AN1146
APPLICATION NOTE

I²C Communication between ST52x301 and EEPROM

Authors: V. Marino, C. Vinci

1. Introduction

This application note shows an example of how to use ST52x301 to communicate with an EEPROM memory with an I²C bus protocol. In this example, an M24C04 EEPROM (4K bit) is taken into account, but the following considerations can be applied to applications with any kind of M24CXX memory.

Two software routines are proposed, the Byte Write and the Random Read Access, for ST52x301 microcontroller configured in single master communication.

2. Communicating Protocol

The M24CXX is an EEPROM supporting the I²C protocol. The M24CXX can communicate with a microcontroller, the master, only by a serial data I/O line (SDA) and a serial clock (SCL).

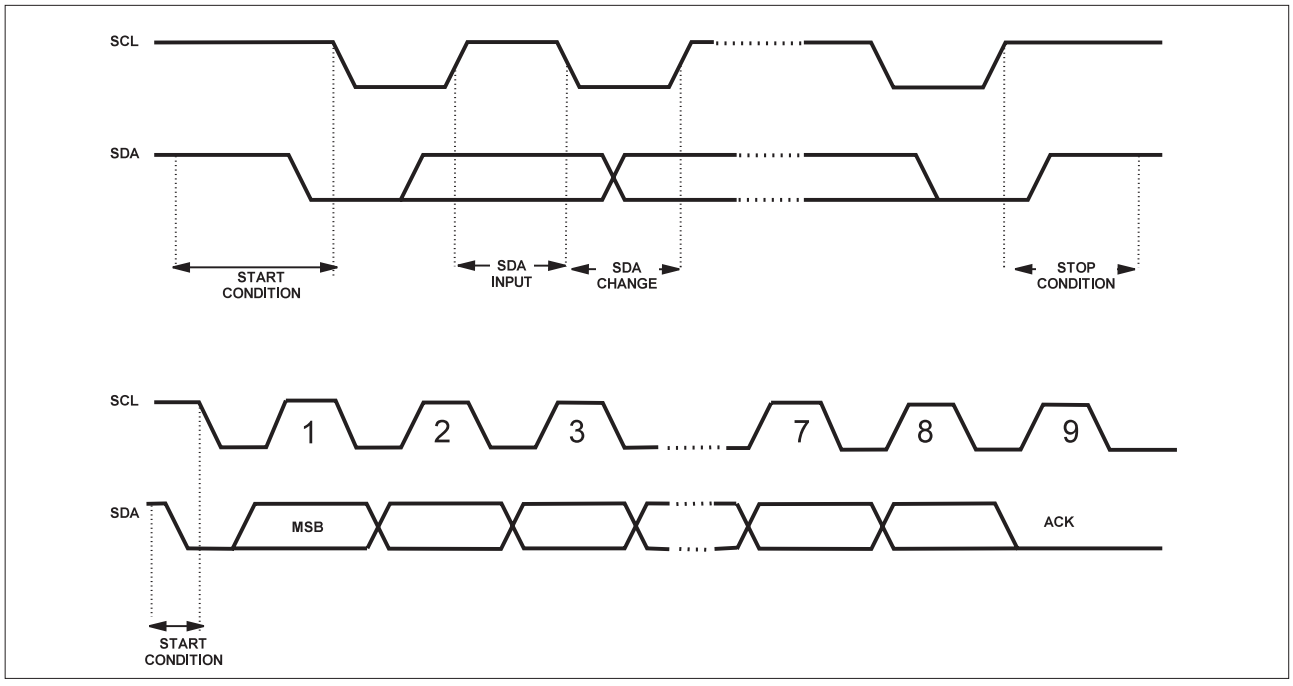
During each data transfer, the M24CXX samples the SDA bus signal on the rising edge of the clock signal SCL. The SDA signal must be stable during the clock low to high transition and the data must change only when the SCL line is low. Changes in the data line while the clock is high are interpreted as a START and STOP conditions.

START is identified by a high to low transition of the SDA line while the clock is stable in the high state. A START condition must precede any data transfer command.

After the START, ST52x301 sends onto the SDA bus line 8 bits (MSB first): the first 7 bits to select the device, the last (RW bit) to indicate if it is a read (RW high) or write (RW low) operation.

After sending each 8 bits data stream, the master releases the SDA bus; during the 9th clock pulse period the receiver pulls the SDA bus low to acknowledge the receipt of the 8 bit data. A complete data transfer is always terminated by a STOP, identified by a low to high transition of the SDA line while the clock SCL is stable in the high state.

