Simple I2C routines for PIC16

1   INTRODUCTION

There are numerous I2C routines available on the web, from Microchip and other sources. The great many of them have one thing in common, there are quite complex to understand and use. As a matter of fact they are much too complex for most needs.

It is true that in some systems this complexity is useful and even required, but in most cases one master controls 1 to 3 or 4 peripherals, one at a time, and no complexity is required in these cases.

Interrupts driven routines are also not typically required for simple cases when things unfold quite linearly as the program executes. So here we do not use interrupts.

This document does not explain what is I2C, and its operation, there is plenty of information not only on the web, but in nearly any data sheet for devices using I2C. Here, we show a few simple routines, to address most needs in a system with 1 master and 1 or more peripherals.

These examples have been used on a Microchip PIC16(L)F15376 but have also been used on other PIC16 devices with just a few modification in register names.

2   BUILDING BLOCKS

Looking in data sheets for devices using I2C, the same "actions" are always required. They are:

- I2C initialize
- Start
- Stop
- Repeated Start
- Read I2C (8 bits)
- Write I2C (8 bits)

So if we have a routine for each of these actions, we have a working I2C interface.

In most systems, there will be no need for bus busy detection, collision detection or other advanced functions such as error handling and multiple masters control.

In Microchip parlance, I2C interfaces are called MASTER SYNCHRONOUS SERIAL PORT (MSSPx) MODULES. Here we are using the MSSP1 for I2C communications.

This can be set up using the MCC. The MCC is a Microchip GUI, provided with their IDE, that can be used to configure the various sections of the MCU, in particular here, the MSSP1. The MCC generates ready to use source files for the compiler. These source files, not only provide initialization routines, but provide complete I2C routines and examples of usage. However these are quite complex and largely undocumented and so are not usable

for this author. However, the initialization routine is totally relevant. The other way to produce this initialization routine is to write it, simply by reading the relevant section of the data sheet and setting up the needed registers.

2.1 Let's look first at the initialization routine.

```
#include "i2c.h"

void I2C1_Initialize (void)
{
    SSP1STAT = 0x00;   // SMP High Speed; CKE disabled;
    SSP1CON1 = 0x28;   // SSPEN enabled; SSPM FOSC/4_SSPxADD_I2C;
    SSP1CON3 = 0x00;   // SBCDE, BOEN, SCIE , PCIE, DHEN disabled; SDAHT 100ns; AHEN disabled;
    SSP1ADD = 0x04;    // SSPADD 4; 400 kHz
    // clear the interrupt flags
    PIR3bits.SSP1IF = 0;
    PIR3bits.BCL1IF = 0;
}
```

Figure 1 I2C initialization routine

`#include "i2c.h"` introduces the functions prototypes to the compiler.

`SSP1STAT = 0x00;` This register mainly reflects the status of MSSP1. Two bits needs to be set[1], the SMP bit for high speed (400 kHz here) and the CKE bit for the voltage thresholds to be I2C compatible.

`SSP1CON1 = 0c28;` One of the control registers. Here the SSPEN bit enables the MSSP1 to act as an I2C interface, setting up the SDA and SCL pins and the SSPM bits set the interface to "I2C Master mode, clock = FOSC / (4 * (SSPxADD+1))", in this case 400 kHz.

`SSP1CON2` is not set up here as it is used only during I2C operation as we will see below.

`SSP1CON3 = 0x00;` Another control register. SBCDE disabled, we are not concerned with bus collision as we have a simple 1 master, 1 or more slaves. Bus collisions cannot occur. Note that this bit is for slave mode, the master turns into a slave when it is served data, but again this does not concern our simple case. BOEN disabled is also of no concern in our simple case. SCIE disabled of no concern since we do not use interrupts for I2C operation. PCIE disabled since we do not use interrupts. DHEN disabled, this is not needed in our simple and straightforward case. SDAHT 100 ns, this is the normal case. One could change this if it is required by one of the slave devices used. AHEN disabled, this is the normal case.

`SSP1ADD = 0x04;` This sets the clock rate at 400 kHz. SCL pin clock period = ((ADD<7:0> + 1) *4)/ FOSC. This system is running at 8 MHz, this has to be changed for different clock speeds.

Note that in recent implementations of the MCC[2], Microchip disables the MSSPx by default. So to use it, do not forget to add[3] :

```
SSP1CON1bits.SSPEN = 1;  // enable I2C
```

---

1. I am using "set" to mean adjusted and not in the sense set (to 1) the opposite of reset (set to 0).
2. This is written in 2021.
3. This is also the case for the SPI if used.

