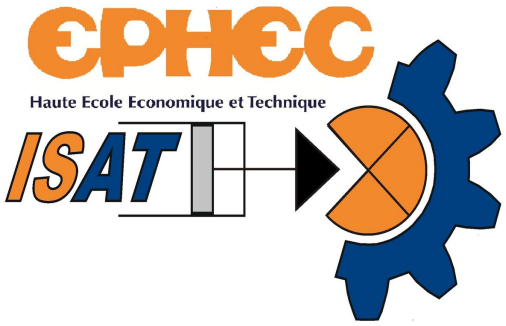


UE A211 : SYSTÈMES EMBARQUÉS I

Professeur : COSTA EMILE

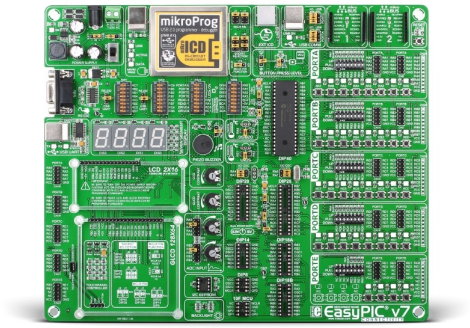
RAPPORT DE LABORATOIRE

Etudiants : KORKUT CANER

TP5 A - Le bus I^2C

Table des matières

1 Objectif du TP



Lors de cette séance, nous avons été amenés à manipuler le mode de communication I^2C. Pour ce faire nous avons introduit les concepts de base qui composent cette norme avant de réaliser un projet associé à cette dérnière.

On nous demande de : réaliser un programme permettant d'envoyer les valeurs de l'ADC dans une E^2PROM[1] et d'ensuite y accéder par le biais du bus I^2C. L'envoi des données du μC vers l'E^2PROM se fait suite à l'appui sur le *Bouton 1*, tandis que le *Bouton 2* permet d'interrompre cet envoi. L'envoi des données pourra être interrompu à tout moment par ce dernier.

Un troisième bouton *Bouton 3* permet quant à lui de charger les valeurs présentes dans l'E^2PROM et ce depuis l'adresse 0 à la dernière valeur enregistrée. Pendant l'opération de chargement, on nous demande également d'envoyer la trame reçue vers un afficheur LCD **ET** vers une interface de communication *UART*. A noter que l'envoi et la lecture depuis l'E^2PROM ne peuvent aller que jusqu'à l'adresse 100 ou encore $0x64$. Pendant toutes ces opérations, le mode dans lequel nous nous trouvons devra être spécifié de la sorte sur le LCD :

- **Attente**
- **Start**
- **Stop**
- **Dumping**

1. **NM24C08 :** Il s'agit d'une mémoire non volatile avec bus de communication I^2C

2 Introduction

Comme dans tout protocole structuré, la communication I^2C comprend des définitions des niveaux électriques haut et bas, des conditions de changement d'états, des conditions de départ START et d'arrêt STOP, des procédures d'acquittement, d'arbitrage, etc.

Dans cette section nous reverrons les fondements du mode de communication I^2C sans réellement entrer dans les détails. Nous présenterons brièvement le protocole de communication et les spécifités matérielles. Dans la suite du travail, nous analyserons également une trame I^2C afin de faire le lien entre la théorie et la pratique et de nous initaliser à la notion de *debugging*.

Un peu de théorie

Le mode de communication I^2C a été mis en place pour la première fois par *Philips Semiconductors*. Il s'agit d'un moyen permettant de relier différents circuits à un microprocesseur par le biais seuelement de 3 fils : **SDA**, **SCL**, et **GND**.