Fig. 1 - I²C Bus Protocol



3. Hardware Description

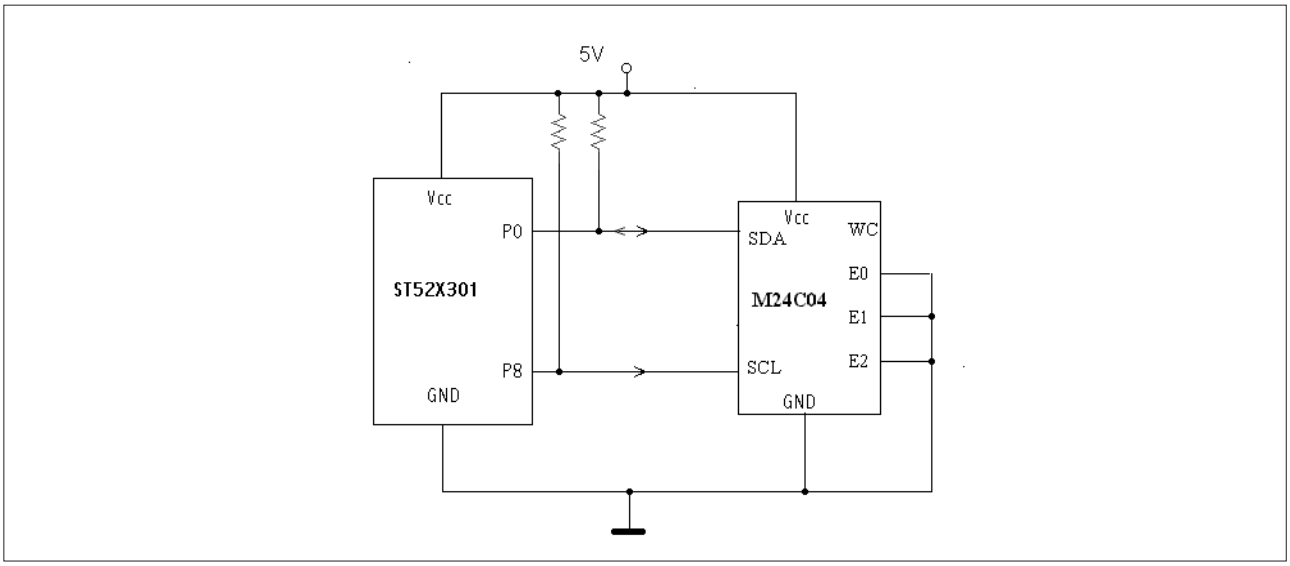
The connection scheme between ST52x301 and EEPROM is shown in Figure 2. Pin P0 of ST52x301 is used to transfer data to and from the memory (SDA); pin P8 for data syncronization (SCL).

In this scheme M24C04 inputs E0, E1 and E2 are tied to Vss (Device Select Code is A0h), therefore, being E0 low, only 256 byte of memory, the low part, can be addressed. To address also the memory high part (address 1xxxxxxxx), E0 must be dynamically driven by using another I/O pin of ST52X301.

A maximum of four memories of the type M24C04 can be addressed by a microcontroller on the same two wire bus, setting E1 and E2 inputs (00, 01, 10, 11): each device is identified by its Device Select Code and will only respond to the correct selection.

Pin WC, used to protect the contents of the memory from inadvertent erase/write operations, is unconnected, therefore with this scheme, write operations are always allowed. To allow enabling (WC\=V_{IL}) and disabling (WC=V_{IH}) write operations, WC must be driven dynamically.

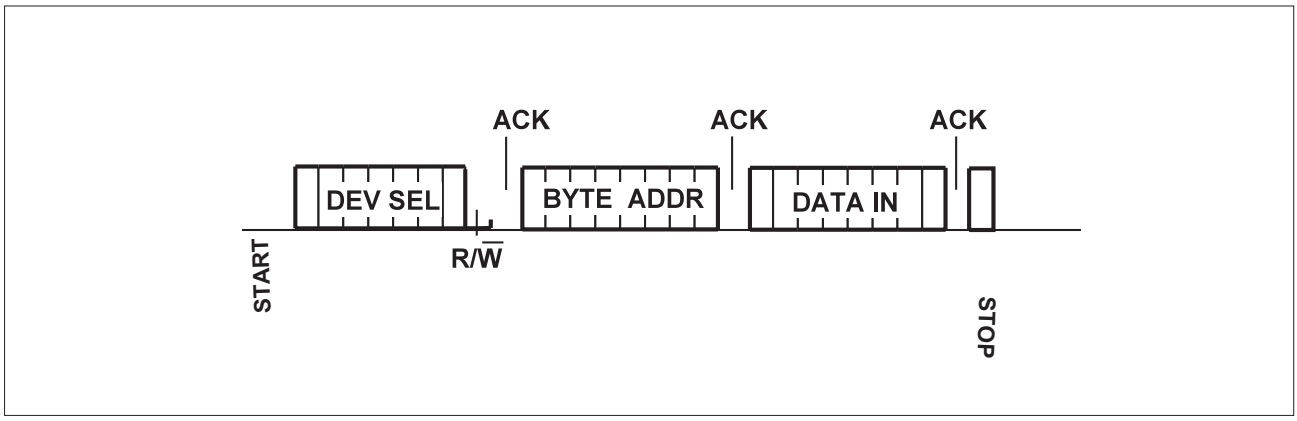
Fig. 2 - ST52x301/EEPROM I^2C Schematics



3.1 Byte Write Mode

In WRITE mode, after the START condition, ST52x301 sends a Device Select Code with the RW} bit set to '0', as shown in Figure 3. The memory acknowledges the reception and waits for an address byte from the master, to which it responds with an acknowledge. After the acknowledge, ST52x301 sends the data byte to be written in the defined memory location. ST52x301 terminates the transfer by generating a STOP condition.

