J.D. Nicoud, Mouette 5
CH-1092 Belmont, Switzerland
Tel +41 21 728-6156, Fax 728-6157
Email info@didel.com, www.didel.com

DIDEL

This file: www.didel.com/doc/DopicI2C.pdf

I²C / SM-Bus PIC routines for PIC 16F84, 16F870, ..

Table of content

1. Introduction

I²C has been proposed by Philips as an efficient way of having a processor controlling a set of I/O devices over 2 signal lines. It is used inside PCs mostly for reading temperature sensors and power controller, and has been renamed by Intel as the SM-Bus (www.sbs-forum.org/smbus/specs/).

Multi-master transfers are possible, but our objective here is only to learn how to control existing I²C circuits, and program a PIC as a slave. We will not explain all the features of I²C Those having undersood this document will be ready to read Philips or SM-Bus documentation.

This note shows how to use I²C transfers on Microchip PIC processors which do not have dedicated hardware for this. From the examples I have seen, it is indeed easier to use our routines than to understand and program the I²C control registers of a 16F76 or 16F877. So, unless you have to be multitask and are ready to spend a lot of time learning to handle correctly a set of interrupts, consider using our simple routines.

We first explain the I2C timings and how to implement them with macros and routines. We show how to use them with two I²C circuits: the PCF8584 parallel port interface and the 24C01 EEPROM. A simple PIC as a slave (a low cost 12C508 for instance) adds some timing constraints and requires to slow down the transfer down to 10 kBit/s.

We use CALM assembly language notations. CALM (Common Assembly Language for Microprocessors) has been developed at the EPFL since 1976 and supports 15 processors with a simple explicit orthogonal syntax. CALM is close to Motorola notations, with more explicit addressing modes. For programmers with Intel or AVR experience, several differences (order of operands, mnemonics) make the transitions more difficult. For beginners, CALM has proven its efficiency, due to its explicit syntax: when Microchip writes btfsc Reg,bit one should remember that this mean "Test in register eg" the bit "bit" and Skip if this bit is set. CALM writes TestSkip,BS Port:#bit which means exacly this. CALM uses the # sign for immediate addressing instead if a "i" letter in the mnemonic. The number of instruction mnemonics is greatly reduced, the addressing mode being very explicit. Going from one processor to another is easy, since only the specificity of the architecure and a few special instructions have to be learned.

For those not familiar with CALM, but having some experience with the PIC, the reference card available at www.didel.com/doc/Pic84Calm.pdef lists the instructions in both

notations.

For those without experience with PIC, but with a good understanding on microcontrolers and assembly language, a detailed document in english is available at www.didel.com/doc/Pic84E.html. For absolute beginners, we have the **PIC***Génial* documentation in french (www.didel.com/PicGenial.html)

2. I²C principle and macros

When sending an 8-bit word in serial, one needs to know when the data starts and when it stops. The great idea of I²C is to use 2 lines, one for the clock (so there is no precise bit rate to select) and one for the data. Valid data must be stable when the clock is at level one. Start and stop bits result from a violation of that rule, easy to generate and decode. In order to simplify, we take all the long timings to 5 µs.

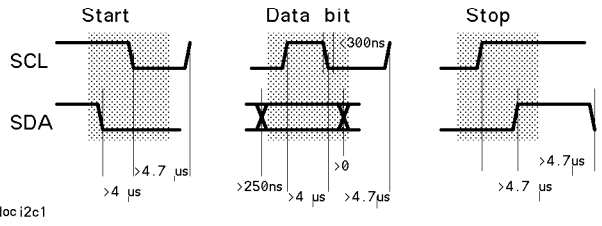


Fig. 1 I²C principle and timings

2.1. Delays

At 4 MHz, the SDA to SCL delay of 250 ns will result from two consecutive instructions for processors up to 16MHz. Short delays are generated by Nop instructions, which takes 1 µs at 4 MHz. The instruction Jump APC+1 takes 2 µs since it jumps to the next instruction (APC is the assembler program counter value), and takes more time due to the instruction pipeline being broken. The following macros generate the delays we will need.

```
.Macro  Nop2
        Jump    APC+1
.Endmacro
```

```
.Macro  Nop3
        Nop
        Jump    APC+1
.Endmacro
```

```
.Macro  Nop3
        Nop
        Jump    APC+1
.Endmacro
```