Cactéristiques *applicatives* du mode I^2C

Ce mode de communication possède 3 modes de fonctionnement : le **standard-mode** ($100kbps$), le **fast-mode** ($400kbps$), et **high-speed mode** ($3,2Mbps$). Nous nous focaliserons ici sur le *standard-mode* qu'on a utilisé. En ce qui concerne les spécificités de la couche *applicative* de cette communication, on peut noter les éléments suivants :

- **Une** adresse par périphérique
- Bus multi-maître : plus d'un maître peut tenter de commander le bus.
- Bus série (8bits) bidirectionnel
- Deux lignes de communication SDA et SCL + une masse.

Cactéristiques du protocole I^2C

Principe : Au repos : les lignes SDA et SCL sont au niveau **haut**. On ne peut les forcer qu'au niveau bas. Ainsi, lorsque le maître souhaite transmettre

un message, voici les opérations qui ont lieu :

1. Il transmet le bit de poids fort $D7$ sur SDA
2. Il valide la donnée en appliquant un niveau **haut** sur SCL
3. Lorsque SCL retombe au niveau bas, il poursuit avec $D6$, jusqu'au dernier octet
4. Il envoie le bit ACK en scrutant l'état réel de SDA
5. L'esclave impose un état bas pour signaler que la transmission s'est déroulée correctement
6. Il (le maître) voit le niveau bas et peut passer à la suite

3 Schéma de principe

Le schéma de principe ainsi que la simulation ont été réalisés sur Proteus. Voici les composants utilisés :

- PIC18F45K22 ;
- 3 Boutons poussoir avec résistances 1kΩ ;
- 1 Potentiomètre 1kΩ ;
- 1 LCD 16x2 ;
- Alimentation 5V ;
- Masse ;
- 1 Debugger I^2C ;
- 1 COMPIM (RS232) ;
- 1 E^2PROM NM24C08.

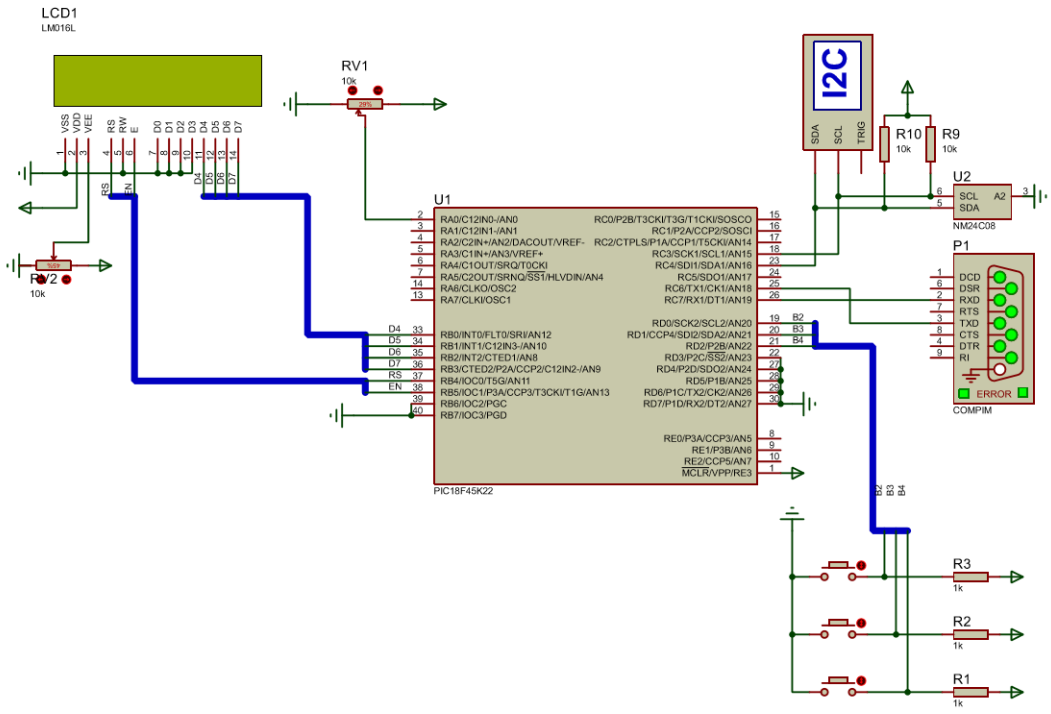


FIGURE 1 – Schéma de principe sur Proteus

4 Algorigramme

Toujours en partant du travail précédent, nous avons restructuré notre algorigramme afin qu'il réponde au programme à coder. Afin de ne pas surcharger le rapport, nous ne présenterons ici que les nouvelles fonctions créées.

