The following document contains information on Cypress products. The document has the series name, product name, and ordering part numbering with the prefix “MB”. However, Cypress will offer these products to new and existing customers with the series name, product name, and ordering part number with the prefix “CY”.

How to Check the Ordering Part Number

1. Go to www.cypress.com/pcn.
2. Enter the keyword (for example, ordering part number) in the **SEARCH PCNS** field and click **Apply**.
3. Click the corresponding title from the search results.
4. Download the Affected Parts List file, which has details of all changes

For More Information

Please contact your local sales office for additional information about Cypress products and solutions.

About Cypress

Cypress is the leader in advanced embedded system solutions for the world's most innovative automotive, industrial, smart home appliances, consumer electronics and medical products. Cypress' microcontrollers, analog ICs, wireless and USB-based connectivity solutions and reliable, high-performance memories help engineers design differentiated products and get them to market first. Cypress is committed to providing customers with the best support and development resources on the planet enabling them to disrupt markets by creating new product categories in record time. To learn more, go to www.cypress.com.

AN205323

FR, MB91460, I2C

I2C is a two-wire serial bus. There is no need for chip select or arbitration logic, making it cheap and simple to implement in hardware. The two I2C signals are serial data (SDA) and serial clock (SCL) and are both bi-directional. These are open drain output hence one need to connect pull up register on the SDA and SCL line.

Contents

1 Introduction

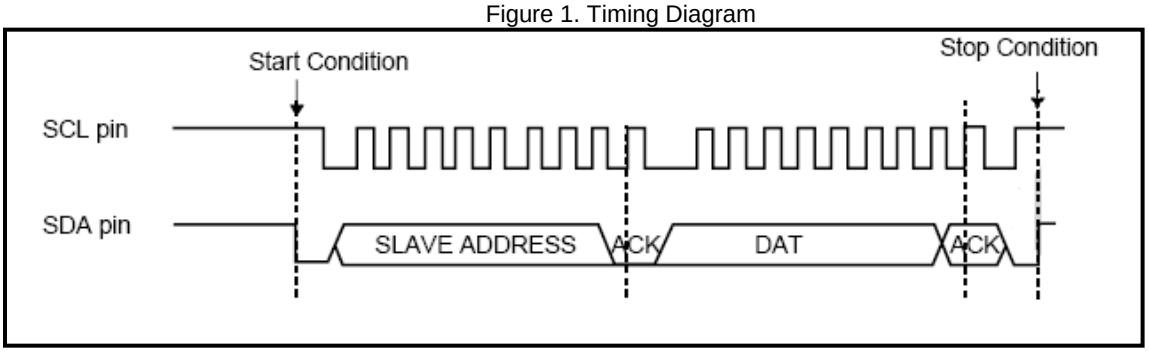
I2C is a two-wire serial bus. There is no need for chip select or arbitration logic, making it cheap and simple to implement in hardware. The two I2C signals are serial data (SDA) and serial clock (SCL) and are both bi-directional. These are open drain output hence one need to connect pull up register on the SDA and SCL line.

The device that initiates a transaction on the I2C bus is termed the master. The master normally controls the clock signal. A device being addressed by the master is called a slave.

Each I2C-compatible hardware slave device comes with a predefined device address, the lower bits of which may be configurable at the board level. The master transmits the device address of the intended slave at the beginning of every transaction. Each slave is responsible for monitoring the bus and responding only to its own address.

Data transfer is initiated with the START bit (S) when SDA is pulled low while SCL stays high. Then, SDA sets the transferred bit while SCL is low and the data is sampled (received) when SCL rises. When the transfer is complete, a STOP bit (P) is sent by releasing the data line to allow it to be pulled up while SCL is constantly high.

Figure 1. Timing Diagram



The master begins the communication by issuing the start condition (S). The master continues by sending a unique 7-bit slave device address, with the most significant bit (MSB) first. The eighth bit after the start, read/not-write (0/1), specifies whether the slave is now to receive (0) or to transmit (1). The receiver, acknowledging receipt of the previous byte, issues an ACK bit. Then the transmitter (slave or master, as indicated by the bit) transmits a byte of data starting with the MSB. At the end of the byte, the receiver (whether master or slave) issues a new ACK bit. This 9-bit pattern is repeated if more bytes need to be transmitted.

In a write transaction (slave receiving), when the master is done transmitting all of the data bytes it wants to send, it monitors the last ACK and then issues the stop condition (P). In a read transaction (slave transmitting), the master does not acknowledge the final byte it receives. This tells the slave that its transmission is done. The master then issues the stop condition.

This application note describes how to communicate via I2C with a Serial EEPROM. In this note a 24C04 EEPROM from Turbo IC is used.

Please note, that this document only gives a rough overview about the communication. The described source codes were written for understanding not for code size or speed. Neither interrupts nor timers were used. Time critical program code is always performed by simple flag polling or wait loops.

2 Registers

2.1 Bus Control Register (IBCR0)

The IBCR0 consists of the following bits:

Bit No.	Bit Name	Initial Value	Value	Description
15	BER	0	0	Clear bus error interrupt flag*
			1	No effect*
			0	No bus error detected+
			1	One of the error conditions described below detected+
14	BEIE	0	0	Bus error interrupt request disabled
			1	Bus error interrupt request enabled
13	SCC	0	0	No effect
			1	Generate repeated start condition during master transfer
12	MSS	0	0	Go to slave mode
			1	Go to master mode, generate start condition and send address data byte in IDAR0 register
11	ACK	0	0	The interface will not acknowledge on data byte reception
			1	The interface will acknowledge on data byte reception
10	GCAA	0	0	The interface will not acknowledge on general call address byte reception
			1	The interface will acknowledge on general call address byte reception
9	INTE	0	0	Interrupt request disabled
			1	Interrupt request enabled

Bit No.	Bit Name	Initial Value	Value	Description
8	INT	0	0	Clear transfer end interrupt flag*
			1	No effect*
			0	Transfer not ended or not involved in current transfer or bus is idle+
			1	Set at the end of a 1-byte data transfer or reception including the acknowledge bit under the following conditions: • Device is bus master. • Device is addressed as slave. • General call address received. • Arbitration loss occurred. Set at the end of an address data reception (after first byte if seven bit address received, after second byte if ten bit address received) including the acknowledge bit if the device is addressed as slave. +

+ Read Access

* Write Access

2.2 Bus Status Register (IBSR0)

The IBCR0 consists of the following bits:

Bit No.	Bit Name	Initial Value	Value	Description
7	BB	0	0	Stop condition detected (bus idle)
			1	Start condition detected (bus in use)
6	RCS	0	0	Repeated start condition not detected
			1	Bus in use, repeated start condition detected
5	AL	0	0	No arbitration loss detected
			1	Arbitration loss occurred during master sending
4	LRB	0	0	Receiver acknowledged
			1	Receiver did not acknowledge
3	TRX	0	0	Not transmitting data
			1	Transmitting data
2	AAS	0	0	Not addressed as slave
			1	Addressed as slave
1	GCA	0	0	General call address not received as slave
			1	General call address received as slave
0	ADT	0	0	Incoming data is not address data (or bus is not in use)
			1	Incoming data is address data

2.3 Ten Bit Slave Address Register (ITBA0)

This register (ITBAH0 / ITBAL0) designates the ten bit slave address.

Bit No.	Bit Name	Initial Value	Value	Description
15-10	RES	0		
9-0	TAn	1		Address bits

2.4 Ten Bit Address Mask Register (ITMK0)

This register contains the ten bit slave address mask and the ten bit slave address enable bit

Bit No.	Bit Name	Initial Value	Value	Description
15	ENTB	0	0	Ten bit slave address disabled
			1	Ten bit slave address enabled
14	RAL	0	0	Addressed as seven bit slave.
			1	Addressed as ten bit slave
13-10	RES	1		
9-0	TMn	1	0	Bit is not used in slave address comparison
			1	Bit is used in slave address comparison.

2.5 Seven Bit Slave Address Register (ISBA0)

This register (ITBAH0 / ITBAL0) designates the ten bit slave address.

Bit No.	Bit Name	Initial Value	Value	Description
7	RES	0		
6-0	SAn	1		Address bits

2.6 Seven Bit Address Mask Register (ISMK0)

This register contains the ten bit slave address mask and the ten bit slave address enable bit

Bit No.	Bit Name	Initial Value	Value	Description
15	ENSB	0	0	Ten bit slave address disabled
			1	Ten bit slave address enabled
14-8	SMn	1	0	Bit is not used in slave address comparison
			1	Bit is used in slave address comparison.

