Freescale Semiconductor
应用笔记

Document Number: AN4342
Rev 0, 08/2011

在 ColdFire+和 Kinetis 上使用内部集成电路

适用于 MCF51JF128 的 I2C 驱动器

作者: Ju Yingyi

内容

1 简介

本应用笔记说明如何在 Kinetis 和 Coldfire+芯片上使用最新的内部集成电路（I2C 或 I^2C）模块。

虽然本文内容是基于 MCF51JF128 的 I^2C 模块且所有示例代码均在此器件上测试，但文中所述之驱动器只需略加修改即可移植到任意其他 Coldfire+或 Kinetis 芯片上。

飞思卡尔 Kinetis 和 Coldfire+器件上的最新 I^2C 模块兼容 *I^2C* 总线规范和系统管理总线（*SMBus*）规范版本 *2*。最新模块还添加了 DMA 支持，从而降低了 MCU 负载。以下页面介绍这些新功能，并提供 I^2C 驱动器示例代码。

有关此应用笔记的更新信息，请参见最新的芯片和演示板文档。本文根据 www.freescale.com 上的最新文档撰写而成。I^2C 或 SMBus 规范请参见 www.i2c-bus.org 。

2 I^2C 模块概述

本节说明了一种可在多个器件之间建立通信的方法。

内部集成电路——通常缩写为 IC、I^2C 或 IIC——模块提供了一种可在多个器件之间实现通信的方法。该接口适用于在总线加载和时序最高的情况下以高达 100 kbit/s 的速度运行。该器件能够以更高的波特率运行，最高波特率为总线时钟除以 20，同时总线负载更低。最长通信距离和可连接的设备数量受最高 ***400 pF*** 的总线电容限制。I^2C 模块还符合系统管理总线（*SMBus*）规范版本 *2*。

注

飞思卡尔的 I^2C 符合 I^2C 规范的标准模式（100 kHz）和快速模式（400 kHz）要求。最长通信距离和可连接的设备数量受最高 400 pF 的总线电容限制。

此模块不需要特殊的电路来实现上拉 SCL 和 SDA 线来获取更高速的模式（最高 3.4MHz）；然而，I^2C 模块的最大波特率可设为总线时钟除以 20。

2.1 特性

I2C 模块包括以下特性：

- 符合 *I2C* 总线规范
- 多主机操作
- 可通过软件对 64 种不同串行时钟频率的其中之一进行编程
- 可通过软件选择应答位
- 由中断驱动的逐字节数据传输
- 通过从主机模式自动切换至从机模式，生成仲裁丢失中断
- 调用地址识别中断
- 开始和停止信号的生成和检测
- 可重复的开始信号生成和检测
- 应答位的生成和检测
- 可检测总线忙状态

相比 Coldfire V2 器件（MCF52xx、MCF5225x），这款最新的 I2C 模块还支持：

- 可扩展 10 位地址
- 可识别通用调用
- 可编程的毛刺输入过滤器

相比最新的 Coldfire V1 器件（MCF51CN128、MCF51MM256/MCF51MM128），这款全新设计的模块还支持：

- 系统管理总线*(SMBus)*规范（第 *2* 版）
- 从机地址匹配时的低功耗唤醒
- 从机地址的适用范围
- DMA 操作

2.2 工作模式

各种低功耗模式下的 I^2C 模块工作情况如下所示：

- 运行模式：这是基本工作模式。要在该模式下节能，则禁用该模块。
- 等待模式：该模块可在内核处于等待模式时继续工作且可提供唤醒中断。
- 停止模式：在停止模式下，除非地址匹配使能，否则该模块处于无效状态以降低功耗。STOP 指令不影响 I^2C 模块的寄存器状态。在任何 VLLSx 模式下，寄存器内容都会复位。

注

有关 Kinetis 和 Coldfire+器件功耗模式的更多信息，请参考 www.freescale.com 上提供的器件参考手册。

2.3 框图

图 1 是 I^2C 模块的框图。

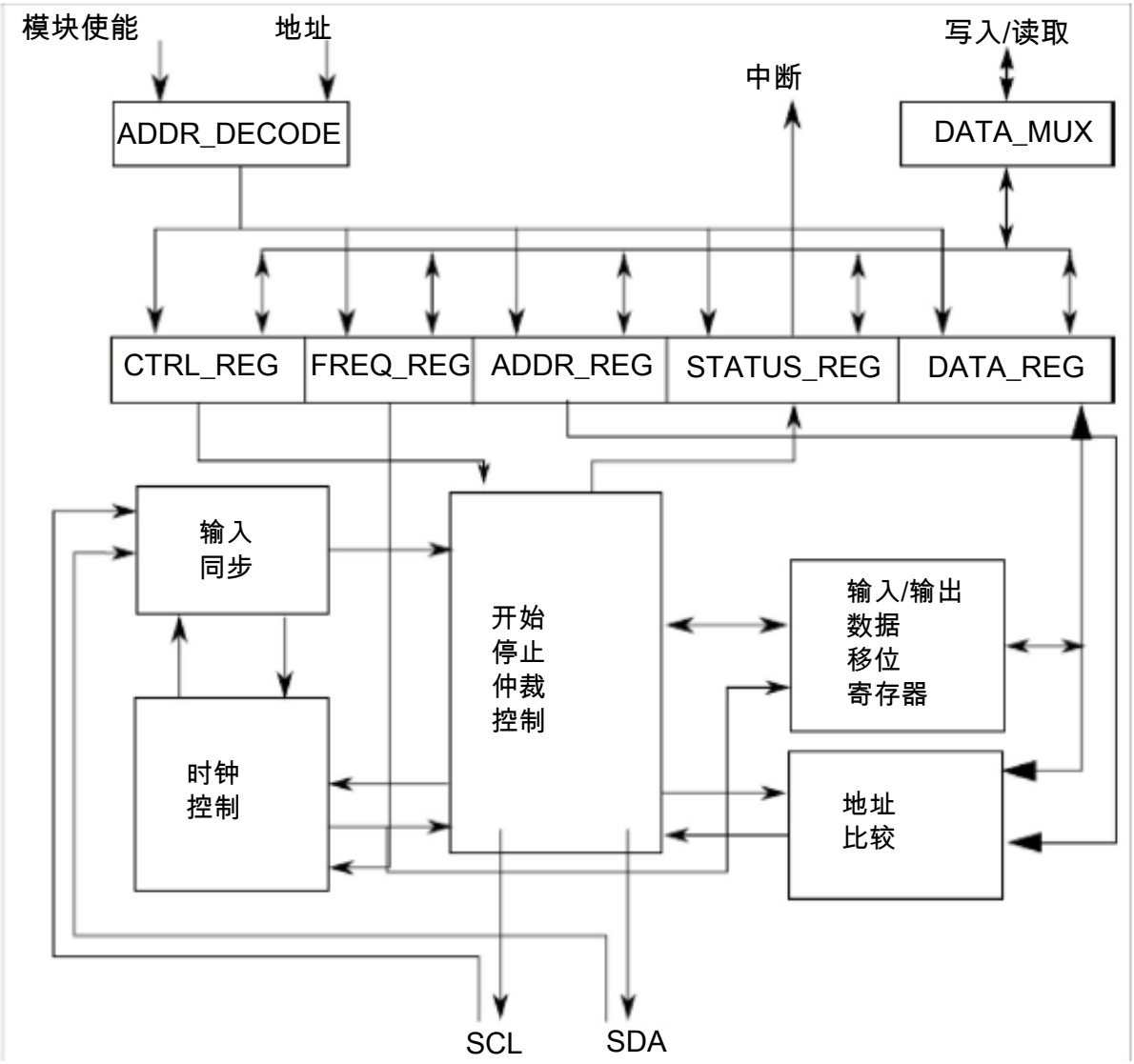

图 1. I²C 功能框图

3 功能说明

本节中，我们将跳过 I²C 协议，来重点关注最新 I²C 模块的一些特殊功能。

3.1 I²C 波特率

根据参考手册的 I²C 分频器寄存器（I2Cx_F）说明，I²C 数据传输波特率由总线时钟和 I2Cx_F 寄存器确定。波特率可用下列等式进行计算：

I²C 波特率 = 总线速度(Hz) ÷ (mul × SCL 分频器)

I2Cx_F[MULT] = 0b00、01 或 10 时，mul = 1、2 或 4。

SCL 分频器：由 I2Cx_F[ICR]确定，数值请参见器件参考手册中 I²C 部分的“I²C 分频器和保持值”章节。

您可能会发现，设置该寄存器有时候可能不会产生您需要的确切波特率。这种情况下可以使用最接近的数值。

例如，假设总线时钟为 25 MHz。为了在 SCL 板上产生 50 kHz 时钟，设置 I2Cx_F = 0x97、0x63 或 0x2B 可能会得到 48828 Hz。

*int _i2c_set_bps(uint8 channel, uint32 bps)*函数可用于查找 I2Cx_F 产生的最接近要求值的波特率数值。详情请参考 i2c.c 中的函数 *int_i2c_set_bps()*。