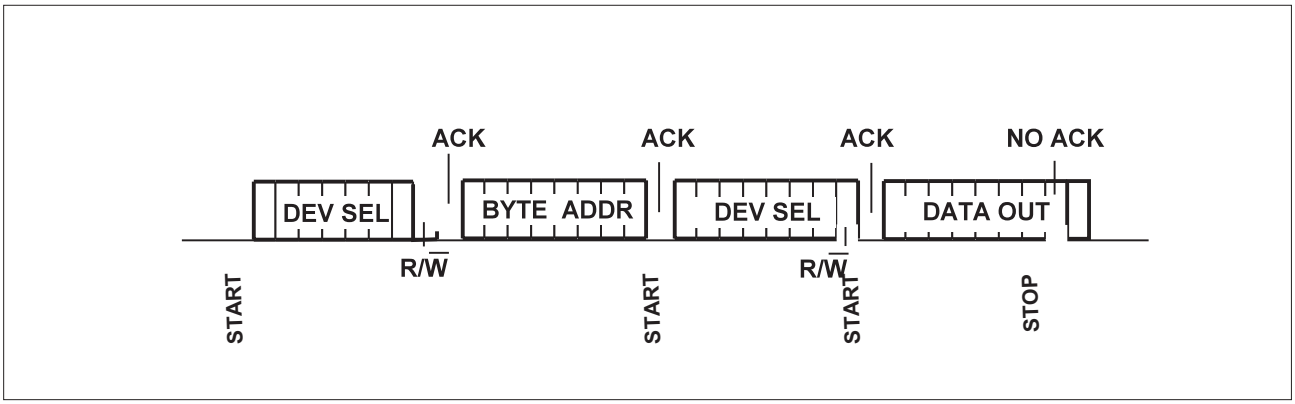
Fig, 3 - Byte Write Mode Sequences with WC=0 (data write enabled)



3.2 Random Address Read Mode

In order to read a byte from an address of the memory, ST52x301 performs a dummy write to load the address, as shown in figure 4. Then, without sending a STOP condition, ST52x301 sends another START condition and resends the Device Select Code, with the RW\ bit set to '1'. After acknowledgment, the memory provides onto the bus the contents of the addressed byte. The master is not required to acknowledge the byte output and terminates the transfer with a STOP condition.

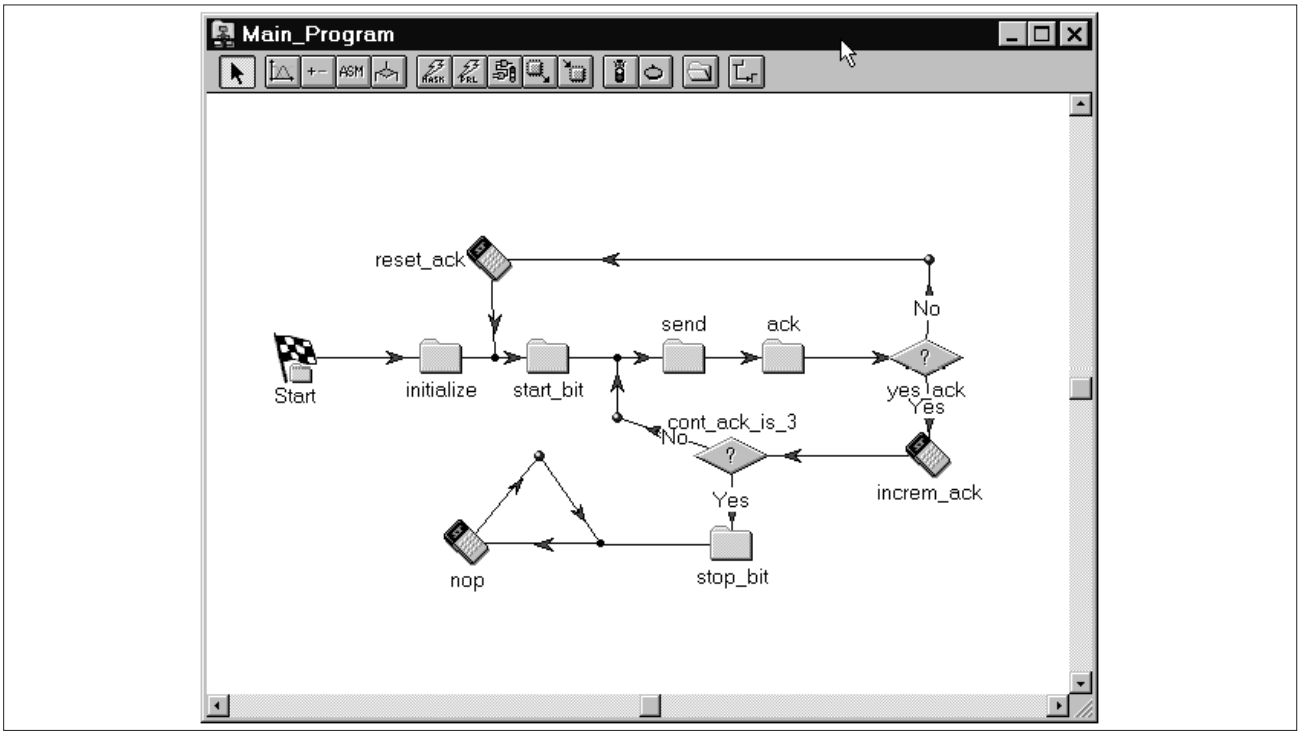
Fig. 4 - Read Mode Sequence



4. Software (I²C communication routines between ST52x301 and M24C04 EEPROM)

The software project for ST52X301 communicating with an EEPROM M24C04 is described in the followings. Two routines are developed with FUZZYSTUDIO™3.0 to show how to perform operations of write and read of a single byte at a specific address. To manage more bytes, a simple modification of this program must be done.

Fig. 5 - Main Window



4.1 Byte Write software routine

The main flow chart program of ST52x301 Wryte Byte routine, as developed in FUZZYSTUDIO™3.0 environment, is shown in Figure 5.