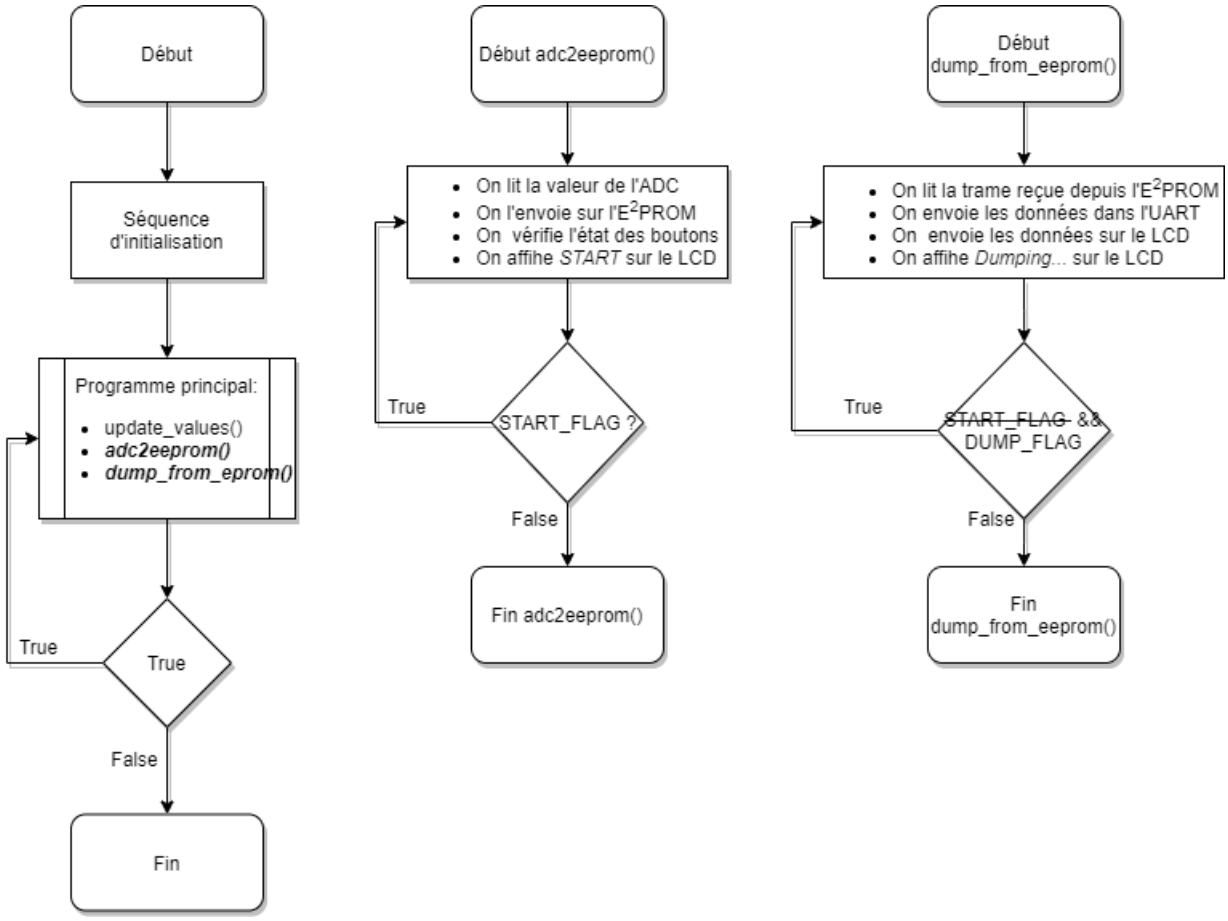


FIGURE 2 – Algorigramme du programme principal

5 Code source

```
// Constantes
#define EEPROM_WR        0xA0
#define EEPROM_RD        0xA1
#define SOT 2
#define EOT 3
#define LF 10
#define CR 13

// Connexions du module LCD
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// Fin des connexions du module LCD

unsigned short index = 0 ;
unsigned short last_index = 0;
unsigned short dump_index = 1;
// Lecture de l'etat des boutons

bit old_state_1;
bit old_state_2;
bit old_state_3;
bit start_flag;
bit dump_flag;

char content_adc[16];
char pot_val;

// Fonction d'ecriture d'un octet dans l'Eeprom
void WriteToEEprom(unsigned short Address, unsigned short Data) {
    I2C1_Start();
```

```
42      I2C1_Wr(EEPROM_WR);
43      I2C1_Wr(Address);
44      I2C1_Wr(Data);
45      I2C1_Stop();
46  }
47
48  // Fonction de lecture d'un octet de l'Eeprom
49  unsigned short ReadFromEEprom(unsigned short Address) {
50    unsigned short temp;
51    I2C1_Start();
52    I2C1_Wr(EEPROM_WR);
53    I2C1_Wr(Address);
54    I2C1_Repeated_Start();
55    I2C1_Wr(EEPROM_RD);
56    temp = I2C1_Rd(0u);
57    I2C1_Stop();
58    return temp;
59  }
60
61  void update_values(){
62      if (Button(&PORTD, 0, 1, 1)){
63        old_state_1 = 1;
64      }
65        if (Button(&PORTD, 0, 1, 0 ) && old_state_1) {
66          start_flag = 1;
67          old_state_1 = 0;
68        }
69
70      if (Button(&PORTD, 1, 1, 1)){
71        old_state_2 = 1;
72      }
73        if (Button(&PORTD, 1, 1, 0 ) && old_state_2) {
74          start_flag = 0;
75          LCD_Out(2,6,"Stop        ");
76          old_state_2 = 0;
77      }
78
79      if (Button(&PORTD, 2, 1, 1)){
80        old_state_3 = 1;
81      }
82        if (Button(&PORTD, 2, 1, 0 ) && old_state_3) {
83          dump_flag = 1;
84          old_state_3 = 0;
```

```
85      }
86  }
87
88  /*Envoi de l'ADC dans l'E2PROM
89  selon la sequence demandee*/
90  void adc2eeprom(){
91     unsigned int adc;
92     while(start_flag && index <= 100 && (last_index!=index)){
93      adc = ADC_Read(0);
94      WriteToEEprom(index, adc);
95      Delay_10ms();
96      update_values();
97      LCD_Out(2,6,"Start          ");
98      last_index = index ;
99     }
100 }
101
102 // Envoi des donnes dans l'UART1
103 void send_values(char val_to_send){
104     UART1_Write(SOT);
105     UART1_Write_Text(val_to_send);
