AN12841

GPIO Simulated I^{2}C for S08

Rev. 0 — May 2020

Application Note

Contents

1 Introduction

The I^{2}C protocol is a 2-wire serial communication interface implemented by hardware in numerous devices. However, MC9S08PL16S does not support I^{2}C peripheral. In this situation, it is necessary to simulate the protocol using the General Purpose Input Outputs (GPIOs) of the microcontroller, usually to act as a master device.

This application note introduces how to implement GPIO simulated I^{2}C as a master based on S08PB/S08PLS.

The software tool in this document is based on CodeWarrior 11.1 IDE (must install the service pack: CodeWarrior MCU 11.1 Service Pack for S08PB and S08PLS), and the hardware is S08PB16-EVK board.

2 I^{2}C protocol introduction

I^{2}C bus is a multi-master bus. Serial, 8-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the Standard mode, and up to 400 kbit/s in the Fast mode. I^{2}C uses a Serial Data Line (SDA) and a Serial Clock Line (SCL) for data transfers. The SDA and SCL of master devices and slave devices are connected to the SDA and SCL of I^{2}C bus. For more details about I^{2}C bus protocol specification, refer to *I^{2}C-bus Specification and User Manual* (document UM10204).

All devices connected to I^{2}C must have open drain or open collector outputs. A logic AND function is exercised on both lines with external pull-up resistors. The master device initiates a transfer, generates clock signals, addresses the slave device, and terminates a transfer. Figure 1 illustrates the timing of I^{2}C bus data transmission.

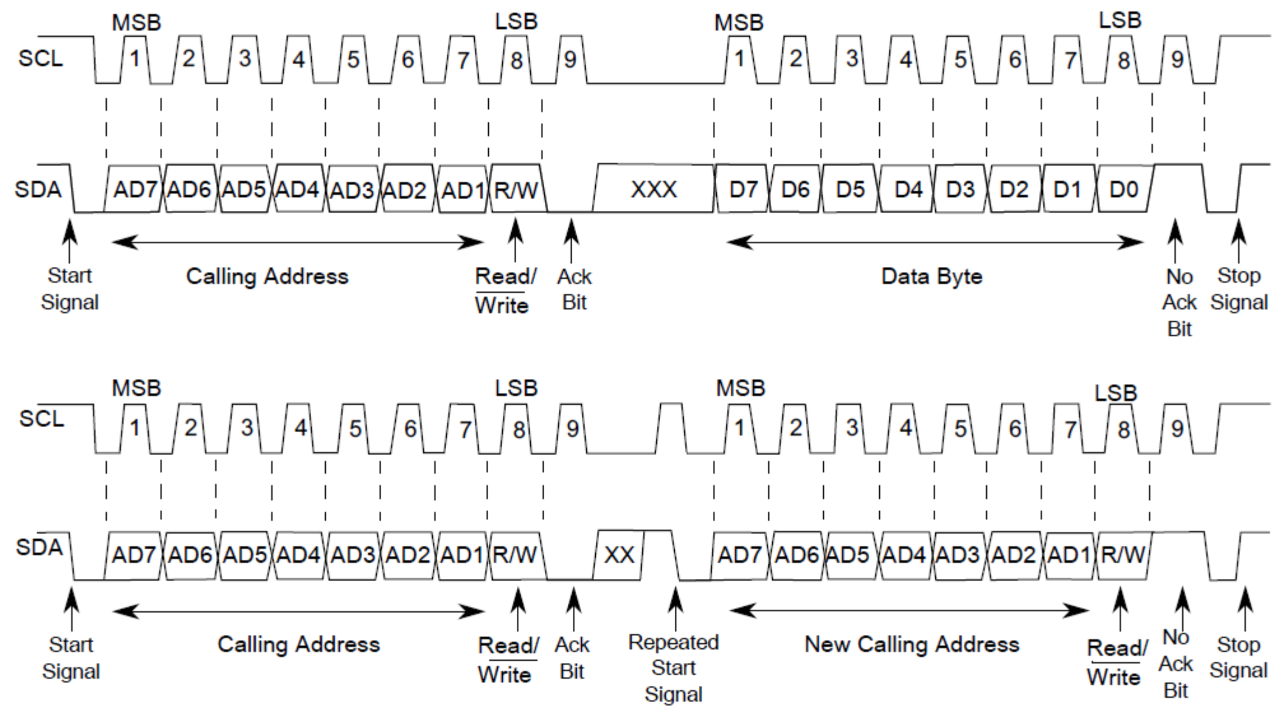

Figure 1. I²C bus data transmission

Table 1 lists part features of the standard instance of I²C transmission.

