Freescale Semiconductor
Application Note

Document Number: AN4342
Rev. 0, 08/2011

Using the Inter-Integrated Circuit on ColdFire+ and Kinetis

I^{2}C driver for the MCF51JF128

by: Ju Yingyi
32-bit Applications
Microcontroller Solutions Group

This application note describes how to use the new inter-integrated circuit (I^{2}C) module on Kinetis and Coldfire+ chips. Though the contents of this document are based on the I^{2}C module of the MCF51JF128 and all sample code has been tested on this device, the driver described here can be migrated to any other Coldfire+ or Kinetis chip with little modification.

Contents

1 Introduction

The new I^{2}C module on Freescale's Kinetis & Coldfire+ devices is compatible with both *The I^{2}C-Bus Specification* and also *System Management Bus (SMBus) Specification, version 2*. The new module also adds DMA support to reduce MCU loading. The following pages introduce these new features and provide sample I^{2}C driver code as well.

Refer to the latest silicon and board documentation for updates to the information in this application note. This document was written using the most recent documentation available from www.freescale.com. The I^{2}C or SMBus specifications at www.i2c-bus.org.

2 Overview of the I^{2}C module

The inter-integrated circuit—frequently abbreviated IC, I^{2}C, or IIC—module provides a method of communication between a number of devices. The interface is designed to operate at up to 100 kbit/s with maximum bus loading and timing. The device is capable of operating at higher baud rates, up to a maximum of bus clock ÷ 20, with reduced bus loading. ***The maximum communication length and the number of devices that can be connected are limited by a maximum bus capacitance of 400 pF***. The I^{2}C module also complies with the *System Management Bus (SMBus) Specification, version 2*.

NOTE

Freescale's I^{2}C is targeted for compliance to I^{2}C specifications for standard (100kHz) and fast mode (400kHz). The maximum communication length and the number of devices that can be connected are limited by a maximum bus capacitance of 400 pF

This module does not have a special circuit to pull up the SCL and SDA lines to achieve high speed mode (up to 3.4MHz) compliance, although you may set the maximum baud rate of the I^{2}C module up to bus clock ÷ 20.

2.1 Features

The I^{2}C module has these distinctive features:

- *The I^{2}C-Bus Specification* compatibility
- Multimaster operation
- Software-programmable operation for one of 64 different serial clock frequencies
- Software-selectable acknowledge bit
- Interrupt-driven byte-by-byte data transfer
- Arbitration-lost interrupt with automatic mode switching from master to slave
- Calling address identification interrupt
- START and STOP signal generation and detection
- Repeated START signal generation and detection
- Acknowledge bit generation and detection
- Bus busy detection

Compared with Coldfire V2 devices (MCF52xx, MCF5225x), this new I^{2}C module also supports:

- 10-bit address extension
- General call recognition
- Programmable glitch input filter

Compared with latest Coldfire V1 devices (MCF51CN128, MCF51MM256/MCF51MM128), this newly-designed module also supports:

- *System Management Bus (SMBus) Specification, version 2* compatibility
- Low power mode wakeup on slave address match
- Use of a range slave address
- DMA operation

2.2 Modes of operation

The following is a list of the I^{2}C module's operation in various low power modes:

- Run mode: This is the basic mode of operation. To conserve power in this mode, disable the module.
- Wait mode: The module continues to operate when the core is in wait mode and can provide a wakeup interrupt.
- Stop mode: The module is inactive in stop mode for reduced power consumption, except that address matching is enabled. The STOP instruction does not affect the I^{2}C module's register states. In any VLLSx mode, the register contents are reset.

NOTE

For more information on the power modes of Kinetis & Coldfire+ devices, please refer to the individual device reference manuals, available from www.freescale.com

2.3 Block diagram

Figure 1 is a block diagram of the I^{2}C module.

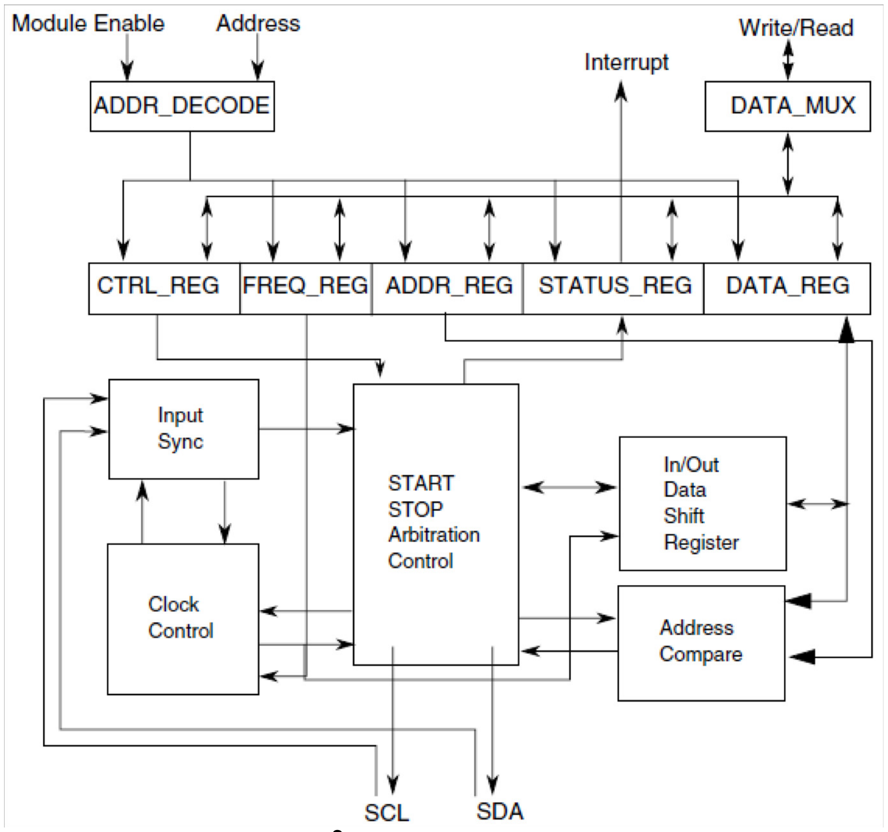


Figure 1. I^{2}C functional block diagram

3 Function description

In this section, we will skip I^{2}C protocol in order to focus on some special features of the new I^{2}C module.

