Fujitsu Semiconductor (Shanghai) Co., Ltd.
Application Note

MCU-AN-500034-E-11

F²MC-8FX FAMILY
8-BIT MICROCONTROLLER
MB95200 SERIES

I²C SW REALIZATION USING GPIO

APPLICATION NOTE

FUJITSU

Revision History

Date	Author	Change of Records
2009-02-04	Folix	V1.0, First draft
2009-03-24	Folix	V1.1, Modify the format.

This manual contains 25 pages.

CONTENTS

1 Introduction

In this document, we will introduce how to use the GPIO to realize I^2C function on the MB95200 series.

2 Overview of I^2C

2.1 Background

In consumer electronics, telecommunications and industrial electronics, there are often many similarities among different designs. The following blocks are nearly included in every system.
Some intelligent control, usually a single-chip microcontroller
General-purpose circuits like LCD drivers, remote I/O ports, RAM, E2PROM, or data converters
Application-oriented circuits, such as digital tuning and signal processing circuits for radio and video systems, or DTMF generators for telephones with tone dialing

In these blocks listed above, the control and communication among different chips may be performed. To maximize hardware efficiency and circuit simplicity, Philips developed a simple bi-directional 2-wire bus for efficient inter-IC control. This bus is called "inter IC" or "I^2C-bus".

Now, Philips' IC includes more than 150 CMOS and bipolar I^2C-bus compatible used to perform corresponding functions mentioned above. All I^2C-bus compatible with devices build in an on-chip interface which supports them to communicate directly each other via I^2C-bus. This concept has solved many problems related to interface when designing digital control circuits.
Here are some features of I^2C-bus:
Only two lines are required: a serial data line (SDA) and a serial clock line (SCL).
For each device connected to the bus, only an independent address and a simple master/slaver relationship are needed. The master can operate as a transmitter or a receiver.
A true multi-master mode can be realized only by bus arbitration and collision detection.
In 8-bit bi-directional serial transfer, data transfer rate can be up to 100 Kbit/s in standard-mode and up to 400 Kbit/s in fast-mode and up to 3.4 Mbit/s in high speed mode.
On-chip bus filtering function can preserve the integrity of communication data on the bus.
The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance of 400 pF.

2.2 Protocol

2.2.1 START Condition and STOP Condition

In I^{2}C communication, START (S) condition and STOP (P) condition should be set respectively.

SDA line is changed from HIGH to LOW while SCL is HIGH; this case is defined as a START condition.

SCL is changed from LOW to HIGH while SCL is HIGH; this case is defined as a STOP condition.

The START condition and the STOP condition are always generated by the master. The bus is busy when the START condition is generated and is free when the STOP condition is generated.

The bus also remains busy if a repeated START (Sr) is generated instead of a STOP condition. In the case above, the function of the START (S) condition and the repeated START (Sr) condition are regarded as the same.

It is easy to detect the START condition and the STOP condition by devices connected to the bus, if they build in necessary hardware interface. However, if the controller does not have such interface, it must sample the SDA line at least twice per clock period so that the master can recognise the START condition and the STOP condition.