Table 1. Part features of I²C transmission

Feature	Description
START condition	Falling edge on SDA while SCL is held high.
Slave address	Indicates the address of the slave to whom the master wants to communicate.
R/W bit	The R/W bit tells the slave the desired direction of data transfer.
ACK (Acknowledge bit)	Held low at every ninth SCL clock cycle if the receiver successfully receives the data from transfer. Held high otherwise.
Data transfer	Proceed on a byte-by-byte basis in the direction specified by the R/W bit sent.
STOP condition	Rising edge on SDA while SCL is held high.

3 Simulating I²C master

This chapter gives a detailed description of how to simulate I²C features using GPIO.

It is not intended to implement all the features simulation of an I²C bus. It provides only the basic functionality of how to use two GPIOs to implement a master device data transmission (writes data to a slave device, reads data from a slave device). The advantage of this method is it uses GPIOs on any S08 MCU.

NOTE

PTA4 cannot be used to simulate I²C, as it is the output-only pin of S08PB16 or S08PL16S.

This application provides the following functionalities:

- Single master transmitter and receiver

- Communication with 7-bit addressing mode of I^{2}C slave device
- Simulated I^{2}C baud rate no more than 150 kHz
- Checking of acknowledgment signal
- Multiple data bytes within a serial transfer

3.1 I/O pins selection

I^{2}C bus transmission can be simulated by controlling two digital I/O pins. These two I/O pins can simulate SCL line and SDA line of I^{2}C bus.

- If the selected I/O pins are push-pull pins, users need to enable internal pull up function of the push-pull pins or add external pull-up resistors.
- If the selected I/O pins are open-drain pins, users must add pull-up resistors at the open-drain pins to make the pins to achieve stable high voltage output.

For S08PB/S08PLS, the two open-drain pins PTB6/PTB7 are selected to make an example.

Select PTB6 pin to simulate SDA line and PTB7 pin to simulate SCL line. Users can change different pins according to their own conditions.

The macro definitions for the PTB6 and PTB7 pins are shown as follows:

```
#define SDA                             PTB6
#define SCL                             PTB7
/* PTB6/PTB7 output enable and input enable */
#define i2c_SDA_Output()                 PORT_PTBOE_PTBOE6 =1
#define i2c_SCL_Output()                PORT_PTBOE_PTBOE7 =1
#define i2c_SDA_Input()                 PORT_PTBIE_PTBIE6 =1
#define i2c_SCL_Input()                 PORT_PTBIE_PTBIE7 =1

/* PTB6/PTB7 output disable */
#define i2c_SDA_Output_Off()            PORT_PTBOE_PTBOE6 =0
#define i2c_SDA_Input_Off()             PORT_PTBIE_PTBIE6 =0

/* set PTB6/PTB7 data to high or low */
#define i2c_SDA_High()                  PORT_PTBD_PTBD6   = 1
#define i2c_SDA_Low()                      PORT_PTBD_PTBD6   = 0
#define i2c_SCL_High()                  PORT_PTBD_PTBD7   = 1
#define i2c_SCL_Low()                      PORT_PTBD_PTBD7   = 0

/* PTB6/PTB7 internal pull up enable */
#define i2c_SDA_PullUp()                   PORT_PTBPE_PTBPE6   = 1;
#define i2c_SCL_PullUp()                   PORT_PTBPE_PTBPE7   = 1;

/* PTB6 data */
#define i2c_SDA_Status()                  PORT_PTBD_PTBD6
```

It is the process to initialize the I/O pins to simulate I^{2}C.

1. Make sure that two external pull-up resistors have been connected open-drain pins PTB6, PTB7 in S08PB16-EVK board. For pull-push pins, it needs to enable the internal pull-up if external pull-up resistors are not added to the pins.
2. Configure the data direction of these two I/O pins as output.