3.1 I^{2}C baud rate

According to the reference manual's I^{2}C Frequency Divider register (I2Cx_F) description, the I^{2}C data transfer baud rate is determined by the bus clock and the I2Cx_F register. It can be calculated with the following the equation:

I^{2}C baud rate = bus speed (Hz) ÷ (mul × SCL divider)
mul = 1, 2, or 4 when I2Cx_F[MULT] = 0b00, 01, or 10.
SCL divider: determined by I2Cx_F[ICR], values found in the I^{2}C Divider and Hold Values table in the I^{2}C chapter of your device's reference manual. ***Eqn. 1***

You may find that setting this register sometimes does not produce the exact baud rate that you need. In this case, the closest value could be used.

For example, assume that the bus clock is 25 MHz. In order to generate a 50 kHz clock on the SCL pad, setting I2Cx_F = 0x97, 0x63, or 0x2B could result in 48828 Hz.

The *int _i2c_set_bps(uint8 channel, uint32 bps)* function can be used to find the value of I2Cx_F that will generate the baud rate closest to the one that you require. Please read function *_i2c_set_bps*() in i2c.c for details.

3.2 Interrupts

The I^2C module generates an interrupt when any of the events in the following table occur, provided that the IICIE bit is set. The interrupt is driven by the IICIF bit (of the I^2C Status Register) and masked with the IICIE bit (of the I^2C Control Register 1). The IICIF bit must be cleared (by software) by writing 1 to it in the interrupt routine. The SMBus timeouts interrupt is driven by SLTF and masked with the IICIE bit. The SLTF bit must be cleared by software by writing 1 to it in the interrupt routine. You can determine the interrupt type by reading the status register.

Table 1. Interrupt summary

Interrupt source	Status	Flag	Local enable
Complete 1-byte transfer	TCF	IICIF	IICIE
Match of received calling address	IAAS	IICIF	IICIE
Arbitration lost	ARBL	IICIF	IICIE
SMBus SCL low timeout interrupt flag	SLTF	IICIF	IICIE
SMBus SCL high SDA low timeout interrupt flag	SHTF2	IICIF	IICIE & SHTF2IE
Wakeup from stop interrupt	IAAS	IICIF	IICIE & WUEN

Figure 2 below is the typical I^2C interrupt routine. You can follow this routine to design the I^2C driver.

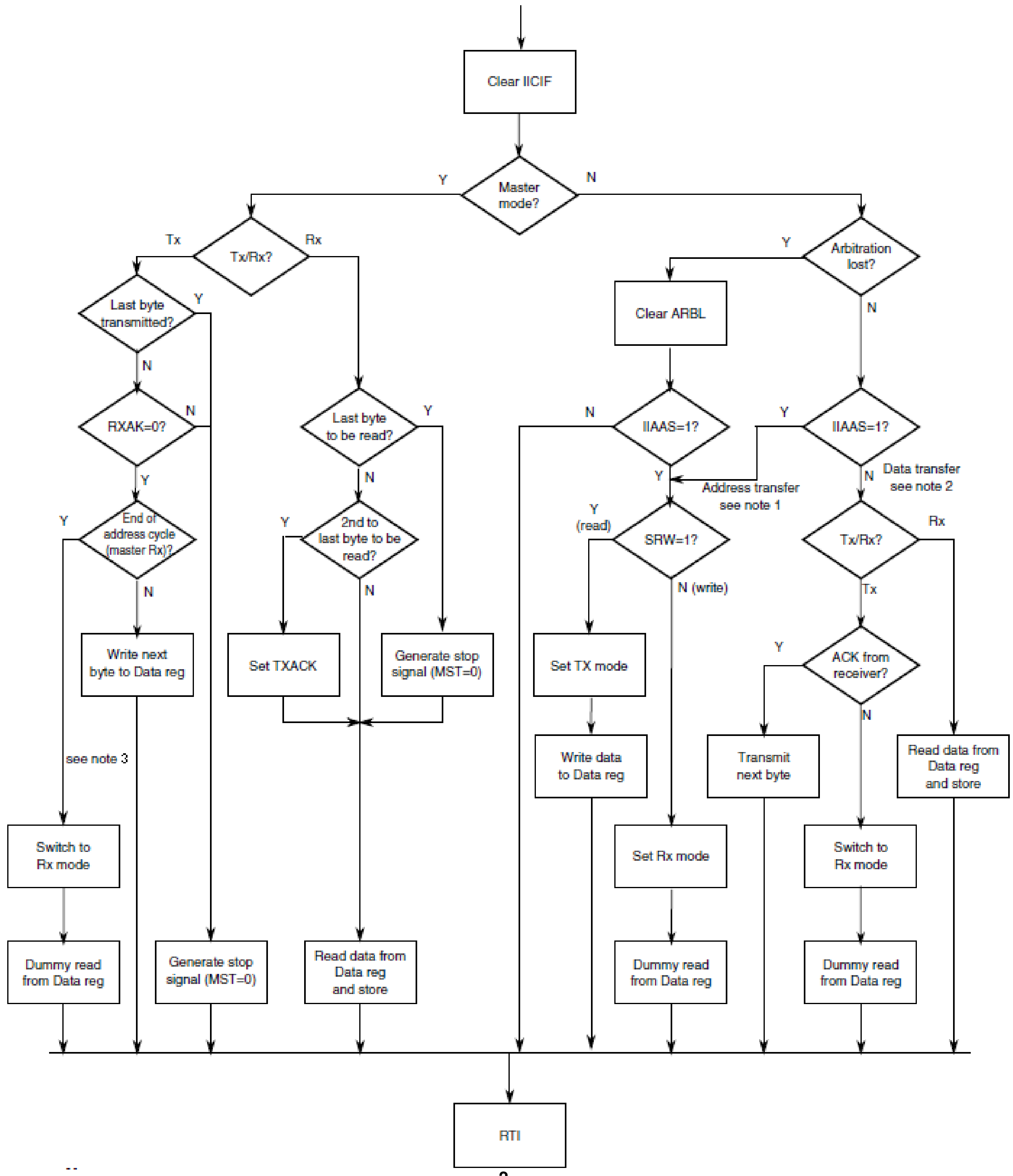


Figure 2. Typical I²C interrupt routine

1 If general call is enabled, check to determine whether the received address was a general call address (0x00). If the received address was a general call address, the general call must be handled by user software.

2 When 10-bit addressing addresses a slave, the slave sees an interrupt following the first byte of the extended address. Ensure that, for this interrupt, the contents of the data register are ignored and not treated as a valid data transfer.

3 When in master receive mode and there is only 1 byte to be received, TXACK should be set before dummy read.

3.3 SMBus support

The new I^{2}C module can support SMBus well. For more detailed information, please refer to the latest reference manual for your Kinetis or Coldfire+ MCU. For details on the SMBus specification, please refer to *System Management Bus (SMBus) specification, version 2.0*.