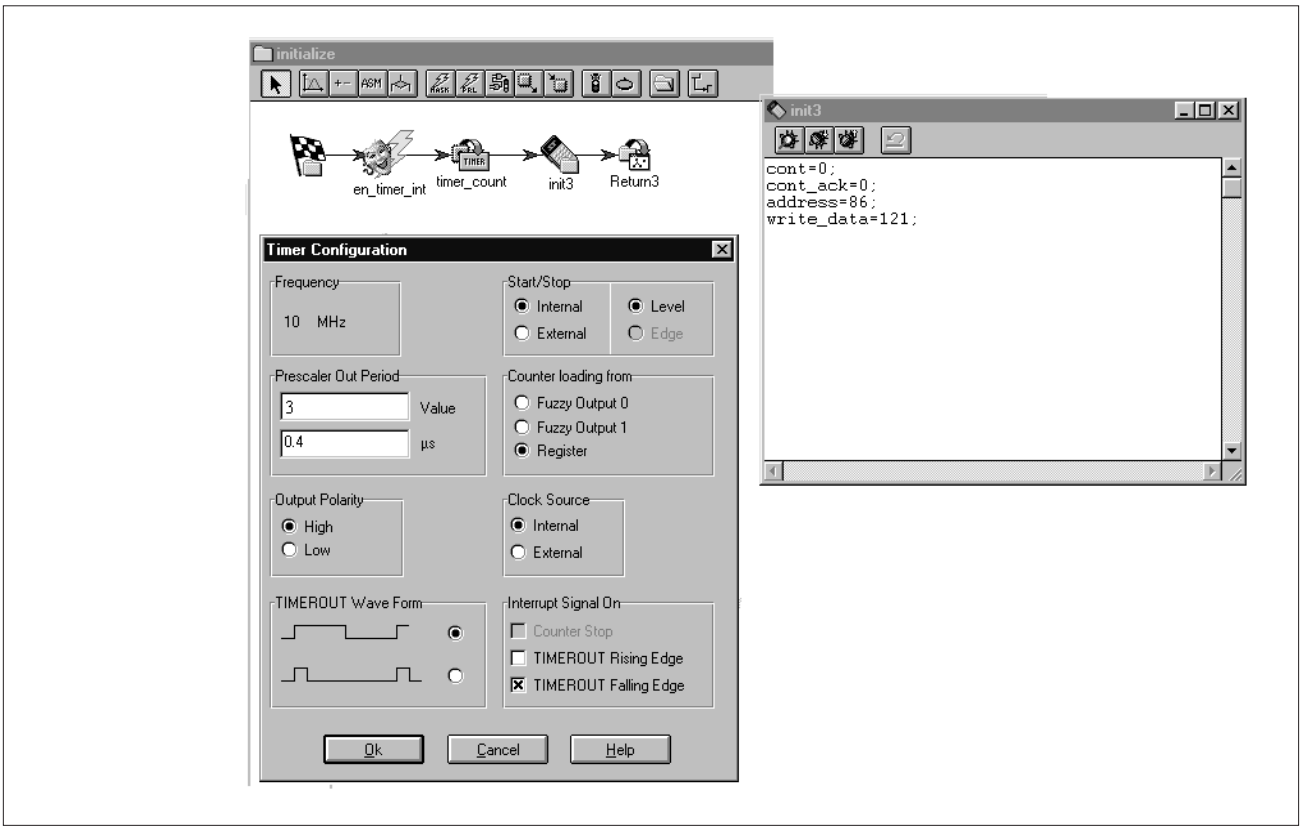
After the block '*initialize*', where the communication speed, the address and the value of the byte to write are set, the START condition is performed in the block '*start bit*'.

Then three bytes are serially sent onto SDA bus, respectively the Device Select Code, the Address byte and the Data byte to be written into the memory address location: after each byte is sent, ST52x301 waits for M24C04 acknowledgement; at each ACK receipt, the variable '*cont_ack*', initially set to zero, is incremented of one unit. When '*cont_ack'* reaches 3, a STOP condition is sent by ST52x301 to the memory. If the signal ACK is not returned from the memory, the program provides to restart the communication protocol.

4.1.1 'Initialize' block

This block allows to enable the timer interrupt, to set the timer counter and to initialize the variables (Figure 6). The Timer peripheral has been configured in order to have a timer clock T = 0.4ms by setting the prescaler value equal to 3 and the master clock frequency to 10Mhz. The choice to load the counting from the the *timer_count* register, where the user can write values ranging from 10 to 255, allows to set the communication speed within the range [250KHz (*fast mode*), 9.8KHz (*standard mode*)].

Fig. 6 - Peripheral Initialization



4.1.2 'Start_bit' block

This block performs the START condition: at first, the SDA line is pulled in high state through sending of the number '1' into the parallel port (block 'data_high'); then the SCL line is pulled high by setting pin P8 high (block 'clk_high'); finally SDA is put low to complete the START bit communication, as required from the protocol (Figure 2).

Time delays among transitions between each block in the *'Start_bit'* routine are managed thanks to the timer peripheral. In fact each block is performed only after each Timer count end: each Wait block allows this synchronization waiting for the Timer interrupt generated on the falling edge of the Timer out signal.