If you have a faster processor, the xDel macro and routine in appendix (www.didel.com/XDelM.asm and www.didel.com/XDelR.asm) is an elegant way to generate delays between 1 and 20 instruction cycles (or more if you add instructions in the XDly routine). The macro inserts a maximum of two instructions and the associated XDly routine is quite short and does not use any register or modify W.

2.2. Start and stop

Now we can control the delays, it is easy to generate the sequence (these are not the definitive macros, see later).

```
.Macro  Start
        ClrSDA  ; SDA ´´´\.....
        Nop4
        ClrSCK  ; SCK ´´´´´´´\.
        Nop2
.Endmacro
```

```
.Macro  FullSCK
        SetSCK  ; SCK .../´´´\..
        Nop4
        ClrSCK
.Endmacro
```

```
.Macro  Stop
        ClrSDA
        Nop
        SetSCK  ; SDA ....../´´
        Nop2
        SetSDA  ; SCK ../´´´´´´
.Endmacro
```

The function of ClrSDA and the other macros referred inside these macros is evident. How to implement them? The naive way would be to initialize two ports bits as output, and set or clear these bits. A special handling would be necessary for read transfer and acknowledge. I²C bus is specified as open-collector and we have to do it for the SDA line at least in order to avoid from time to time a short circuit between the master and slave SCL outputs. Open collector means that the zero are active, but not the ones; pull-ups will assert the one. This is easily implemented on portA for instance by initializing the port to zero, and playing on the direction to get an output at level zero and ones: When a port line is initialized as output, a zero is active. When the port line is initialized as input, the pull-up resistor sets a one (if no other output on the bus forces a zero).

2.3. Bit selection

Let us define bSCL and bSDA the bit number of the port, and PortI2C the port on which the two lines are wired (the same port is highly preferable). For instance, if SCL and SDA are connected on PortC bits 3 and 4 of the 16F870 (to be compatible with an hardware implementation on the 16F876), one should declare:

```
bSCK     = 3
bSDA     = 4
DirC     = 2´10011001     ; bits 3 and 4 are important
PortI2C  = 7      ; PortC
TrisI2C  = 7      ; Bank1  TrisC
```

2.4. Caution

Before any call of a I²C macro or routine, one should be sure that the PortI2C bits bSCL and bSDA are at zero. If there is no interaction with another I/O using the same port, the bits are initialized in the beginning and will stay, but pay attention with the Set bit and Clr bit of other bits on the same port!

We can write now the basic macros for changing the lines. These macros **have to be executed in register bank 1**. This will be under the control of the Start and Stop conditions, hence the above routines will be modified. It is important that wireing constraints do not appear in any macro or routine. The macro and routine files must be applicable to any PIC processor and any wireing option, with the only constraint that SCL and SDA have to be on the same port.

```
.Macro  SetSDA   ; SDA= 1 and Inp
        Set      TrisI2c:#bSDA
.EndMacro
.Macro  ClrSDA   ; SDA = 0 if outp
        Clr      TrisI2c:#bSDA
.EndMacro
```