3.2 中断

当发生下表中的任意事件时，如果已设置 IICIE 位，I²C 模块将生成中断。中断由（I²C 状态寄存器的）IICIF 位驱动，通过（I²C 控制寄存器 1 的）IICIE 位屏蔽。在中断例程中，必须通过向 IICIF 位写入 1 的方式由软件清零 IICIF 位。SMBus 超时中断由 SLTF 驱动，并用 IICIE 位屏蔽。在中断例程中，必须通过向 SLTF 位写入 1 的方式由软件清零 SLTF 位。可通过读取状态寄存器确定中断类型。

表 1. 中断汇总

中断源	状态	标志	本地使能
完成 1 个字节的传输	TCF	IICIF	IICIE
已接收的调用地址与主/从机地址匹配	IAAS	IICIF	IICIE
仲裁丢失	ARBL	IICIF	IICIE
SMBus SCL 低电平超时中断标志	SLTF	IICIF	IICIE
SMBus SCL 高电平 SDA 低电平超时中断标志	SHTF2	IICIF	IICIE 和 SHTF2IE
从停止中断唤醒	IAAS	IICIF	IICIE 和 WUEN

下面的图 2 是典型的 I²C 中断例程。您可按照该例程设计 I²C 驱动器。

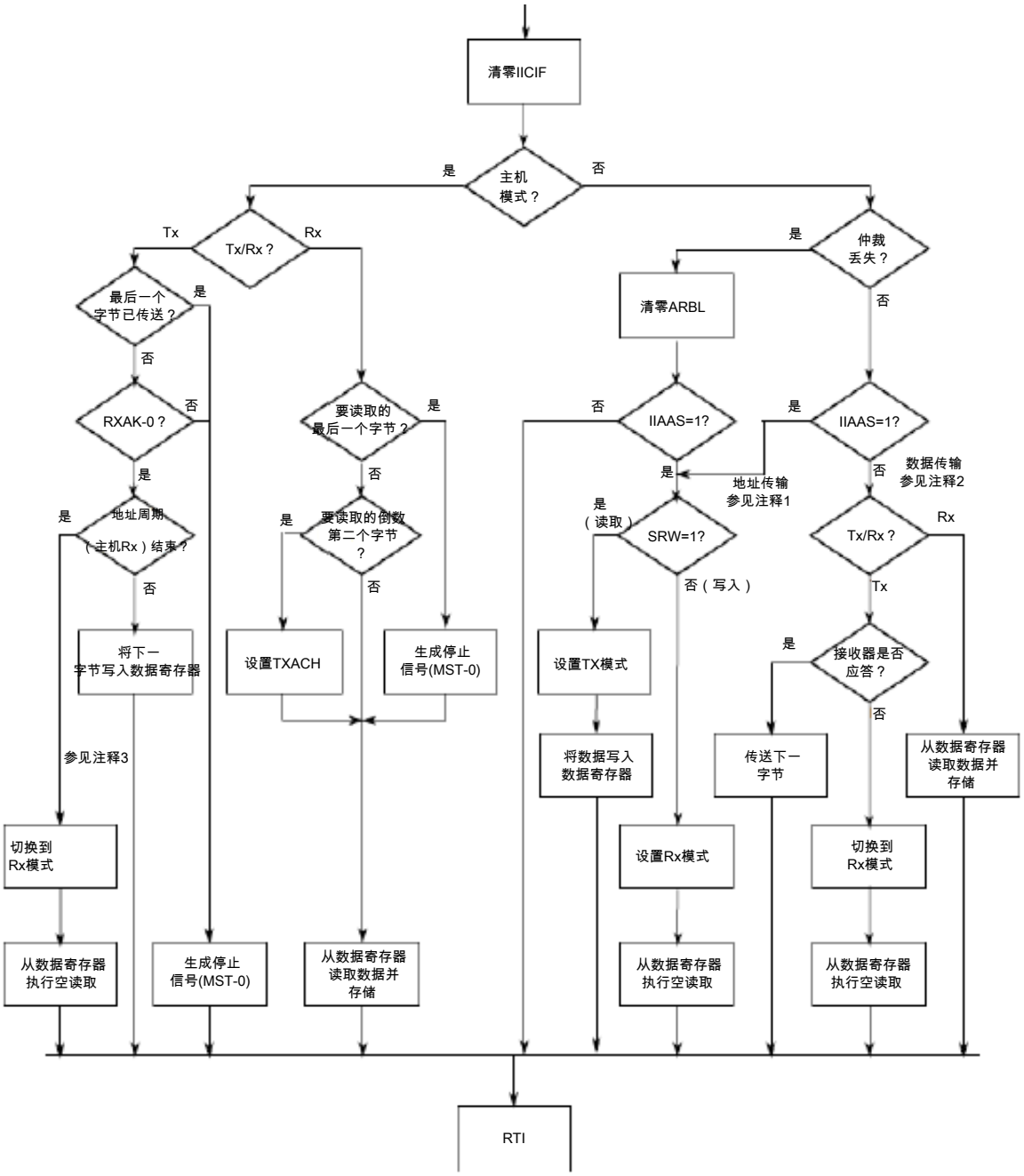

图 2. 典型的 I²C 中断例程

注

1. 如果已使能通用调用，则检查以确定接收到的地址是否是通用调用地址(0x00)。如果接收到的地址是通用调用地址，则通用调用必须由用户软件处理 。
2. 如果 10 位寻址进行的是从机寻址，则在扩展地址的首个字节后，该从机会遭遇中断。对于此中断，必须确保忽略数据寄存器的内容，且不将其当作有效数据传输。
3. 在主机接收模式下且仅有 1 个字节待接收时，空读取前应置位 TXACK。

3.3 SMBus 支持

最新的 I^2C 模块可很好地支持 SMBus。有关更多详情，请参见最新的 Kinetis 或 Coldfire+ MCU 参考手册。有关 SMBus 规范的详情，请参见系统管理总线（*SMBus*）规范（第 *2.0* 版）。

3.3.1 超时

根据 $T_{TIMEOUT,MIN}$ 参数，主机或从机可判断是否有缺陷器件将时钟无限期地保持在低电平，还是主机有意驱动器件脱离总线。从器件检测到任意单个时钟保持在低电平的时间长于 $T_{TIMEOUT,MIN}$ 时，强烈建议释放总线（停止驱动总线并使 SCL 和 SDA 悬空为高电平）。已检测到此条件的器件必须复位其通信并且能在 $T_{TIMEOUT,MAX}$ 的时间范围内接收新的 START 条件。

SMBus 定义 35 ms 的时钟低电平超时 $T_{TIMEOUT}$，将 $T_{LOW:SEXT}$ 指定为从器件的累积时钟低电平延长时间，将 $T_{LOW:MEXT}$ 指定为主器件的累积时钟低电平延长时间。

3.3.2 FACK 和 NACK

为了提高可靠性和通信的稳定性，SMBus 设备是否实现数据包差错校验(PEC)是可选的，但对于需要参与地址解析协议(ARP)过程的器件，PEC 则是必需的。PEC 是一个基于所有消息字节的 CRC-8 错误校验字节。PEC 由提供最后一个数据字节的器件追加到消息中。如果 PEC 存在但不正确，接收器将发送 NACK，否则发送 ACK。为了通过软件计算 CRC-8，在接收到第 8 个 SCL（第 8 位）之后，如果该字节是数据字节，此模块将使 SCL 线保持低电平。当 FACK（快速 ACK/NACK）位已使能，就可以通过软件确定应当向总线发送 ACK 还是 NACK，即置位还是清除 TXAK 位。