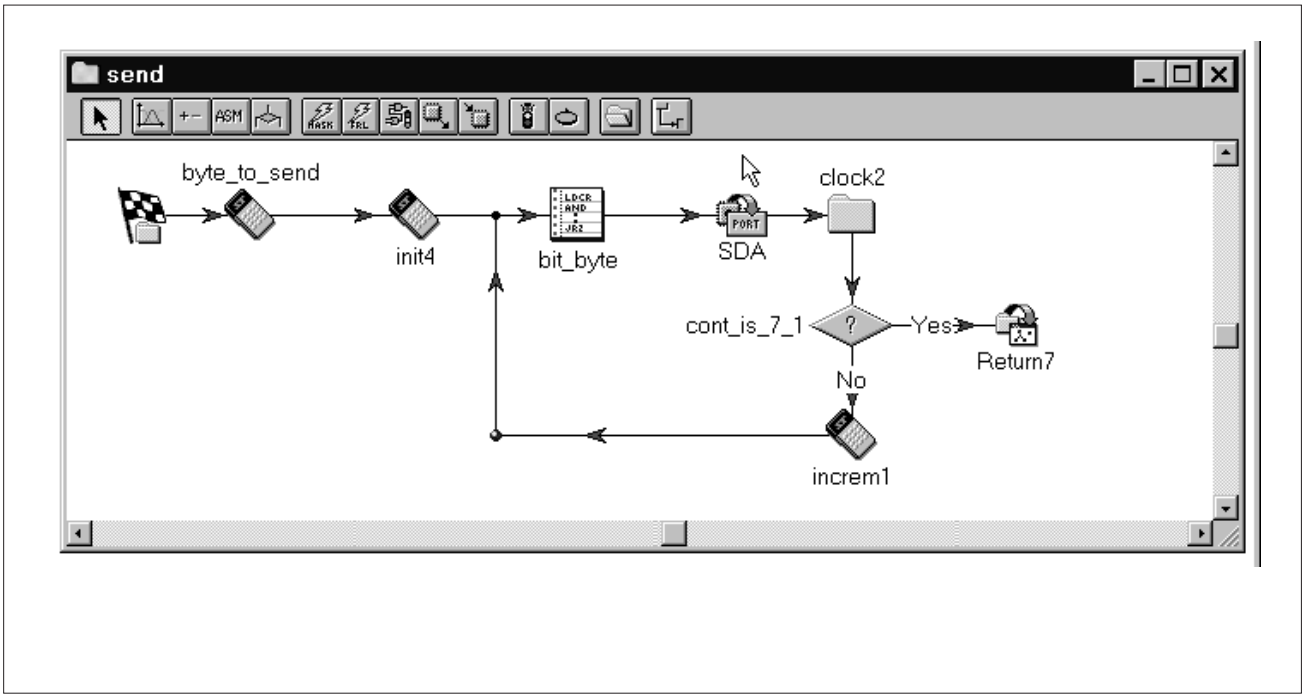
4.1.3 'Send' block

To decide whether the Device Select code byte (160 in this case), or the address byte or the data byte must be sent (in the arithmetical block 'byte_to_send') the variable '*cont_ack'* is used (Figure 7).

To send each byte, bit by bit, a cycle scanned by a '*cont*' variable (incremented from 0 to 7) is performed. At each step of the cycle, one data bit is sent onto SDA bus (in the '*SDA*' block) and the SCL line put low to high and again to low ('*clock2*' block), through port P8, to let the memory read the single bit data on the rising edge of SCL clock signal pulse.

The block '*bit_byte*', written in assembler code for quick operations allows to send the single bit of the byte stream (see assembler code routine in Appendix 1).

