以使能接收功能（RXE = 1），并发出接收中断请求（RIE = 1）。

```
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. 接收数据寄存器已满时（RDRF = 1），如果使能了接收中断，将生成一个接收中断，此时可以从 RDR 中读取 ID 字段。

   数据接收过程和 ID 字段接收过程是相同的。

图 22. LIN 从设备模式下数据传输的时序图



5.5 底层 API

下面显示的是 PDL 的 LIN 驱动程序 API，它被放置在 `mfs.c/h` 文件内。

- 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()`通过使用自己专用的 LIN 配置参数`#stc_mfs_lin_config_t` 来初始化 LIN 模式的 MFS 实例。该函数仅设置了基本的 LIN 配置。`Mfs_Lin_DeInit()`用于复位与 MFS LIN 相关的所有寄存器。

`Mfs_Lin_EnableIrq()`用于使能由中断类型`#en_lin_irq_sel_t` 所选择的 LIN 中断源；
`Mfs_Lin_DisableIrq()`则用于禁用由`#en_lin_irq_sel_t` 中断类型所选择的 LIN 中断源。

`Mfs_Lin_SetBaudRate()`在初始化 LIN 后能够更改 LIN 波特率。

`Mfs_Lin_GenerateBreakField()`用于生成一个 LIN 间隔场，该间隔场也能够自己被检测。

`Mfs_Lin_EnableFunc()`使能了由 `Mfs_Lin_EnableFunc#enFunc` 参数所选择的 LIN 功能；
`Mfs_Lin_DisableFunc()`则禁用了 LIN 功能。

`Mfs_Lin_GetStatus()`用于读取由 `Mfs_Lin_GetStatus#enStatus` 所选择的 LIN 状态；
`Mfs_Lin_ClrStatus()`清除所选择的 LIN 状态。某些状态只能通过硬件自动清除。

`Mfs_Lin_SendData()`将一个字节数据写入到 LIN 发送缓冲区内；`Mfs_Lin_ReceiveData()`则从 LIN 接收缓冲区中读取一个字节数据。

`Mfs_Lin_ResetFifo()`复位 LIN 硬件 FIFO。

`Mfs_Lin_SetFifoCount()`在初始化 LIN 后能够更改 FIFO 的大小。

`Mfs_Lin_GetFifoCount()`读取 FIFO 中的当前数据值。

5.6 示例代码

根据底层驱动程序 API，这里提供了一个示例用于说明如何使用中断标志轮询方法通过 LIN 传输数据。该方法将 LIN ch.0 作为 LIN 主设备，用于传输 10 字节。

使用 LIN 前，先要根据下面内容配置 LIN 的引脚功能：

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

然后配置 LIN 配置结构，并初始化 LIN 通道。

- 模式：主设备模式
- 波特率：9600 bps
- 间隔场长度：13 位
- 间隔符长度：1 位
- 停止位长度：1 位

```
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 */
}
```

如果 ID 字段被设为 0x3A，则以下代码会发送 10 字节数据。

```
/*****************************************************************/
/*   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 总结

MFS 接口具有很高的灵活性，能够应用于各种不同的串行通信类型。

文档修订记录

文档标题： AN99218 — FM MCU 的多功能串行接口

文档编号： 002-09818

版本	ECN	变更者	提交日期	变更说明
**	5026158	JCUI	11/27/2015	本文档版本号为 Rev**，译自英文版 001-99218 Rev**。

全球销售和设计支持

赛普拉斯公司具有一个由办事处、解决方案中心、厂商代表和经销商组成的全球性网络。要想查找离您最近的办事处，请访问赛普拉斯所在地。

产品

汽车级产品	cypress.com/go/automotive
时钟与缓冲器	cypress.com/go/clocks
接口	cypress.com/go/interface
照明与电源控制	cypress.com/go/powerpsoc
存储器	cypress.com/go/memory
PSoC	cypress.com/go/psoc
触摸感应	cypress.com/go/touch
USB 控制器	cypress.com/go/usb
无线/射频	cypress.com/go/wireless

PSoC®解决方案

psoc.cypress.com/solutions

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

赛普拉斯开发者社区

社区 | 论坛 | 博客 | 视频 | 培训

技术支持

cypress.com/go/support

PSoC 是赛普拉斯半导体公司的注册商标。此处引用的所有其他商标或注册商标归其各自所有者所有。

赛普拉斯半导体公司
198 Champion Court
San Jose, CA 95134-1709

电话 ：408-943-2600
传真 ：408-943-4730
网址 ：www.cypress.com

©赛普拉斯半导体公司，2015。此处所包含的信息可能会随时更改，恕不另行通知。除赛普拉斯产品内嵌的电路外，赛普拉斯半导体公司不对任何其他电路的使用承担任何责任。也不会根据专利或其他权利以明示或暗示的方式授予任何许可。除非与赛普拉斯签订明确的书面协议，否则赛普拉斯不保证产品能够用于或适用于医疗、生命支持、救生、关键控制或安全应用领域。此外，对于可能发生运转异常和故障并对用户造成严重伤害的生命支持系统，赛普拉斯不授权将其产品用作此类系统的关键组件。若将赛普拉斯产品用于生命支持系统中，则表示制造商将承担因此类使用而招致的所有风险，并确保赛普拉斯免于因此而受到任何指控。
该源代码（软件和/或固件）均归赛普拉斯半导体公司（赛普拉斯）所有，并受全球专利法规（美国和美国以外的专利法规）、美国版权法以及国际条约规定的保护和约束。赛普拉斯据此向获许可者授予适用于个人的、非独占性、不可转让的许可，用以复制、使用、修改、创建赛普拉斯源代码的派生作品、编译赛普拉斯源代码和派生作品，并且其目的只能是创建自定义软件和/或固件，以支持获许可者仅将其获得的产品依照适用协议规定的方式与赛普拉斯集成电路配合使用。除上述指定的用途外，未经赛普拉斯明确的书面许可，不得对此类源代码进行任何复制、修改、转换、编译或演示。
免责声明：赛普拉斯不针对此材料提供任何类型的明示或暗示保证，包括（但不限于）针对特定用途的适销性和适用性的暗示保证。赛普拉斯保留在不做出通知的情况下对此处所述材料进行更改的权利。赛普拉斯不对此处所述之任何产品或电路的应用或使用承担任何责任。对于可能发生运转异常和故障，并对用户造成严重伤害的生命支持系统，赛普拉斯不授权将其产品用作此类系统的关键组件。若将赛普拉斯产品使用于生命支持系统中，则表示制造商将承担因此类使用而招致的所有风险，并确保赛普拉斯免于因此而受到任何指控。
产品使用可能受限于赛普拉斯软件许可协议。