106     UART1_Write(EOT);
107     UART1_Write(LF);
108     UART1_Write(CR);
109 }
110
111 // Reinitalise l'adresse de chargement de l'E2PROM
112 void reset_if_last_pos(){
113       if (dump_index == index){
114          dump_flag = 0;
115          dump_index = 0;
116       }
117       dump_index++;
118 }
119
120 /*Charge les donnees depuis l'EEPROM
121  selon la s quence demand e*/
122 void dump_from_eeprom(){
123     while (dump_flag && !start_flag){
124       char content_eeprom[16];
125       unsigned int eeprom_val;
126       eeprom_val = ReadFromEEprom(dump_index);
127       sprintf(content_eeprom,"Data: %u",eeprom_val);
```

```
128         send_values(content_eeprom);
129         LCD_Out(1,1,content_eeprom);
130         LCD_Out(2,6,"Dumping...     ");
131         reset_if_last_pos();
132       }
133 }
134
135 // Routine d'interruption
136 void Interrupt(){
137   if (TMR0IF_bit){
138     TMR0IF_bit = 0;
139     TMR0H  = 0x0B;
140     TMR0L  = 0xDC;
141     if (start_flag){
142        index++;
143     }
144   }
145 }
146
147 // Message d'initialisation du LCD
148 void LCD_init_message(){
149   Lcd_Cmd(_LCD_CLEAR);
150   Lcd_Cmd(_LCD_CURSOR_OFF);
151   Lcd_Out(1,1,"Initialisation...");
152   Delay_ms(1000);
153   Lcd_Cmd(_LCD_CLEAR);
154   Lcd_Out(2,1,"Mode: Attente");
155 }
156
157 // Sequence d'initalisation
158 void init(){
159   // Selection des ports analogiques
160   ANSELA  = 0b00000001;
161   ANSELB = 0;
162   ANSELD  = 0;
163   ANSELC = 0;
164
165   // Desactive les comparateurs
166   C1ON_bit = 0;
167   C2ON_bit = 0;
168
169   // Initialise les entrees et sorties
170   TRISD = 0b00000111;
```

```
171   TRISA = 0b00000001;
172   TRISB = 0;
173
174   // Initialise les flag
175   start_flag = 0;
176   dump_flag = 0;
177   old_state_1 = 0;
178   old_state_2 = 0;
179
180   // Timer0 interruption
181   T0CON  = 0x85;
182   TMR0H  = 0x0B;
183   TMR0L  = 0xDC;
184   GIE_bit  = 1;
185   TMR0IE_bit   = 1;
186
187   /*Initialise les objets lies
188   au bibliotheques utilisees*/
189   ADC_Init();
190   UART1_Init(9600);
191   Lcd_Init();
192   I2C1_Init(100000);  // initalise le I2C
193   Delay_100ms();
194   LCD_init_message();
195 }
196
197 //Fonction principale
198 void main(){
199     init();
200   //Programme principal
201   for(;;){
202     update_values();
203     adc2eeprom();
204     dump_from_eeprom();
205   }
206 }
```