Fig. 7 - Send block

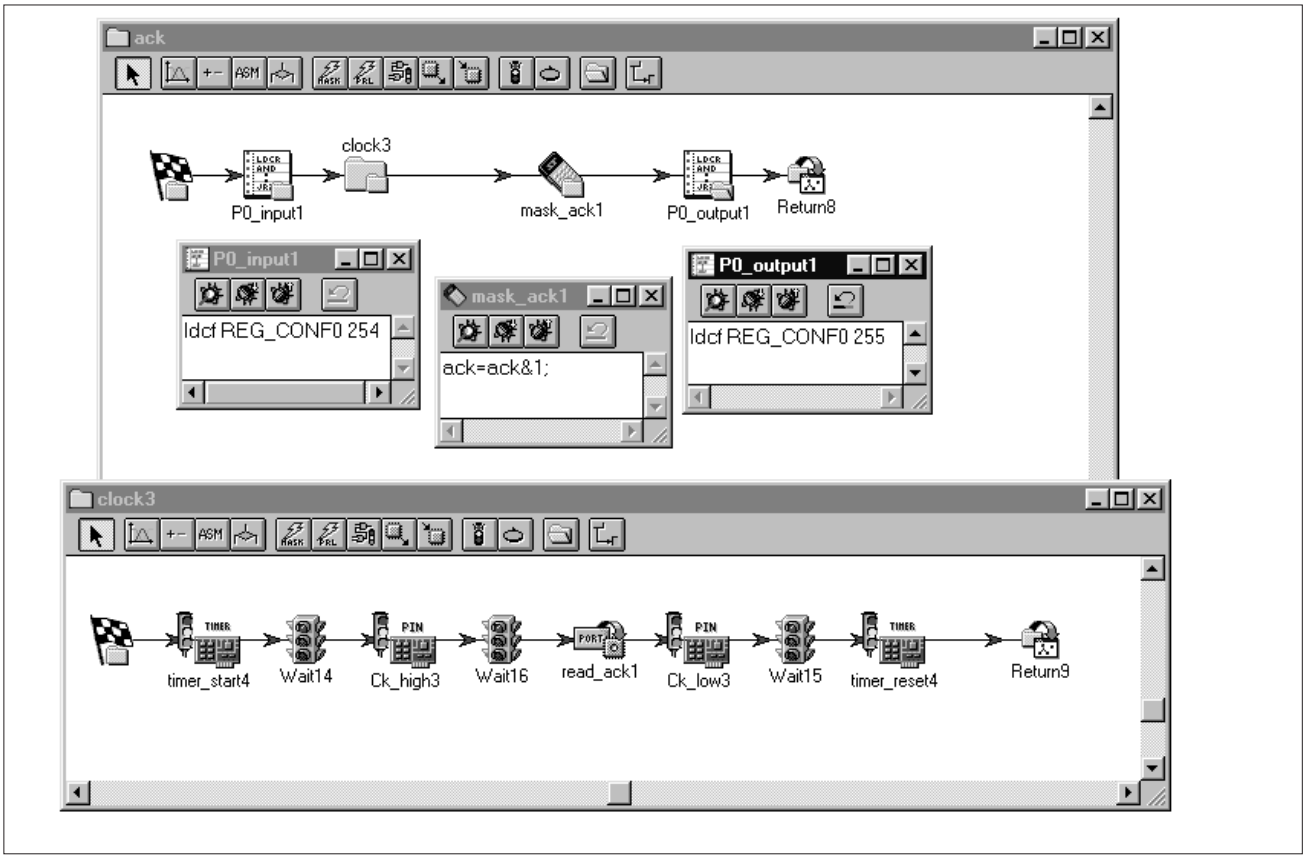


4.1.4 'Ack' block

After sending 8 bits data, ST52x301 releases the SDA bus, setting the pin P0 in input. REG_CONF0 is used to set each pin of the parallel port in input or in output.

During the 9-th clock pulse period, the EEPROM pulls the SDA bus low to acknowledge reception of the data byte: the byte read onto ST52x301 parallel port is stored into an '*ack*' variable ('*read_ack1'* block). To understand if the LSB is 0 (on P0), therefore if the EEPROM has correctly received the data, the 8 bit variable '*ack*' is masked. If P0 bit results to be 1, an error has occurred and the communication is restarted.

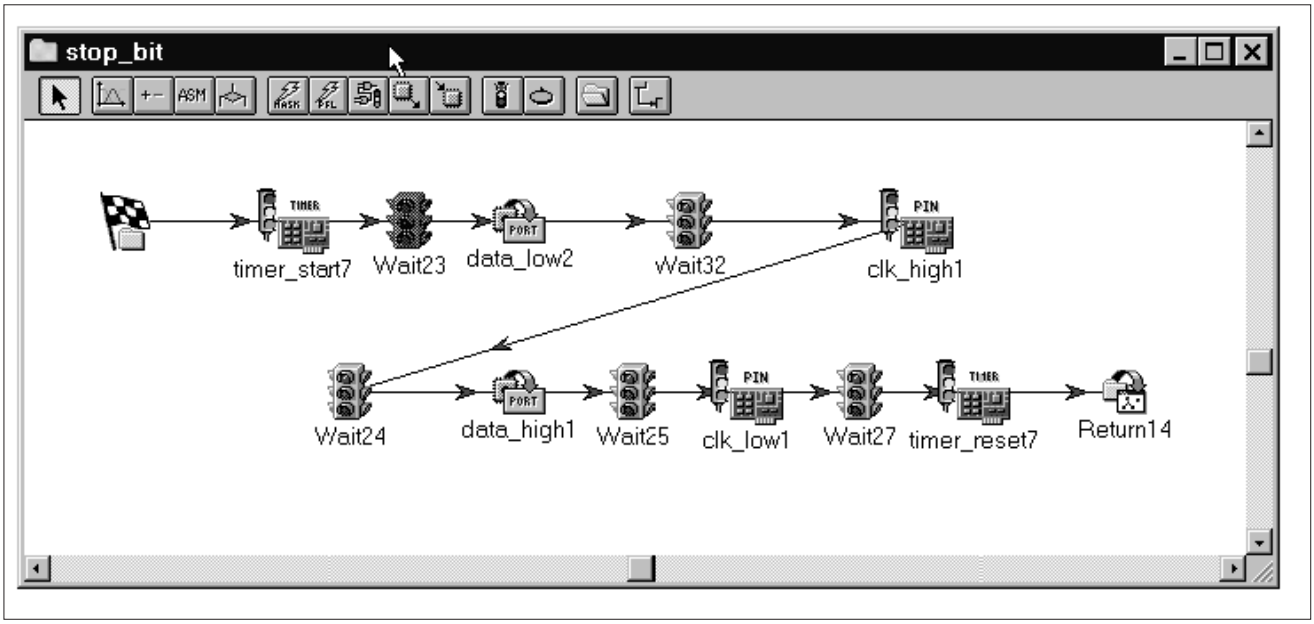
Fig. 8 - Acknowledgement Block



4.1.4 'Stop_bit' block

A data transfer is always terminated by a STOP condition, that is identified by a low to high transition of the SDA line while the clock SCL is stable in the high state. This condition is performed with the '*stop_bit'* block (Figure 9).

Fig. 9 - Stop block



4.2 Read software routine

The flow chart of the 'Read Memory' routine, as developed with FUZZYSTUDIO™3.0 is shown in Figure 10. After the block '*Initialize*', where the communication speed and the address of the byte to be read are set, the block '*start bit*' executes a START condition.

Three bytes are then sent serially onto SDA bus. A cycle variable '*cont_ack*' is used to discriminate which byte is going to be sent: the Device Select Code (160) if '*cont_ack'* is '0' the address byte if it is '1', the Device Select Code again with the RW\ bit set to '1' (i.e. a byte corresponding to 161), after a new START condition, if '*cont_ack'* is '2'. If '*cont_ack*' is '3', the data is read and the STOP condition is executed.