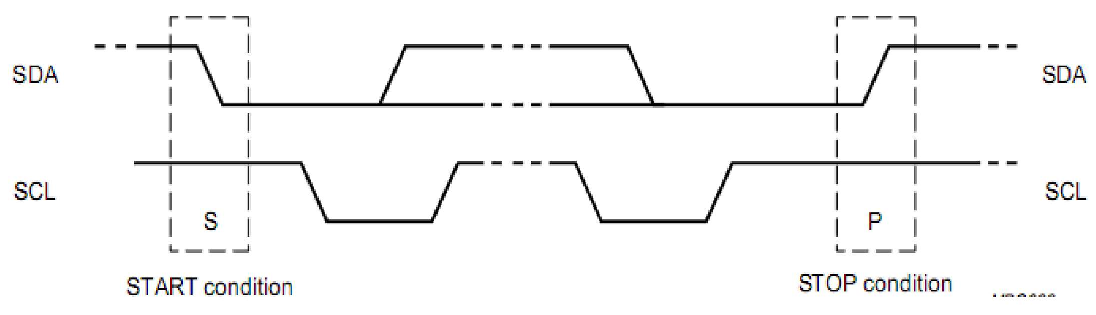

Figure 1 START Condition and STOP Condition

2.2.2 Byte Format

Each byte on the SDA line should be 8bits long. The number of bytes that can be transferred at a time is arbitrary. Each byte should be followed by an acknowledge bit. Data transfer starts from the most significant bit (MSB first). If the slaver can not receive or transfer a complete byte data, for example, an internal interrupt processing, SCL line should be set as LOW to force the master into a wait sate. Data transfer can be performed until the slaver is ready for next byte data and releases SCL line.

In some cases, you can use a format different from I^{2}C-bus format (for example, CBUS compatible devices). A message which starts with such an address can be terminated by generating a STOP condition, even during a byte transfer. In this case, a no acknowledge is generated.

2.2.3 Acknowledge

An acknowledge is necessary for data transfer. The clock pulse related to acknowledge is generated by the master. During the acknowlede clock pulse, the transmitter releases the SDA line (HIGH), the receiver pulls down the SDA line and always remains LOW during the HIGH period of the clock pulse. Please note that setup time and hold time should also be considered.

Usually, a receiver which has been addressed generates an acknowledge every time after a byte data is received. An exception is that the receiver accesses on CBUS addressing and sends a message.

If a slaver can not recognise the slave address (for example, it is performing some real-time function), the SDA line should be left HIGH by the slaver. In this case, the master sholuld terminate data transfer by generating a STOP condition or start a new transfer by generating a repeated START condition.

If a slave-receiver does not receive any byte data for some time after recognising the slave address, the master must terminate data transfer again.

If the slaver generates a no acknowledge immediately after the first byte, the slaver will set SDA as HIGH, so the master should generate a STOP condition or a repeated START condition.

If the master-receiver also performs data transfer, it should send a no acknowlede to the slave-transmitter when the slaver has sent the last data. The slave-transmitter must release the SDA line so that the master can generate a STOP condition or a repeated START condition.

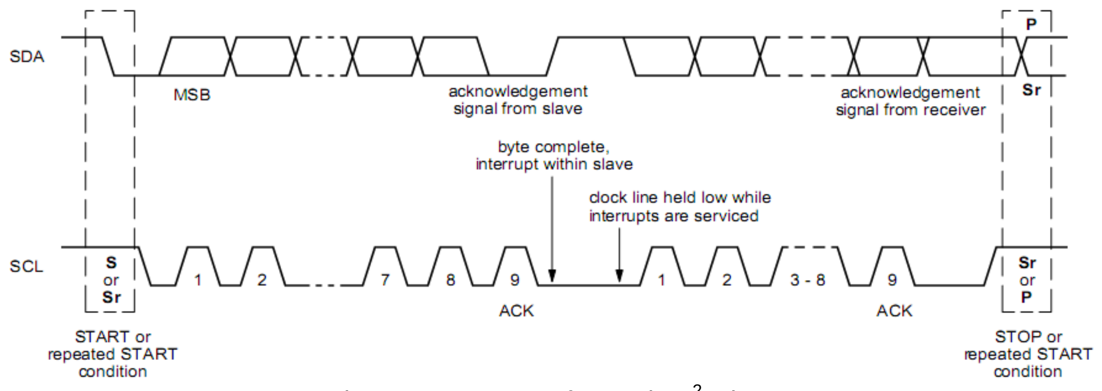
SDA
SCL
MSB
acknowledgement signal from slave
acknowledgement signal from receiver
byte complete, interrupt within slave
clock line held low while interrupts are serviced
S or Sr
1
2
7
8
9
ACK
1
2
3 - 8
9
ACK
P
Sr
Sr or P
START or repeated START condition
STOP or repeated START condition