3.3.1 Timeouts

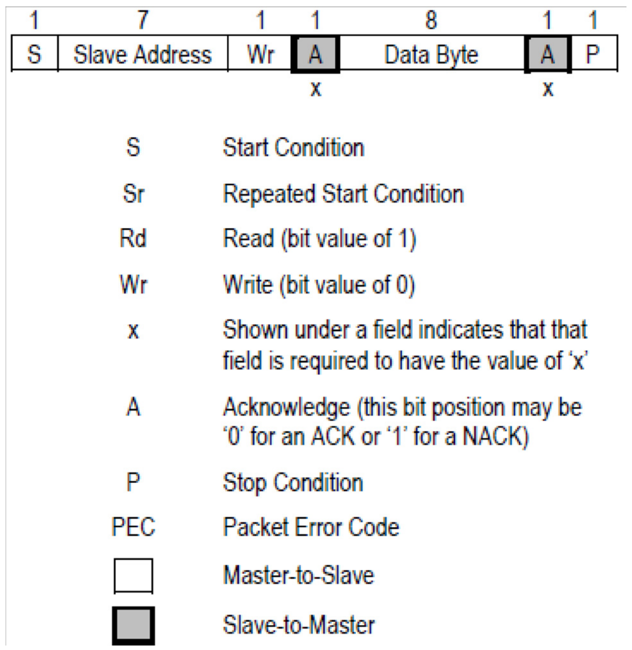
The $T_{TIMEOUT,MIN}$ parameter allows a master or slave to conclude that a defective device is holding the clock low indefinitely or a master is intentionally trying to drive devices off of the bus. It is highly recommended that a slave device release the bus (stop driving the bus and allow SCL and SDA to float high) when it detects any single clock held low longer than $T_{TIMEOUT,MIN}$. Devices that have detected this condition must reset their communication and be able to receive a new START condition within the timeframe of $T_{TIMEOUT,MAX}$.

SMBus defines a clock low timeout, $T_{TIMEOUT}$, of 35 ms, specifies $T_{LOW:SEXT}$ as the cumulative clock low extend time for a slave device, and specifies $T_{LOW:MEXT}$ as the cumulative clock low extend time for a master device.

3.3.2 FACK and NACK

To improve reliability and communication robustness, packet error checking (PEC) implementation is optional for SMBus devices, but required for devices participating in and only during the address resolution protocol (ARP) process. The PEC is a CRC-8 error checking byte, calculated on all the message bytes. The PEC is appended to the message by the device that supplied the last data byte. If the PEC is present but not correct, a NACK is issued by the receiver. Otherwise an ACK is issued. In order to calculate the CRC-8 with software, this module can hold the SCL line low after receiving the eighth SCL (8th bit) if this byte is a data byte. So software can determine whether an ACK or NACK should be sent to the bus by setting or clearing the TXAK bit if the FACK (fast ACK/NACK enable) bit is enabled.

Figure 3 below is SMBus packet protocol diagram element key.



1 7 1 1 8 1 1
S | Slave Address | Wr | A | Data Byte | A | P
x x
S Start Condition
Sr Repeated Start Condition
Rd Read (bit value of 1)
Wr Write (bit value of 0)
x Shown under a field indicates that that field is required to have the value of 'x'
A Acknowledge (this bit position may be '0' for an ACK or '1' for a NACK)
P Stop Condition
PEC Packet Error Code
Master-to-Slave
Slave-to-Master

Figure 3. SMBus packet protocol diagram element key

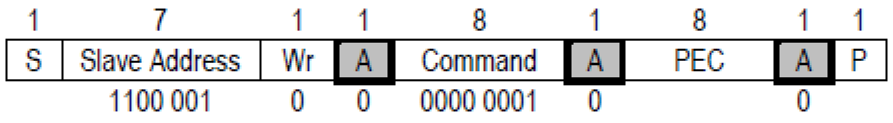


Figure 4. Standard SMBus send byte protocol with PEC

SMBus requires that a device always acknowledge its own address as a mechanism to detect the presence of a removable device (such as a battery or docking station) on the bus. In addition to indicating a slave device busy condition, SMBus uses the NACK mechanism to indicate the reception of an invalid command or invalid data. Because such a condition may occur on the last byte of the transfer, SMBus devices are required to have the ability to generate the not acknowledge after the transfer of each byte and before the completion of the transaction. This requirement is important because SMBus does not provide any other resend signaling. This difference in the use of the NACK signaling has implications on the specific implementation of the SMBus port, especially in devices that handle critical system data such as the SMBus host and the SBS components.

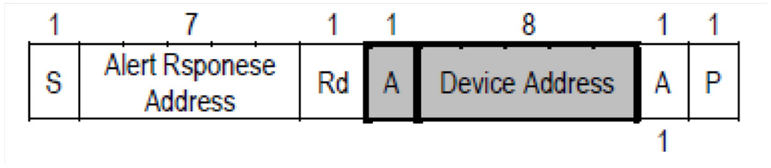
NOTE

In the last byte of master receive slave transmit mode, the master must send a NACK to the bus, so FACK must be switched off before the last byte transmits.

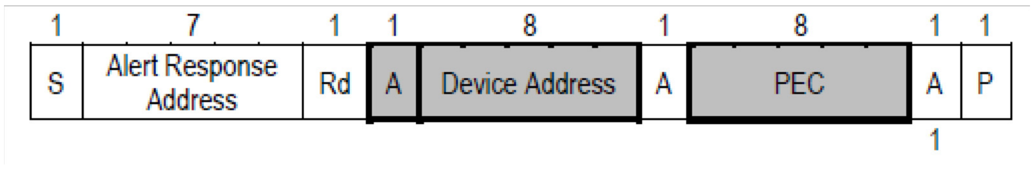
3.3.3 ALERTEN and SIICAEN

These two bits are found in the I2C*x*_SMB register. To support SMBus specified protocols, ARP, ARA (Alert Response Address), these two bits should be set, and I2C*x*_A2 should be set to the corresponding address (SMBus Device Default Address 0x61<<1, SMBus Alert Response Address 0x0C<<1).

However, an additional interrupt line (SMBALERT#) between the SMBus host and device may be needed to support ARA. A slave can signal the host, through this line, that it wants to talk. Figure 5 below shows the ARA format. For more details, please refer to *System Management Bus (SMBus) Specification, version 2*.