下面的图 3 是 SMBus 数据包协议图关键的元素。

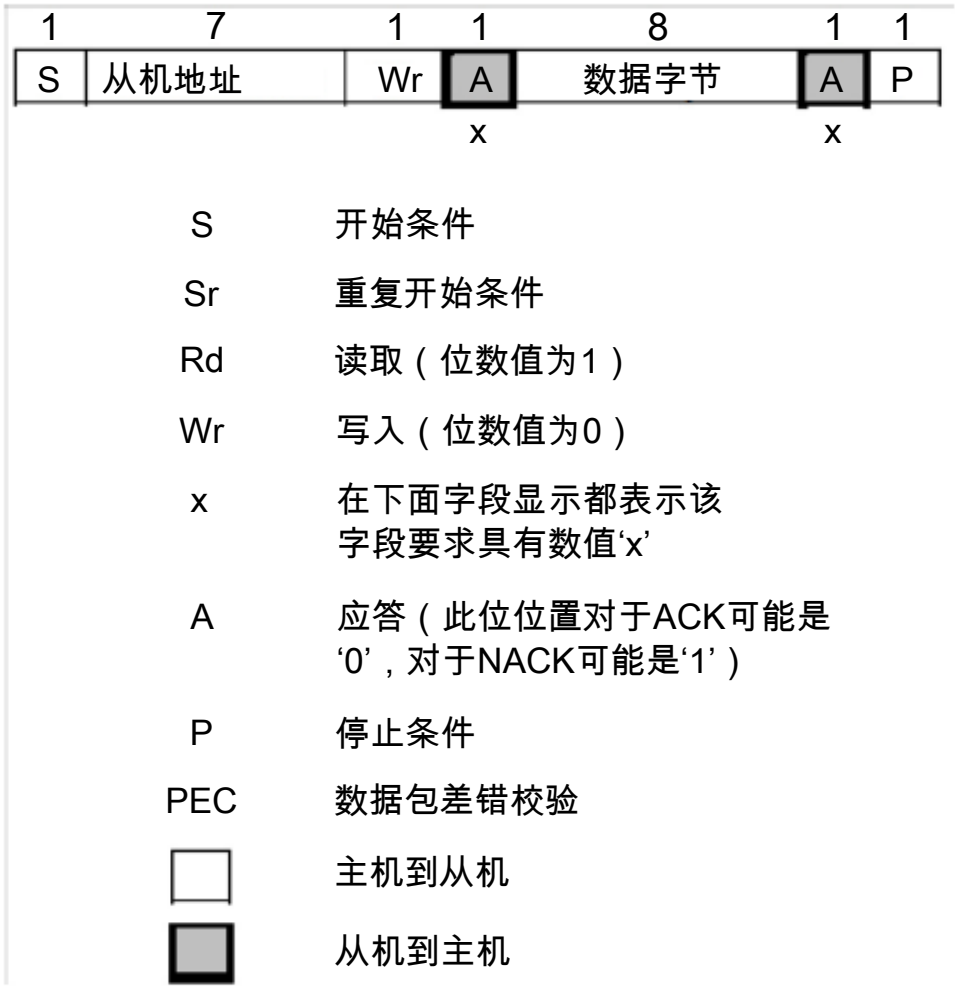

图 3. SMBus 数据包协议图关键的元素

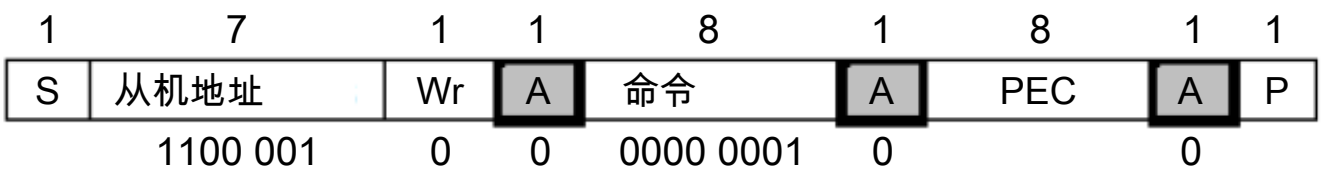

图 4. 标准 SMBus 发送字节协议是包含 PEC 的

SMBus 要求设备始终应答其自己的地址，作为检测总线上可移除设备（如电池或插接站）存在与否的机制。除了指示从设备繁忙条件以外，SMBus 还使用 NACK 机制来表示接收到无效命令或数据。由于这种条件可能出现在最后一个字节时，因此 SMBus 设备必须有能力在传输每个字节之后及完成处理之前产生不应答信号。因为 SMBus 不提供其他重新发送命令，所以这一要求非常重要。使用 NACK 信令中的这一差别对 SMBus 端口的具体实现有影响，尤其是对于 SMBus 主机和 SBS 器件等处理关键系统数据的设备。

注

在主机接收从机传送模式的最后一个字节中，主机必须向总线发送一个 NACK，因此在传送最后一个字节之前必须关闭 FACK。

3.3.3 ALERTEN 和 SIICAEN

这两个位可在 I2Cx_SMB 寄存器中找到。为了支持 SMBus 规定的协议，ARP 和 ARA（报警响应地址）这两位必须置位，且 I2Cx_A2 应设为相应地址（SMBus 器件默认地址 0x61<<1，SMBus 报警响应地址 0x0C<<1）。

然而，为了支持 ARA，可能需要在 SMBus 主机和器件之间创建额外中断线路（SMBALERT#）。从机可通过该线路对主机提供信号。下面的图 5 显示的是 ARA 格式。有关更多详情，请参见系统管理总线（*SMBus*）规范（第 2 版）。

一个7位可寻址器件响应ARA

一个7位可寻址器件以PEC响应ARA

图 5. SMBus ARA 命令格式

3.3.4 典型的 SMBus 中断例程

图 6 是一个典型的 SMBus 中断例程，附加其运行过程相关的说明。

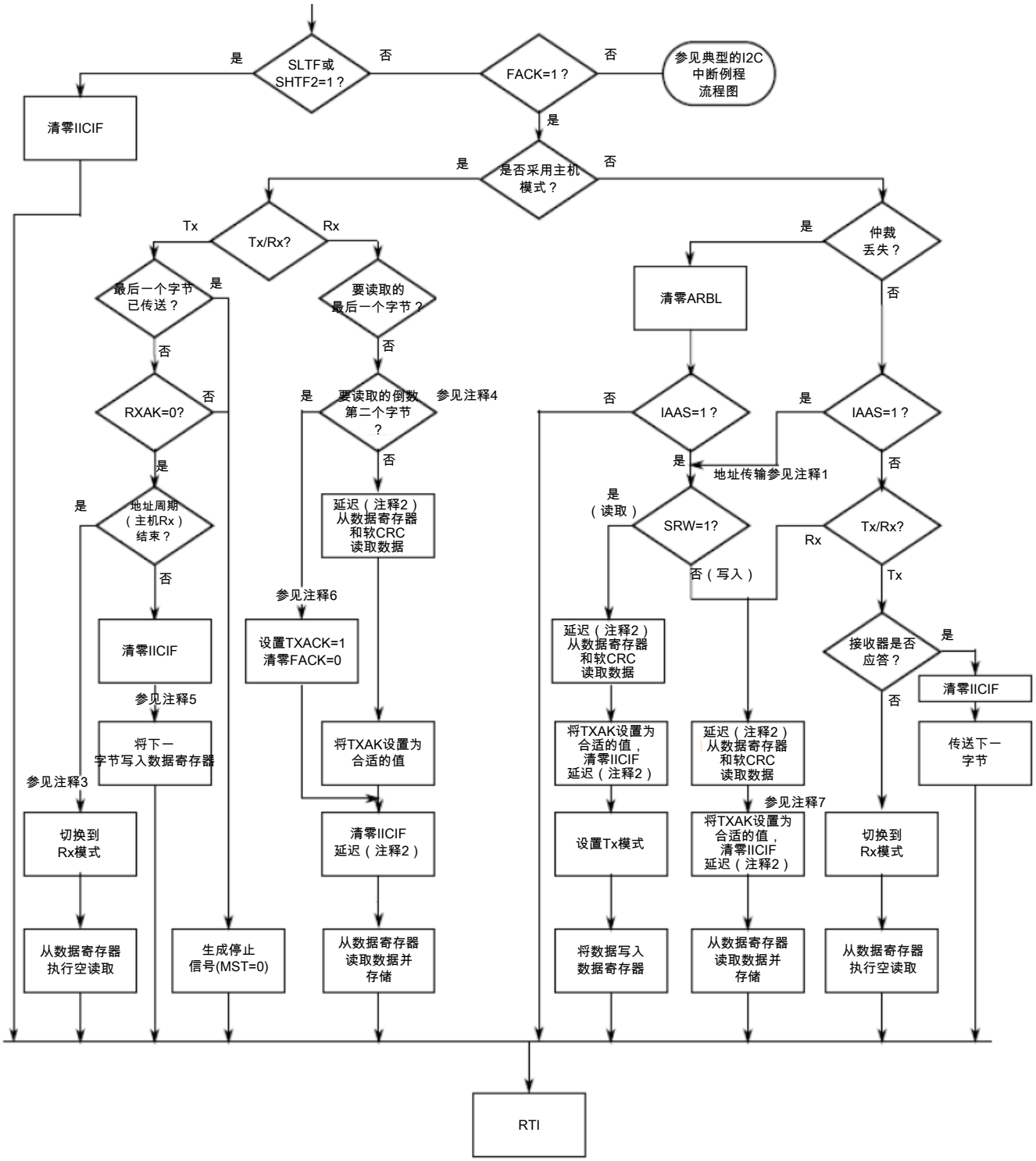

图 6. 典型的 SMBus 中断例程

注

1. 如果已使能通用调用或 SIICAEN，则进行检查，以确定接收到的地址是通用调用地址(0x00)还是 SMBus 器件默认地址。无论是哪一种，都必须由用户软件来处理。
2. 在接收模式下，进行第一次和第二次数据读取前，可能需要一个数据位的时间延迟。