Figure 2 Data Transfer on the I^{2}C-bus

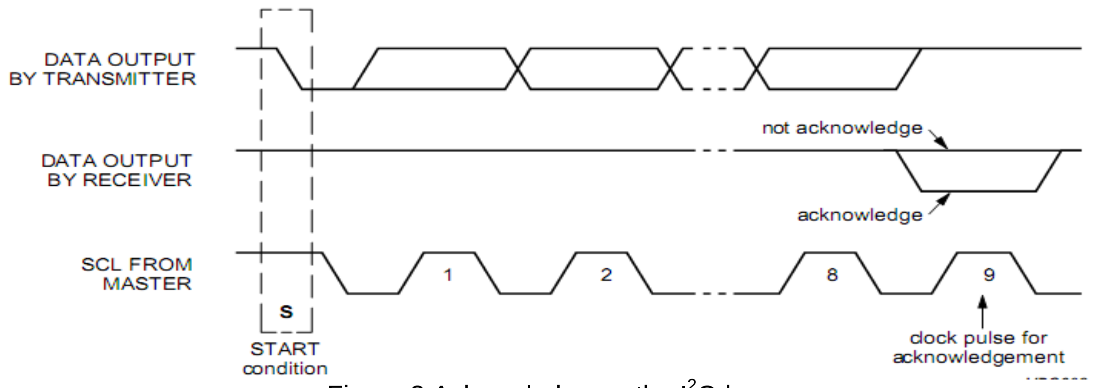

Figure 3 Acknowledge on the I^{2}C-bus

2.2.4 A Complete Data Transfer

Data transfer format is shown in Figure.4 as below. A slave address is transferred after the START condition (S) is generated. This address is 7 bits long followed by the eighth bit which is a data direction bit (R/W) ("0": Write, ‘1’: Read). Data transfer is always terminated by the STOP condition (P) generated by the master. However, if the master still needs to perform data transfer on the bus, it can generate a repeated START condition (Sr) and address another slaver without generating a STOP condition first. This way above can be used for various read/write formats.

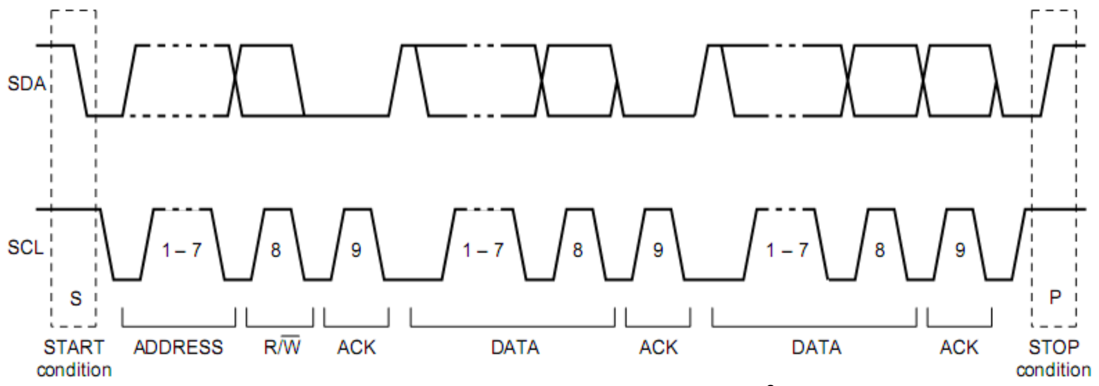

Figure 4 A Complete Data Transfer on the I^{2}C-bus

3 I^{2}C Driver

3.1 Peripheral Usage

The MCU pins are used as below.
P62, used as SDA;
P63, used as SCL;

3.2 Driver Code

3.2.1 General Definition

```
typedef unsigned char   BOOLEAN;
typedef unsigned char   INT8U;          /* Unsigned   8 bit quantity */
typedef signed     char   INT8S;          /* Signed       8 bit quantity */
typedef unsigned int      INT16U;        /* Unsigned 16 bit quantity */
typedef signed     int      INT16S;        /* Signed      16 bit quantity */
typedef unsigned long   INT32U;        /* Unsigned 32 bit quantity */
typedef signed     long   INT32S;        /* Signed      32 bit quantity */

#define BOOL              BOOLEAN
#define BYTE              INT8U
#define UBYTE             INT8U
#define WORD              INT16U
#define UWORD             INT16U
#define LONG              INT32S
#define ULONG            INT32U
#define UCHAR            INT8U
#define UINT              INT16U
#define DWORD             INT32U

#define TRUE               1
#define FALSE              0

#define SDA        PDR6_P62
#define SCL        PDR6_P63
```