A 7-bit addressable device responds to an ARA

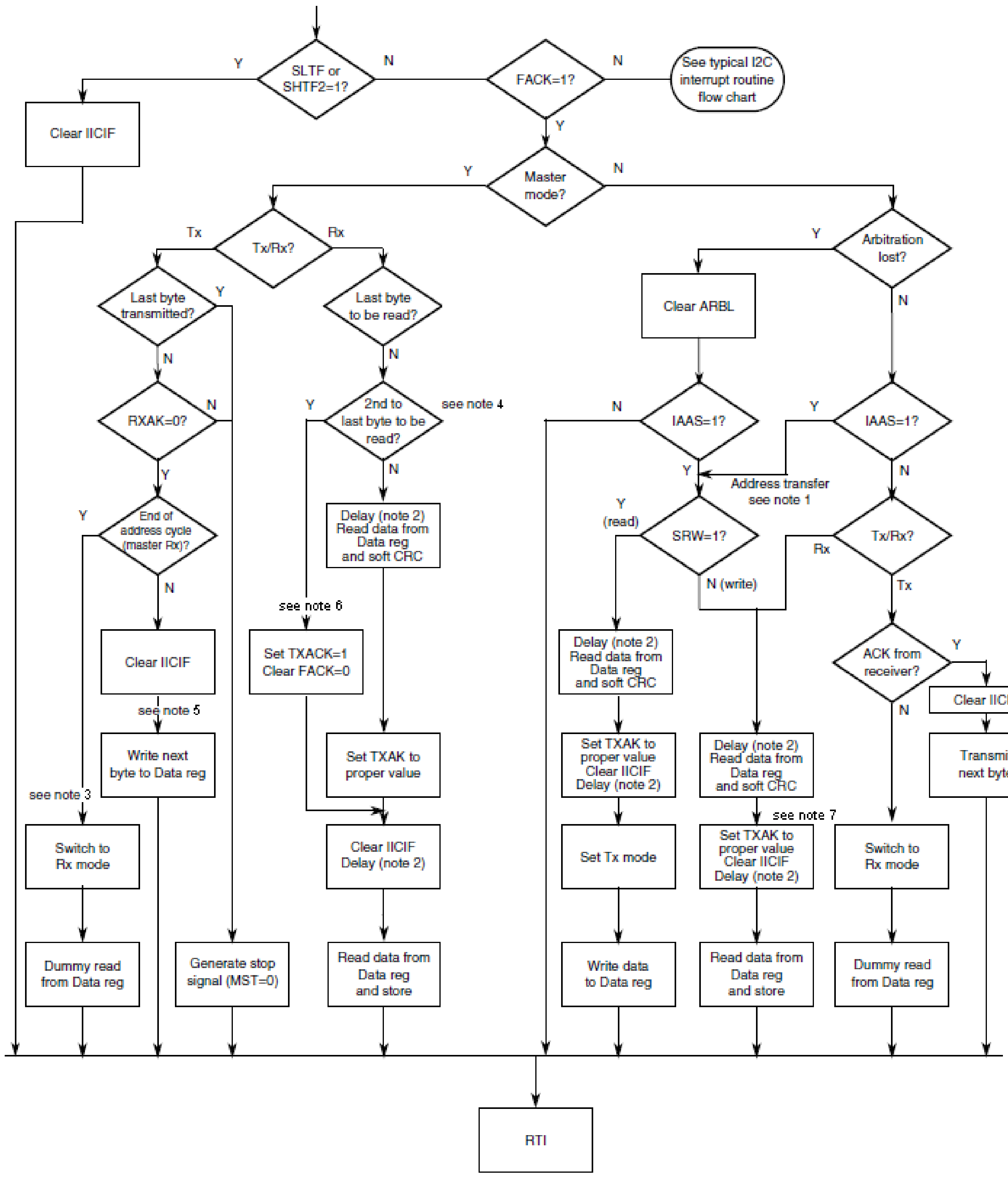


A 7-bit addressable device responds to an ARA with PEC

Figure 5. Format of SMBus ARA Command

3.3.4 Typical SMBus interrupt routine

Figure 6 below is the typical SMBus interrupt routine, appended with additional notes on its operation.



SLTF or SHTF2=1?
Y
N
FACK=1?
N
See typical I2C interrupt routine flow chart
Y
Clear IICIF
Master mode?
Y
N
Tx/Rx?
Tx
Rx
Arbitration lost?
Y
N
Clear ARBL
Last byte transmitted?
Y
N
Last byte to be read?
N
RXAK=0?
N
Y
2nd to last byte to be read?
Y
see note 4
N
IAAS=1?
N
Y
IAAS=1?
Y
N
Address transfer see note 1
End of address cycle (master Rx)?
Y
N
Delay (note 2) Read data from Data reg and soft CRC
Y (read)
SRW=1?
N (write)
Tx/Rx?
Rx
Tx
see note 6
Clear IICIF
Set TXACK=1 Clear FACK=0
Delay (note 2) Read data from Data reg and soft CRC
ACK from receiver?
Y
N
Clear IICIF
see note 5
Write next byte to Data reg
Set TXAK to proper value
Set TXAK to proper value Clear IICIF Delay (note 2)
Delay (note 2) Read data from Data reg and soft CRC
Transmit next byte
see note 3
see note 7
Switch to Rx mode
Clear IICIF Delay (note 2)
Set Tx mode
Set TXAK to proper value Clear IICIF Delay (note 2)
Switch to Rx mode
Dummy read from Data reg
Generate stop signal (MST=0)
Read data from Data reg and store
Write data to Data reg
Read data from Data reg and store
Dummy read from Data reg
RTI

Figure 6. Typical SMBus interrupt routine

1 If general call or SIICAEN is enabled, check to determine if the received address is a general call address (0x00) or an SMBus device default address. In either case, they must be handled by user software.

2 In receive mode, one bit time delay may needed before the first and second data reading.

3 When in master receive mode and there is only 1 byte to be received, TXACK should be set before dummy read.

[4] Because, when processing second-to-last byte transferring, FACK has been cleared, there is no need to process last byte transferring case here, the last byte transferring will follow the normal I^{2}C interrupt routine.

[5] If sending second-to-last byte, FACK should be cleared here.

[6] Before setting TXACK to 1 and clearing FACK, you should delay 1 bit CLK, read data from the data register, perform a soft CRC, ACK the current byte by setting TXACK = 0, and delay 1 bit CLK again.

[7] A second-to-last byte transfer check routine needs to be added here. If yes, then ACK the current byte by setting TXACK = 0 and delaying the 1-bit CLK. Also, you must check whether SIICAEN is set or not; if yes, just clear FACK and set TXACK = 1.