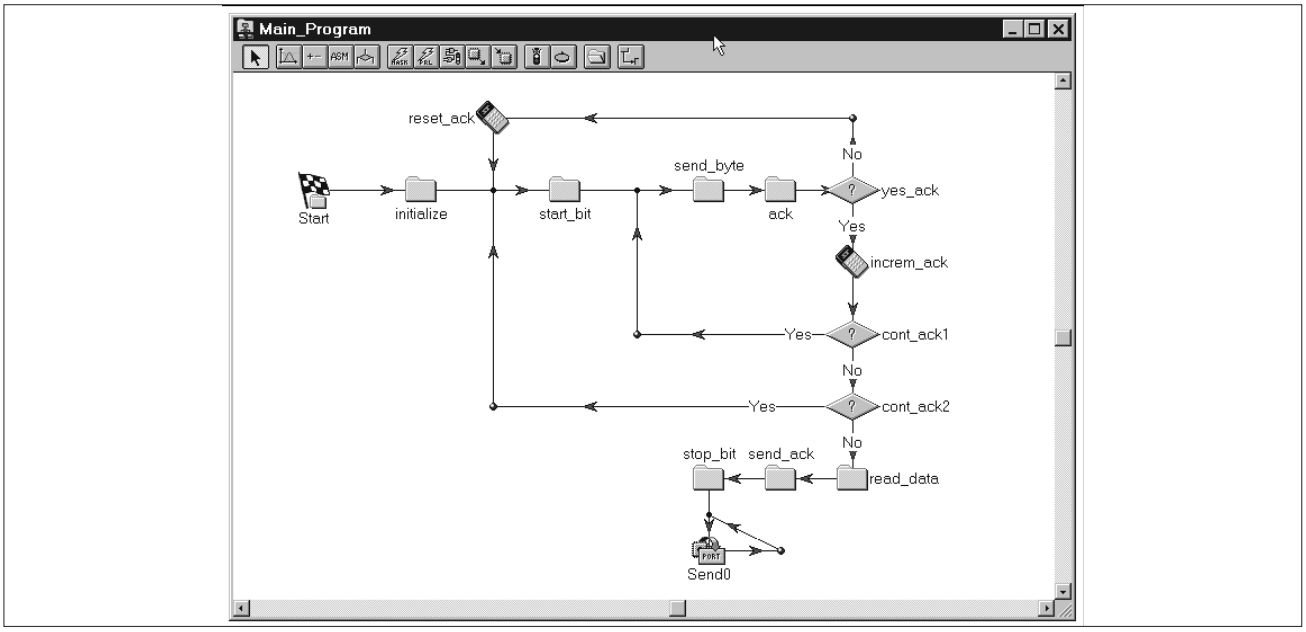
After each byte is sent, the microcontroller waits an acknowledgement from the memory: if the ACK is not returned, the program provides to restart the communication protocol.

4.2.1 'Read_data' block

To read the eigth bit data sent from the memory, ST52x301 performs a cycle scanned by a variable '*cont*': at each step *'cont'* is incremented from 0 to 7 and one bit of the data is received into pin P0 (block 'Receive0') after the rising edge of the clock SCL (pin P8 of ST52x301).

The block '*bit_data*' allows to build the data byte contained in the addressed location from each bit received from the memory (See Appendix 2 for assembler code").

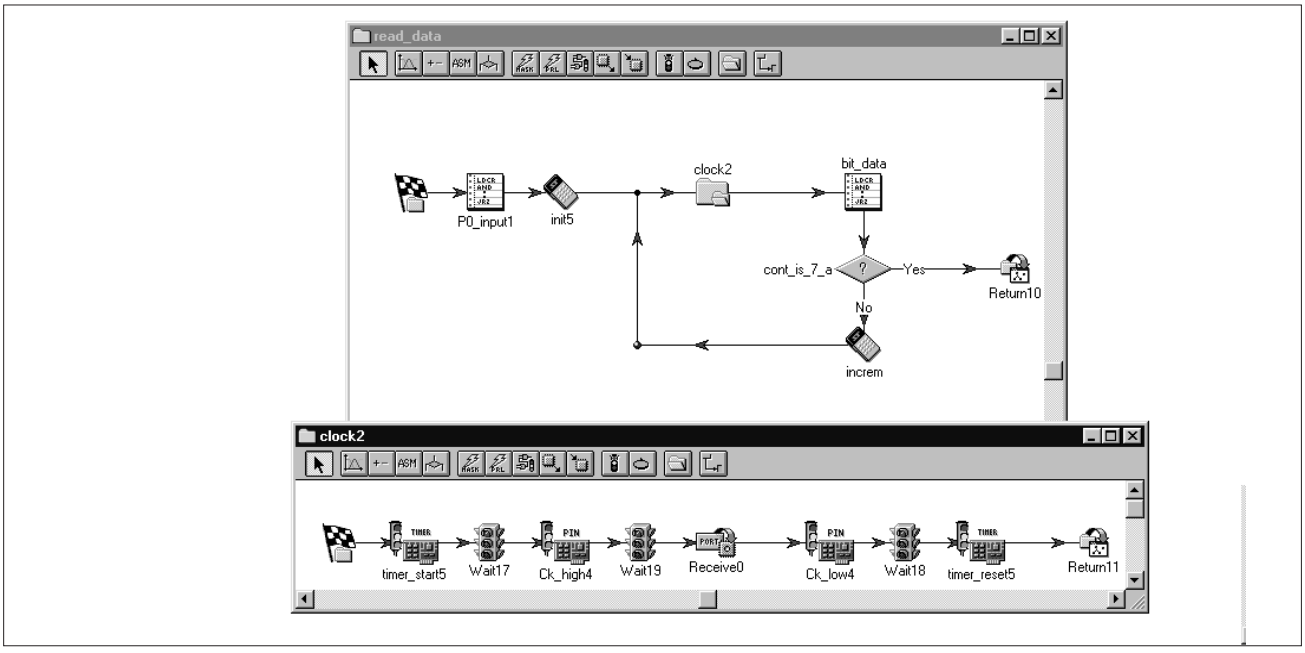
Fig. 10 - Read_data block



4.2.2 'Send_ack' block

In accordance with the I²C communicating protocol, at the end of the data byte receipt, ST52x301 puts the pin P0 high, before sending the STOP condition.