2.7 Data Register (IDAR0)

The data register is used in serial data transfer, and transfers data MSB-first.

Bit No.	Bit Name	Initial Value	Value	Description
15-8	RES	0		
7-0	Dn	0	Data bits	

2.8 Clock Control Register (ICCR0)

The clock control register (ICCR0) has the following functions:

- Enable IO pad noise filters
- Enable I2C interface operation
- Setting the serial clock frequency

Bit No.	Bit Name	Initial Value	Value	Description
15	RES	0		
14	NSF	0	This bit enables the noise filters built into the SDA and SCL IO pads	
13	EN	0	0	Interface disabled
			1	Interface enabled
12-8	CSn	0	Clock pre-scalar	

3 I2C Operation

This Chapter Describes Operation of I2C

3.1 Start Condition

When the bus is free (IBSRn: BB = 0, IBCRn: MSS = 0), writing ‘1’ to the IBCRn: MSS bit places the I2C interface in master mode and generates a start condition and the contents of the IDAR0 register (which should be address data) is sent.

Repeated start conditions can be generated by writing ‘1’ to the IBCRn: SCC bit when in bus master mode and interrupt status (IBCRn: MSS=1 and IBCRn: INT=1).

If a 1 is written to the IBCR0: `MSS` bit while the bus is in use (IBSRn: BB=1 and IBSRn: TRX=0; IBCRn: MSS=0 and IBCRn: INT=0), the interface waits until the bus is free and then starts sending.

Writing ‘1’ to the MSS bit or SCC bit in any other situation has no significance.

3.2 Stop Condition

Writing ‘0’ to the IBCRn: `MSS` bit in master mode (IBCRn: MSS=1 and IBCRn: INT=1) generates a stop condition and places the device in slave mode. Writing ‘0’ to the IBCRn: `MSS` bit in any other situation has no significance.

After clearing the IBCRn: `MSS` bit, the interface tries to generate a stop condition which might fail if another master pulls the SCL line low before the stop condition has been generated. This will generate an interrupt after the next byte has been transferred.

3.3 Slave Address Detection

In slave mode, after a start condition is generated the IBSRn: BB is set to ‘1’ and data sent from the master device is received into the IDARn register. After the reception of eight bits, the contents of the IDARn register is compared to the ISBAn register using the bit mask stored in ISMKn if the ISMKn: ENSB bit is ‘1’.

If a match results, the IBSRn: AAS bit is set to ‘1’ and an acknowledge signal is sent to the master. Then bit 0 of the received data (bit 0 of the IDARn register) is inverted and stored in the IBSRn: TRX bit.

If the ITMKn: ENTB bit is ‘1’ and a ten bit address header is detected, the interface sends an acknowledge signal to the master and stores the inverted last data bit in the IBSRn: TRX register. No interrupt is generated. Then, the next transferred byte is compared (using the bit mask stored in ITMKn) to the lower byte of the ITBAn register. If a match is found, an acknowledge signal is sent to the master, the IBSRn: AAS bit is set and an interrupt is generated.

3.4 Slave Address Masking

Only the bits set to ‘1’ in the mask registers (ITMKn / ISMKn) are used for address comparison, all other bits are ignored. The received slave address can be read from the ITBAn (if ten bit address received, ITMKn: RAL=1) or ISBAn (if seven bit address received, ISMKn: RAL=‘0’) register if the IBSRn: AAS bit is ‘1’.

If the bitmasks are cleared, the interface can be used as a bus monitor since it will always be addressed as slave. Note that this is not a real bus monitor because it acknowledges upon any slave address reception, even if there is no other slave listening.

3.5 Addressing Slaves

In master mode, after a start condition is generated the IBSRn: BB and IBSRn: TRX bits are set to ‘1’ and the contents of the IDARn register is sent in MSB first order. After address data is sent and an acknowledge signal was received from the slave device, bit 0 of the sent data (bit 0 of the IDARn register after sending) is inverted and stored in the IBSRn: TRX bit.

Acknowledgement by the slave may be checked using the IBSRn: LRB bit. This procedure also applies to a repeated start condition.

In order to address a ten bit slave for write access, two bytes have to be sent. The first one is the ten bit address header which consists of the bit sequence ‘1 1 1 1 0 A9 A8 0’, it is followed by the second byte containing the lower eight bits of the ten bit slave address (A7 - A0). A ten bit slave is accessed for reading by sending the above byte sequence and generating a repeated start condition (IBSRn: SCC) followed by a ten bit address header with read access (1 1 1 1 0 A9 A8 1).

Summary of the address data bytes:

7 bit slave, write access: Start condition - A6 A5 A4 A3 A2 A1 A0 0

7 bit slave, read access: Start condition - A6 A5 A4 A3 A2 A1 A0 1

10 bit slave, write access: Start condition - 1 1 1 1 0 A9 A8 0 - A7 A6 A5 A4 A3 A2 A1 A0

10 bit slave, read access: Start condition - 1 1 1 1 0 A9 A8 1 - A7 A6 A5 A4 A3 A2 A1 A0 - repeated start - 1 1 1 1 0 A9 A8 1

3.6 Acknowledgement

Acknowledge bits are sent from the receiver to the transmitter. The IBCRn: ACK bit can be used to select whether to send an acknowledgment when data bytes are received.

When data is sent in slave mode (read access from another master), if no acknowledgement is received from the master, the IBSRn: TRX bit is set to ‘0’ and the device goes to receiving mode. This enables the master to generate a stop condition as soon as the slave has released the SCL line.

In master mode, acknowledgement by the slave may be checked by reading the IBSRn: LRB bit.

4 Interfacing to EEPROM

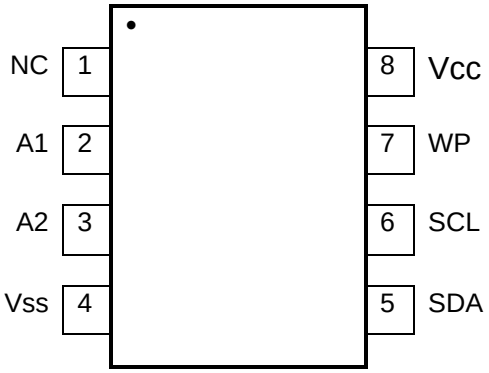
This Chapter Describes How to Communicate With the 24C04 EEPROM

4.1 EEPROM

The 24LC04 serial EEPROM from Turbo IC has 4096-Bit memory size, organized as 512 x 8 Bits.

The 24LC04 has the following pin-out:

Figure 2. Pin Diagram of EEPROM 24LC04



Pin names:

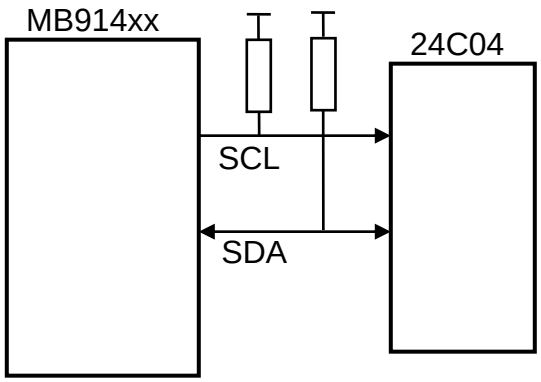
Table 1. Pin names of EEPROM 24LC04

NC	Not connected
A1	Address select 1
A2	Address select 2
Vss	Ground
SDA	Serial Data
SCL	Serial Clock
WP	Write Protect
Vcc	Power Supply (+ 5 volts)

4.2 Connection to MB914xx

The EEPROM can be connected as in the following schematic. Please note, that no power supply pins and other MCU-Pins are drawn than those for the connection to the EEPROM.

Figure 3. Connection block diagram



4.3 Addressing

The EEPROM uses the 8-bit addressing scheme of the I2C bus. The 4 most significant bits are fixed to “1010”. The A1 and A2 lines of the chip select the next two address bits. The next bit is the least significant bit of the memory address. The last address bit select Read or Write operation, as defined by the I2C standard.