2.2 Lets now look at the Start routine.

```c
void Send_I2C_Start(void)
{
    PIR3bits.SSP1IF = 0;      // clear interrupt flag
    SSP1CON2bits.SEN = 1;     // send start bit
    while(!PIR3bits.SSP1IF);  // Wait for the SSPIF bit to return high
}
```

Figure 2 Our Start routine.

Things are now getting considerably simpler. Here we first make sure the interrupt flag is cleared. We do not use interrupts, but this flag will be set when the start sequence is finished. We then initiate a Start condition (SEN = 1;) and we wait for the interrupt flag to return to 1, indicating the action is complete.

2.3 The Stop routine.

```c
void Send_I2C_Stop(void)
{
    PIR3bits.SSP1IF = 0;      // clear interrupt flag
    SSP1CON2bits.PEN = 1;     // Initiate the Stop condition
    while(!PIR3bits.SSP1IF);  // Wait for the SSPIF bit to return high
}
```

Figure 3 Our Stop routine.

The same comment as above applies, but here we initiate a Stop condition (PEN = 1).

2.4 The Restart routine.

```c
void Send_I2C_R_Start(void)
{
    PIR3bits.SSP1IF = 0;      // clear interrupt flag
    SSP1CON2bits.RSEN = 1;    // initiate restart condition
    while(!PIR3bits.SSP1IF);  // Wait for the SSPIF bit to return high
}
```

Figure 4 The Restart routine.

Not used very often, it makes use of the RSEN bit.

2.5 The Read routine.

```c
char Read_I2C (char ack) // 1 = ACK, 0 = NAK
{
    PIR3bits.SSP1IF = 0;         // clear interrupt flag
    SSP1CON2bits.RCEN = 1;       // set the receive enable bit to initiate a read
    while(!PIR3bits.SSP1IF);     // Wait for interrupt flag to return high
    if (ack){SSP1CON2bits.ACKDT = 0;} else {SSP1CON2bits.ACKDT = 1;}
    PIR3bits.SSP1IF = 0;         // clear interrupt flag
    SSP1CON2bits.ACKEN = 1;      // send acknowledge sequence
    while (!PIR3bits.SSP1IF) ;   // wait for a flag to be set
    return (SSP1BUF);            // Received data is now in the SSP1BUF
}
```

Figure 5 Reading 1 byte from the slave terminating either with a ACK or a NACK.

Reading a byte from a slave devices may either need to be terminated by a ACK (acknowledge) or a NACK (non-acknowledge). This routine does either by calling it with the `ack` parameter at 1 or 0.

As previously we first clear the SSP1IF interrupt flag and we initiate the byte transfer (`RCEN = 1`).

We then wait for the transfer to be complete (SSP1IF returns high). At this stage we set the ACKDT bit to either send an ACK or a NACK, and after resetting the interrupt flag, we initiate the ACK/NACK operation. When this is complete (SSP1IF returns high), we simply return the content of the receive buffer.

2.6 The Write routine.

```
void Write_I2C (char data)
{
    PIR3bits.SSP1IF = 0;          // clear interrupt flag
    SSP1BUF = data;               // Transmit begins as soon as SSP1BUF is written
    while (!PIR3bits.SSP1IF);     // Wait for interrupt flag to return high
}
```

Figure 6 The write routine

Finally our I2C write routine. Writing the data to transmit in the SSP1BUF initiate the transfer, which is complete once the interrupt flag returns to high.

These are all the building block we need to write our I2C routines.

Let's now look at the required routines for a few devices below, to demonstrate the use of these blocks.