3. 在主机接收模式下且仅有 1 个字节待接收时，空读取前应置位 TXACK。
4. 因为处理倒数第二个字节传输时 FACK 已清零，所以无需处理最终字节传输；最终字节的传输将遵循正常 I2C 中断例程。
5. 如果发送倒数第二个字节，则应当清零 FACK。
6. 将 TXACK 设为 1 并清零 FACK 之前，应当延迟 1 位 CLK，从数据寄存器读取数据，执行软 CRC，通过设置 TXACK = 0 来对当前字节执行 ACK，并再次延迟 1 位 CLK。
7. 此处需添加一个第二到最终字节传输检查例程。如果正确，则设置 TXACK = 0 来对当前字节执行 ACK，并延迟 1 位 CLK。此外，必须检查 SIICAEN 是否置位；如果已置位，则只需清零 FACK，并设置 TXACK = 1。

3.4 可编程输入毛刺过滤器

I^2C 毛刺过滤器已添加到传统 I^2C 模块的外部以及 I^2C 封装内。该过滤器可吸附 I^2C 时钟和 I^2C 模块数据线路上的毛刺。可根据 (半) 总线时钟周期数指定待吸收毛刺的宽度。提供单个可编程输入毛刺过滤器控制寄存器。实际上，I^2C 模块通常会忽略数据线路上的任意下降-上升-下降或上升-下降-上升转换 (在该寄存器编程的时钟周期数范围内)。程序员必须指定毛刺的大小 (根据总线时钟周期)，以便过滤器吸收毛刺，并且阻止毛刺通过。

3.5 地址匹配唤醒

当 I^2C 模块处于从接收模式下时，如果发生主机地址、范围地址或通用调用地址匹配情况，则 MCU 将从低功耗模式 (无外围总线在此模式下运行) 中唤醒。设置地址匹配 IAAS 位后，将在地址匹配结束时发送中断以唤醒内核。必须在时钟恢复之后清除 IAAS 位。

注

- 系统恢复后，当系统处于运行模式中时，如有必要可重启此 I^2C 模块。SCL 线路不保持在低电平状态，直至 I^2C 模块在地址匹配之后复位。
- 从停止模式执行 I^2C 地址匹配唤醒时，有一个漏洞：如果第一个数据包包含错误的地址，则 MCU 将不会唤醒，即使后续数据包包含正确的地址亦是如此。器件的最新勘误表报告中记录了这一漏洞。

3.6 DMA 支持

如果 DMAEN 位清零，且 IICIE 位置位，则中断条件生成中断请求。如果 DMAEN 位和 IICIE 位置位，则中断条件转而生成 DMA 请求。由传输完成标志(TCF)生成 DMA 请求。

如果 DMAEN 位置位，则仅丢失另一个 I^2C 模块的仲裁 (错误)，且 SCL 低电平超时 (错误) 产生 CPU 中断。所有其他事件启动一次 DMA 传输。

注

- 在主机接收模式的最后一个字节之前，TXAK 必须置位，以在最后一个字节传输之后发送一条 NACK 信号。因此，必须在最后一个字节传输之前禁用 DMA。
- 在 10 位地址模式发送过程中，地址发送占用 2-3 个字节。在此次传输期间，必须禁用 DMA，因为 C1 寄存器写入了发送一条重复开始指令或变更传输方向指令。

4 I²C 驱动程序示例

本节提供与可用于 Coldfire+和 Kinetis 器件的 I²C 驱动程序相关的信息。

本节介绍独立式典型 I²C 驱动程序以供参考。所有 API 原型可参见 *i2c.c*。

该驱动程序已在 MCF51JF128 上进行测试。可完全用于 Coldfire+或 Kinetis 器件。

4.1 关键宏

所有预定义的关键宏均可在各演示项目的 i2c_cfg.h 中找到。

4.1.1 I2C_POLLING_MODE

如果定义为 1，则使用轮询方法；否则使用中断方法。

4.1.2 I²C_BUFFER_SIZE

定义 I²C TX 和 RX 缓冲器大小。

4.1.3 I2C_DEBUG

如果定义为 1，则将通过默认 UART 端口输出调试消息（会影响性能）。

4.2 全局变量

4.2.1 i2c_tx_buffer

I2C TX 缓冲器。

原型：

```
I2C_BUFFER i2c_tx_buffer;
```

备注：

```
/* Structure for storing I2C transfer data */
typedef struct {
       int tx_index;          /* TX index */
       int rx_index;          /* RX index */
       int data_present;      /* Data present flag */
       uint16 length;         /* Length of the buffer in bytes */
uint8 buf[I2C_BUFFER_SIZE];   /* Data buffer */
} I2C_BUFFER;
```

传输前应填充缓冲器结构。

4.2.2 i2c_rx_buffer

I²C RX 缓冲器。

原型:

```
I2C_BUFFER i2c_rx_buffer;
```

备注:

请参照 i2c_tx_buffer 的备注。

4.3 API

4.3.1 i2c_init

初始化指定 I2C 通道。地址模式为 7 位。

原型:

void *i2c_init*(uint8 *channel*, uint8 *addr*, uint32 *bps*)

参数:

channel: 用户指定的 I²C 通道数

addr: 主从机地址

bps: 波特率 (Hz)

返回值:

无

备注:

IICEN、IICIE 设置取决于 I2C_POLLING_MODE 的数值。如果 I2C_POLLING_MODE = 1，则置位 IICEN 和 IICIE。

4.3.2 i2c_10bit_init

初始化指定 I2C 通道。地址模式为 10 位。

原型:

void *i2c_10bit_init*(uint8 *channel*, uint8 *addr*, uint32 *bps*)

参数:

channel: 用户指定的 I²C 通道数

addr: 主从机地址

bps: 波特率 (Hz)

返回值:

无

备注:

IICEN、IICIE 设置取决于 I2C_POLLING_MODE 的数值。如果 I2C_POLLING_MODE = 1，则置位 IICEN 和 IICIE。

4.3.3 smbus_init

初始化指定 I²C 通道。此通道特性遵循 SMBus 协议。

原型：

void *iic_smbus_init*(uint8 *channel*, uint8 *addr*, uint8 *sec_addr*, uint32 *bps*)

参数：

channel：用户指定的 I²C 通道数

addr：主从机地址

sec_addr：SMBus 地址

bps：波特率（Hz）

返回值：

无

4.3.4 i2c_master

执行 I²C 主机传输的通用函数。

原型：

void *i2c_master* (uint8 *channel*, uint8 *mode*, uint16 *slave_address*)

参数：

channel：用户指定的 I²C 通道数

mode：有效模式包括 I2C_TX、I2C_RX、I2C_TXRX、I2C_10BIT_TX、I2C_10BIT_RX、I2C_10BIT_TXRX（所有模式均在 i2c.h 中定义）

slave_address：从机地址

返回值：

无

4.3.5 i2cx_isr

通用 I²C 处理程序，该函数的创建遵循流程图图 2 和图 6 。

原型：

interrupt void *i2cx_isr*(void)