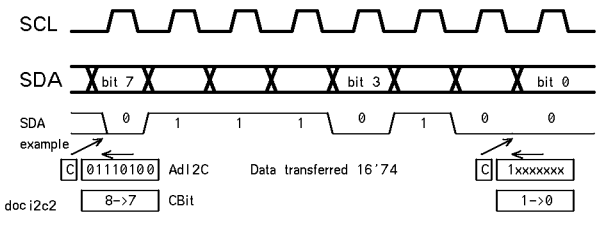
```
.Macro  SetSCK   ; SCK = 1
        Set      TrisI2c:#bSCK
.EndMacro
.Macro  ClrSCK   ; SCK = 0
        Clr      TrisI2c:#bSCK
.EndMacro
```

```
.Macro  FullSCK
        Nop
        Set      TrisI2c:#bSCK
        Nop4
        Clr      TrisI2c:#bSCK
        Nop
.EndMacro
```

We repeat that one should be careful with open-collector programmed lines. A Set or Clr on another bit of the same port will read the byte on PortI2C, modify the assigned bit, and write the byte. This mean that if SCL for instance is one on the line, it will be copied in the port register, and the line will go to one when selected as output. The clock line will stay at one and no more clock will appear!

2.5. 8-bit write

The module to write 8 bits prepared in the ADi2c register is quite similar to all serial transfer modules (fig 2). Register AdI2C will contain addresses or data, hence its name.



doci2c2

Fig. 2 8-bit serial transfer

```
; Write a byte module
 in:  AdI2C
     Move        #8,W
     Move        W,CBit
L$:  RLC         ADi2c
     Skip,CC
     SetSDA      ; Carry = 1
     Skip,CS
     ClrSDA      ; Carry = 0
     FullSCk
     DecSkip,EQ  CBit
     Jump        L$
```

2.6. 8-bit read

The module to read a byte is quite similar: the data line is tested when the clock is active and copied into the AdI2C register via the Carry. In order to test the SDA bit, one have to switch back to bank 0 for the time of the test. Macro Bank1to0 correspond to the instruction Clr Status:#RP0 and the other is a Set.

```
;Read a byte
  out: AdI2C = W
       Move       #8,W
       Move       W,CBit
       SetSDA
 L$:
       ClrC
       Bank1to0
```

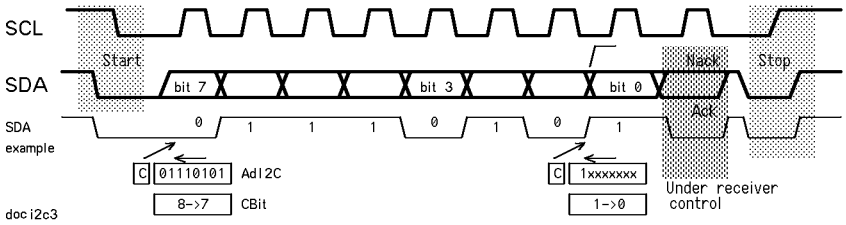
```
TestSkip,BC   PortI2C:#bSDA; SCK à 0
SetC
RLC           ADi2c
Bank0to1
FullSCK
DecSkip,EQ    CBit
Jump          L$
Move          AdI2c,W
```

2.7. Acknowledge

An interesting feature of I²C is the unit receiving the data has to send a positive or negative acknowledge. This occurs immediately after the 8 data clock. The sender leaves the data line at one and the slave generates a zero (Ack) or leaves the line at one (NAck). The sender generates a clock, during which it reads the data line and sees if a unit is replying (Ack) or not (NAck), which may mean that the receiver is not here, or can not/will not accept any more data for this transaction.

2.8. Basic I²C protocol

An I²C write transfer consists of a start condition, 8 data bits, and acknowledge and a stop bit, as shown in Fig 3. The acknowledge is activated at the negative edge of the last data clock, before the sender has deactivated its possible zero. Hence the importance of an open collector, to avoid a "short circuit" visible on the scope as a medium level, without consequence on the correct transfer operation, as long the collision has desappeared when the next clock is active. The sender takes then again the control to generate the stop condition.



doci2c3

Fig. 3 Basic write transfer

With our macros, all the transfer must be executed on register bank 1, and the best is to include the bank switching in the start and stop macros. The main program will stay in bank 0.