3.4 Programmable input glitch filter

An I^{2}C glitch filter has been added outside legacy I^{2}C modules but within the I^{2}C package. This filter can absorb glitches on the I^{2}C clock and data lines for the I^{2}C module. The width of the glitch to be absorbed can be specified in terms of the number of (half) bus clock cycles. A single Programmable Input Glitch Filter control register is provided. Effectively, any down-up-down or up-down-up transition on the data line that occurs within the number of clock cycles programmed in this register is ignored by the I^{2}C module. The programmer must specify the size of the glitch (in terms of bus clock cycles) for the filter to absorb and not pass.

3.5 Address matching wakeup

When a primary, range, or general call address match occurs with the I^{2}C module in slave receive mode, the MCU wakes from low power mode with no peripheral bus running. After the address matching IAAS bit is set, an interrupt is sent at the end of address matching to wake the core. The IAAS bit must be cleared after the clock recovery.

NOTE

After the system recovers and is in run mode, restart the I^{2}C module if necessary. The SCL line is not held low until the I^{2}C module resets after address matching.

NOTE

There is a bug associated with I^{2}C address matching wakeup from stop mode: if the first data package contains the wrong address, the MCU will not wake up, even if the subsequent data package contains the correct address. This bug is documented in the most recent revision of the errata report for your device.

3.6 DMA support

If the DMAEN bit is cleared and the IICIE bit is set, an interrupt condition generates an interrupt request. If the DMAEN and IICIE bits are set, an interrupt condition generates a DMA request instead. DMA requests are generated by the transfer complete flag (TCF).

If the DMAEN bit is set, the only arbitration lost is to another I^{2}C module (error), and SCL low timeouts (error) generate CPU interrupts. All other events initiate a DMA transfer.

NOTE

Before the last byte of master receive mode, TXAK must be set to send a NACK after the last byte's transfer. Therefore, the DMA must be disabled before the last byte's transfer.

NOTE

In 10-bit address mode transmission, the addresses to be sent occupy 2–3 bytes. During this transfer period, the DMA must be disabled because the C1 register is written to send a repeat start or to change the transfer direction.

4 Sample I^{2}C driver

This section introduces the standalone typical I^{2}C driver for your reference. All the prototypes of APIs can be found in *i2c.c.*

This driver has been tested on the MCF51JF128. It may be fully reused for Coldfire+ or Kinetis devices.

4.1 Key macros

All predefined key macros can be found in i2c_cfg.h of each demo project.

4.1.1 I2C_POLLING_MODE

Use the polling method if it is defined to 1, else use the interrupt method.

4.1.2 I2C_BUFFER_SIZE

Defines I^{2}C TX and RX buffer size.

4.1.3 I2C_DEBUG

Will output debugging message via default UART port if it is defined to 1 (affects performance).

4.2 Global Variables

4.2.1 i2c_tx_buffer

I^{2}C TX buffer.

Prototype:

```
I2C_BUFFER i2c_tx_buffer;
```

Remarks:

```
/* 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;
```

You should fill the buffer structure before transferring.

4.2.2 i2c_rx_buffer

I^{2}C RX buffer.

Prototype:

I2C_BUFFER i2c_rx_buffer;

Remarks:

Please see the remarks of i2c_tx_buffer.

4.3 API

4.3.1 i2c_init

Initializes the specified I^{2}C channel. Address mode is 7-bit.

Prototype:

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

Parameters:

channel: I^{2}C channel number specified by user
addr: primary slave address
bps: baud rate (Hz)

Return value:

none

Remarks:

IICEN, IICIE settings depends on the value of I2C_POLLING_MODE. If I2C_POLLING_MODE = 1, then IICEN & IICIE will be set.

4.3.2 i2c_10bit_init

Initializes the specified I^{2}C channel. Address mode is 10-bit.

Prototype:

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

Parameters:

channel: I^{2}C channel number specified by user

addr: primary slave address

bps: baud rate (Hz)

Return value:

none

Remarks:

IICEN, IICIE settings depends on the value of I2C_POLLING_MODE. If I2C_POLLING_MODE = 1, then IICEN & IICIE will be set.

4.3.3 smbus_init

Initializes the specified I^{2}C channel. Behavior of this channel will follow SMBus protocol.

Prototype:

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

Parameters:

channel: I^{2}C channel number specified by user

addr: primary slave address

sec_addr: SMBus address

bps: baud rate (Hz)

Return value:

none

4.3.4 i2c_master

General function for performing I^{2}C master transfers.

Prototype:

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

Parameters:

channel: I^{2}C channel number specified by user

mode: Valid modes include I2C_TX, I2C_RX, I2C_TXRX, I2C_10BIT_TX, I2C_10BIT_RX, I2C_10BIT_TXRX (all modes defined in i2c.h)

slave_address: the slave address

Return value:

none

4.3.5 i2c*x*_isr

General I²C handler, this function is created by following the flowchart of Figure 2 and Figure 6.

Prototype:

interrupt void ***i2cx_isr***(void)

Parameters:

none:

Return Value:

none:

Remarks:

If SMBus mode isn't used, calling i2c_handler() in interrupt service routine instead of i2csmbus_handler() is OK.

4.4 Using the I²C driver

The sample code is for CodeWarrior 10.x only. To open the test project, extract the code first, then select "File→Import…" as in Figure 7 below.

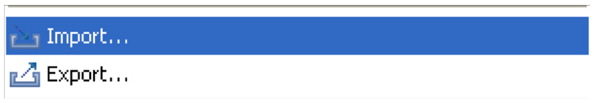


Figure 7. How to open sample project in CW10.x, step 1

Next, select "Existing Projects into Workspace" and click "Next >".

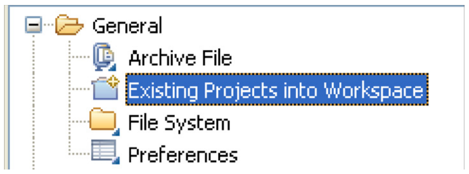


Figure 8. How to open sample project in CW10.x, step 2

Then select the project folder and click "Next >". All sample projects will be listed, then you can select one, some or all of them to open.

4.4.1 Polling mode

Sample code can be found in the i2c_basic project.

First, define ***I2C_POLLING_MODE*** to 1 in i2c_cfg.h.

I²C channel 3, 7-bit address, master TX mode, 50000bps, transfer 64 bytes to I²C slave which address is 0x21 (defined in i2c_cfg.h):

```
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);
}
```

Figure 9 shows how i2c_master() works when the module is put into master TX mode using polling mode.