1	0	1	0	A2	A1	A0	R/$\overline{W}$

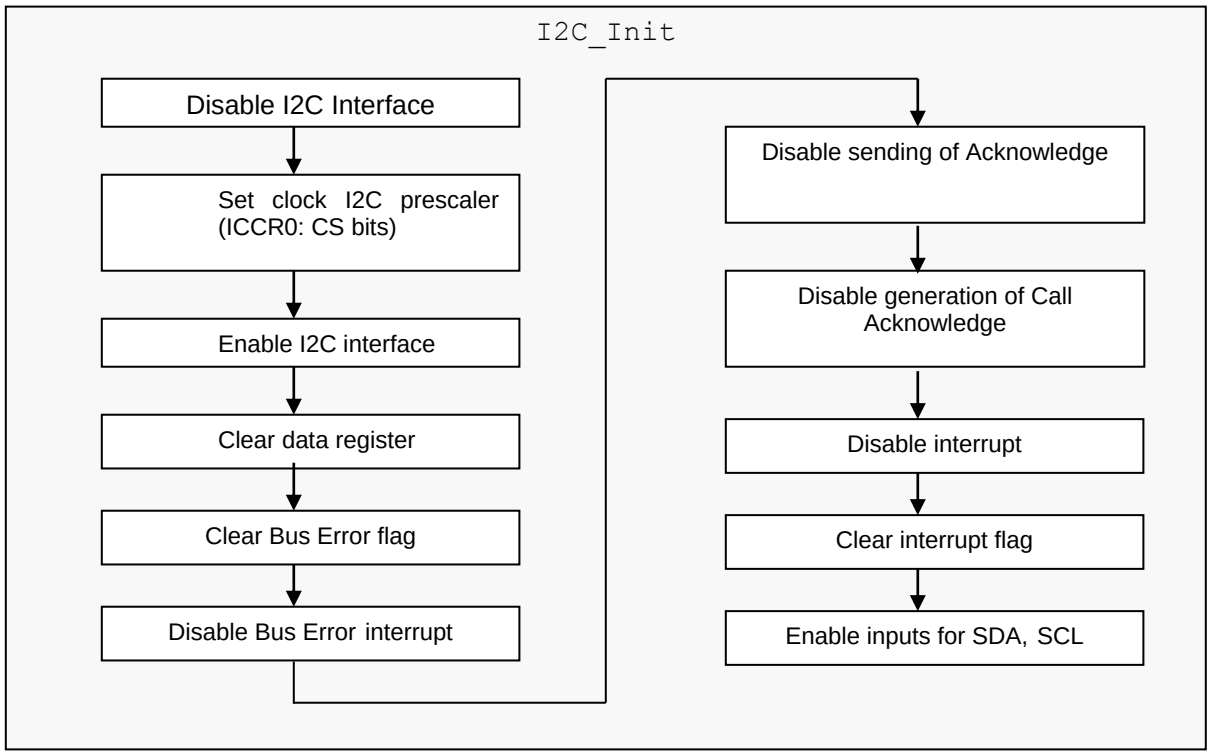
The actual address that is written to or read from is transmitted as the first data byte. It can only be set by a write operation, i.e. if a certain memory address needs to be read, a write access for that address is required before reading it.

4.4 Example Code *

The following code shows how to establish a communication to and from the EEPROM using the MB91F467D.

4.4.1 Initial Functions and Declarations

4.4.1.1 Flowchart



4.4.1.2 C Code

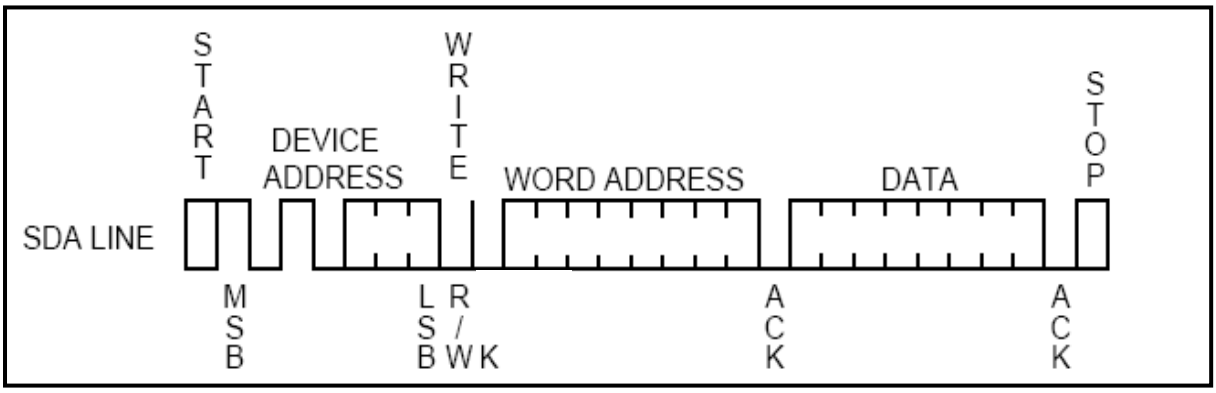
```
void I2C_Init(void)
{

  PFR22 = 0x30;                 // set port function register for I2C

  ICCR0_EN  = 0;                // stop I2C interface
  ICCR0_CS4 = 1;                // CS4..0 : set prescaler
  ICCR0_CS3 = 1;
  ICCR0_CS2 = 1;
  ICCR0_CS1 = 1;
  ICCR0_CS0 = 1;
  ICCR0_EN  = 1;                // enable I2C interface

  IDAR0 = 0;                    // clear data register

  IBCR0_BER  = 0;               // clear bus error interrupt flag
  IBCR0_BEIE = 1;               // bus error interrupt enabled
  IBCR0_ACK  = 0;               // no acknowledge generated
  IBCR0_GCAA = 0;               // no call acknowledge is generated
  IBCR0_INTE = 1;               // enable interrupt
  IBCR0_INT  = 0;               // clear transfer interrupt request flag
}
```

4.4.2 Write EEPROM

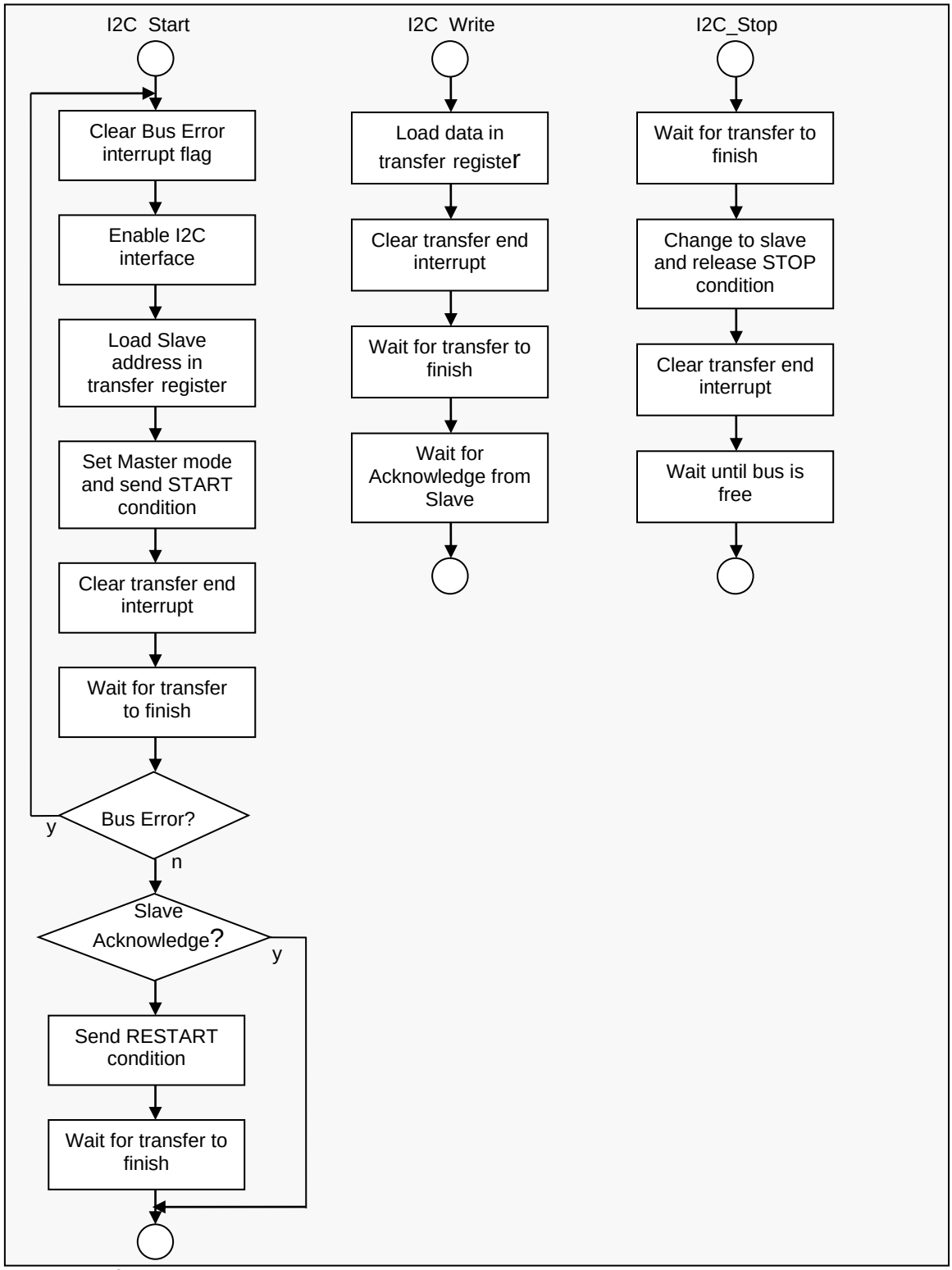
Writing to the EEPROM is done in several steps. First, the I2C controller is set to Master mode and the slave address is configured. Then, the memory address is transmitted to the EEPROM. Next, actual data to write is transmitted. Finally, the I2C controller is switched back to slave mode.