The I/O pins (PTB6, PTB7) initialization reference code is as follows:

```
/* I/O initialization */
void i2c_Init(void)
{
    /* enable I/O pins internal pull up */
    //i2c_SDA_PullUp();
    //i2c_SDA_PullUp();

    /* enable PTB6(SDA)/PTB7(SCL) output */
    i2c_SCL_Output();
    i2c_SDA_Output();
}
```

3.2 Baud rate of simulate I2C

3.2.1 Using hardware timer to simulate I^{2}C serial clock

This document gives an example about how to use the hardware timer MTIM0 to simulate I^{2}C serial clock.

The bus clock of MTIM0 is configured to 16 M in the example code. In theory, if user wants to get a baud rate of 80 k, the function **MTIM0_Init**(16000000, 160000) must be called to initialize the timer to configure MTIM0 frequency. The time of half SCL clock cycle is simulated by calling the function **SCL_Delay**().

The simulate I^{2}C baud rate is theoretically equal to half of the MTIM0 frequency. But actually, the actual baud rate is quite different from the theory baud rate. Table 2 shows part of the comparative data between theoretical and actual baud rate.

Table 2. Comparison of theoretical and actual I^{2}C baud rate

MTIM0 configure	MTIM0_CLK_CLKS = 0, bus clock:16M MTIM0_CLK_PS = 2, prescaler:4										
MTIM0 frequency (Hz)	40 k	80 k	100 k	120 k	160 k	200 k	240 k	280 k	320 k	360 k	400 k
Theory simulate baud rate (Hz)	20 k	40 k	50 k	60 k	80 k	100 k	120 k	140 k	160 k	180 k	200 k
Actual simulate baud rate (Hz)	18 k	33 k	40 k	46 k	56 k	65 k	74 k	81 k	87 k	91 k	97 k

If MTIM0 frequency is increased, the simulate i2c baud rate is also increased. But the maximum GPIO simulate i2c baud rate can only up to about 150k when MTIM0 frequency up to 16 M.

The example functions of simulate serial clock are as follows:

```
/* mtim0 intialize, frequency range: 16k - 4M */
void MTIM0_Init(dword busCLKHz, dword mtim0Frq)
{
     byte dummy;

    /* select bus clock as mtim0 clock source */
    MTIM0_CLK_CLKS = 0x0;

    /* prescale is 4 */
    MTIM0_CLK_PS = 0x2;

    dummy = (byte)(busCLKHz/4/mtim0Frq);
    /* set modulo value */
    MTIM0_MOD = (dummy-1);

    /* start count */
    MTIM0_SC_TSTP   = 0;
}

/* time delay of SCL pulse */
void SCL_Delay(void)
{
   /* counter reset to 0x00, TOF is cleared*/
    MTIM0_SC_TRST = 1;
    while(!(MTIM0_SC & MTIM0_SC_TOF_MASK));
}
```

The frequency range of MTIM0 is about 16 k to 4 M in the above code. If user wants to get other frequency, please configure `MTIM0_CLK_CLKS` and `MTIM0_CLK_PS` to appropriate values.

3.2.2 Using software timer to simulate I^{2}C serial clock

Using software delay can also simulate I^{2}C serial clock. The time of half SCL clock cycle is simulated by the function **`SCL_Delay`**`()`. It only takes 0.0625 µs to execute **`asm`**`(nop)` instruction. But it takes 5.7 µs to execute the following **`SCL_Delay`**`()` function. The baud rate of the simulated I^{2}C is set to approximately 88 kHz when using the following `SCL_Delay()` function.

The delay time function is as follows:

```
/* delay time is about 5.7us */
void SCL_Delay(void)
{
    unsigned int i;
    for(i=0;i<2;i++)
    {
        asm(nop);
    }
}
```

3.3 Simulating START condition

According to Figure 2, a HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition.

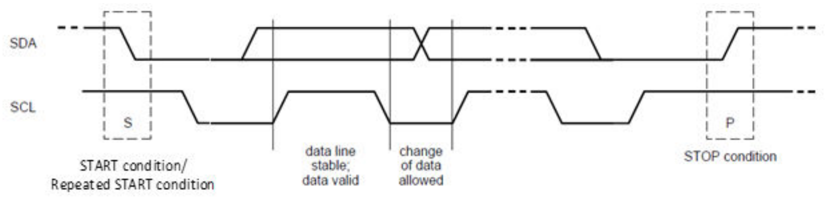

Figure 2. START, repeated START and STOP conditions

When using PTB6 (SDA) and PTB7 (SCL) to simulate the timing of START condition, please note that:

- It needs set to PTB7 (SCL) to low first in **`i2c_Start`**`()` function, then set PTB6 (SDA) to high, and PTB7 (SCL) to high last. Configuration in the above order can prevent to generate unwanted STOP condition. If PTB6 (SDA) is changed from low to high during PTB7 (SCL) is high, the unwanted STOP condition is generated.

