AN1328
APPLICATION NOTE

I²C COMMUNICATION PROTOCOL WRITTEN IN FUZZYSTUDIO™4.0 FOR ST52X430

Author: G. Rascona'

1. INTRODUCTION

In this application note we implement two routines to read/write one single byte of data at a time from/into an EEPROM memory (in this case the ST24x04) using ST52x430 microcontroller. The protocol used to set up this communication link is the well-known I²C (Inter Integrated Circuit), in which the master is always the micro, while the memory is always the slave.

The procedures can be used with all the ST52 micros and with all the memories compatible with the one we used for this application. Eventually, some changes have to be done on the software to match different device characteristics, but due to the program structure flexibility, these are very easy to perform.

The routines were created with FUZZYSTUDIO™4.0 and are easy to be read and structured for future extensions and improvements. Moreover, with some care, they can be imported directly in any user program written with FUZZYSTUDIO™4.0 and ready to be used to build up the communication link.

2. THE ST24X04 EEPROM MEMORY

2.1 Functional Description

The ST24x04 is a 4Kbit Electrically Erasable Programmable Memory (EEPROM), organized as 2 blocks of 256 x8 bits. These devices are compatible with the I²C standard two-wire serial interface, which uses a bi-directional data bus and serial clock. The memories carry a built-in 4 bit, unique device configuration identification code (1010) corresponding to the I²C bus definition. This is used together with 2 chip enable inputs (E1, E2) so that up to 4 x4K devices may be attached to the I²C bus and selected individually. The memories behave as a *slave device* in the I²C protocol with all memory operations synchronized by the serial clock. Read and write operations are initiated by a START condition generated by the bus master.

The START condition is followed by a stream of 7 bits (identification code 1010) plus one read/write bit and terminated by an acknowledge bit. When writing data to the memory it responds to the 8 bits received by asserting an acknowledge bit during the 9th bit time. When the data is read by the bus master, it acknowledges the receipt of the data bytes in the same way. Data transfers are terminated with a STOP condition.

In Figure 1 the memory logic diagram is shown. Note that in our application note some functionalities of the memory are not implemented and, then, some pins are of not concern. For major details refer to the memory datasheet.

Figure 1. Logic Diagram
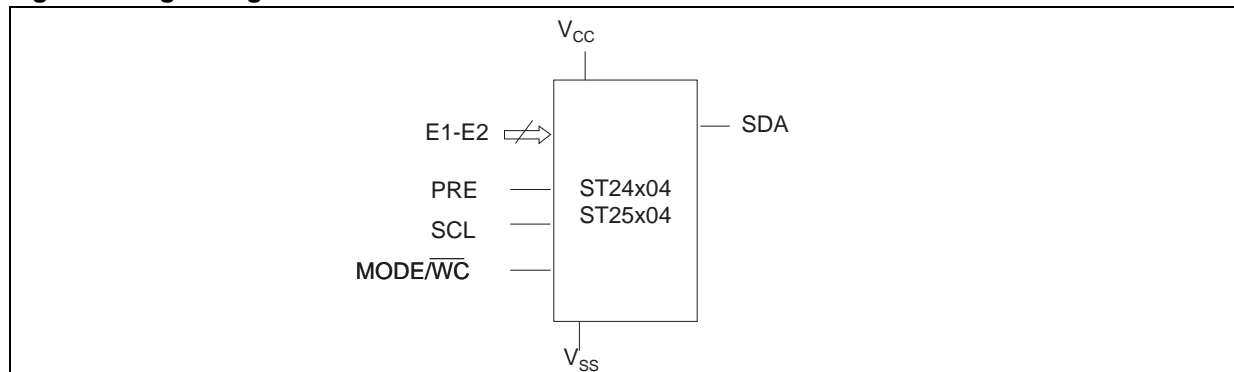
Note: $\overline{WC}$ signal is only available for ST24/25W04 products

2.2 Signal Description

As it was said previously, this application note implements the more simple way to set up a communication between the micro and an EEPROM, letting to the final user an eventual extension of all other functionalities this type of memories allow. Below, you will find a brief description of all the signals used in our implementation with no reference to the different modes to write and protect the memory.

Serial Clock (SCL). The SCL input pin is used to synchronize all data in and out of the memory. A resistor can be connected from the SCL line to Vcc acting as a pull up (see Figure 2).