Fig. 11 - Send_ack block



Appendix 1

Bit_byte Assembler Block

This block is designated to write the data byte into the memory; the data is composed of eigth bits, then, in order to write it, a cycle of eight steps is realized (Fig.7).

At the first step the cycle counter '*cont*' is 0, then intruction '*send_byte_bit_7'* is executed and the program jumps to label '*out*'; at the second step, since cont is 1, the intruction '*send_byte_bit_6*' is executed before jumping to label '*out*', and so on.

At this point, at label '*out*', the variable 'mask' contains one value among 128, 64, 32, 16, 8, 4, 2, 1. A logical AND betwen the variables '*mask*' and '*send_byte*' will allow to determine if a 0 or a 1 data bit must be sent onto P0 by block 'SDA' (Fig.7). The variable *data* contains the bit to be written.

In this manner data is converted from parallel to serial format.

```
//*************************************************************************
ldrc temp 0
sub  temp cont
jpnz a0                 //cont!=0

//******send_byte-bit7*****
ldrc mask 128           //cont=0
jp out
//************************

a0:
ldrc temp 1
sub  temp cont
jpnz a1                 //cont!=1

//******send_byte-bit6*****
ldrc mask 64            //cont=1
jp out
//************************

a1:
ldrc temp 2
sub  temp cont
jpnz a2                 //cont!=2

//******send_byte-bit5*****
ldrc mask 32            //cont=2
jp out
//************************
```

```
a2:
ldrc temp 3
sub  temp cont
jpnz a3                 //cont!=3

//******send_byte-bit4*****
ldrc mask 16            //cont=3
jp out
//************************

a3:
ldrc temp 4
sub  temp cont
jpnz a4                 //cont!=4

//******send_byte-bit3*****
ldrc mask 8             //cont=4
jp out
//************************

a4:
ldrc temp 5
sub  temp cont
jpnz a5                 //cont!=5

//******send_byte-bit2*****
ldrc mask 4             //cont=5
jp out
//************************

a5:
ldrc temp 6
sub  temp cont
jpnz a6                 //cont!=6

//******send_byte-bit1*****
ldrc mask 2             //cont=6
jp out
//************************
```

```
a6:
ldrc temp 7
sub  temp cont
jpnz out                //cont!=7

//******send_byte-bit0*****
ldrc mask 1             //cont=7
//************************

out:
ldrc data 0
and  mask send_byte
jpz  b0
ldrc data 1
b0:
//*************************************************************************************
```

Appendix 2

Bit_data Assembler Block

This block is designated to read the data from the memory; the data is composed of eigth bits, then, in order to read it, a cycle of eigth step is realized(Fig.11).

At the first step the cycle counter '*cont*' is zero, then the intruction '*data-bit7*' is executed and the program jumps to label '*out*'; at the second step, the intruction '*data-bit6*' is executed before jumping to label 'out', and so on.

At this point, at label '*out*', the variable '*mask*' contains one value among 128, 64, 32, 16, 8, 4, 2, 1; if at step n of the eight bit cycle, the variable '*read_bit*' is 1, the variable '*data*' is increased adding the variable '*mask*' contents.

In this manner data is converted from serial to parallel format.

```
//*************************************************************************
ldrc  temp 1

and  read_bit temp      //read_bit will contain '0' or '1'


ldrc temp 0
sub  temp cont
jpnz c0                  //cont!=0

//******data-bit7*****
ldrc  mask 128          //cont=0
Jp    out
//************************

c0:
ldrc temp 1
sub  temp cont
jpnz c1                 //cont!=1

//******data-bit6*****
ldrc  mask 64           //cont=1
jp out
//************************

c1:
ldrc temp 2
sub  temp cont
jpnz c2                 //cont!=2
```

```
//******data-bit5*****
ldrc  mask 32           //cont=2
jp out
//************************

c2:
ldrc temp 3
sub  temp cont
jpnz c3                 //cont!=3

//******data-bit4*****
ldrc  mask 16
                        //cont=3
jp out
//************************

c3:
ldrc temp 4
sub  temp cont
jpnz c4                 //cont!=4

//******data-bit3*****
ldrc  mask 8
                        //cont=4
jp out
//************************

c4:
ldrc temp 5
sub  temp cont
jpnz c5                 //cont!=5

//******data-bit2*****
ldrc  mask 4            //cont=5
jp out
//************************

c5:
ldrc temp 6
sub  temp cont
jpnz c6                 //cont!=6

//******data-bit1*****
ldrc  mask 2            //cont=6
jp out
//************************
```

```
c6:
ldrc temp 7
sub  temp cont
jpnz out                //cont!=7

//******data-bit0*****
ldrc  mask 1            //cont=7

//************************

out:

and read_bit            read_bit
jpz   d0
add  data       mask
d0:
//*************************************************************************
```