3.2.2 I^{2}C Routines

void I2C_BusInit()

Return : None.
Parameters : None.
Description : Initialize the I^{2}C.
Example :
I2C_BusInit ();

```
void I2C_BusInit()
{
  DDR6_P62=1; //SDA-OUT
  DDR6_P63=1; //SCL-OUT

  PDR6_P62=0; //SDA-0
  PDR6_P63=0; //SCL-0
}
```

void DelayUS(UINT nDly)

Return : None.
Parameters :
nDly,delay US.
Description : Delay nDly on the I^{2}C.
Example :
DelayUS(3);

```
void DelayUS(UINT nDly)
{
  for(;nDly>0;nDly--){}
}
```

Note: The clock loop varies depending on MCU, please change the 'DelayUS()' function to realize the s delay.

void Start ()

Return : None.
Parameters : None.
Description : Start I^{2}C transfer.
Example :
Start ();

```
void Start()
{
  SCL=0;
  SDA=1;
  DDR6_P62=1; //SDA-OUT
  DelayUS(2);
  SCL=1;
  DelayUS(2);
  SDA=0;
  DelayUS(2);
}
```

void Stop()

Return : None.
Parameters : None.
Description : Stop I^{2}C transfer.
Example :
Stop ();

```
void Stop()
{
  SCL=0;
  SDA=0;
  DDR6_P62=1; //SDA-OUT
  DelayUS(2);
  SCL=1;
  DelayUS(2);
  SDA=1;
  DelayUS(2);
}
```

void SetAck()

Return	: None.
Parameters	: None.
Description	: Set an acknowledge on the I^{2}C.
Example	:

SetAck ();

```
void SetAck()
{
  SCL=0;
  DelayUS(2);
  SDA=0;
  DDR6_P62=1; //SDA-OUT
  DelayUS(2);
  SCL=1;
  DelayUS(2);
}
```

BOOL GetAck()

Return	: 0,failure 1,success.
Parameters	: None.
Description	: Get an acknowledge from the I^{2}C.
Example	:

BOOL bAck;
bAck=GetAck ();

```
BOOL GetAck()
{
  SCL=0;
  DelayUS(2);
  SDA=1;
  DDR6_P62=0; //SDA-IN
  SCL=1;
  DelayUS(2);

  return SDA? FALSE:TRUE;
}
```

void NoAck()

Return : None.
Parameters : None.
Description : Set a no acknowledge on the I2C.
Example :
NoAck ();

```
void NoAck()
{
  SCL=0;
  DelayUS(2);
  SDA=1;
  DDR6_P62=1; //SDA-OUT
  DelayUS(2);
  SCL=1;
  DelayUS(2);
}
```

void Write8Bit(UCHAR uDat)

Return : None.
Parameters :
uDat, 8-bit data.
Description : Write a 8-bit data on the I^{2}C.
Example :
Write8Bit (0x63);

```
void Write8Bit(UCHAR uDat)
{
  UCHAR i;

  SCL=0;
  DDR6_P62=1; //SDA-OUT
  for(i=8;i!=0;i--)
  {
    DelayUS(2);
    SDA=(uDat&0x80)? 1:0;
    DelayUS(2);
    SCL=1;
    DelayUS(2);
    SCL=0;
    uDat<<=1;
  }
}
```

UCHAR Read8Bit()

Return	: Data from the I^{2}C.
Parameters	: None.
Description	: Read a 8-bit data from the I^{2}C.
Example	:

UCHAR   tmp;
tmp=Read8Bit ();

```
UCHAR Read8Bit()
{
  UCHAR i;
  UCHAR uRByte=0;

  SCL=0;
  SDA=1;
  DDR6_P62=0; //SDA-IN
  for(i=8;i!=0;i--)
  {
    DelayUS(2);
    SCL=1;
    DelayUS(2);
    uRByte<<=1;
    uRByte|=(UCHAR)(SDA);
    SCL=0;
  }

  return uRByte;
}
```

4 Typical Application

4.1 HW Design

In this application, MB95F204K is used as MCU chip and E2PROM (AT24C08) is used as the I^{2}C device. The HW is designed as Figure.5.