The corrected Sart and Stop macros are

```
.Macro  Bank0to1
        Set       Status:#RP0
.Endmacro
.Macro  Bank1to0
        Clr       Status:#RP0
.Endmacro
```

```
.Macro  Start
        Bank0to1
        ClrSDA      ; SDA ´´´\....
        Nop4
        ClrSCK      ; SCK ´´´´´´´\
        Nop2
.Endmacro
```

```
.Macro  Stop
        ClrSDA
        Nop
        SetSCK      ; SDA  ...../
        Nop2
        SetSDA      ; SCK  ../´´´´
        Bank1to0
.Endmacro
```

The routine to write a byte on a I2C slave uses ADi2c as input and the carry bit as output.

Routine **WrI2C** **Write a byte and test acknowledge**

```
  in:  ADi2c address or data
  out: Carry = 0 if Ack, Carry = 1 if NAck
WrI2C:
       Start
       Move       #8,W          ; Transmit ADi2c
       Move       W,CBit
 L$:   RLC        ADi2c
       Skip,CC
       SetSDA     ; Carry = 1
       Skip,CS
       ClrSDA     ; Carry = 0
       FullSCk
       DecSkip,EQ  CBit
       Jump       L$
```

```
       SetSDA                  ; An Ack is expected
       ClrC
       Bank1to0    ; Acknowledge test
       TestSkip,BC PortI2C:#bSDA
       SetC
       Bank0to1
       FullSCK
       Nop3
       ClrSDA
       Ret
```

2.9. Important note on variables

The two variables we use, ADi2c and CBit are accessed inside the routine at a time bank 1 is active. It is of no importance on the 16F84, since the access to variables does not depend on the bank. But on the 74C76 or 76C87x, there are different variables on bank 0, 1, 2, 3. However, variables at addresses 16´70 to 16´FF are identical on all banks, and this is the place we have to define our I2C variables.

2.10. Testing the write routine

A scope or any I²C slave circuit can be used to test our routine. Let us suppose that this slave is 16´70. Let us also have a LED on e.g bit 0 of PortB to show if there is an acknowledge or not. The program that writes continuously on the slave chip is given below, and available on www.didel.com/doc/Xi2cT0.asm. This program insert the I2C macros and the write routine Xi2cWR.asi, explained in previous section. In the future, all our I²C routines will be put inside a Xi2cR.asi file, and what will be important to check every time a routine is called are the parameters in, out and modified, and of course its functionnality.

Program **Xi2cT0** **Simple write test**

```
;JDN 280301
.Proc 16F84
RB0   = 5          ; bit on Status register that control Ba
PortA
bSCK  = 0          ; I2C bits
bSDA  = 1
DirA  = 2´00011              ; Bits 0 and 1 as input
PortI2C          = 5                  ; I2C bits on
TrisI2C          = PortI2c
SlaveAdress = 16´70          ; 8574 I/O expander
.Ins  Xi2cM.asi  ; Inserts I2C macros
PortB
bLed  = 0          ; Led bright if output=0
DirB  =2´00000000            ; Bit 0 as output
.Macro LedOn
      Clr          PortB:#bLamp
.Endmacro
.Macro LedOff
      Set          PortB:#bLamp
.Endmacro
Variables
.Loc  16´0C      ; begin of variables
CBit:            .Blk.16      1
ADi2c:           .Blk.16      1
```

```
Programme
.Loc  0          ; program start
Deb:  Move       #DirA,W
      Move       W,TrisA
      Move       #DirB,W
      Move       W,TrisB
      Clr        PortA             ; important for SCK SDA
Loop: Move       #SlaveAddress,W
      Call       WriteI2C
      Skip,CS
      LedOn                        ; Ack
      Skip,CC
      LedOff                       ; NAck
      Jump       Wait              ; space on the scope
                                   ; to ease synchronization
      Jump       Loop
.Ins  Xi2c0.asi

Routine Wait 1ms wait
Wait: Clr        W
 W$:  Add        #-1,W             ; 4 us loop
      Skip,EQ    ; repeated 256 times
      Jump       W$
      Ret
.End
```

If the I²C slave (here the 8574 described later, but any I2C chip can be used it the SlaveAddress is correctly declared) is correctly connected and powered, the LED will bright. By disconnecting SCL or SDA, or will not. Have also a look with a scope. For adjusting the timings and for any real time programming a scope, even an old one, is essential.