参数：

无

返回值：

无

备注

如果未使用 SMBus，则可调用中断服务例程中的 i2c_handler()，取代 i2csmbus_handler()。

4.4 使用 I²C 驱动程序

示例代码仅供 CodeWarrior 10.x 使用。若要打开测试项目，应首先解压缩代码，然后选择“File > Import…”，如下面的图 7 所示。

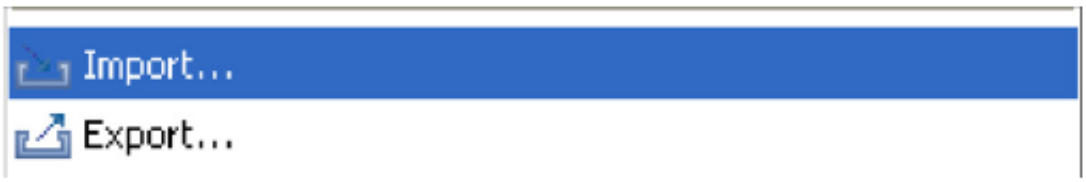

图 7. CW10.x 中如何打开示例项目，步骤 1

下一步，选择“Existing Projects into Workspace”，然后单击“Next >”。

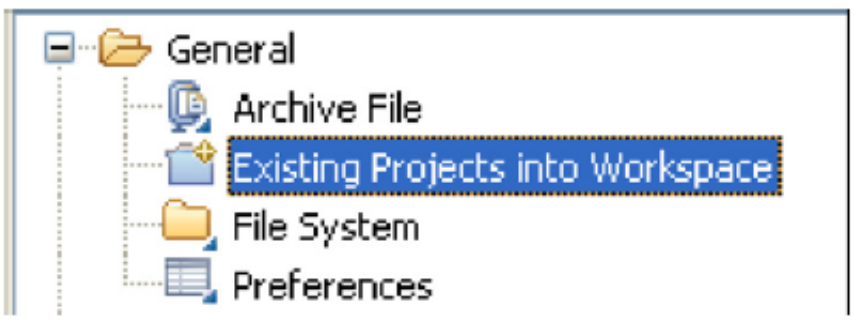

图 8. CW10.x 中如何打开示例项目，步骤 2

然后，选择项目所在文件夹，并单击“Next >”。将列出所有示例项目，您可选择打开其中的一个、多个或全部。

4.4.1 轮询模式

示例代码可在 i2c_basic 项目中找到。

首先，将 i2c_cfg.h 中的 *I2C_POLLING_MODE* 定义为 1。

I²C 通道 3，7 位地址，主机 TX 模式，50000bps，传输 64 个字节至 I²C 从机，从机地址为 0x21（在 i2c_cfg.h 中定义）：

```
void i2c_master_TX_test(void)
{
    uint8 i;

    printf("***Mini I2C Basic Master TX_Polling Test***\r\n");
    i2c_tx_buffer.tx_index = 0;
    i2c_tx_buffer.rx_index = 0;
    i2c_tx_buffer.data_present = TRUE;
    i2c_tx_buffer.length = 64;
    // for demo only, first byte is slave address
    i2c_tx_buffer.buf[0] = (uint8)(I2C_SLAVE_ADDR&0x0FF);

    for(i=1;i<64;i++)
    {
        i2c_tx_buffer.buf[i] = i;
```

```
    }
    // I2C channel 3, 50000bps
    i2c_init(3, I2C_MASTER_ADDR, 50000);
    i2c_master(3, I2C_TX, I2C_SLAVE_ADDR);
}
```

图 9 显示了 i2c_master()工作流程（当模块进入主机 TX 模式后使用轮询模式）。

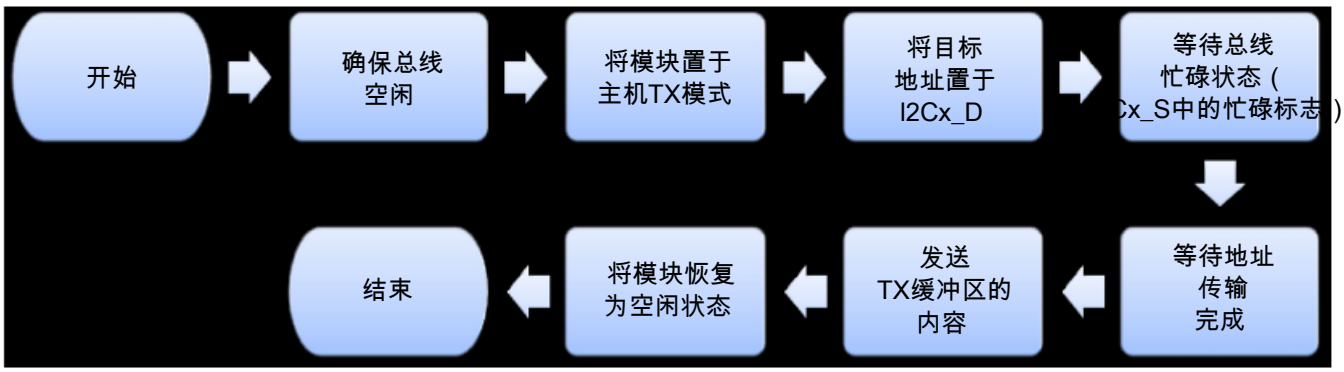

图 9. 主机 TX 轮询模式流程图

I²C 通道 0，7 位地址，主机 RX 模式，50000bps，从从机接收 64 个字节，从机地址为 0x21（在 i2c_cfg.h 中定义）：

```
void i2c_master_RX_test(void)
{
    uint8 i;

    printf("***Mini I2C Basic Master RX_Polling Test***\r\n");
    i2c_rx_buffer.length = 64;
    i2c_init(0, I2C_MASTER_ADDR, 50000);
    i2c_master(0, I2C_RX, I2C_SLAVE_ADDR);
}
```

下图显示了 i2c_master()工作流程（当模块进入主机 RX 模式后使用轮询模式）。

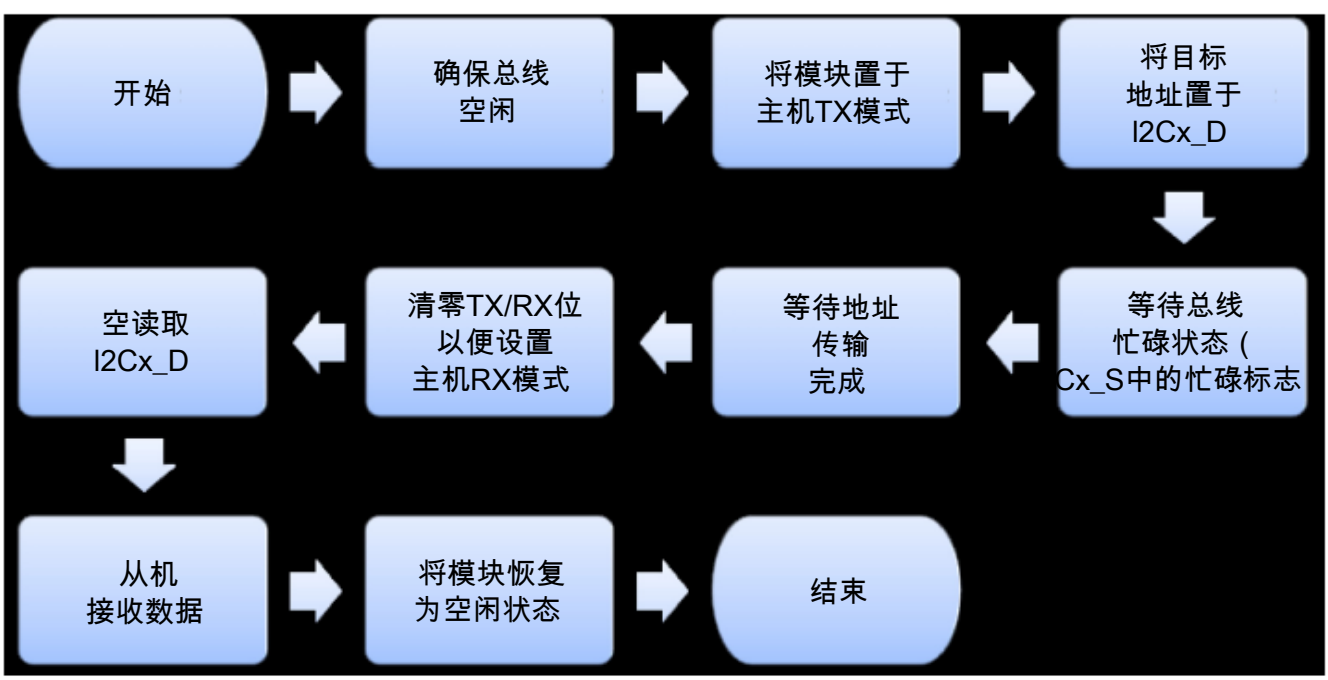

图 10. 主机 RX 轮询模式

注

在“从机接收数据”阶段，必须遵循典型 I²C 中断例程的主机 RX 例程。

4.4.2 使用中断

示例代码可在 i2c_10bit 项目中找到。对于 7 位地址模式，请打开 i2c_interrupt 项目。

首先，将 *I2C_POLLING_MODE* 定义为 0，然后在 i2c_cfg.h 中，在 exceptions.c 里，用 i2cx_isr 中断函数填充向量表。另外，请记得使能所用 I²C 通道的中断。

I²C 通道 0，10 位地址，主机 TX 模式，50000bps，传输 64 个字节至 I²C 从机:

```
void i2c_10bit_master_tx_test(unsigned short mst_addr, unsigned short slv_addr)
{
   uint8 i;

   printf("***Mini I2C 10-bit Addr Master TX Test***\r\n");
   i2c_tx_buffer.tx_index = 0;
   i2c_tx_buffer.rx_index = 0;
   i2c_tx_buffer.data_present = TRUE;
   i2c_tx_buffer.length = 64;
   // for demo only, first byte to be transferred is slave address
   i2c_tx_buffer.buf[0] = (uint8)(I2C_SLAVE_ADDR&0x0FF);

   for(i=1;i<64;i++)
   {
     i2c_tx_buffer.buf[i] = i;
   }

   iic_10bit_init(0, mst_addr, 50000);
   i2c_master(0, I2C_10BIT_TX, slv_addr);
}
```

I²C 通道 0，10 位地址，主机 RX 模式，50000 bps，从 I²C 从机接收 64 个字节。请注意下述代码的注释:

```
void i2c_10bit_master_rx_test(unsigned short mst_addr, unsigned short slv_addr)
{
     printf("***Mini I2C 10-bit Addr Master RX Test***\r\n");
     i2c_tx_buffer.tx_index = 0;
     i2c_tx_buffer.rx_index = 0;
     i2c_tx_buffer.data_present = TRUE;

     // when using 10-bit address mode, the tx length must be set to 1,
     // and the first byte to be transferred should be
     // the low 8-bit of 10-bit slave address.
     // when using 7-bit address mode, if you do not want to transfer data
     // to the slave, just set tx length to 0,
     // else use I2C_TXRX_MODE instead.
     i2c_tx_buffer.length = 1;
     i2c_tx_buffer.buf[0] = (uint8)(slv_addr&0x0FF);

     i2c_rx_buffer.length = 64;

     iic_10bit_init(0, mst_addr, 50000);
     i2c_master(0, I2C_10BIT_RX, slv_addr);
}
```