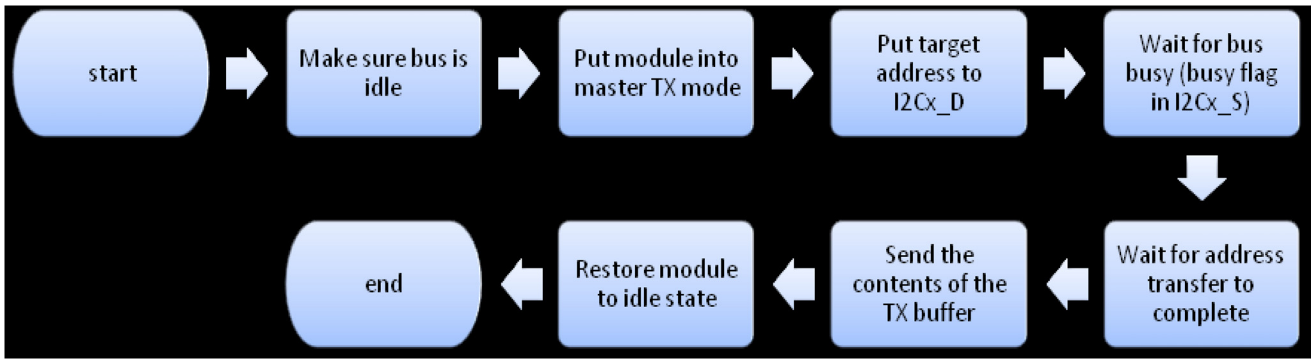

start
Make sure bus is idle
Put module into master TX mode
Put target address to I2Cx_D
Wait for bus busy (busy flag in I2Cx_S)
Wait for address transfer to complete
Send the contents of the TX buffer
Restore module to idle state
end

Figure 9. Flowchart of Master TX Polling Mode

I²C channel 0, 7-bit address, master RX mode, 50000bps, receive 64 bytes from slave which address is 0x21 (defined in i2c_cfg.h):

```
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);
}
```

The following figure shows how i2c_master() works when the module is put into master RX mode using polling mode.

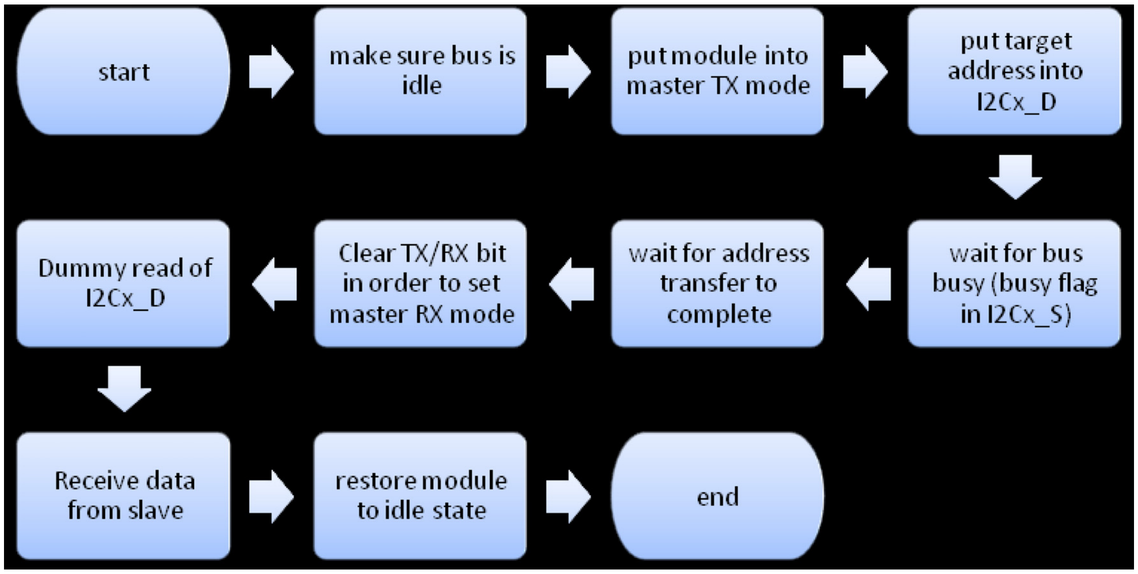

start
make sure bus is idle
put module into master TX mode
put target address into I2Cx_D
wait for bus busy (busy flag in I2Cx_S)
wait for address transfer to complete
Clear TX/RX bit in order to set master RX mode
Dummy read of I2Cx_D
Receive data from slave
restore module to idle state
end

Figure 10. Master RX polling mode

NOTE

At the “Receive data from slave” stage, you must follow the master RX routine of a typical I²C interrupt routine.

4.4.2 Using interrupts

Sample code can be found in the i2c_10bit project. For 7-bit address mode, please open the i2c_interrupt project.

First, define ***I2C_POLLING_MODE*** to 0 and, in i2c_cfg.h, fill in the vector table with i2c*x*_isr in exceptions.c. Also, don’t forget to enable the interrupt of the I²C channel you are using.

I²C channel 0, 10-bit address, master TX mode, 50000bps, transfer 64 bytes to I²C slave:

```
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);
    }
```

I^{2}C channel 0, 10-bit address, master RX mode, 50000bps, receive 64 bytes from I^{2}C slave. Please take note of the comments in the code below:

```
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);
}
```

I^{2}C channel 0, 10-bit address, slave RX mode, 50000 bit/s (actually, when in slave mode, I2C*x*_F will be ignored), receive 64 bytes from I^{2}C master.

```
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);
}
```

I^{2}C channel 0, 10-bit address, slave TX mode, 50000 bit/s (actually, when in slave mode, I2C*x*_F will be ignored), transfer 64 bytes to I^{2}C master after addressed by master.

```
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);
}
```

NOTE

Please refer to Figure 2 for a diagram of a typical I^{2}C interrupt routine.

Commonly speaking, the slave baud rate follows the master baud rate. However, the SBRC field in register I2C*x*_C2 is a special bit used to force clock stretching on SCL in very fast I^{2}C modes.

4.4.3 Using DMA

This module works smoothly with DMA. Open the i2c_dma project for sample code. Please refer to your device's reference manual for more detailed information about the DMA module.

NOTE

Sample code uses the polling method for DMA transfer.

I^{2}C channel 0, 7-bit address, master TX mode, 50000bps, transfer 64 bytes to I^{2}C slave.

```
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;
        }
```

Figure 11 shows an example of how to use DMA to transfer data in master TX polling mode.