3 EEPROM ACCESS THE 24AA02

Microchip has a range of serial access EEPROMs. Here is one of the simpler one with only a 8-bit address: the 24AA02.
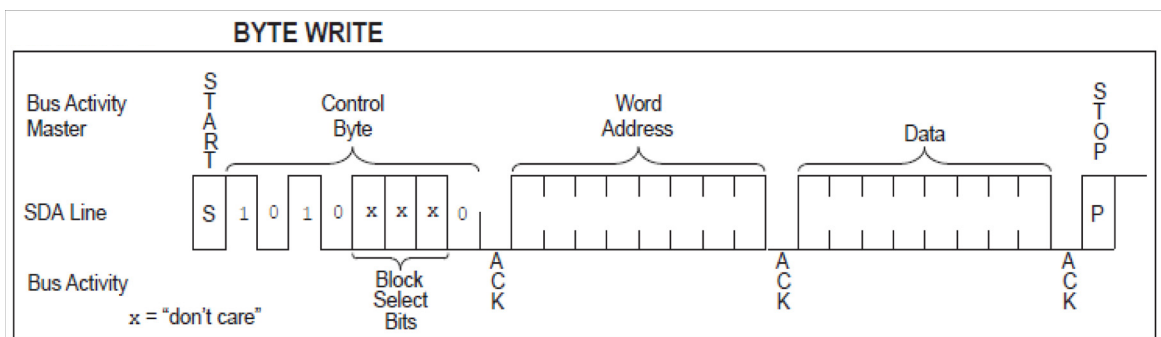
3.1 Accessing the 24AA02: byte write.

Figure 7 Byte write sequence for the 24AA02 EEPROM.

Setting up our write access routine consists in assembling our building blocks according to this figure.

The single byte write sequence requires: Start, EEPROM address, with 3 don't care address bits (x) set at zero. Then a 1 byte of memory address, data transfer and Stop. This is shown below.

```c
void EEPROM_write (char addr, char mydata)
{
    Send_I2C_Start();
    Write_I2C(EEP | WRITE);
    Write_I2C(addr);
    Write_I2C(mydata);
    Send_I2C_Stop();
}
```

Figure 8 Single byte write to the 24AA02.

The following #defines are needed:

```c
// EEP (I2C) definitions
#define EEP   0xa0  // EEPROM address MSB justified
#define READ  1
#define WRITE 0
```

Note that since this device does not use NACK, it is not defined nor used in these routines, we'll see an example of NACK usage later.

Following are a few considerations on the device address.

An I2C address normally consists of 7 bits plus 1 for R/$\overline{\text{W}}$. There are 2 ways to represent this address, left justified or right justified. Microchip calls it the control byte and represents it as shown below left. In this design, the chip select bits are all at zero.
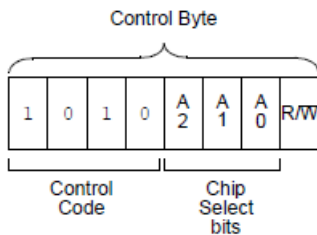
disregarding the R/$\overline{\text{W}}$ bit, this address can be written as:

0 1 0 1 0 0 0 0 = 0x50

1 0 1 0 0 0 0 x = 0xA0 for writing
and 0xA1 for reading

It is then shifted left by 1 and OR'd with the R/$\overline{\text{W}}$ bit.

Figure 9 Two representations of the chip address

The representation at right, seen in some documentation is both misleading and needlessly complex, requiring a shift operation and an OR operation to obtain the actual byte to be sent to the device. The one on the left is simpler, requiring only an OR operation to obtain the final address. This is the one used in this document. Here EEP is declared as 0xA0 and OR'd with either WRITE (which is declared as 0) or READ (which is declared as 1).

3.2 Accessing the 24AA02: page write.

In this device the pages are relatively small, being 8 bytes in size. Looking at the data sheet required sequence for page write, we can easily write the necessary routine.
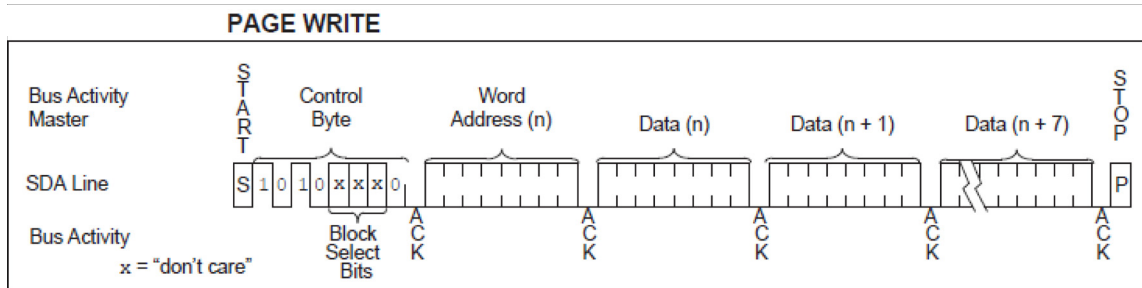
Figure 10 Page write sequence for the 24AA02 EEPROM.

```c
void EEPROM_write_p (char addr, char size, char a[])
{
char myi;
    Send_I2C_Start();
    Write_I2C(EEP | WRITE);
    Write_I2C(addr);
    for (myi = 0; myi < size; myi++){
    Write_I2C(a[myi]);  // address auto increments
    }
    Send_I2C_Stop();
}
```

Figure 11 Page write routine for the 24AA02

The first parameter passed to the routine is the base address of the page. Next is the size of the page to be written, and finally a buffer containing the data to be written.