I²C 通道 0，10 位地址，从机 RX 模式，50000 bit/s（事实上，处于从机模式时，I2Cx_F 将被忽略），从 I²C 主机接收 64 个字节。

```
 void i2c_10bit_slave_rx_test(unsigned short addr)
 {
    printf("***Mini I2C 10-bit Addr slave RX Test***\r\n");
    iic_10bit_init(0, addr, 50000);
 }
```

I²C 通道 0，10 位地址，从机 TX 模式，50000 bit/s（事实上，处于从机模式时，I2Cx_F 将被忽略），主机寻址后向 I²C 主机传输 64 个字节。

```
void i2c_10bit_slave_tx_test(unsigned short mst_addr, unsigned short slv_addr)
{
   uint8 i;
   printf("***Mini I2C 10-bit Addr slave TX Test***\r\n");

   i2c_tx_buffer.tx_index = 0;
   i2c_tx_buffer.rx_index = 0;
   i2c_tx_buffer.data_present = TRUE;
   i2c_tx_buffer.length = 64;
   i2c_tx_buffer.buf[0] = (uint8)(mst_addr&0x0FF);

   for(i=1;i<64;i++)
   {
       i2c_tx_buffer.buf[i] = i;
   }

   iic_10bit_init(0, slv_addr, 50000);
 }
```

注

请参见图 2，了解典型 I²C 中断例程示意图。

一般而言，从机波特率随主机波特率变化。然而，寄存器 I2Cx_C2 中的 SBRC 字段是一个特殊位，用于在极高速 I²C 模式下迫使 SCL 时钟拉伸。

4.4.3 使用 DMA

此模块可与 DMA 协同运行。如需查看示例代码，请打开 i2c_dma 项目。有关 DMA 模块的更多详情，请参考器件参考手册。

注

示例代码采用轮询方式实现 DMA 传输。

I²C 通道 0，7 位地址，主机 TX 模式，50000 bps，传输 64 个字节至 I²C 从机。

```
void i2c_dma_master_tx(uint8 channel)
{
    struct dma_tcd tcd1;
    uint8 i;

    /* Init transmit buffer */
    i2c_tx_buffer.tx_index = 0;
    i2c_tx_buffer.rx_index = 0;
    i2c_tx_buffer.data_present = TRUE;
    i2c_tx_buffer.length = 64;
    // fill the data buffer to be transferred.
    for(i=0;i<64;i++)
    {
        i2c_tx_buffer.buf[i] = i;
    }

    iic_init(channel, I2C_MASTER_ADDR, 50000);
    I2C_C1(channel) |= I2C_C1_DMAEN_MASK; //enable dma request

    SIM_SCGC4 |= SIM_SCGC4_DMA_MASK;
    tcd1.channel_no = channel;
    tcd1.ctrl = (0 | DMA_DCR_ERQ_MASK
              | DMA_DCR_SINC_MASK
              | DMA_DCR_CS_MASK
              | DMA_DCR_SSIZE(1)
              | DMA_DCR_DSIZE(1)
```

```
                | DMA_DCR_D_REQ_MASK);
        tcd1.daddr = &(I2C_D(channel));
        tcd1.saddr = &(i2c_tx_buffer.buf[0]);
        tcd1.nbytes = 63;
        dma_config(CONFIG_BASIC_XFR, &tcd1);

        if(channel>3)
        {
            printf("***invalid I2C channel***\r\n");
            return;
        }
        DMA_REQC &= ~(0x0F000000>>(channel*8));
        DMA_REQC |= (0x84000000>>(channel*8)); // dman, id 4

        /* Make sure bus is idle */
        while (I2C_S(channel) & I2C_S_BUSY_MASK);
        /* Put module in master TX mode (generates START) */
        I2C_C1(channel) |= (I2C_C1_MST_MASK | I2C_C1_TX_MASK);
        I2C_D(channel) = ( 0 | (I2C_SLAVE_ADDR<<1) | I2C_TX);

        dma_config(WAIT_FOR_XFR, &tcd1);
        /* Restore module to it's idle (but active) state */
        while (!(I2C_S(channel) & I2C_S_TCF_MASK));
        I2C_C1(channel) = 0x80;
}
```

图 11 中的示例显示如何在主机 TX 轮询模式下使用 DMA 传输数据。

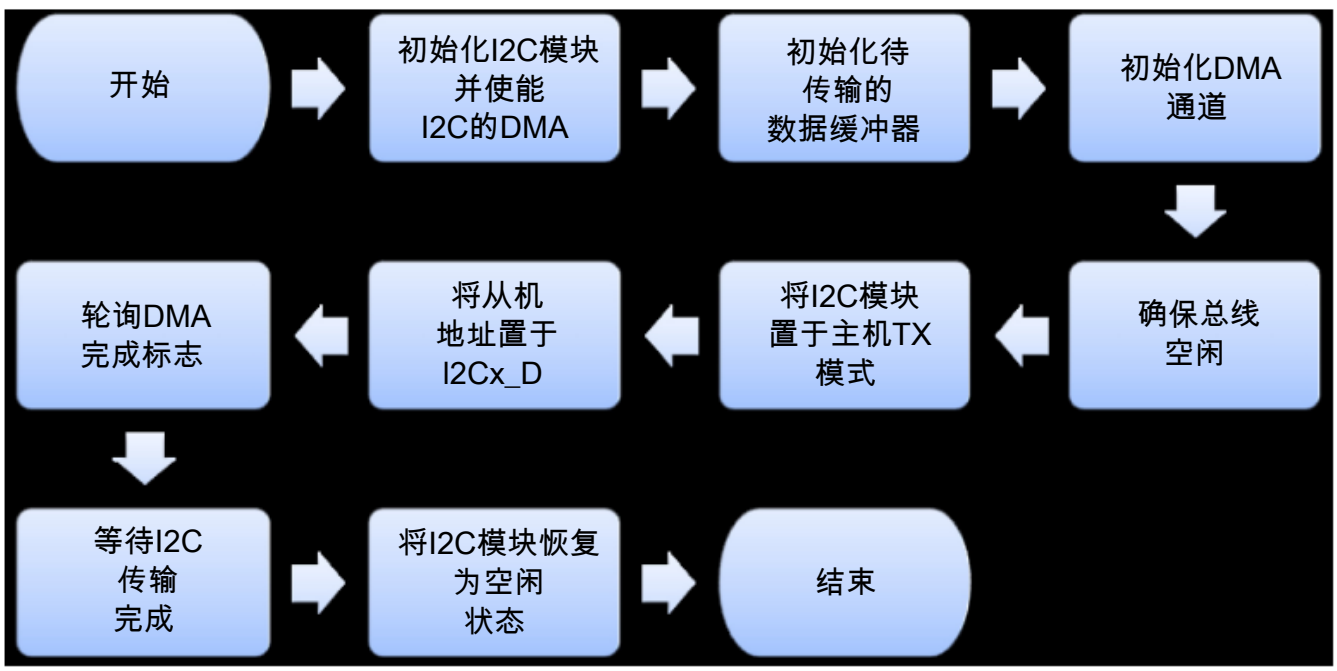

图 11. 在主机 TX 轮询模式下使用 DMA

注

由于必须有一个完整的 1 字节传输中断才能触发 DMA，地址字节必须由 CPU 自行填入 I2Cx_D。

tcd1.nbytes = 63 而非 64，因为如果使用两个相同的 MCF51JF128 板进行测试且从机使用 DMA 接收数据，则从机侧由 DMA 接收的第一个字节将是 addresslTX 字节。有关更多详情，请参见 DMA 从机 RX 源代码 i2c_dma_slave_rx()。