4.4.2.1 Timing Diagram

Figure 4. Byte write



4.4.2.2 Flowchart



4.4.2.3 Code

```
void I2C_Acknowlegde()
{
  while(IBSR0_LRB == 1);          // no answer from slave, program stucks here
                                  // a timeout mechanism should be implemented her
}

void I2C_Start(unsigned char slave_address)
{
  do
  {
    IBCR0_BER = 0;                // clear bus error interrupt flag
    ICCR0_EN  = 1;                // enable I2C interface
    IDAR0     = slave_address;    // slave_address is sent out with start condition

    IBCR0_MSS = 1;                // set master mode and set start condition
    IBCR0_INT = 0;                // clear transfer end interrupt flag

    while(IBCR0_INT == 0);        // look if transfer is in process
  }
  while (IBCR0_BER == 1);         // retry if Bus-Error detected

  while(IBSR0_LRB == 1)           // no acknowledge means device not ready
  {                               // maybe last write cycle not ended yet
    IBCR0_SCC = 1;                // try restart (= continue)
    while (IBCR0_INT == 0);       // wait that transfer is finished
  }
}
```

```
void I2C_Stop(void)
{
  while (IBCR0_INT == 0);         // wait that transfer is finished
  IBCR0_MSS = 0;                  // change to slave and release stop condition
  IBCR0_INT = 0;                  // clear transfer end interrupt flag
  while (IBSR2_BB);               // wait till bus free
}

void I2C_Write(unsigned char value)
{
  IDAR0 = value;                  // load data or address in to register
  IBCR0_INT = 0;                  // clear transfer end intrerupt flag
  while (IBCR0_INT == 0);         // look if transfer is in process
  I2C_Acknowlegde();              // wait for Acknowledge
}
```

```
void main(void) {
    __EI();                         /* enable interrupts */
    __set_il(31);                   /* allow all levels */
    InitIrqLevels();                /* init interrupts */

    PORTEN = 0x3;                   /* enable I/O Ports */
                                    /* This feature is not supported by MB91V460A */
                                    /* For all other devices the I/O Ports must be enabled*/

    I2C_Init();                     /* initialize the I2C */

    /* write to EEPROM */
    I2C_Start(Slave_Addr | I2C_WRITE);

    I2C_Write(0x00);                    /* Address where to write to */

    I2C_Write(0x11);                      /* Databytes 'A' */
    I2C_Write(0x22);                      /* Databytes 'B' */
    I2C_Write(0x33);                      /* Databytes 'C' */
    I2C_Write(0x44);                      /* Databytes 'D' */
    I2C_Write(0x55);                      /* Databytes 'E' */
    I2C_Write(0x66);                      /* Databytes 'F' */

    I2C_Stop();
```

```
    while(1) {                      /* endless loop */


        HWWD_CL = 0;

        /* feed hardware watchdog */
        /* (Only for devices with hardware (R/C based) watchdog) */
        /* The hardware (R/C based) watchdog is started */
        /* automatically after power-up and can not be stopped */
        /* If the hardware watchdog is not cleared frequently */
        /* a reset is generated. */
    }
}
```

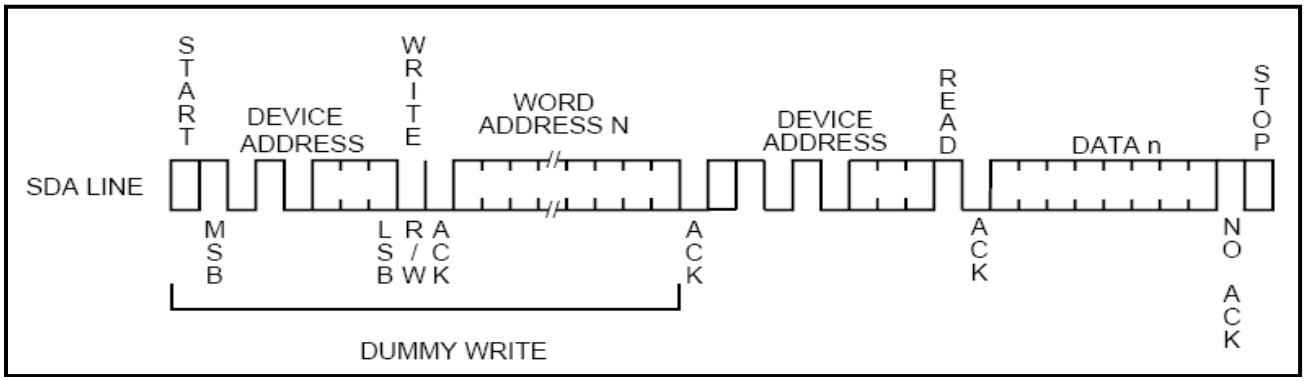
4.4.3 Read from EEPROM

Reading from the EEPROM is performed in several steps. First, a START condition is generated. A write command containing the memory address is sent next. This is followed by a RESTART condition and the actual read cycles. Please note that after the last read byte, no-acknowledge has to be generated by the Master. Finally, a STOP condition is generated.

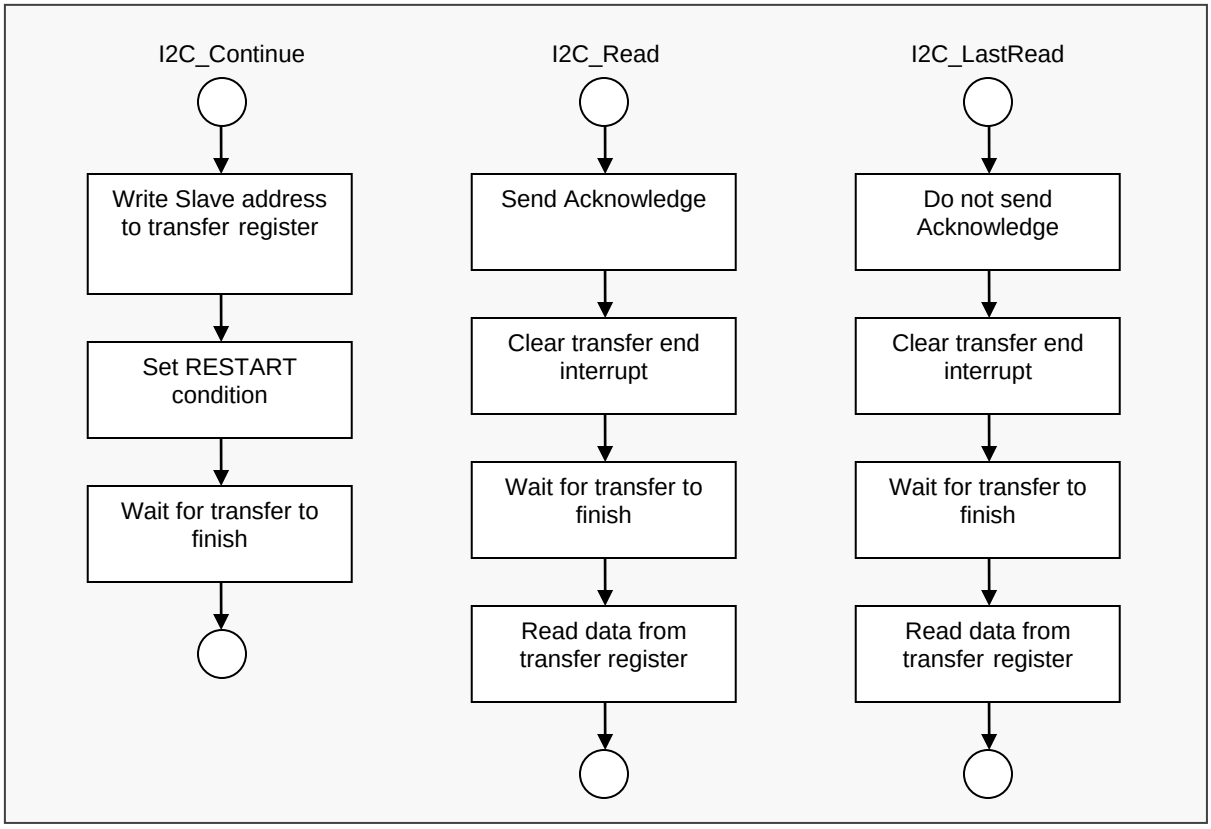
The START, write, and STOP functions are the same as for writing data to the EEPROM. In the following, only the flowchart of the RESTART condition and the actual read function is shown. For the other functions, please refer to the write command.