Figure 2. Maximum R_L Value versus Bus Capacitance (C_{BUS}) for an I²C Bus
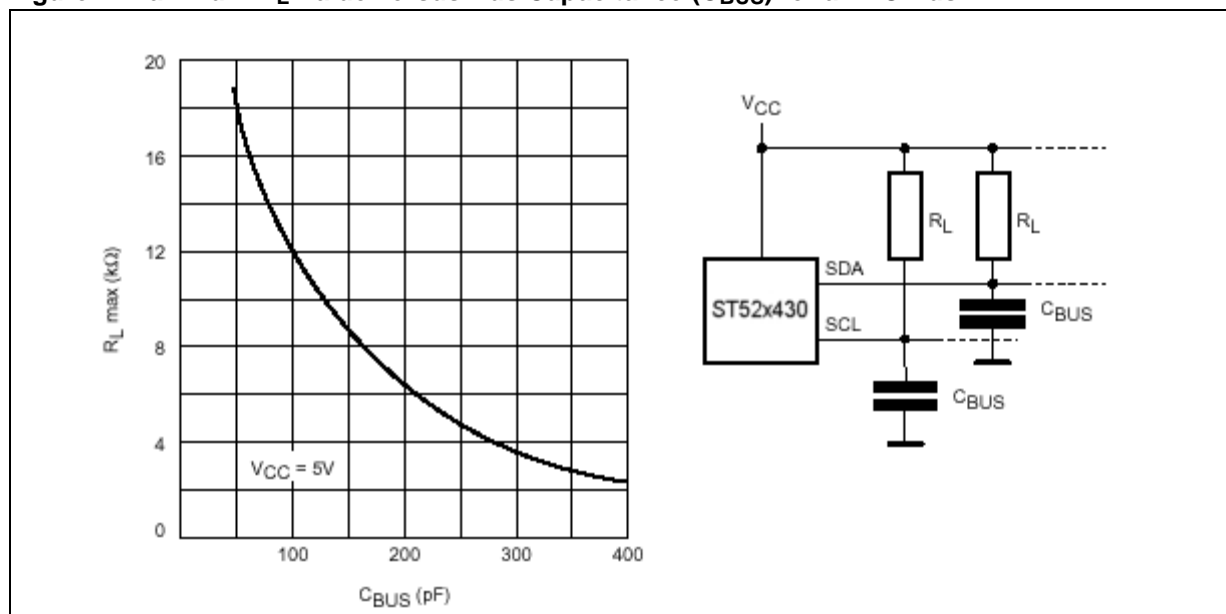
Serial Data (SDA). The SDA pin is bi-directional and is used to transfer data in or out of memory. It is an open drain output that may be *wire-OR'ed* with other open drain or open collector signals on the bus. A resistor must be connected from the SDA bus line to Vcc to act as pull up (see Figure 2).

Chip Enable (E1 - E2). This chip enable inputs are used to set the 2 least significant bits (b2,b3) of the 7-bit device select code. These inputs may be driven dynamically or tied to *Vcc* or *Vss* to establish the device select code. With these two pins, up to 4 memory chips, sharing the same bus, can be addressed.

3. I²C BUS BACKGROUND

ST24x04 supports the I²C protocol. This protocol defines any device that sends data onto the bus as a *transmitter* and any device that reads the data as a *receiver*. The device that controls the data transfer is known as the *master* and the other as the *slave*. In our case the master will be the microcontroller and it will always initiate a data transfer and will provide the serial clock for synchronization. The memories are always slave devices in all communications.

Start condition

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

Stop condition

STOP is identified by a low to high transition of the SDA line while the clock SCL is stable in the high state. A STOP condition terminates communication between the memory and the bus master. A STOP condition at the end of a READ command, after and only after a NO ACKNOWLEDGE, forces the standby state. A STOP condition at the end of a WRITE command triggers the internal EEPROM write cycle. Figure 3 shows the bus and clock sequences for START and STOP commands.