START condition reference code is as follows: :

```
/* START condition */
void i2c_Start(void)
{
    i2c_SCL_Low();
    i2c_SDA_High();
    i2c_SCL_High();
    SCL_Delay();
    /* SDA change from high to low when SCL is high, i2c start */
    i2c_SDA_Low();
}
```

3.4 Simulating STOP condition

According to Figure 2, a LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.

When using PTB6 (SDA) and PTB7 (SCL) to simulate the timing of STOP condition, please note that:

- It needs set PTB7 (SCL) to low first in `i2c_Stop()` function, then set PTB6 (SDA) to low and PTB7 (SCL) to high. Configuration in the above order can prevent to generate unwanted START condition. If PTB6 (SDA) is changed from high change to low during PTB7 (SCL) is high, the unwanted START condition is generated.

STOP condition reference code is as follows:

```
/* STOP condition */
void i2c_Stop(void)
{
    i2c_SCL_Low();
    i2c_SDA_Low();
    i2c_SCL_High();
    SCL_Delay();
    /* SDA change from low to high when SCL is high, i2c stop */
    i2c_SDA_High();
}
```

3.5 Simulating repeated START condition

According to Figure 2, the START and repeated START conditions are functionally identical. After starting a communication, before sending a stop signal, if the host sends a repeated START condition, the host and slave can be changed without releasing the bus. The timing simulation of the repeated START condition is the same as the START condition.

```
#define i2c_RepeatedStart()    i2c_start();
```

3.6 Simulating ACK and NACK

The acknowledge takes place after every byte. The acknowledge bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent. The master generates all clock cycles, including the acknowledge ninth clock cycle.

When SDA remains LOW during this ninth clock cycle, this is defined as the Acknowledge (ACK) signal. When SDA remains HIGH during this ninth clock cycle, this is defined as the Not Acknowledge (NACK) signal.

3.6.1 Master reads ACK signal from slave

If a master receives low-level pulse (ACK signal) from a slave in the ninth clock cycle after sending a byte, the data transmission is successful. If a master receives high-level pulse (NACK signal) from a slave in the ninth clock cycle after sending a byte, the data transmission failed.

When using PTB6 (SDA) and PTB7 (SCL) to simulate the process of master reads ACK signal from slave, please note that:

- The master reading the ACK signal is actually reading the input data of PTB6 (SDA) in the ninth clock cycle.

  Before reading the status of PTB6 (SDA), the data direction of PTB6 (SDA) needs to be set as input. Please disable the output of PTB6 (SDA) before enabling the input of PTB6 (SDA) to change the data direction to input.

  After reading the status of PTB6 (SDA), the data direction of PTB6 (SDA) needs to be changed to output soon. Please disable the input of PTB6 (SDA) before enabling the output of PTB6 (SDA) to change the data direction to output.
- In the ninth clock cycle, it recommends read the status of PTB6 (SDA) during PTB7 (SCL) is high. At this time, the ACK signal is in stable status.
- When the ninth clock is over, PTB7 (SCL) needs to be kept low, and PTB6 (SDA) is pulled high. The bus enters to wait state.

A master reads ACK signal code is as follows:

```
/* master reads ACK signal */
uint8_t i2c_ReadACK(void)
{
    uint8_t ack;

    /* the 9th clock start */
    i2c_SCL_Low();

    /* disable PTB6(SDA) output, enable PTB6(SDA) input */
    i2c_SDA_Output_Off();
    i2c_SDA_Input();

    SCL_Delay();
    i2c_SCL_High();
    /* read ACK signal when SCL is high */
    ack = i2c_SDA_Status();
    SCL_Delay();
    /* the 9th clock end */

    /* disable PTB6(SDA) input, enable PTB6(SDA) output */
    i2c_SDA_Input_Off();
    i2c_SDA_Output();

    /* SCL hold low to wait */
    i2c_SCL_Low();
    i2c_SDA_High();

    return ack;
}
```

3.6.2 Master sends ACK signal or NACK signal to slave

If the master sends low-level pulse (ACK signal) to the slave after receiving a byte, it means that the master wishes to continue reading the next byte, prompting the slave to prepare the next data. If the master sends high-level pulse (NACK signal) to the slave after receiving a byte, the data transmission is considered to be over, and no more data is sent. At this time, the master will generate a STOP signal to release the I2C bus or generate a repeated START signal to start a new transmission.