4.4.3.1 Timing Diagram

Figure 5. Random Read



4.4.3.2 Flowchart



4.4.3.3 Code

```
unsigned char I2C_LastRead(void)
{
  IBCR0_ACK = 0;                    // acknowledge has to be sent after last byte
  IBCR0_INT = 0;                    // clear transfer end interrupt flag
  while(IBCR0_INT == 0);            // wait that transfer is finished
  return(IDAR0);                    // read received data out
}


void I2C_StartWrite (unsigned char slave_address)
{
    while(IBSR0_BB);                // wait till bus is free

    IBCR0_BER = 0;                  // clear bus error interrupt flag
    ICCR0_EN  = 1;                  // enable I2C interface

    IDAR0     = slave_address;      // slave_address is sent out with start condition
    IBCR0_MSS = 1;                  // set master mode and set start condition
    IBCR0_INT =0;
}
```

```
void I2C_Continue(unsigned char slave_address)
{
  IDAR0     = slave_address;        // slave_address is sent out with start condition
  IBCR0_SCC = 1;                    // restart (= continue)
  while (IBCR0_INT == 0);           // wait that transfer is finished
}
```

```
void main(void) {
    __EI();                         /* enable interrupts */
    __set_il(31);                   /* allow all levels */
    InitIrqLevels();                /* init interrupts */

    PORTEN = 0x3;                   /* enable I/O Ports */
                                    /* This feature is not supported by MB91V460A */
                                    /* For all other devices the I/O Ports must be enabled*/

    I2C_Init();                     /* initialize the I2C */
```

```
    /* read from EEPROM */
    I2C_Start(Slave_Addr | I2C_WRITE);

    I2C_Write(0x00);                        /* Address where to read from */

    I2C_Continue(Slave_Addr | I2C_READ);    /* Restart, with READ comand */

    i = 0;

    result[i++] = I2C_Read();       /* receive data from EEPROM */
    result[i++] = I2C_Read();       /* receive data from EEPROM */
    result[i++] = I2C_Read();       /* receive data from EEPROM */
    result[i++] = I2C_Read();       /* receive data from EEPROM */
    result[i++] = I2C_Read();       /* receive data from EEPROM */
    result[i++] = I2C_LastRead();   /* receive last byte from EEPROM */

    I2C_Stop();

    while(1) {                      /* endless loop */


        HWWD_CL = 0;

        /* feed hardware watchdog */
        /* (Only for devices with hardware (R/C based) watchdog) */
        /* The hardware (R/C based) watchdog is started */
        /* automatically after power-up and can not be stopped */
        /* If the hardware watchdog is not cleared frequently */
        /* a reset is generated. */
    }
}
```

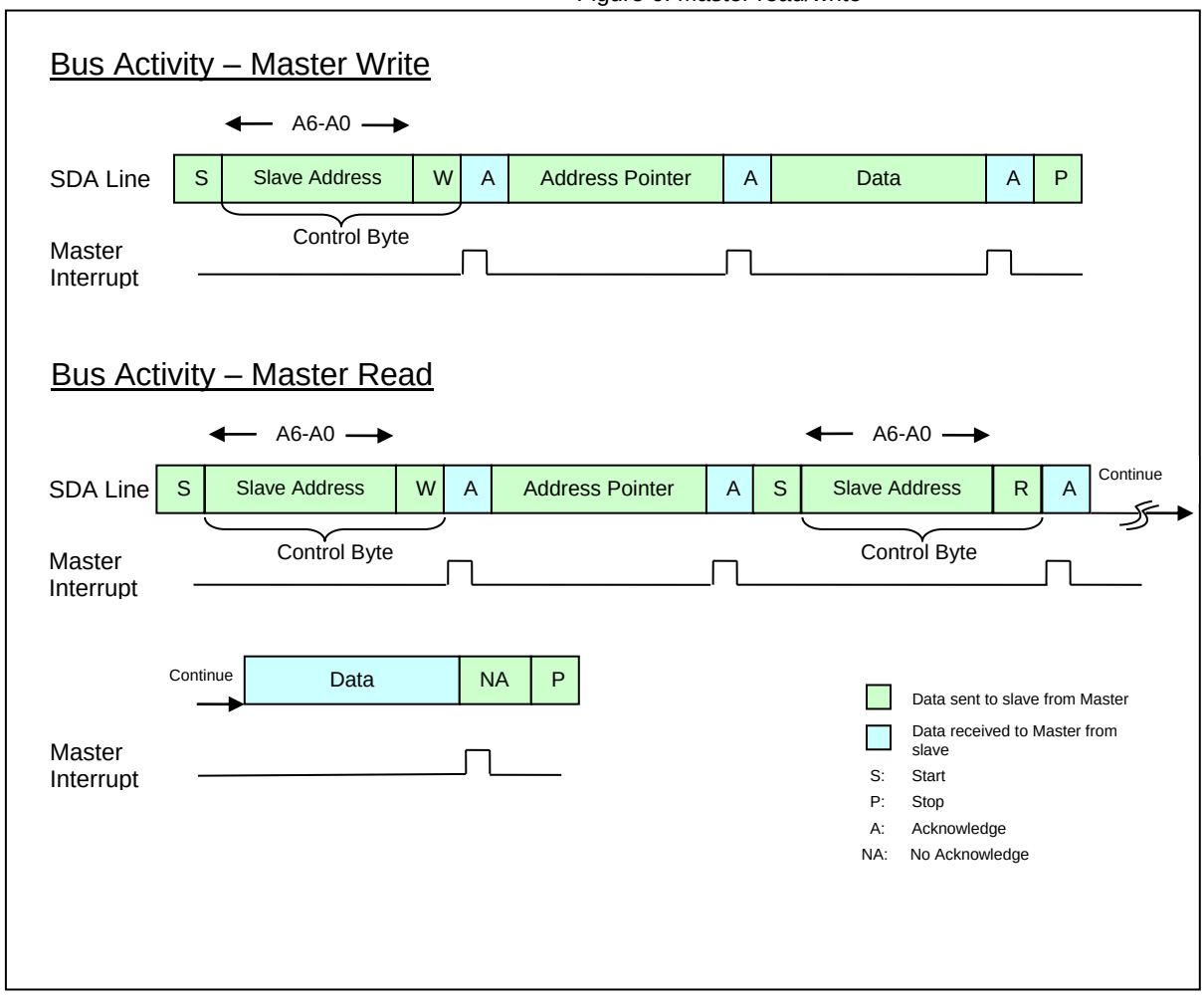
4.4.4 Master mode using Interrupt

Here actual read and write of data is carried out in Interrupt service routine.