Figure 3. Bus Timing START/STOP
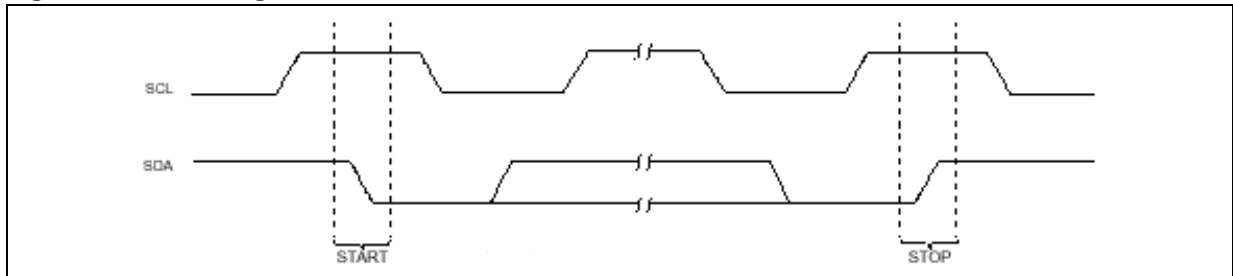
Acknowledge bit (ACK)

An acknowledge signal is used to indicate a successful data transfer. The bus transmitter, either master or slave, will release the SDA bus after sending 8 bits of data. During the 9th clock pulse period the receiver pulls the SDA bus **low** to acknowledge the receipt of the 8 bits of data.

Data Input

During data input the memory samples the SDA bus signal **on the rising edge** of the clock SCL. Note that, for correct device operation, 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.

Memory Addressing

To start communication between the bus master and the slave ST24x04, the master ST52x430 must initiate a START condition. Following this, the master sends onto the SDA bus line 8 bits (MSB first) corresponding to the *device select code* (7 bits) and a *R/W* bit. The figure 4 below shows the bit format of this byte called "*device selector*".
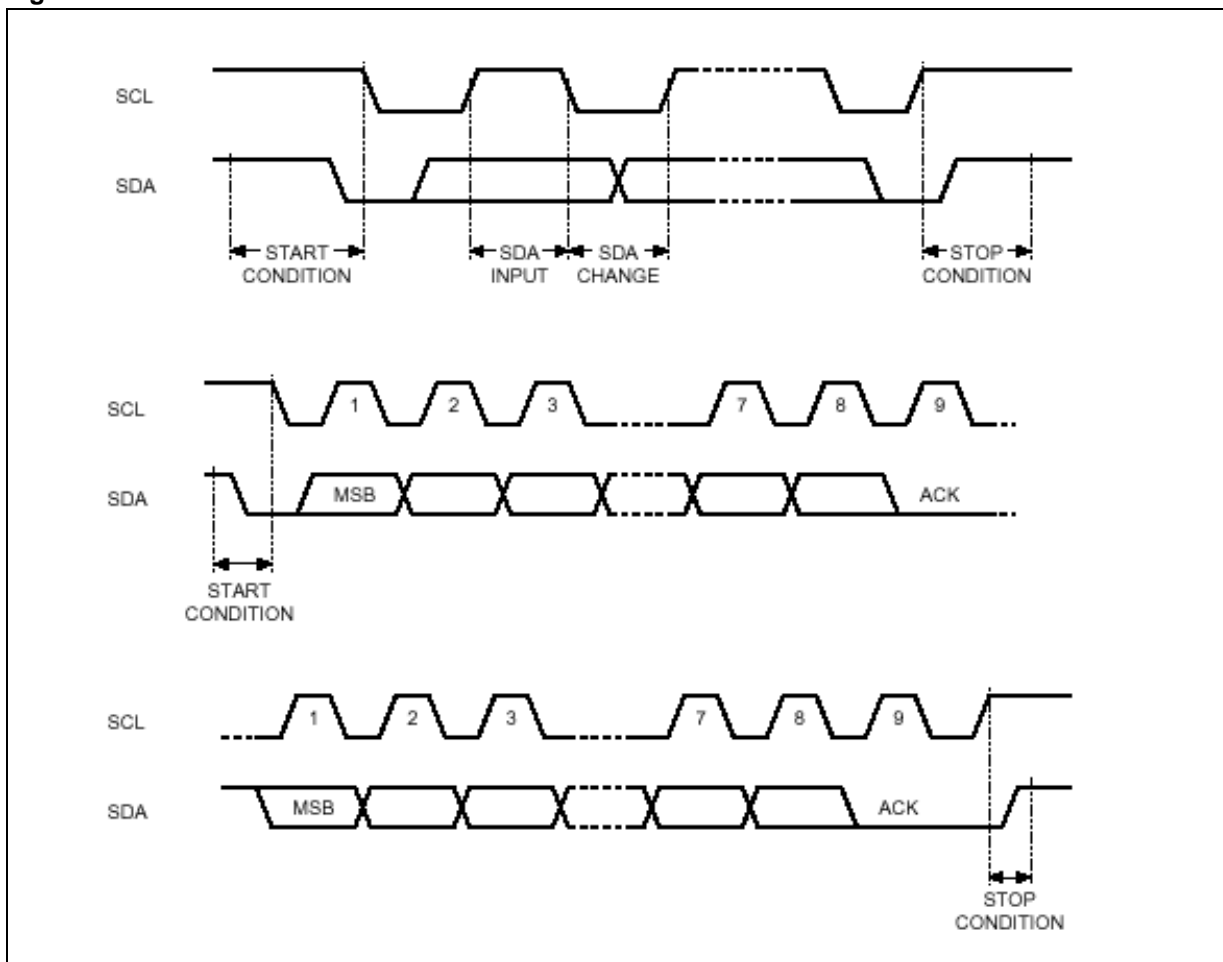
If a match is found, the corresponding memory will acknowledge the identification on the SDA bus during the 9th bit time.