3. I2C main features

As mentionned above, we will not detail all the I²C features. Most applications just control several I²C slaves and we want to do it in a simple and efficient way.

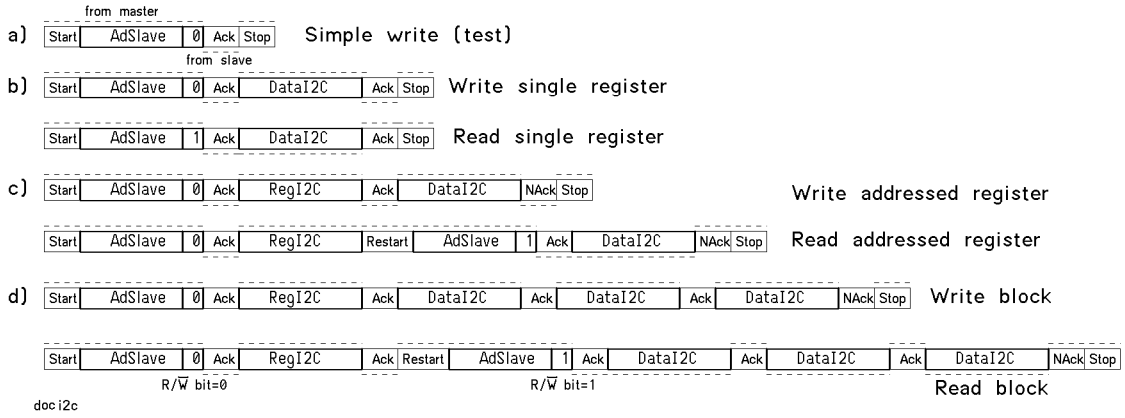
3.1. Transactions

The write transfer we have seen is drawn in a simplified way on Fig 4 a). It selects a slave, and has no other application than an existence test.

The simplest I²C slaves (Fig 4 b) accept an 8-bit data word and can provide an 8-bit information back. This is the case of the PCF8574 parallel port we will see later. Reading needs to repeat the address, with the least significant bit (the $\overline{R}$/W bit) set to one. All I²C addresses are even. The read transfer sequence is double: a write transfer to write the selected register, and then a write cycle with the read address, followed by a read cycle. Start/Stop can be used for both transfers, but a slightly shorter Restart condition can be used inbetween.

More complex I²C slaves have a set of 8-bit registers. The register is selected before sending the data (Fig 4 c). Read transfers are terminated by a NAck, to mention the master the transfer is complet. The reason for this is multiple data transfers can be done in the same transaction (Fig 4 d).

An EEPROM or a RAM for instance has an address register which points to a data byte that can be read or written. A data transfer increments automatically the address counter, and consecutive locations can be transferred (bloc transfer). There may be limitations on the length of the bloc, and delays after writing, in the case of an EEPROM. Other I²C circuits, e.g. a real clock, are indeed similar; they have many registers, which can be addressed independantly, or in sequence as a block transfer to consecutive addresses.



doci2c4 *Fig. 4 I²C transfer sequences*

3.2. Addressing

I²C addresses are 7 bits on bits 7..0. Bit 0 is a Read/$\overline{\text{Write}}$ indication: read addresses are odd. Many I²C circuits take the 2 or 3 low address bit on circuit pins, in order to hardwire these address bits and support several identical chips in parallel (or try to avoid an address used by another slave).