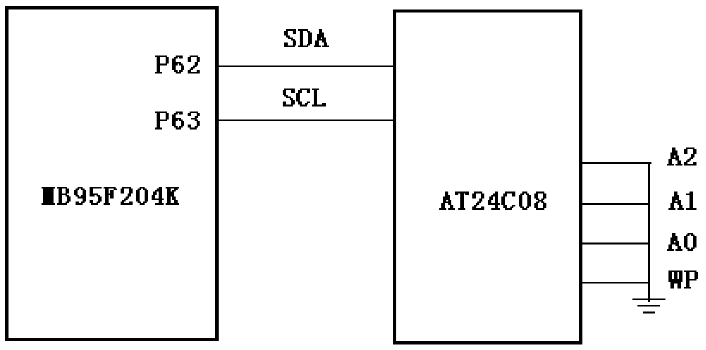

Figure 5 Hardware Design

4.2 Sample Code

The codes are applied to AT24C08. Please refer to the AT24C08 manual for more information. In the demo, the address for AT24C08 is set as 0xA0 and the baudrate is set as 1 kbit/s.

BOOL Rom_Write(UBYTE romAddr,UBYTE datAddr,UBYTE data)

Return : 0,failure 1,success.
Parameters :
romAddr,the AT24C08 address;
datAddr,the memory space address;
data, data written to the AT24C08.
Description : Write a data to the AT24C08 by I^{2}C.
Example :
BOOL bSuc;
bSuc=Rom_Write(0xA0,0x35,0x62);

```
BOOL Rom_Write(UBYTE romAddr,UBYTE datAddr,UBYTE data)
{
  Start();
  Write8Bit(romAddr&0xFE); //I2C write address
  if(!GetAck())
    return FALSE;

  Write8Bit(datAddr);
  if(!GetAck())
    return FALSE;
  Write8Bit(data);
  if(!GetAck())
    return FALSE;
  Stop();
  return TRUE;
}
```

UBYTE Rom_Read(UBYTE romAddr,UBYTE datAddr)

Return : data from AT24C08.
Parameters :
romAddr,the AT24C08 address;
datAddr,the memory space address;
Description : Read a data from the AT24C08 by I^2C.
Example :
UBYTE tmp;
tmp=Rom_Read (0xA1,0x35);

```
UBYTE Rom_Read(UBYTE romAddr,UBYTE datAddr)
{
  UBYTE value;

  Start();
  Write8Bit(romAddr&0xFE); //I2C write address
  if(!GetAck())
    return FALSE;
  Write8Bit(datAddr);
  if(!GetAck())
    return FALSE;

  Start();
  Write8Bit(romAddr|0x01); //I2C read address
  if(!GetAck())
    return FALSE;

  value=Read8Bit();
  NoAck();
  Stop();

  return value;
}
```