When using PTB6 (SDA) and PTB7 (SCL) to simulate the master sends ACK or NACK signals to slave, please note that:

- In the ninth clock cycle, PTB6 (SDA) is set to 0 during PTB7 (SCL) is low to enable master sends an ACK signal to the slave. In the ninth clock cycle, PTB6 (SDA) is set to 1 during PTB7 (SCL) is low to enable master sends a NACK signal to the slave. The value of SDA can only be changed when SCL is low.

- When the ninth clock is over, PTB7 (SCL) needs to be kept low, and PTB6 (SDA) is pulled high. The bus enters to wait state.

The reference code is as follows:

```
/* master sends ACK to slave */
void i2c_SendACK(void)
{
    /* the 9th clock start */
    i2c_SCL_Low();
    /*send ACK signal */
    i2c_SDA_Low();
    SCL_Delay();
    i2c_SCL_High();
    SCL_Delay();
    /* the 9th clock start */

    /* SCL hold low to wait */
    i2c_SCL_Low();
    i2c_SDA_High();
}


/* master sends NACK to slave */
void i2c_SendNACK(void)
{
    /* the 9th clock start*/
    i2c_SCL_Low();
    /*send NACK signal */
    i2c_SDA_High();
    SCL_Delay();
    i2c_SCL_High();
    SCL_Delay();
    /* the 9th clock start */

    /* SCL hold low to wait */
    i2c_SCL_Low();
    i2c_SDA_High();
}
```

3.7 Simulating master to write a byte

I^2C transmissions are sequences of 8-bit bytes. When using GPIO to simulate the master writes a byte, master should write data to PTB6 (SDA) bit by bit when PTB7 (SCL) is low. It takes eight clock cycles to implement 8-bit data transmission.

When using PTB6 (SDA) and PTB7 (SCL) to simulate the master writes a byte to slave, please note that:

- If the master has finished sending 8-bit data to the slave, it needs set PTB7 (SCL) to low to prevent the unwanted START or STOP conditions from being generated due to the PTB6 (SDA) voltage changes.
- It must check whether the master receives an ACK from the slave when the master finished sending 8-bit data. If the ACK signal is not received, the master will generate a STOP condition and end the communication.

The master writes a byte and checks ACK signal codes as follows:

```
/* master writes a byte to salve and check ACK signal */
void i2c_write_byte(unsigned char data)
{
    unsigned char i;
    /* write 8 bits data */
    for(i=0;i<8;i++)
    {
        i2c_SCL_Low();
        SCL_Delay();

        /* Data can be changed only while SCL is low */
        if(data & 0x80)
        {
            i2c_SDA_High();
        }
        else
        {
            i2c_SDA_Low();
        }

        /* Data must be held stable while SCL is high */
        i2c_SCL_High();
        SCL_Delay();
        data = data<<1;
    }
     /* Must set SCL to low. If SCL is high, SDA change to High/Low will cause Stop/Start signal.*/
    i2c_SCL_Low();
    /* check ACK signal, ACK signal is 0, NACK signal is 1 */
    if(i2c_ReadACK() == 1)
    {
        /* receive NACK signal,stop data transfer */
        i2c_Stop();
        while(1);
    }
}
```

3.8 Simulating master to read a byte

I^{2}C transmissions are sequences of 8-bit bytes. When using GPIO to simulate the master reads a byte from the slave, master should read data from PTB6 (SDA) bit by bit when PTB7 (SCL) is high. It takes eight clock cycles to implement 8-bit data transmission.

When using PTB6 (SDA) and PTB7 (SCL) to simulate the master reads a byte to slave, please note that:

- If the master has finished receiving 8-bit data from the slave, it needs set PTB7 (SCL) to low to prevent the unwanted START or STOP conditions from being generated due to the PTB6 (SDA) voltage changes.
- The master reads data by bit, which is actually reading the input data of PTB6 (SDA).