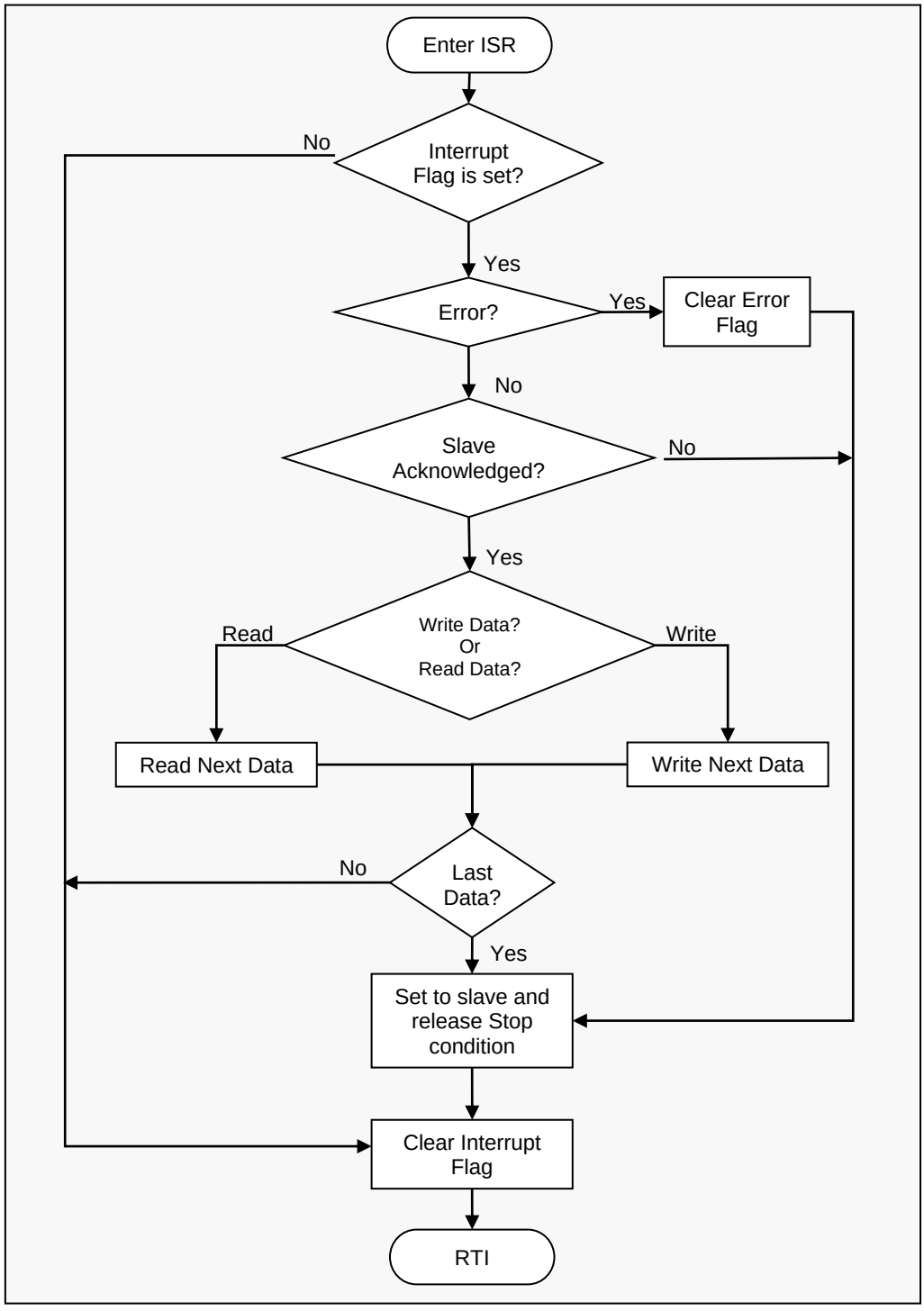
In master mode, after a start condition is generated the contents of the Data (IDARn) register are sent in MSB first order. After address data is sent and an acknowledge signal was received from the slave device, bit 0 of the sent data (bit 0 of the IDAR0 register after sending) is inverted and stored in the TRX bit. Acknowledgement by the slave may be checked using the LRB bit in the IBSR0 register. This procedure also applies to a repeated start condition.

4.4.4.1 Timing Diagram

Figure 6. Master read/write



4.4.4.2 Flowchart



4.4.4.3 Code

```
__interrupt void I2CInterrupt(void)
{
    if (IBCR0_INT)                                  /* ISR request ? */
    {
        if (!IBCR0_BER)                             /* check for Bus error */
        {
            if (IBSR0_GCA==1 || IBSR0_AAS==1)       /* general call as slave or addressed as slave ? */
            {
                /* implementation behaviour as slave */
            }
            else
            {
                /* Master mode send/get data ! */
                if (IBSR0_AAS==0)                   /* addressed not as slave ? */
                {
                    if(IBSR0_AL)                    /* arbitration lost ? */
                    {
                        error = 3;                  /* arbitration error master mode */
                    }
                    else
                    {
                        if(IBSR0_LRB && !get_value) /* check if got Acknowlegde from slave  */
                        {
                            error = 5;              /* slave did not ACK */
                            IBCR0_MSS = 0;          /* Set to slave and release stop condition */
                        }
                        else
                        {
                            if (IBSR0_TRX)          /* check if send or receive data in Master mode*/
                            {
                                switch (cnt)        /* sending data in Master mode */
                                {
                                    case 0 :                                    /* send data address */
                                        IDAR0 = adr_buf;                        /* write Address */
                                        cnt++;
                                        break;

                                    case 1 :                                    /* write DATA or restart send slave address to read data */
                                        if (get_value)                          /* data read or write ? */
                                        {
                                            IDAR0     = Slave_Addr | I2C_READ;  /* slave_address is sent out with start condition  */
                                            IBCR0_SCC = 1;                      /* restart (= continue) */
                                            cnt++;
                                        }
                                        else
                                        {
                                            IDAR0 = write_buf[cnt-1];           /* write DATA */
```

```
                                    cnt++;
                                }
                                break;

                            case 7 :                                    /* release bus after succesful write cycle */
                                IBCR0_MSS = 0;                          /* Set to slave and release stop condition */
                                ready = 1;                              /* transmission ready */
                                break;

                            default: // case 2, 3, 4, 5
                                IDAR0 = write_buf[cnt-1];               /* write DATA */
                                cnt++;
                                break;
                        } // switch (cnt)
                    }
                    else // if (IBSR0_TRX)
                    {
                        switch (cnt)                    /* receiving data in Master mode */
                        {
                            case 2 :
                                IBCR0_ACK = 1;          /* send acknowledge request data byte */
                                cnt++;
                                break;

                            case 7  :
                                rec_buf[cnt-3]=IDAR0;   /* get data */
                                IBCR0_ACK = 0;          /* do not send acknowledge on next byte */
                                cnt++;
                                break;
```

```
                            case 8 :                    /* read transfer finished, release bus */
                                rec_buf[cnt-3]=IDAR0;   /* get data */
                                IBCR0_MSS = 0;          /* Set to slave and release stop condition */
                                ready = 1;              /* set transfer ready flag */
                                break;

                            default: // case 3, 4, 5, 6
                                rec_buf[cnt-3]=IDAR0;   /* get data */
                                IBCR0_ACK = 1;          /* send acknowledge request next data byte */
                                cnt++;
                                break;
                        } // switch (cnt)
                    } // if (IBSR0_TRX) - else
                } // if(IBSR0_LRB && !get_value) - else
            } // if(IBSR0_AL) - else
        } // if (IBSR0_AAS==0)
    } // if (IBSR0_GCA==1 || IBSR0_AAS==1) - else
} // if (IBCR0_BER)
else // if (IBCR0_BER)
{
    error=1;            /* Bus error */
    IBCR0_MSS = 0;      /* Set to slave and release stop condition */
    IBCR0_BER = 0;      /* clear bus error flag */
} // if (IBCR0_BER) - else
} // if (IBCR0_INT)

IBCR0_INT = 0;          /* clear Int flag */

} // __interrupt void I2CInterrupt(void)
```

```
void Write_MemoryINT (unsigned char mem_adr)
{
    adr_buf = mem_adr;                  /* store memory address */

    cnt          = 0;                   /* set buffer count to '0' */
    ready        = 0;                   /* clear transfer flag */
    error        = 0;                   /* clear error flag */
    get_value    = 0;                   /* clear Read_flag */

    I2C_StartWrite( Slave_Addr | I2C_WRITE );    /* start WRITE transfer */
}



void  Read_MemoryINT (BYTE mem_adr)
{
    adr_buf = mem_adr;                  /* store memory address */

    cnt          = 0;                   /* set buffer count to '0' */
    ready        = 0;                   /* clear transfer flag */
    error        = 0;                   /* clear error flag */
    get_value    = 1;                   /* set Read_flag */

    I2C_StartWrite( Slave_Addr | I2C_WRITE );    /* start READ transfer, write slave Address */
}
```

```
/*************************@INCLUDE_START************************/
#include "mb91467d.h"
#include "global.h"

/* I2C */
#define I2C_READ     0x01
#define I2C_WRITE    0x00
#define Slave_Addr   0xA0     /* I2C Slave device address */
/*************************@INCLUDE_END*************************/

/*********************@GLOBAL_VARIABLES_START*******************/
volatile int i;
BYTE cnt, ready, error, getchar, get_value;
BYTE adr_buf, write_buf[10], rec_buf[10];
/*********************@GLOBAL_VARIABLES_END*********************/
```

```
void main(void)
{
    __EI();                         /* enable interrupts */
    __set_il(31);                   /* allow all levels */
    InitIrqLevels();                /* init interrupts */

    PORTEN = 0x3;                   /* enable I/O Ports */
                                    /* This feature is not supported by MB91V460A */
                                    /* For all other devices the I/O Ports must be enabled*/

    I2C_Init();                     /* initialize the I2C */

    /* write transmission buffer 'write_buf[]' to EEPROM */
    write_buf[0] = 0x11;
    write_buf[1] = 0x22;
    write_buf[2] = 0x33;
    write_buf[3] = 0x44;
    write_buf[4] = 0x55;
    write_buf[5] = 0x66;

    Write_MemoryINT (0x00);

    while(!ready);                  /* wait until transmission ready */

    /* delay loop for internal write cycle time of EEPROM */
    /* e.g. X24C08 typ. 5 ms / max. 10 ms */

    for(i=0;i<40000;i++);

    /* read from EEPROM to reception buffer 'rec_buf[]' */

    Read_MemoryINT (0x00);

    while(!ready);                  /* wait until reception ready */

    while(1)                            /* endless loop */
    {
        HWWD_CL = 0;

        /* feed hardware watchdog */
        /* (Only for devices with hardware (R/C based) watchdog) */
        /* The hardware (R/C based) watchdog is started */
        /* automatically after power-up and can not be stopped */
        /* If the hardware watchdog is not cleared frequently */
        /* a reset is generated. */
    } // while (1)
} // main
```