4. I2C routines

In order to be compatible with different addressed devices and different transaction formats, we have defined a set of routines which make the code as flexible and short as possible. One needs to define 6 registers at bank-independant addresses:

```
CBit:          .Blk.16  1
AdI2C:         .Blk.16  1
AdSlave:       .Blk.16  1
RegI2C:        .Blk.16  1
DataI2C:       .Blk.16  1
```

AdSlave must be loaded with the slave address. It has to be reloaded by the 2 instructions below only when it is necessary to select a new slave.

```
Move    #AdNextSlave,W
Move    W,AdSlave
```

RegI2c is the first byte written, usually the register address, DataI2c is the data. The write low level routine uses ADi2c register.

The source code of our routines (www.didel.com/doc/i2c.html) is not detailed here; we recommend their reading, but they can be used without their understanding if the parameters they need are correctly prepared.

Routine **StartWrite** **Send the slave addresse, the register internal address and the data**

```
; A block of data may follow. A Stop must terminate the transfer.
  in:   AdSlave slave address
  in:   RegI2C  register address
  in:   DataI2c  dats written
  mod: W, ADi2c
```

Routine **WriteI2C** **Send address or data**

in: ADi2c byte to be transmitted
out: Carry = 1 if Ack = 0 if Nack
mod: W, ADi2c

Routine **StartRead** **Select a register in a esclave and reselect for a data read**

; An Ack or Nack must be given. A data block may follow. A Stop must terminate.
in: AdSlave slave address
in: RegI2C register address
out: W = Adi2c data read
mod: W, ADi2c

Routine **ReadI2C** **An Ack or Nack must be given. Plus a Stop if last read**

in: AdSlave slave address
out: W = ADi2c data read
mod: W, ADi2c

Routine **GiveAckR** **Master acknowlege after a read**

mod: -

Routine **GiveNAckR** **Negative acknowledge**

mod: -

Routine **GiveNAckStop** **Negative acknowledge and stop condition**

mod: -

Routine **StartR** **Routine Start**

mod: -

Routine **StopR** **Routine Stop**

mod: -

The StartR, StopR routines have the same effect as the Start and Stop macros, but they insert only one byte of code.

Transaction of Fig 4 b) are not supported, since our routines have been optimized for complex I²C circuits. For this case, new routines have to be written, as shown in next section.

Transaction of Fig 4 c) is programmed as below.

For writing:

```
AdSlave OK, load RegI2C and DataI2C
Call      StartWrite
Call      StopR
```

For reading:

```
Load AdSlave, load RegI2C
Call      StartRead
Call      GiveNAckStop
The data read is both in ADi2c and W
```

Transaction of Fig 4 d) is programmed as below.

For writing:

```
AdSlave OK, load RegI2C and first data in DataI2C
Call      StartWrite
Load second data in DataI2C
Call      WriteI2C
Load third data in DataI2C
Call      WriteI2C
Call      StopR
```

For reading:

```
AdSlave OK, load RegI2C
Call      StartRead
Handle first data in W
Call      ReadI2C
Handle second data in W
Call      ReadI2C
Handle third data in W
Call      GiveNAckStop
The data read is both in ADi2c and W
```

5. Example 1: 8-bit I/O expander PCF 8574

The I²C I/O expander PCF 8574 is easy to implement and convenient to get more inputs and outputs from a microcontroller system. The Max 1608/Max1609 circuis have a quite similar functionnality. There is no direction to assign, since the outputs are open-collector: an input is an output at state one.