I^2C 通道 0，7 位地址，主机 RX 模式，50000 bit/s，从 I^2C 从机接收 64 个字节。

```
void i2c_dma_master_rx(uint8 channel)
{
    struct dma_tcd tcd1;
    uint8 i;
    printf("*** i2c master dma rx test ***\r\n");
    /* Init transmit buffer */
    i2c_rx_buffer.tx_index = 0;
    i2c_rx_buffer.rx_index = 0;
    i2c_rx_buffer.data_present = FALSE;
    i2c_rx_buffer.length = 64;

    iic_init(channel, I2C_MASTER_ADDR, 50000);

    SIM_SCGC4 |= SIM_SCGC4_DMA_MASK;
    tcd1.channel_no = channel;
    tcd1.ctrl = (0 | DMA_DCR_ERQ_MASK
            | DMA_DCR_CS_MASK
            | DMA_DCR_SSIZE(1)
            | DMA_DCR_DINC_MASK
            | DMA_DCR_DSIZE(1)
            | DMA_DCR_D_REQ_MASK);
    tcd1.saddr = &(I2C_D(channel));
    tcd1.daddr = &(i2c_rx_buffer.buf[0]);
    tcd1.nbytes = 64-2;
    dma_config(CONFIG_BASIC_XFR, &tcd1);

    if(channel>3)
    {
      printf("***invalid I2C channel***\r\n");
      return;
    }
    DMA_REQC &= ~(0x0F000000>>(channel*8));

    /* Make sure bus is idle */
    while (I2C_S(channel) & I2C_S_BUSY_MASK);
    /* Put module in master TX mode (generates START) */
    I2C_C1(channel) |= (I2C_C1_MST_MASK | I2C_C1_TX_MASK);
    /* Put target address into IBDR */
    I2C_D(channel) = ( 0 | (I2C_SLAVE_ADDR<<1) | I2C_RX);
     /* Wait for I2SR[IBB] (bus busy) to be set */
    while (!(I2C_S(channel) & I2C_S_BUSY_MASK));

    /* Wait for address transfer to complete */
    while (!(I2C_S(channel) & I2C_S_IICIF_MASK));
    I2C_S(channel) |= I2C_S_IICIF_MASK;

    /* Clear TX/RX bit in order to set receive mode */
    I2C_C1(channel) &= ~I2C_C1_TX_MASK;

    I2C_C1(channel) |= I2C_C1_DMAEN_MASK; //enable dma request
    DMA_REQC |= (0x84000000>>(channel*8)); // dman, id 4

    /* Dummy read to signal the module is ready for the next byte */
    I2C_D(channel);

    dma_config(WAIT_FOR_XFR, &tcd1);
    while (!(I2C_S(channel) & I2C_S_TCF_MASK));
    I2C_C1(channel) |= I2C_C1_TXAK_MASK;

    I2C_C1(channel) &= ~I2C_C1_DMAEN_MASK; //disable dma request
    DMA_REQC = 0; // disable dma req

    i2c_rx_buffer.buf[62] = I2C_D(channel);

    /* receive last byte */
    /* Wait for transfer to complete */
    while (!(I2C_S(channel) & I2C_S_IICIF_MASK));
    I2C_S(channel) |= I2C_S_IICIF_MASK;

    /* Generate STOP */
```

```
        I2C_C1(channel) &= ~I2C_C1_MST_MASK;

        i2c_rx_buffer.buf[63] = I2C_D(channel);

        /* Restore module to it's idle (but active) state */
        I2C_C1(channel) = 0x80;

         for(i=0;i<64;i++)
         {
         if(i%16 == 0)
         {
            printf("\r\n");
         }
         printf("%02x ", i2c_rx_buffer.buf[i]);
 }
}
```

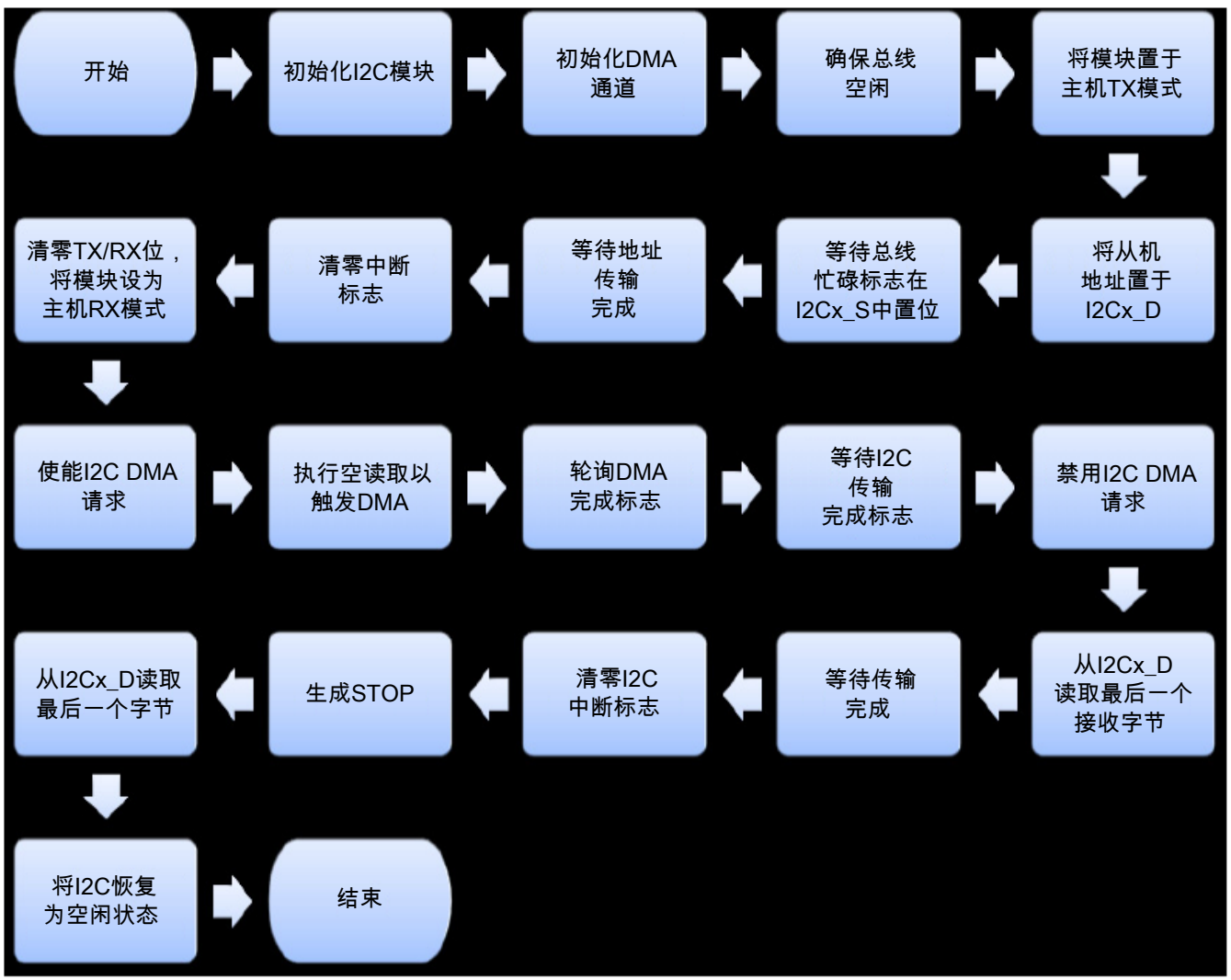

图 12. 在主机 RX 轮询模式下使用 DMA

注

由于在主机 RX 模式下，执行地址周期后必须置位某些必要的控制位，可在完成地址周期和传输方向发生改变后开始 DMA 传输。

接收倒数第二个字节之前，必须停止 DMA 传输。此时应设置 TXACK。

I²C 通道 0，7 位地址，从机 TX 模式，从 I²C 主机接收 64 个字节。

```
    void i2c_dma_slave_tx(uint8 channel)
    {
        struct dma_tcd tcd1;
        uint8 i;

        printf("**** i2c dma slave rx test ****\r\n");
        /* Init transmit buffer */
        i2c_rx_buffer.tx_index = 0;
        i2c_rx_buffer.rx_index = 0;


        i2c_rx_buffer.data_present = FALSE;
        i2c_rx_buffer.length = 64;

        iic_init(channel, I2C_SLAVE_ADDR, 50000);

        SIM_SCGC4 |= SIM_SCGC4_DMA_MASK;
        tcd1.channel_no = channel;
        tcd1.ctrl = (0 | DMA_DCR_ERQ_MASK
                    | DMA_DCR_CS_MASK
                    | DMA_DCR_SSIZE(1)
                    | DMA_DCR_DINC_MASK
                    | DMA_DCR_DSIZE(1)
                    | DMA_DCR_D_REQ_MASK);
        tcd1.saddr = &(I2C_D(channel));
        tcd1.daddr = &(i2c_rx_buffer.buf[0]);
        tcd1.nbytes = 64;
        dma_config(CONFIG_BASIC_XFR, &tcd1);

        if(channel>3)
        {
           printf("***invalid I2C channel***\r\n");
           return;
        }
        DMA_REQC &= ~(0x0F000000>>(channel*8));

        I2C_C1(channel) |= I2C_C1_DMAEN_MASK; //enable dma request
        DMA_REQC |= (0x84000000>>(channel*8)); // dman, id 4

        dma_config(WAIT_FOR_XFR, &tcd1);

        I2C_C1(channel) &= ~I2C_C1_DMAEN_MASK; //disable dma request
        DMA_REQC = 0; // disable dma req

        for(i=0;i<64;i++)
        {
        if(i%16 == 0)
        {
              printf("\r\n");
        }
        printf("%02x ", i2c_rx_buffer.buf[i]);
    }
}
```