3.3 Polling

Another important routine is needed to complete our description of the write operation. The data sheet mentions that the device is not available during the write cycle which can last up to 5 ms. In order to prevent writing during this interval it is desirable to wait this time out. One way to do this, outlined in the data sheet, is to poll the device until it answers. This is shown in the routine below which only returns once the EEPROM is again available.

```c
void EEPROM_poll (void)
{
  do { Send_I2C_Start();
       Write_I2C(EEP | WRITE);
     } while (SSP1CON2bits.ACKSTAT);  // wait for ack
  Send_I2C_Stop();                    // release SDA
}
```

Figure 12 The polling routine to time out the write operation

3.4 Accessing the 24AA02: byte read.

This sequence is a bit more elaborate, but once again, following the data sheet and assembling the required blocs is easy.

After a Start, we send the address with the WRITE bit set, then the address to be read. After that we send the again the address but with the READ bit set, and we receive the data. The operation is terminated by a Stop.

This is shown in the routine below. Here we need to use a NACK defined as follows:
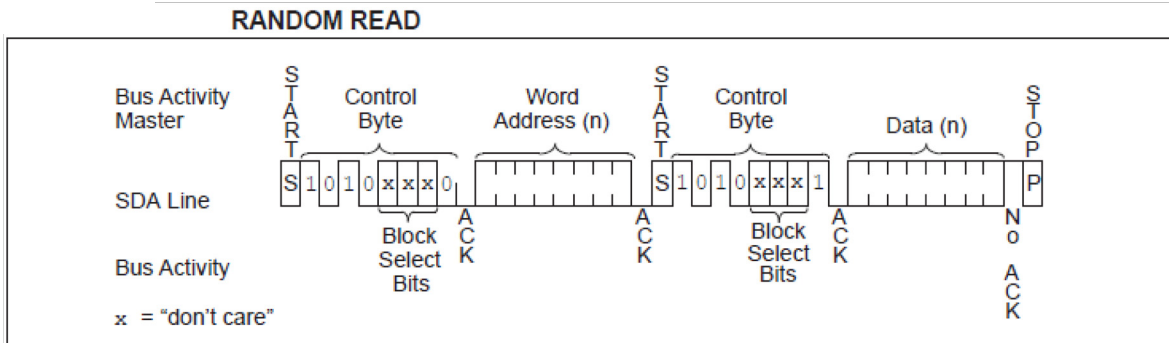
```
#define ACK 1
#define NAK 0
```


Figure 13 Byte read sequence for the 24AA02 EEPROM.

```c
char EEPROM_read (char addr)
{
  char mydata;
    Send_I2C_Start();
    Write_I2C(EEP | WRITE);
    Write_I2C(addr);
    Send_I2C_Start();
    Write_I2C(EEP | READ);
    mydata = Read_I2C(NAK);
    Send_I2C_Stop();
    return mydata;
}
```

Figure 14 Single byte read for the 24AA02.

3.5 Accessing the 24AA02: sequential read.

For reading there is no page size limitation, thus the entire EEPROM can be read in one operation if required. It starts the same as a single byte read, but will keep on sending the next byte (from the next address) as long as a ACK is received. As per the single byte read, the sequence terminates when a NACK is received by the EEPROM.
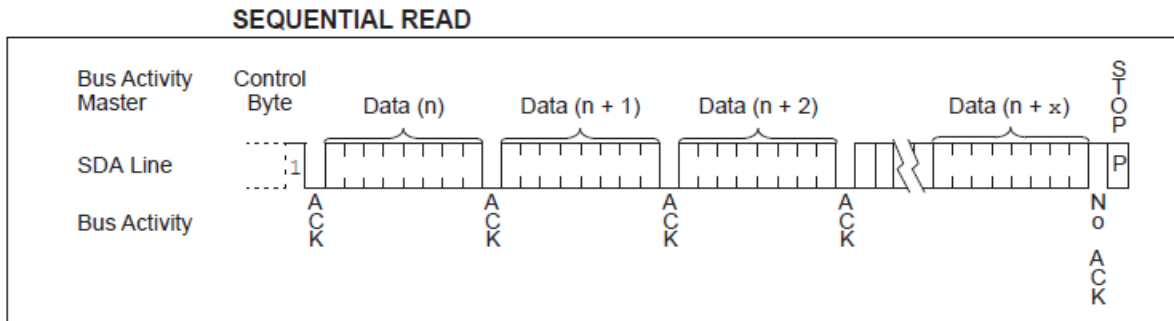

Figure 15 Sequential read sequence for the 24AA02 EEPROM.