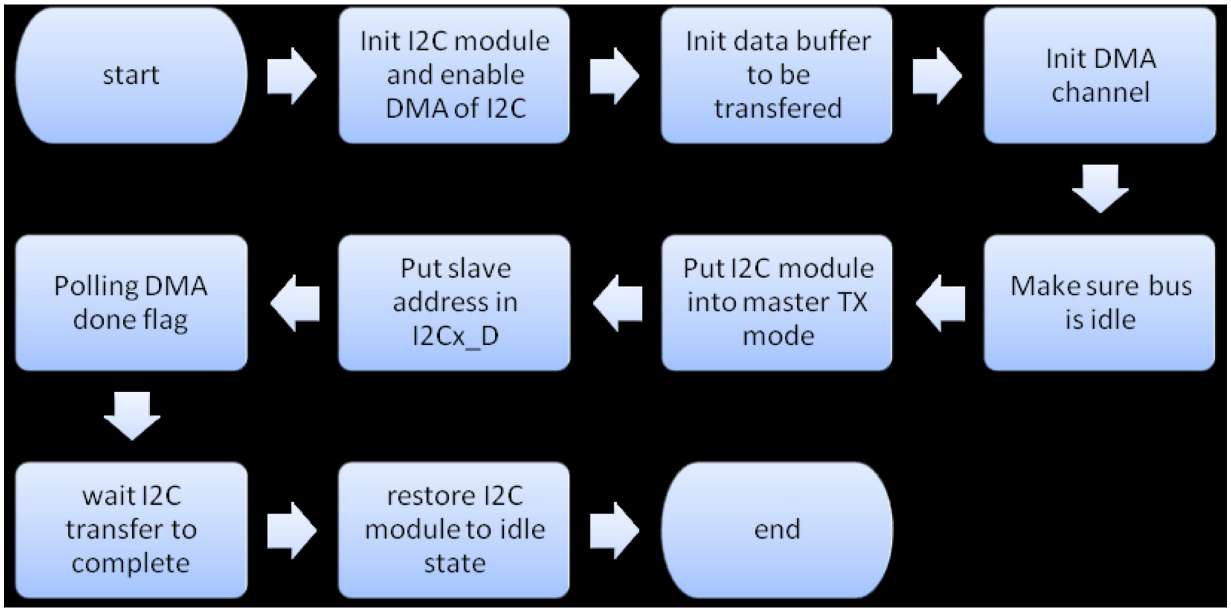

start
Init I2C module and enable DMA of I2C
Init data buffer to be transfered
Init DMA channel
Make sure bus is idle
Put I2C module into master TX mode
Put slave address in I2Cx_D
Polling DMA done flag
wait I2C transfer to complete
restore I2C module to idle state
end

Figure 11. Using DMA in master TX polling mode

NOTE

Because there must be an interrupt of complete 1-byte transfer to trigger the DMA, the address byte should be filled into I2C*x*_D by the CPU itself.

tcd1.nbytes = 63 instead of 64, because if you use two of the same MCF51JF128 boards for test and slave using DMA to receive data, the first byte received by DMA on the slave side will be address|TX byte. For detailed information, please refer to the source code of DMA slave RX `i2c_dma_slave_rx()`.

I²C channel 0, 7-bit address, master RX mode, 50000 bit/s, receive 64 bytes from I²C slave.

```
        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]);
           }
      }
```

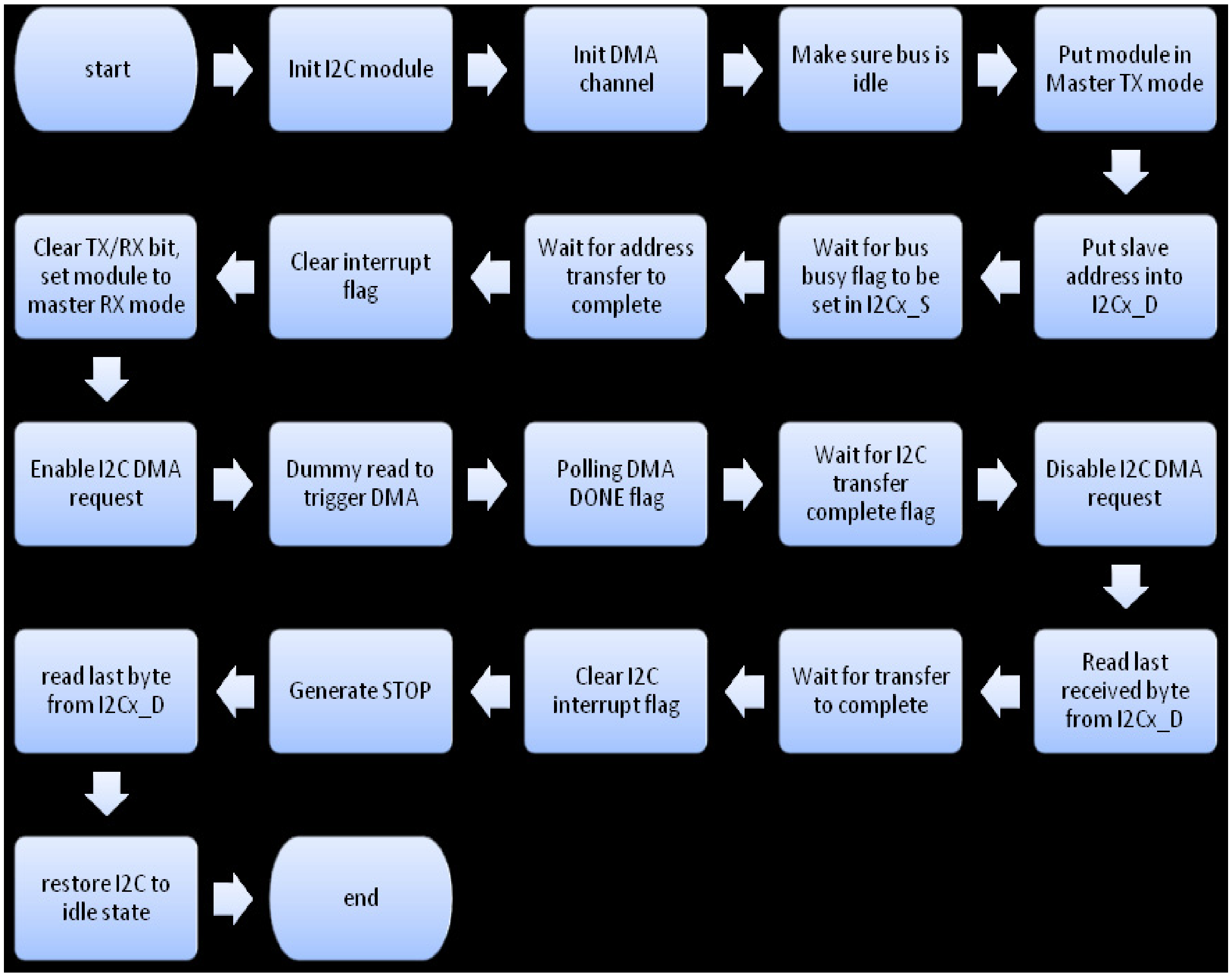

start
Init I2C module
Init DMA channel
Make sure bus is idle
Put module in Master TX mode
Put slave address into I2Cx_D
Wait for bus busy flag to be set in I2Cx_S
Wait for address transfer to complete
Clear interrupt flag
Clear TX/RX bit, set module to master RX mode
Enable I2C DMA request
Dummy read to trigger DMA
Polling DMA DONE flag
Wait for I2C transfer complete flag
Disable I2C DMA request
Read last received byte from I2Cx_D
Wait for transfer to complete
Clear I2C interrupt flag
Generate STOP
read last byte from I2Cx_D
restore I2C to idle state
end

Figure 12. Using DMA in master RX polling mode

NOTE

Because in master RX mode some necessary control bit should be set after the address cycle, DMA transfer can be started after the address cycle and transferring direction is changed.