6 Analyse du projet

6.1 Analyse du code source

Cette section sera consacrée à l'analyse du code et plus précisément aux fonctions void `adc2eeprom` et void `dump_from_eeprom()` puisque ce sont majoritairement elles qui sont venues s'ajouter au programme précédent.

```
50 /*Envoi de l'ADC dans l'E2PROM
51 selon la sequence demandee*/
52 void adc2eeprom(){
53    unsigned int adc;
54    while(start_flag && index <= 100 && (last_index!=index)){
55     adc = ADC_Read(0);
56     WriteToEEprom(index, adc);
57     Delay_10ms();
58     update_values();
59     LCD_Out(2,6,"Start          ");
60     last_index = index ;
61    }
62 }
```

On peut remarquer que la sous-routine associée à cette fonction est exécutée si le flag `start_flag` est à l'état haut, et ce jusqu'à ce que `index` ait pour valeur 100. La condition `last_index ! = index` permet à l'interruption de prendre la main sur le déroulement normal du programme. Une fois dans la sous-routine, on lit successivement la valeur de l'*ADC*, on l'envoie directement vers l'E^2PROM à l'aide de la fonction `WriteToEEprom(index, adc)`. Cette dernière sera expliquée plus bas. Après avoir effectué cette opération on met à jour l'état des flag à l'aide de la fonction `update_values()`. On affiche ensuite le message START avant d'assigner la valeur actuelle de `index` à `last_index`.