Here is the routine to accomplish this.

```
void EEPROM_read_p (char addr, char size, char a[])
{
 int myi;
    size -=1;            // read 0 to size -1 with ACK
    Send_I2C_Start();
    Write_I2C(EEP | WRITE);
    Write_I2C(addr);
    Send_I2C_Start();
    Write_I2C(EEP | READ);
    for (myi = 0; myi < size; myi++){
        a[myi] = Read_I2C(ACK);     // read 0 to size -1 with ACK
        }
    a[size] = Read_I2C(NAK);     // read last byte with NAK
    Send_I2C_Stop();
}
```

Figure 16 Sequential read sequence routine for the 24AA02.

4   EEPROM ACCESS, THE 24AA64

This is a rather large device that requires a 16 bit address and that has 64 bytes page writes. While we'll show the full working routines, we will not describe them is as many details as above.

4.1   Writing to the EEPROM

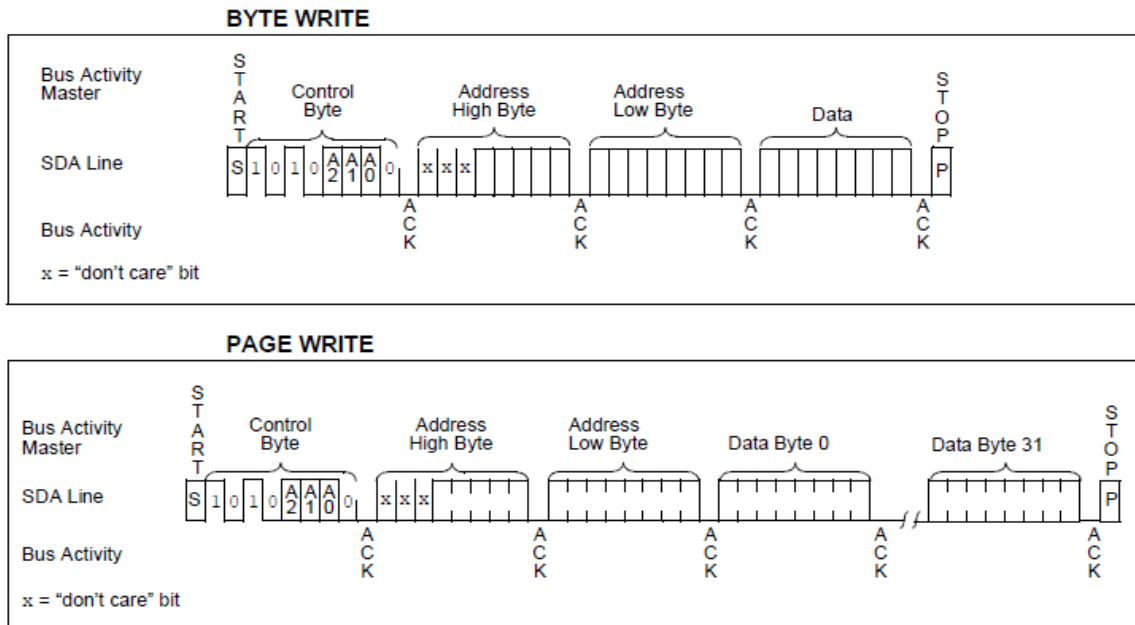The following is the write sequence as shown in the 24AA64 data sheet.

Figure 17 The write sequences for the 24AA64 EEPROM

Setting up our write access routine consists in assembling our building blocks according to this figure.

Single byte write: Start, EEPROM address, 2 bytes of memory address, data transfer and Stop. This is shown below.

```c
void EEPROM_write (int addr,  char mydata)
{
 char msb, lsb;
    lsb = addr & 0x00ff;
    msb = addr >> 8;
    Send_I2C_Start();
    Write_I2C(EEP | WRITE);
    Write_I2C(msb);     //Hi address
    Write_I2C(lsb);     //L0 address
    Write_I2C(mydata);  //data
    Send_I2C_Stop();
}
```

Figure 18 Single byte write to the EEPROM

The call to the routine passes the address of the memory array as an INT and the byte to be written as a CHAR.