4.3 I^2C Wave (E2PROM Read and Write Wave)

4.3.1 Write Wave-->Rom_Write (0xA0,0X11,0X35)

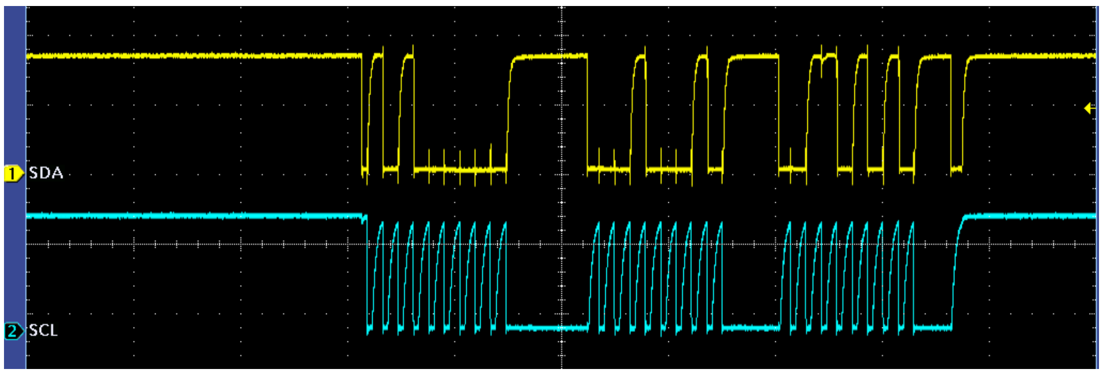

Figure. 6 Write Wave

4.3.2 Read Wave-->Rom_Read (0xA1,0X11)

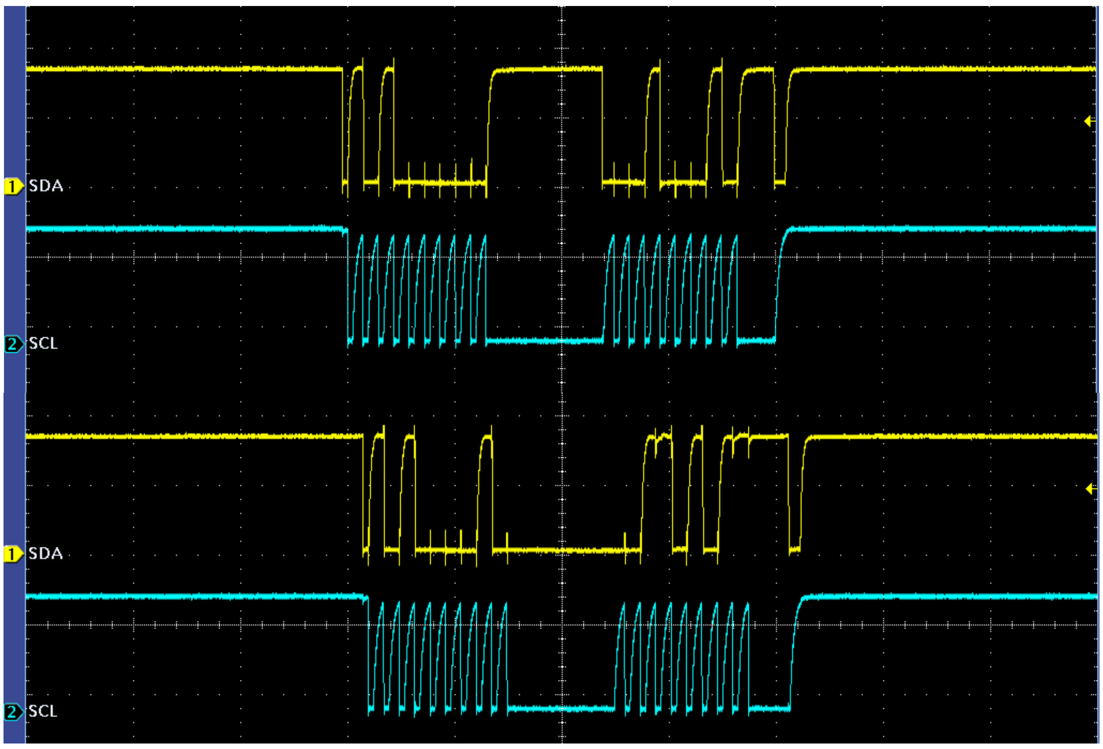

Figure 7 Read Wave

5 Notes on Items

I^{2}C address bit0 setting :
The bit0 is set as "1" in read mode and is set as "0" in write mode.
In the function “void DelayUS(UINT nDly)”, the ‘nDly’ is different from the other MCU. Please set the ‘nDly’ according to MCU.

6 More Information

For more information on FUJITSU MB95200 products, please visit following websites:
English version:

http://www.fujitsu.com/cn/fsp/services/mcu/mb95/application_notes.html

Simplified Chinese Version:

http://www.fujitsu.com/cn/fss/services/mcu/mb95/application_notes.html

7 Appendix