  Before reading the status of PTB6 (SDA), the data direction of PTB6 (SDA) needs to be set as input. Please disable the output of PTB6 (SDA) before enabling the input of PTB6 (SDA) to change the data direction to input.

  After reading the status of PTB6 (SDA), the data direction of PTB6 (SDA) needs to be changed to output soon. Please disable the input of PTB6 (SDA) before enabling the output of PTB6 (SDA) to change the data direction to output.

The reference code is as follows:

```
/* master reads a byte from slave */
unsigned char i2c_read_byte(void)
{
    unsigned char i;
    unsigned char data = 0;

    /* disable PTB6(SDA) output, enable PTB6(SDA) input */
    i2c_SDA_Output_Off();
    i2c_SDA_Input();
    /* write 8 bits data */
    for(i=0;i<8;i++)
    {
        i2c_SCL_Low();
        SCL_Delay();
        /* Data must be held stable while SCL is high */
        i2c_SCL_High();
        data = data<<1;
        /* read SDA data */
        if(i2c_SDA_Status())
        {
            data |= 0x01;
        }
        else
        {
            data &= ~0x01;
        }

        SCL_Delay();

    }

    /* disable PTB6(SDA) input, enable PTB6(SDA) output */
    i2c_SDA_Input_Off();
    i2c_SDA_Output();

    i2c_SCL_Low();      // set SCL to low

    return data;
}
```

4 Implementation details

This is the detailed information on how to realize master data read and write multiple data through I^{2}C simulated by GPIO. Figure 3 shows the data transfer formats of a master.

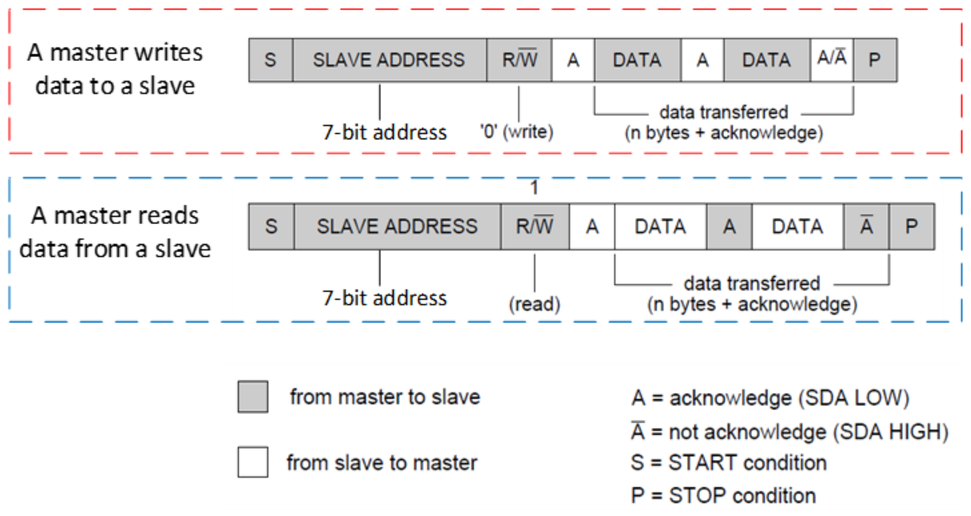

Figure 3. Data transfer formats of a master writes data and a master reads data

Figure 4 shows the flowchart of GPIO simulated I2C communication. Users can write data sending and receiving procedures according to the flowchart.