So in this routine, we first split the 16 bit address into 2 bytes. Then send a Start, then the chip address (control byte) with the WRITE bit set to 0, then the high address, then the low address, then the data, then a Stop, exactly as described in the data sheet sequence reproduces on Figure 17 top.

Below is the routine for page write. Again it follows exactly the sequence represented in Figure 17 bottom. Here we pass the base address of the page to be written to the routine, and the size (number of bytes) of the desired transfer. We also pass a table of the bytes to be written (a[ ]). Note that in C, the table is not passed by value as for the address and size, but is passed by reference. This is not important here, but will be in the case of the page read shown below. So we first split the 16 bit page address in 2 bytes, and following the Figure 17 bottom, we send a Start, the chip address in Write mode, the high page address, the low page address, and now we send the buffer one byte at a time terminated by a Stop.

```c
void EEPROM_write_p (int addr, char size, char a[])
{
 char msb, lsb, myi;
    lsb = addr & 0x00ff;
    msb = addr >> 8;
    Send_I2C_Start();
    Write_I2C(EEP | WRITE);
    Write_I2C(msb);     // Hi address
    Write_I2C(lsb);     // L0 address
    for (myi = 0; myi < size; myi++){
        Write_I2C(a[myi]);  // address auto increments
        }
    Send_I2C_Stop();
}
```

Figure 19 The page write routine

4.2   Reading from the EEPROM

We once again turn to the data sheet to extract the following sequences.
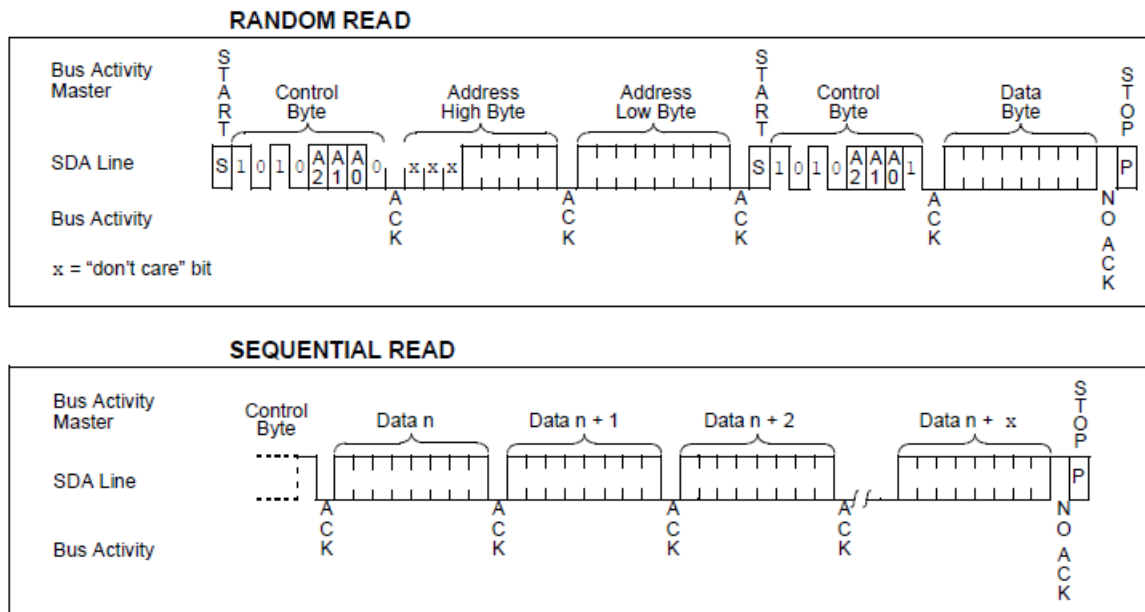

Figure 20 The read sequences for the 24AA64 EEPROM

Let's first look at reading 1 byte from the EEPROM - top figure. The sequence is quite straightforward, except that we need a Start in the middle and a NACK near the end. This is provided by the routine below.

```
char EEPROM_read (int addr)
{
 char msb, lsb, mydata;
    lsb = addr & 0x00ff;
    msb = addr >> 8;
    Send_I2C_Start();
    Write_I2C(EEP | WRITE);
    Write_I2C(msb);          // Hi address
    Write_I2C(lsb);          // Lo address
    Send_I2C_Start();
    Write_I2C(EEP | READ);
    mydata = Read_I2C(NAK);
    Send_I2C_Stop();
    return mydata;
}
```

Figure 21 Single byte reading from the EEPROM

The routine will return the byte read at the address provided.