5 Slave mode using Interrupt

This Chapter Describes How to Use MB91F467D in Slave Mode

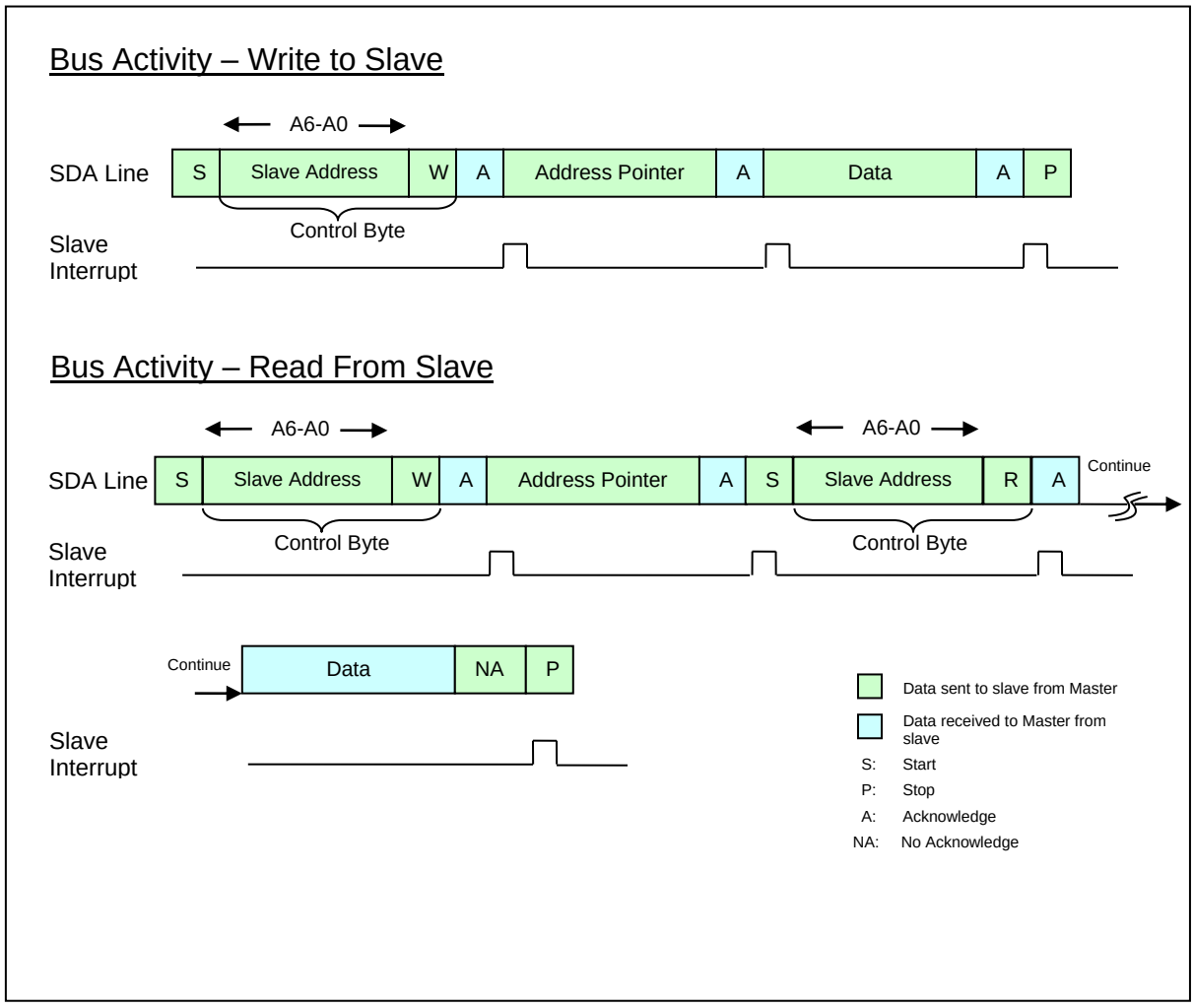
Here MCU is configured in slave mode with seven bit addressing enabled. MCU has two data arrays, write_buf[], where data received from master is stored and read_buf[], from where data to the master is sent.

In slave mode, after a start condition is generated data sent from the master device is received into the Data (IDARn) Register. After the reception of eight bits, the contents of the Data (IDARn) Register is compared to the Seven Bit slave Address (ISBA) register using the bit mask stored in Seven bit slave address Mask (ISMKn), if the ISMKn: ENSB bit is 1. If a match results, an acknowledge signal is sent to the master and an interrupt is generated. For data transfers slave need to generate Acknowledge signal by setting IBCRn: ACK bit to 1.