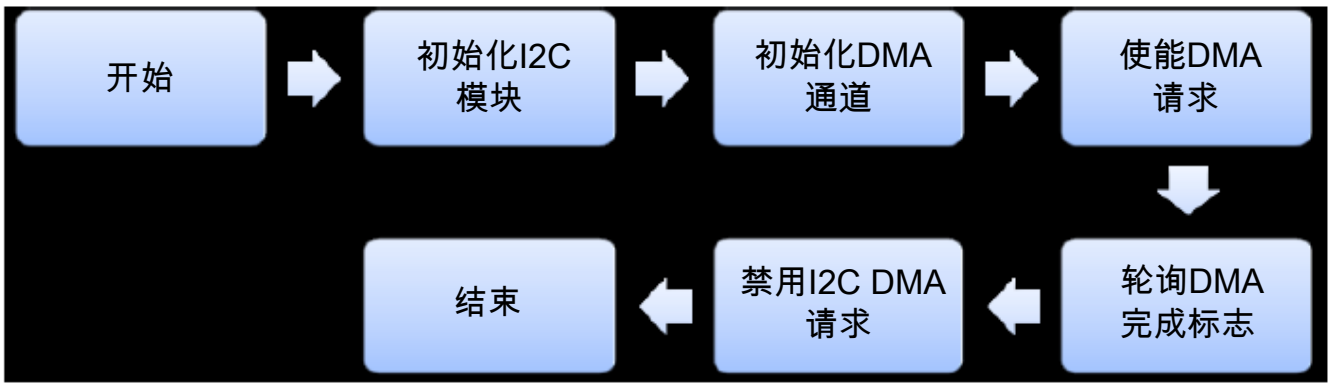

图 13. 在从机 RX 轮询模式下使用 DMA

I^2C 通道 0，7 位地址，从机 TX 模式，从 I^2C 主机接收 64 个字节。

```
void i2c_dma_slave_tx(uint8 channel)
{
  struct dma_tcd tcd1;
  uint8 i;

  /* Init transmit buffer */
  i2c_tx_buffer.tx_index = 0;
  i2c_tx_buffer.rx_index = 0;
  i2c_tx_buffer.data_present = TRUE;
  i2c_tx_buffer.length = 64;
  i2c_tx_buffer.buf[0] = (uint8)(I2C_SLAVE_ADDR&0x0FF);

  for(i=1;i<64;i++)
  {
    i2c_tx_buffer.buf[i] = i;
  }

  iic_init(channel, I2C_SLAVE_ADDR, 50000);

  SIM_SCGC4 |= SIM_SCGC4_DMA_MASK;
  tcd1.channel_no = channel;
  tcd1.ctrl = (0 | DMA_DCR_ERQ_MASK
            | DMA_DCR_SINC_MASK
            | DMA_DCR_CS_MASK
            | DMA_DCR_SSIZE(1)
            | DMA_DCR_DSIZE(1)
            | DMA_DCR_D_REQ_MASK);
  tcd1.daddr = &(I2C_D(channel));
  tcd1.saddr = &(i2c_tx_buffer.buf[1]);
  tcd1.nbytes = 63;
  dma_config(CONFIG_BASIC_XFR, &tcd1);

  if(channel>3)
  {
     printf("***invalid I2C channel***\r\n");
     return;
  }
  DMA_REQC &= ~(0x0F000000>>(channel*8));
  DMA_REQC |= (0x84000000>>(channel*8)); // dman, id 4

  while (!(I2C_S(channel) & I2C_S_IICIF_MASK));
  I2C_S(channel) |= I2C_S_IICIF_MASK;

  if (I2C_S(channel) & I2C_S_IAAS_MASK)
  {
     if (I2C_S(channel) & I2C_S_SRW_MASK)
     {
         /* Set tx_index to 0 */
        i2c_tx_buffer.tx_index = 0;
```

```
            /* Master was reading from slave - Set Transmit Mode. */
            I2C_C1(channel) |= I2C_C1_TX_MASK;

            I2C_C1(channel) |= I2C_C1_DMAEN_MASK; //enable dma request

            I2C_D(channel) = i2c_tx_buffer.buf[0];

            dma_config(WAIT_FOR_XFR, &tcd1);

            I2C_C1(channel) &= ~I2C_C1_DMAEN_MASK; //disable dma request
            DMA_REQC = 0; // disable dma req

            return;
          }
      }
      printf("error occurs!\r\n");
}
```

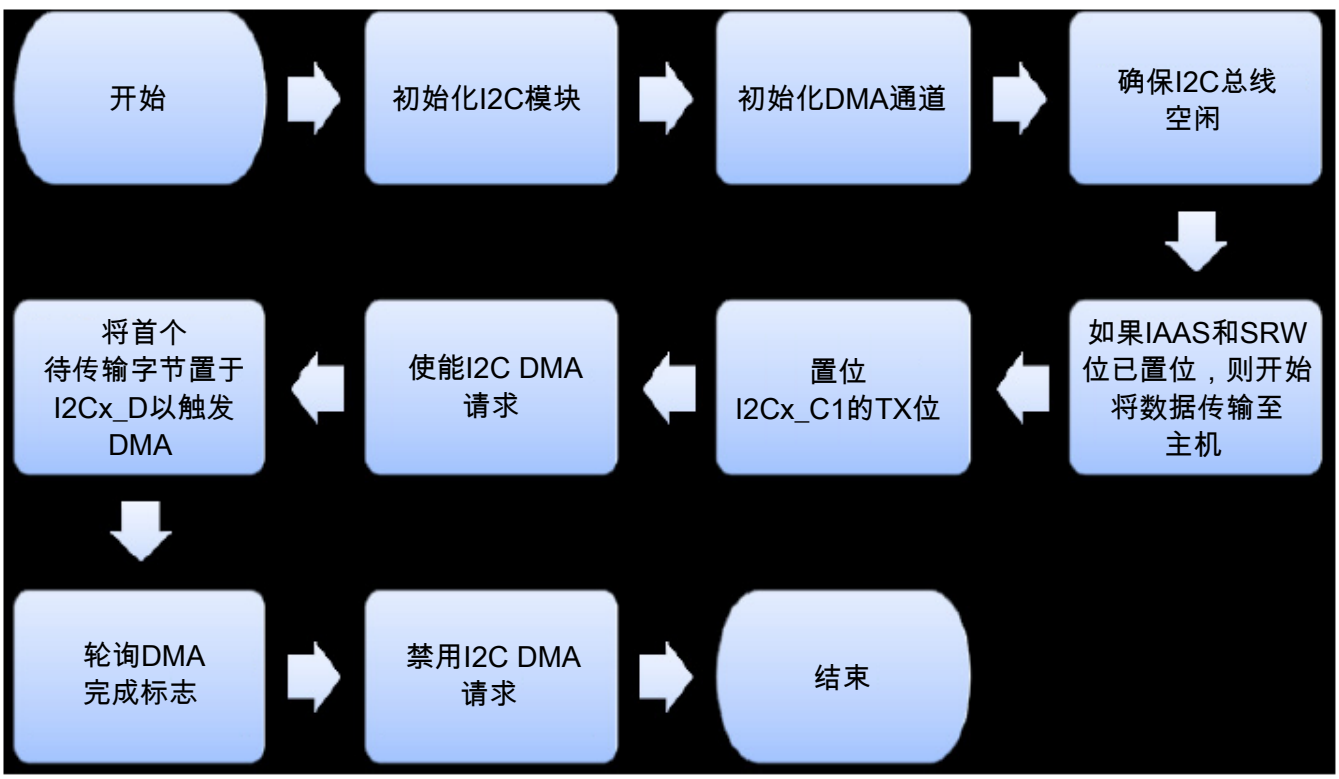

图 14. 在从机 TX 轮询模式下使用 DMA

注

完成地址周期并通过将第一个数据加载至 I2Cx_D 触发 DMA 之后，开始 DMA 传输。

4.4.4 SMBus 相关代码

用于测试 SMBus 的示例代码可在 i2c_smb_basic 项目中找到。常规 I^2C API 和 SMBus API 之间并无太大区别，但初始化 SMBus 时可能需要置位 ALERTEN 和 SIICAEN；可参考章节使用中断 。有关更多详细信息，还可查看 i2c.c 中的 i2csmbus_handler()。

注

根据图 6，部分操作需要 1 位 SCL 延迟。示例代码中，该延迟通过模数定时器 (MTIM) 来实现。有关 MTIM 的更多信息，请参见器件参考手册的“模数定时器”章节。

根据 SMBus 协议（PEC），在主机或从机 RX 模式时，ARP 处理期间需要软 CRC。I2Cx_SMB[ALERTEN]和 I2Cx_SMB[SIICAEN]应正确设置。

5 结论

最新 I^2C 模块支持 10 位地址模式和 7 位地址模式，并添加了对 SMBus 协议支持和 DMA 支持，降低了 CPU 负载。它还允许使用轮询和中断方法。

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number AN4342
Revision 0, 08/2011