As before, we split the address into 2 bytes, and send a Start. Then the chip address with the WRITE bit set to 0, then the 2 byte address. We follow with a Start bit and the chip address in READ mode. We are then instructed to read the byte from the EEPROM with a NACK, which we do, followed by a Stop. This conforms to the requirements outlined on Figure 20 top.

Now let's look at page read. We pass by value the address of the page to be read, the number of bytes to be read, and a pointer to a table where to store the data.

In C, when passing a table to a function (subroutine), it is passed by reference, meaning the routine gets a pointer to the table. This has the implication that the subroutine is able to modify the content of the table which is here exactly what we want to store the data read.

The sequence start as for byte read, by splitting the address in 2 bytes. Now, looking at the sequence on Figure 20 bottom, we notice that the treatment for the last byte is different, it needs a NACK while all the others need an ACK, so we'll decrease size by 1, to use it as a "ACK bytes" counter.

```c
void EEPROM_read_p (int addr, char size, char a[])
{
 char msb, lsb, myi;
    lsb = addr & 0x00ff;
    msb = addr >> 8;
    size -=1;              // read 0 to size -1 with ACK
    Send_I2C_Start();
    Write_I2C(EEP | WRITE);
    Write_I2C(msb);       // Hi address
    Write_I2C(lsb);       // L0 address
    Send_I2C_Start();
    Write_I2C(EEP | READ);
    for (myi = 0; myi < size; myi++){
        a[myi] = Read_I2C(ACK);     // read 0 to size -1 with ACK
        }
    a[size] = Read_I2C(NAK);    // read last byte with NAK
    Send_I2C_Stop();
}
```

Figure 22 Page read from the EEPROM

This is also convenient to store the last byte since when we say our transmission needs 32 bytes (for instance) they are numbered 0 to 31. So after the usual chip address, base page address transmission, we start reading (size − 1) bytes that we store into the table we are pointing to.

This being completed, we read the last byte, with a NAK flag and store it in the last place in the table. We can now send Stop to complete the whole transaction.

5 TEMPERATURE SENSOR

Here is another example, controlling a MCP9808 temperature sensor[1]. This devices uses (mostly) 16 bit transfers.

All transfers are done by first writing to an 8-bit pointer register (only 4 bits used), then the 16 bit register pointed to is either written or read.

Let's first look at reading a register.

---

1. Note that the data sheet for this device provides C-code examples of I2C routines that are very close to the ones described here.

5.1 Reading routine.

Here is the sequence as per the data sheet for the 16-bit CONFIG register. This register is accessed with the pointer at 0x01.
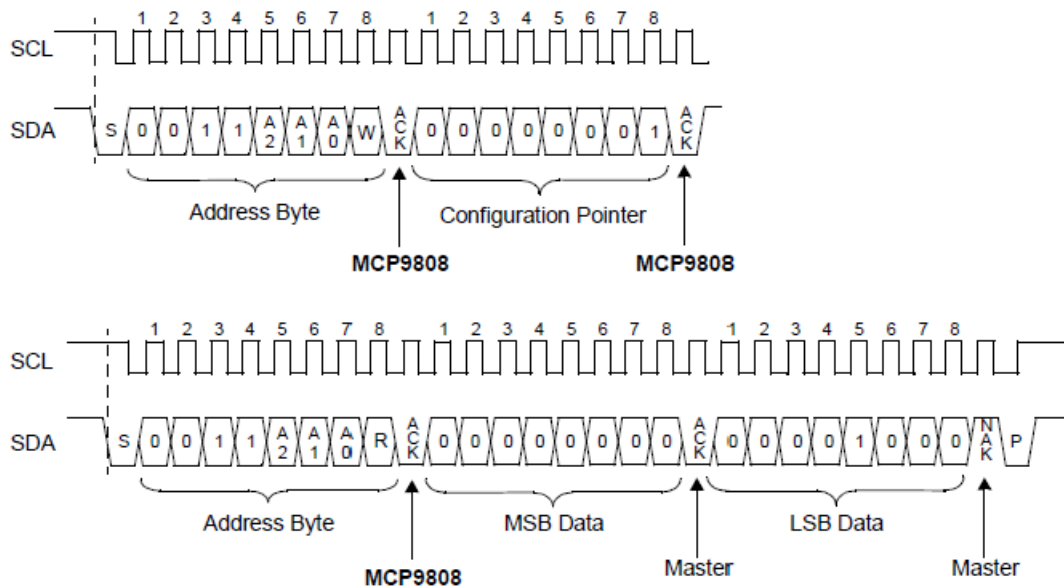