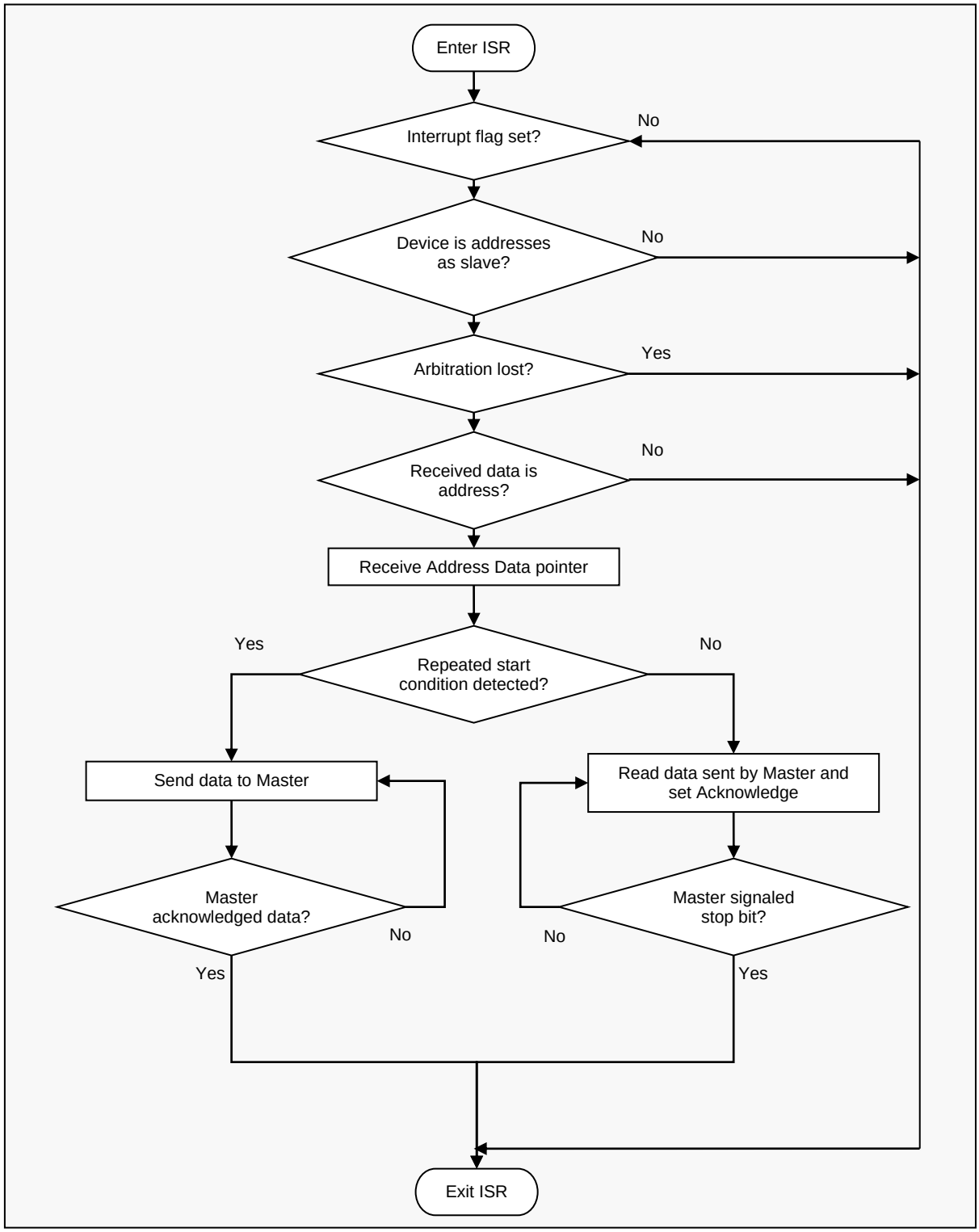
Only the bits set to '1' in the mask registers (ISMKn) are used for address comparison, all other bits are ignored. For correct Master and slave addressing and for setting of (ISBA) register, Please refer 3.3 , 3.4 and 3.5.

5.1.1.1 Timing Diagram

Figure 7. Slave read/write



5.1.1.2 Flowchart



5.1.1.3 Code

```
/*********************Interrupt Service Routines*******************/
/* I2C ISR */
__interrupt void I2CInterrupt(void)
{
    if (IBCR0_INT)                                  /* ISR request ? */
    {
        if (!IBCR0_BER)                             /* check for Bus error */
        {
            if (IBSR0_AAS==1)                       /* addressed as slave ? */
            {
                if(IBSR0_AL)                        /* arbitration lost ? */
                {
                    error = 3;                      /* arbitration error  */
                }
                else
                {
                    switch(cnt)
                    {
                        case 0:
                        if(IBSR0_ADT)                   /* Check if addressed */
                        {
                            IBCR0_ACK = 1;              /* set to ack on next byte*/
                            cnt = 1;
                        }
                        break;

                        case 1:
                        offset = IDAR0;                 /*Set the pointer, to read / write data in / from array*/
                        IBCR0_ACK = 1;                  /* set to ack on next byte */
                        cnt = 2;
                        break;

                        case 2:
                        if(IBSR0_RSC==1)                /*If repeated start condition detected ?*/
                        {
                            if(IBSR0_ADT)               /* Check if addressed */
                            {
                                IDAR0 = read_buf[offset++]; /*Send data*/
                                cnt = 3;
                            }
                        }
                        else if (IBSR0_RSC == 0)
                        {
                            write_buf[offset++] = IDAR0;    /* Read data*/
                            IBCR0_ACK = 1;                  /* Give Ack */
                            cnt = 4;
                            get_data = 1;
                        }
```

```
                        break;

                        case 3:
                        if(!IBSR0_LRB)                  /* check if got Acknowlegde from slave  */
                        {
                            IDAR0 = read_buf[offset++]; /*Send data*/
                        }
                        else
                        {
                            cnt = 0;                    /*If Not ACk*/
                        }
                        break;

                        case 4:
                        {
                            if(IBSR0_ADT == 1)
                            {
                                IBCR0_ACK = 1;                  /* If new control byte is received*/
                                cnt = 1;                        /* Restart the count  */
                            }
                            else
                            {
                                write_buf[offset++] = IDAR0;    /* Read data */
                                IBCR0_ACK = 1;                  /* Set ack */
                            }
                        }
                        break;

                        default:
                        cnt = 0;
                        break;
                    } // switch
                }// if (IBSR0_AL) - else
            }//if (IBSR0_AAS==1)
        }//if (IBCR0_BER)
```

```
        else
        {
            error=1;            /* Bus error */
            IBCR0_MSS = 0;      /* Set to slave and release stop condition */
            IBCR0_BER = 0;      /* clear bus error flag */
        } // if (IBCR0_BER) - else
    } // if (IBCR0_INT)

    IBCR0_INT = 0;          /* clear Int flag */

} // __interrupt void I2CInterrupt(void)
```

```
/************************@INCLUDE_START************************/
#include "mb91467d.h"
#include "global.h"
/************************@INCLUDE_END************************/
/*********************@GLOBAL_VARIABLES_START*********************/
volatile int i;
BYTE cnt, ready, error, offset;
BYTE get_data;
BYTE write_buf[256];
BYTE read_buf [256] = {0xa,0xb,0xc,0xd,0xe,0xf,0x55,0xaa,0x55,0xaa} ;
/*********************@GLOBAL_VARIABLES_END*********************/
```

```
void I2C_Init(void)
{

  PFR22 = 0x30;                    // set port function register for I2C

  ICCR0_EN  = 0;                   // stop I2C interface
  ICCR0_CS4 = 1;                   // CS4..0 : set prescaler
  ICCR0_CS3 = 1;
  ICCR0_CS2 = 1;
  ICCR0_CS1 = 1;
  ICCR0_CS0 = 1;

  IDAR0 = 0;                       // clear data register

  ISBA0 = 0x05;                    // Allow all address
  ISMK0 = 0xFF;                    // 7-bit slave address

  ITMK0 = 0x0000;

  IBCR0_BER  = 0;                  // clear bus error interrupt flag
  IBCR0_BEIE = 1;                  // bus error interrupt enabled
  IBCR0_ACK  = 0;                  // no acknowledge generated
  IBCR0_GCAA = 0;                  // no call acknowledge is generated

  IBCR0_INT  = 0;                  // clear transfer interrupt request flag

  ICCR0_EN  = 1;                   // enable I2C interface

  IBCR0_MSS = 0;                   // set slave mode
  IBCR0_INTE = 1;                  // enable interrupt
}
```

```
void main(void)
{
    __EI();                        /* enable interrupts */
    __set_il(31);                  /* allow all levels */
    InitIrqLevels();               /* init interrupts */

    PORTEN = 0x3;                  /* enable I/O Ports */
                                   /* This feature is not supported by MB91V460A */
                                   /* For all other devices the I/O Ports must be enabled*/

    I2C_Init();                    /* initialize the I2C */

    while(1)                            /* endless loop */
    {


        HWWD_CL = 0;


        /* feed hardware watchdog */
        /* (Only for devices with hardware (R/C based) watchdog) */
        /* The hardware (R/C based) watchdog is started */
        /* automatically after power-up and can not be stopped */
        /* If the hardware watchdog is not cleared frequently */
        /* a reset is generated. */
    } // while (1)
} // main
```

6 Additional Information

INFORMATION about CYPRESS Microcontrollers can be found on the following Internet page:

http://www.cypress.com/cypress-microcontrollers

The software examples related to this application note is:

91460_i2c_400khz-v11

91460_i2c_400khz_irq-v10

91460_i2c_400khz_irq_slave-v10

It can be found on the following Internet page:

http://www.cypress.com/cypress-mcu-product-softwareexamples

Document History

Document Title: AN205323 - FR, MB91460, I2C

Document Number: 002-05323

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	NOFL	02/26/2008	V1.0, First draft, HPi
			03/19/2008	V1.1 Updated Example and Description, HPi
			05/30/2008	V1.2 Updated chapter 3, HPi
			06/11/2008	V1.3 Reworked on source code, HPi
*A	5129236	NOFL	02/07/2016	Converted Spansion Application Note “MCU-AN-300072-E-V13” to Cypress template.
*B	5873188	AESATMP9	09/05/2017	Updated logo and copyright.
*C	6060508	NOFL	02/06/2018	Updated hyperlinks across the document. Updated to new template. Completing Sunset Review.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709