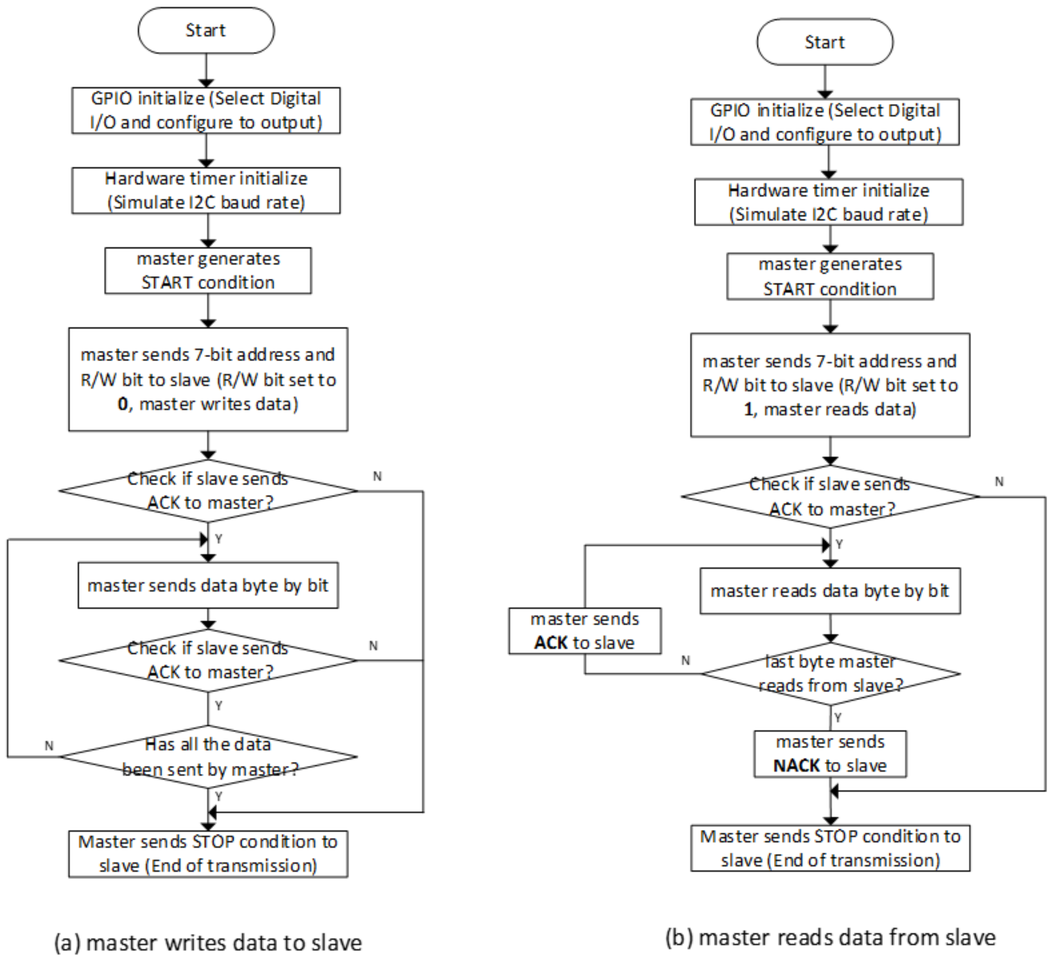
Start
GPIO initialize (Select Digital I/O and configure to output)
Hardware timer initialize (Simulate I2C baud rate)
master generates START condition
master sends 7-bit address and R/W bit to slave (R/W bit set to 0, master writes data)
Check if slave sends ACK to master?
master sends data byte by bit
Has all the data been sent by master?
Master sends STOP condition to slave (End of transmission)
(a) master writes data to slave
master sends 7-bit address and R/W bit to slave (R/W bit set to 1, master reads data)
master reads data byte by bit
last byte master reads from slave?
master sends ACK to slave
master sends NACK to slave
(b) master reads data from slave
Y
N

Figure 4. Flowchart of data transmission

4.1 Master writes data to slave

Prepare a slave device that can receive data normally. Referring to the flowchart shown in Figure 4 (a), there is an example about the master successfully sending `0x0A`, `0x0B` two bytes to the slave device with address `0x56`. The I^{2}C serial clock is simulated by configuring **`MTIM0_Init(16000000L, 80000L);`**. The simulated I^{2}C baud rate is about 55.8 k.

The reference code is as follows:

```
/* master writes data to slave */
void i2c_master_WriteData(void)
{
    unsigned char i;
    unsigned char slaveID;
    char tx_data[16] = {10,11};

    /* shift address in right position, set R/W bit to 0 at end of slave address */
    slaveID = (0x56 << 1) | 0;

    /* send start condition */
    i2c_Start();
    /* send slave address with W/R bit */
    i2c_write_byte(slaveID);

    /* wait for master transmit completed */
    for (i=0; i<2; i++)
    {
        /* master write a byte to salve and check ACK signal */
        i2c_write_byte(tx_data[i]);
    }
    i2c_Stop();
}
```

The functions called in the above code have been explained in detail in Simulating I2C master.