The test program below, available with its complete form at www.didel.com/doc/Xi2c8574.asm, reads 4 switches on the low 4 bits (pull-up are required) and copies the content on the 4 high bits. Simpler write and read routines could be written if there is only one 8574 circuit to be controlled; the 8574 is the only circuit not requiring internal sub-addresses.

```
      Move      #Ad8574,W   ; Declared as 16'40 to
      Move      W,AdSlave
      Move      #2'00001111,W
      Move      W,Datai2c   ; Data direction initialize
      Call      WrSingleI2C
Loop:
      Call      RdSingleI2C ; Result in W and DataI
      Swap      DataI2C,W
      Or        #2'00001111
      Call      WrSingleI2C
      Jump      Loop
```

```
WrSingleI2C:
      Move      AdSlave,W
      Move      W,ADi2c
      Start
      Call      WriteI2C
      Stop
      Ret
RdSingleI2C:
      Move      AdSlave,W
      Move      W,ADi2c
      Start
      Call      WriteI2C
      Call      ReadI2C
      Call      GiveNAckR
      Ret
```

6. Example 2: EEPROM 24LC01 128x8

The 24LC01 from Microchip is an 8-pin circuit (also available in a miniature 5-pin package) that stores up to 128 bytes of data. There a single control register, the address pointer that defines where the next data will be read or written. This address pointer is incremented after every read, and the reading is immediate. For writing one location at a time, a programming time of 10ms must be respected (the circuit will not acknowledge a transfer during this time). Block write are efficient, but one needs to understand their limitation: The EEPROM block is structured as 16 8-byte blocks and has an 8-byte input buffer. Within the same block, one can do a block transfer of 8 byte maximum, if one stays inside the same block (the autoincrement concerns only the 8 low bit of the address pointer). If the complete memory has to be written, it is hence necessary to reload the address pointer every 8 bytes. The interest of that feature is to program up to 8 bytes with a 10 ms programming time.

The routine we have defined are ideally suited for this case. A single position write and a read are programmed as below.

```
Write:
      Move      #Ad24LC01,W
      Move      W,AdSlave
      Move      #eeAddr1,W    ; Selected eeprom addres
      Move      W,RegI2C
      Move      #Data1,W      ; Usually a variable
      Move      W,DataI2C
      Call      StartWrite
      Call      StopR
```

```
Read:
      Move      #Ad24LC01,W   ; If not
      Move      W,AdSlave     ;  done before
      Move      #eeAddr1,W    ; Selected eeprom addres
      Move      W,RegI2C
      Call      StartRead
      Call      GiveNAckStop
result in W and ADi2c
```

Reading or writing a block can be done by calling the routine ReadI2C or WriteI2C before the StopR. For instance, the module to read NN bytes from Addr1 is given below. One notices that intermediate read gets an Ack, but the last read gets a NAck.

```
BlockRead:
        Move    #Ad24LC01,W     ; If not done
        Move    W,AdSlave ; done before
        Move    #eeAddr1,W      ; Selected eeprom address
        Move    W,RegI2C
        Call    StartRead
        Move    #NN-1,W         ; NN is the block lengh
        Move    W,CBlock  ; CBlock variable to be defined
 L$:
        data available in W and ADi2c
        Call    GiveAck
        Call    ReadI2C
        DecSkip,EQ CBlock
```

```
        Jump      L$
        last data available in W and ADi2c
        Call      GiveNAckStop
```

7. Appendix

7.1. I²C macros and routines

Following files, referred in www.didel.com/doc/Xi2c.html will help you to execute the test programs and use our .asi files for you own applications.

- Xi2cV.asi includes definitions and variables. Definitions will be copied in your program, or inside an imported file that defines your hardware.
- Xi2cM.asi are the macros.
- Xi2cR.asi are the routines

7.2. Short delays macro and routine

File XDelR.asi (www.didel.com/doc/XdelR.asi) has to be inserted after the beginning of the program if macros are called by the program, which we do not recommend in applications, but it is of course the case in the test program Xdel.asm.

For instance,

```
Del  3 produce the code:
    Jump      APC+1
    Nop
```

```
Del  7 produces the code
    Nop
    Call      Del7
```

7.3. Long delays routines

File XDelaiR.asi (www.didel.com/doc/XdelaiR.asi) includes two routines for short delays (unit 100 μs) and long delais (unit 20ms, max 5 sec). Test program Xdelai.asm blinks a Led.