On peut maintenant analyser la fonction void `dump_from_eeprom()` :

```
120 /*Charge les donnees depuis l'EEPROM
121  selon la s quence demand e*/
122 void dump_from_eeprom(){
123     while (dump_flag && !start_flag){
124       char content_eeprom[16];
125       unsigned int eeprom_val;
126       eeprom_val = ReadFromEEprom(dump_index);
127       sprintf(content_eeprom,"Data: %u",eeprom_val);
```

```
128         send_values(content_eeprom);
129         LCD_Out(1,1,content_eeprom);
130         LCD_Out(2,6,"Dumping...      ");
131         reset_if_last_pos();
132       }
133 }
```

La sous-routine associée est également effectuée suite à la mise à l'état haut des flags `dump_flag` et `start_flag`. On lit la donnée présente dans l'E^2PROM pour ensuite l'envoyer sur le LCD et l'UART comme demandé dans l'énoncé. Pendant cette opération, on affiche le message *Dumping...* sur le LCD et on met le flag `dump_flag` à l'état bas si ce dernier atteint la position de `index`, à savoir la dernière adresse utilisée dans l'E^2PROM.

6.2 Analyse des trames I^2C

Dans cette section nous analyserons les trames que nous nous attendons à rencontrer lors d'une opération de *sniffing*. Le logiciel PROTEUS disposant d'un **debugger** I^2C, nous avons pu l'exploiter pendant cette opération :

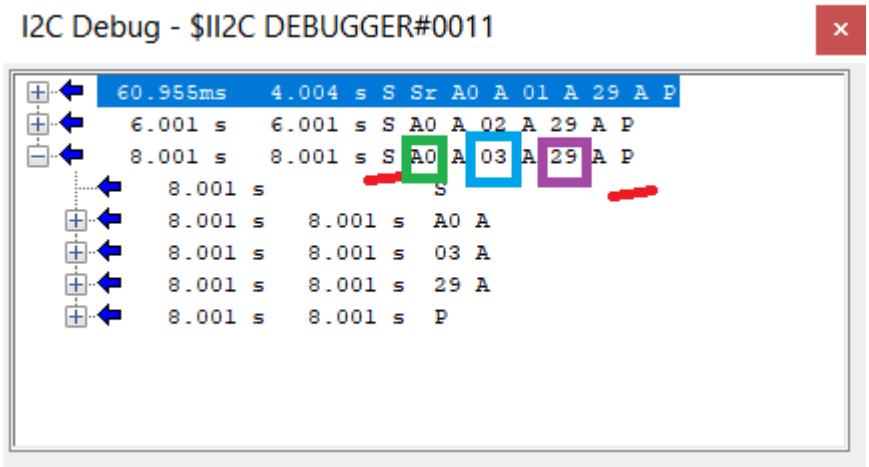


FIGURE 3 – Debugger I^2C de PROTEUS

Il est trivial de faire une analogie directe entre les trames de la figure 3 et notre programme. On voit le présence de bits de START et STOP suivis de 3 données hexadécimales de 8 bits. Voyons de plus près ces valeurs :

— **A0** : signifie qu'on souhaite *écrire* une donnée [2]
— **03** : signifie qu'on souhaite manipuler la $3^{ème}$ *adresse*
— **29** : indique la donnée à écrire

Ces valeurs sont toute suivies du flag *ACK* qui permet, comme expliqué dans l' et dans la datasheet de l'annexe A, d'acquitter l'envoi de la trame par le maître pour que cette dernière puisse être traitée par le récepteur. Pour une question de clareté, nous avons utilisé un code couleur et repris le programme ci-dessous :

```
void WriteToEEprom(unsigned short Address, unsigned short Data) {
    I2C1_Start();
    I2C1_Wr(EEPROM_WR);
    I2C1_Wr(Address);
    I2C1_Wr(Data);
    I2C1_Stop();
}
```

FIGURE 4 – Encapsulation des données selon le protocole I^2C

7 Conclusion

Cette séance nous a permis de nous familiariser avec le mode de communication I^2C depuis la couche phyisque à la couche applicative. Nous y avons mis en place ce mode de communication, analysé les trames, envoyé et reçu des données et enfin, nous avons traité les données acquises avant de les communiquer à l'opérateur par l'intermédiaire d'un écran LCD.