The waveforms of simulated SCL and SDA are captured by a logic analyzer. Figure 5 is captured by a logic analyzer when running the code, the simulated I^{2}C baud rate is about 55.8 k.

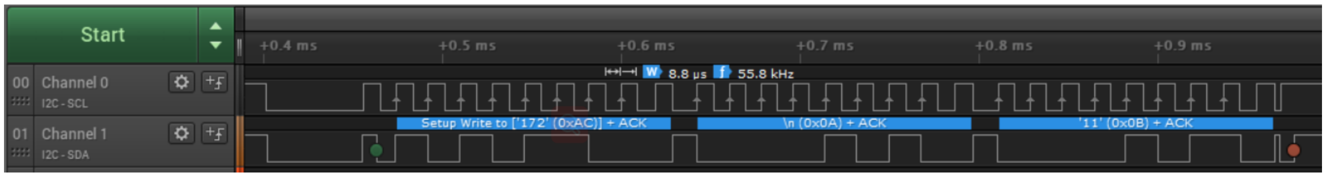

Figure 5. Write two bytes to slave device with `0x56` address

4.2 Master reads data from slave

Prepare a slave device that can send 16 bytes of data (data: from 20 to 35) normally. Referring to the flowchart shown in Figure 4 (b), there is an example about the master successfully receive `0x14`, `0x15` two bytes from the slave device with address `0x56`. The I^{2}C serial clock is simulated by configuring **`MTIM0_Init(16000000L, 80000L);`**. The simulated I^{2}C baud rate is about 55.8 k.

The reference code is as follows:

```
/* master reads data from slave */
void i2c_master_ReadData(void)
{
    unsigned char k;
    unsigned char slaveID;
    char rx_data[16] = {0};

    /* shift address in right position, set R/W bit to 1 at end of slave address */
    slaveID = (0x56 << 1) | 1;

    /* send start condition */
    i2c_Start();

    /* send slave address with W/R bit */
    i2c_write_byte(slaveID);

    /* wait for master transmit completed */
    for (k=0; k<2; k++)
    {
        rx_data[k] = i2c_read_byte();
        if(k == 1)
        {
            /* master sends NACK to slave*/
```

```
                i2c_SendNACK();
                /* Generate STOP condition */
                i2c_Stop();
                break;
            }
            else
            {
                /* master sends ACK to slave*/
                i2c_SendACK();
            }
            /* wait to receive next byte from slave */
            SCL_Delay();
        }
    }
```

The functions called in the above code have been explained in detail in Simulating I2C master.

The waveforms of simulated SCL and SDA are captured by a logic analyzer. Figure 6 is captured by a logic analyzer when running the code. The simulated I^{2}C baud rate is about 55.8 k.

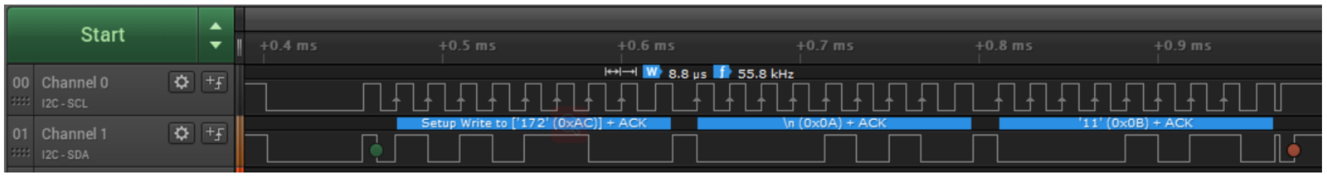

Figure 6. Reading two bytes from slave device with 0x56 address

5 Conclusion

This application note explains how to simulate I^{2}C master with GPIO based on S08P. The simulate I^{2}C master can implement communication with I^{2}C slave device. The maximum communication speed is close to 150 k on S08PB/S08PLS when the system clock is 16 M. The reference code of GPIO simulated I^{2}C can be downloaded at S08PB16-EVK.

6 References

Following references are available on NXP website:

1. *MC9S08PB16 Reference Manual* (document MC9S08PB16RM)
2. *I^{2}C-bus Specification and User Manual* (document UM10204)
3. *Emulating I2C Master Mode* (document AN5264)
4. *IIC Master on the MC9RS08KA2* (document AN3317)

Date of release: May 2020
Document identifier: AN12841