Figure 23 Reading the 16 bit CONFIG register

We must first send a Start followed by the chip address with the WRITE bit set, then the internal 8-bit register pointer, then again after a Start, the chip address, in READ mode, and we now accept 2 bytes, the first one with an ACK, the second one with a NACK.

```
int MCP9808_read (char reg)
{
 int mydata;
    Send_I2C_Start();
    Write_I2C(THR | WRITE);
    Write_I2C(reg);                 // send pointer
    Send_I2C_Start();
    Write_I2C(THR | READ);
    mydata = Read_I2C(ACK);         // MSB with ACK
    mydata = (mydata <<= 8) | Read_I2C(NAK);  // LSB with NACK
    Send_I2C_Stop();
    return mydata;
}
```

Figure 24 Reading from the temperature sensor.

The routine returns the 16 bit value just read.

5.2 Writing routine

We can now look at a write routine.

Let's look at writing the CONFIG register, using the pointer at 0x01.

Writing to the CONFIG Register to Enable the Event Output Pin <0000 0000 0000 1000>b:
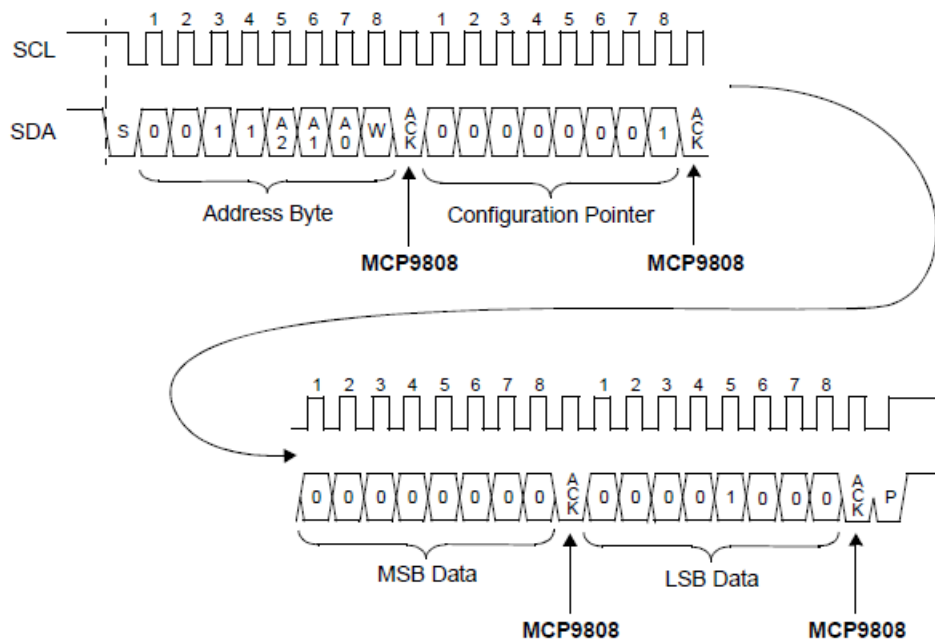

Figure 25 Writing 16-bit data

As before we must first transmit an 8-bit register pointer before writing the 16 bit data.

```
void MPC9808_write (char reg, int mydata)
{
  char msb, lsb;
    lsb = mydata & 0x00ff;
    msb = mydata >> 8;
    Send_I2C_Start();
    Write_I2C(THR | WRITE);
    Write_I2C(reg);         // send pointer
    Write_I2C(msb);         //MSB
    Write_I2C(lsb);         //LSB
    Send_I2C_Stop();
}
```

Figure 26 Routine to write 16 bit data

At this stage, this is self explanatory.

6   CONCLUSION

This document has described I2C routines, based on simple building blocks that are immediately usable for most simple (standard) cases. They assume a single master, one or more peripherals (slave devices), and do not make use of interrupts.

Olivier PILLOUD - Nov. 2019 - Apr 2021.