Nous retenons de cette séance qu'il est indispensable de connaître le mode de fonctionnement du protocole utilisé lors d'une communication. En effet, ayant précédemment mis en oeuvre une communication $UART$, nous constatons que les mécanismes ne sont pas les mêmes au niveau de leurs caractéristiques et leur mise en oeuvre.

2. Voir annexe B

A Datasheet *PIC18F45K22*

15.6 I²C Master Mode

Master mode is enabled by setting and clearing the appropriate SSPxM bits in the SSPxCON1 register and by setting the SSPxEN bit. In Master mode, the SCLx and SDAx lines are set as inputs and are manipulated by the MSSPx hardware.

Master mode of operation is supported by interrupt generation on the detection of the Start and Stop conditions. The Stop (P) and Start (S) bits are cleared from a Reset or when the MSSPx module is disabled. Control of the I²C bus may be taken when the P bit is set, or the bus is Idle.

In Firmware Controlled Master mode, user code conducts all I²C bus operations based on Start and Stop bit condition detection. Start and Stop condition detection is the only active circuitry in this mode. All other communication is done by the user software directly manipulating the SDAx and SCLx lines.

The following events will cause the SSPx Interrupt Flag bit, SSPxIF, to be set (SSPx interrupt, if enabled):

- Start condition detected
- Stop condition detected
- Data transfer byte transmitted/received
- Acknowledge transmitted/received
- Repeated Start generated

Note 1: The MSSPx module, when configured in I²C Master mode, does not allow queueing of events. For instance, the user is not allowed to initiate a Start condition and immediately write the SSPxBUF register to initiate transmission before the Start condition is complete. In this case, the SSPxBUF will not be written to and the WCOL bit will be set, indicating that a write to the SSPxBUF did not occur

2: When in Master mode, Start/Stop detection is masked and an interrupt is generated when the SEN/PEN bit is cleared and the generation is complete.

15.6.1 I²C MASTER MODE OPERATION

The master device generates all of the serial clock pulses and the Start and Stop conditions. A transfer is ended with a Stop condition or with a Repeated Start condition. Since the Repeated Start condition is also the beginning of the next serial transfer, the I²C bus will not be released.

In Master Transmitter mode, serial data is output through SDAx, while SCLx outputs the serial clock. The first byte transmitted contains the slave address of the receiving device (7 bits) and the Read/$\overline{\text{Write}}$ (R/$\overline{\text{W}}$) bit. In this case, the R/$\overline{\text{W}}$ bit will be logic '0'. Serial data is transmitted eight bits at a time. After each byte is transmitted, an Acknowledge bit is received. Start and Stop conditions are output to indicate the beginning and the end of a serial transfer.

In Master Receive mode, the first byte transmitted contains the slave address of the transmitting device (7 bits) and the R/$\overline{\text{W}}$ bit. In this case, the R/$\overline{\text{W}}$ bit will be logic '1'. Thus, the first byte transmitted is a 7-bit slave address followed by a '1' to indicate the receive bit. Serial data is received via SDAx, while SCLx outputs the serial clock. Serial data is received eight bits at a time. After each byte is received, an Acknowledge bit is transmitted. Start and Stop conditions indicate the beginning and end of transmission.

A Baud Rate Generator is used to set the clock frequency output on SCLx. See **Section 15.7 "Baud Rate Generator"** for more detail.

B Datasheet *NM24C08*

Device Operation Inputs (A0, A1, A2)

Device address pins A0, A1, and A2 are connected to V_{CC} or V_{SS} to configure the EEPROM chip address. Table 1 shows the active pins.

Table 1.