DMA transfer must be stopped before receiving second-to-last byte. TXACK should be set at this point.

I^{2}C channel 0, 7-bit address, slave RX mode, receive 64 bytes from I^{2}C master.

```
void i2c_dma_slave_rx(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]);
        }
    }
```

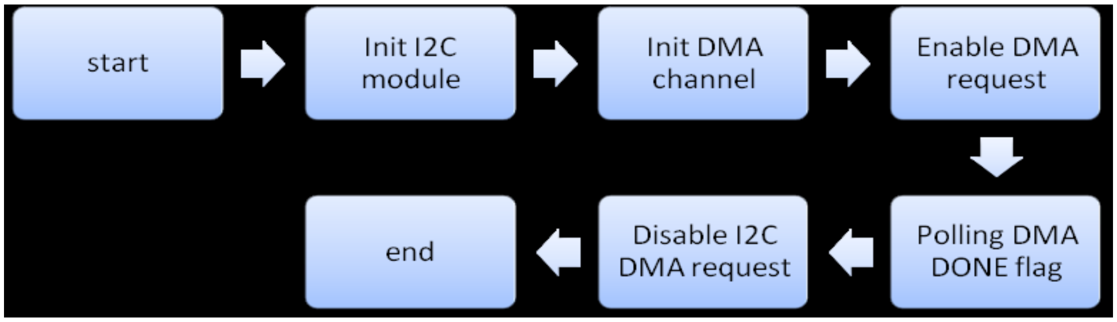


Figure 13. Using DMA in slave RX polling mode

I²C channel 0, 7-bit address, slave TX mode, receive 64 bytes from I²C master.

```
    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");
    }
```

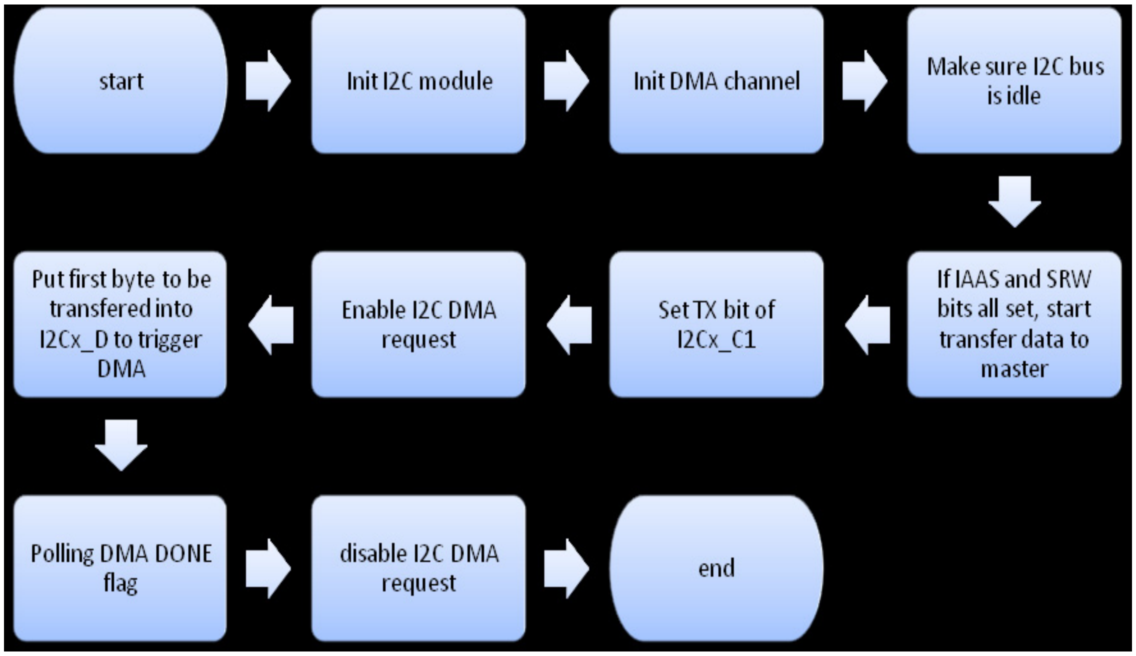

start
Init I2C module
Init DMA channel
Make sure I2C bus is idle
If IAAS and SRW bits all set, start transfer data to master
Set TX bit of I2Cx_C1
Enable I2C DMA request
Put first byte to be transfered into I2Cx_D to trigger DMA
Polling DMA DONE flag
disable I2C DMA request
end

Figure 14. Using DMA in slave TX polling mode

NOTE

Start DMA transfer after the address cycle and trigger DMA by loading first data to I2C*x*_D.

4.4.4 SMBus-related code

Sample code for SMBus test can be found in the i2c_smb_basic project. There is no special difference between normal I²C APIs and SMBus APIs, except that you may need to set ALERTEN and SIICAEN when initializing the SMBus; follow *Section 4.4.2, "Using interrupts"* as a reference. Please also read `i2csmbus_handler()` in i2c.c for more detailed information.

NOTE

According to Figure 6, 1 bit SCL delay is necessary in some operations. In sample code, this delay is accomplished by the Modulo Timer (MTIM). For a more detailed discussion of the MTIM, see the Modulo Timer chapter of your device's reference manual.

According to SMBus protocol (PEC), when in master or slave RX mode, a soft CRC is needed during the ARP process. I2C*x*_SMB[ALERTEN] and I2C*x*_SMB[SIICAEN] should be set properly.

5 Summary

The new I^{2}C module supports 10-bit address mode as well as 7-bit address mode, adds support of SMBus protocol, and adds DMA supports for reducing CPU load. It also allows the use of both the polling and interrupt methods.

Document Number: AN4342
Rev. 0
08/2011