Device	A0	A1	A2	Effects of Addresses
NM24C08/09	x	x	ADR	2^1 = 2; 2*x (4x2K)** = 16K

* Max # of devices on bus ** Number of page blocks per density

The maximum density addressable using the three pin configuration of the I^2C protocol is 16K. Any combination of densities can be used up to this limit.

Background Information (I^2C Bus)

As mentioned, the I^2C bus allows synchronous bidirectional communication between Transmitter/Receiver using the SCL (clock) and SDA (Data I/O) lines. All communication must be started with a valid START condition, concluded with a STOP condition and acknowledged by the Receiver with an ACKNOWLEDGE condition.

As shown below, the EEPROMs on the I^2C bus may be configured in any manner required, the total memory addressed can not exceed 16k (16,384 bits). EEPROM memory address programming is controlled by 2 methods:

- Hardware configuring the A0, A1, and A2 pins (Device Address pins) with pull-up or pull-down to V_{CC} or V_{SS}. **All unused pins must be grounded** (tied to V_{SS}).
- Software addressing the required PAGE BLOCK within the device memory array (as sent in the Slave Address string).

For devices with densities greater than 16K, a different protocol, extended I^2C protocol, is used. Refer to NM24C32 datasheet (for example) for additional details.

Addressing an EEPROM memory location involves sending a command string with the following information: [DEVICE TYPE]–[DEVICE ADDRESS]–[PAGE BLOCK ADDRESS]–[BYTE ADDRESS]

DEFINITIONS	
WORD	8 bits (byte) of data
PAGE	16 sequential addresses (one byte each) that may be programmed during a 'Page Write' programming cycle
PAGE BLOCK	2048 (2K) bits organized into **16 pages of addressable memory. (8 bits) x (16 bytes) x (16 pages) = 2048 bits**
MASTER	Any I^2C device CONTROLLING the transfer of data (such as a

Pin Descriptions

Serial Clock (SCL)

The SCL input is used to clock all data into and out of the device.

Serial Data (SDA)

SDA is a bidirectional pin used to transfer data into and out of the device. It is an open drain output and may be wire–ORed with any number of open drain or open collector outputs.

WP Write Protection (NM24C09 Only)

If tied to V_{CC}, PROGRAM operations onto the upper half of the memory will not be executed. READ operations are possible. If tied to V_{SS}, normal operation is enabled, READ/WRITE over the entire memory is possible.

This feature allows the user to assign the upper half of the memory as ROM which can be protected against accidental programming. When write is disabled, slave address and word address will be acknowledged but data will not be acknowledged.

Device Operation

The NM24C08/09 supports a bidirectional bus oriented protocol. The protocol defines any device that sends data onto the bus as a transmitter and the receiving device as the receiver. The device controlling the transfer is the master and the device that is controlled is the slave. The master will always initiate data transfers and provide the clock for both transmit and receive operations. Therefore, the NM24C08/09 will be considered a slave in all applications.

Clock and Data Conventions

Data states on the SDA line can change only during SCL LOW. SDA state changes during SCL HIGH are reserved for indicating start and stop conditions. Refer to *Figure 2* and *Figure 3* on next page.

Start Condition

All commands are preceded by the start condition, which is a HIGH to LOW transition of SDA when SCL is HIGH. The NM24C08/09 continuously monitors the SDA and SCL lines for the start condition and will not respond to any command until this condition has been met.

Stop Condition

All communications are terminated by a stop condition, which is a LOW to HIGH transition of SDA when SCL is HIGH. The stop condition is also used by the NM24C08/09 to place the device in the standby power mode.

Write Cycle Timing

Acknowledge

C Bibliographie

— https://bit.ly/2wPhnOm
— https://bit.ly/2Jm0Y6H
— https://bit.ly/33WVUPD
— https://bit.ly/2UpYc70
— https://bit.ly/2JkXjGu
— https://bit.ly/341VMyE
— https://bit.ly/39q1v2k
— https://bit.ly/2xygVnX