Figure 4. Device select byte
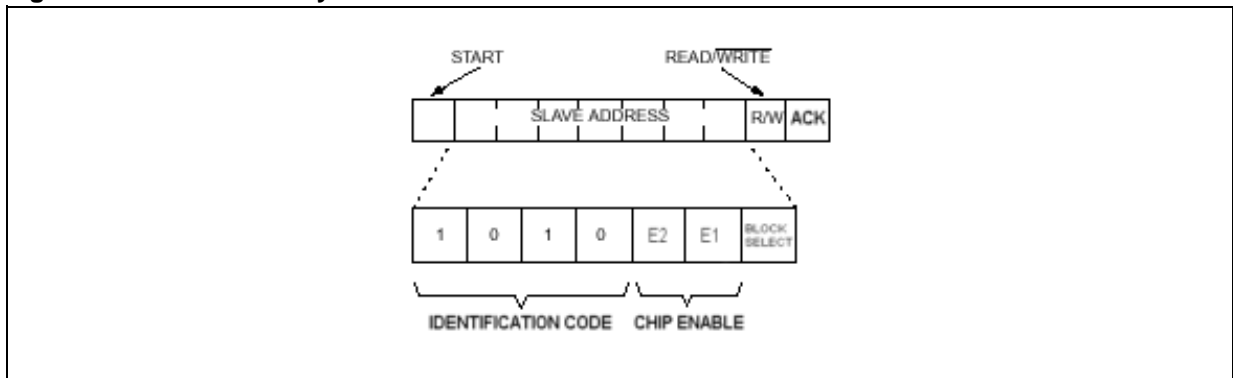
In figure 5 the fundamentals of I^2C protocol is drawn. For the AC timing please refer to the relative datasheet.

Figure 5. I²C Bus Protocol

Byte Write

Although there are different modes for writing more than one byte into the memory with a single command, in this example we implemented the *Single Byte Write* operation. With a few modifications it is then possible to get the *Multybyte* and *Page Write* operations allowed by the I²C protocol (see the ST24C04 datasheet for further information).

Following a START condition, the master sends a device select code with the R/W bit reset to '0'. The memory acknowledges this and waits for a byte address. The byte address of 8 bits provides access to one block of 256 bytes of the memory. After receipt of the byte address, the device again responds with an acknowledge.

After that, the master sends one data byte, which is acknowledged by the memory. The master then terminates the transfer by generating a STOP condition. At this point the *internal memory program cycle* starts: all inputs are disabled until the completion of this cycle and the memory will not respond to any request.

Byte Read

Also in this case we have different ways to read the memory content. Here, what has been implemented is the *Random Address Read*, that allows to read a byte from a specified memory address. To this aim, a dummy write is sent to the memory to load the address into the *address counter*. This is followed by another START condition from the master and the device selector is repeated with the R/W bit set to '1'. The memory acknowledges this and outputs the byte addressed. The master has to NOT acknowledge the byte output, but terminates the transfer with a STOP condition.

The two sequences for *Byte Write* and *Random Byte Read* are reported in Figure 6.

Figure 6. Byte Write and Read sequences
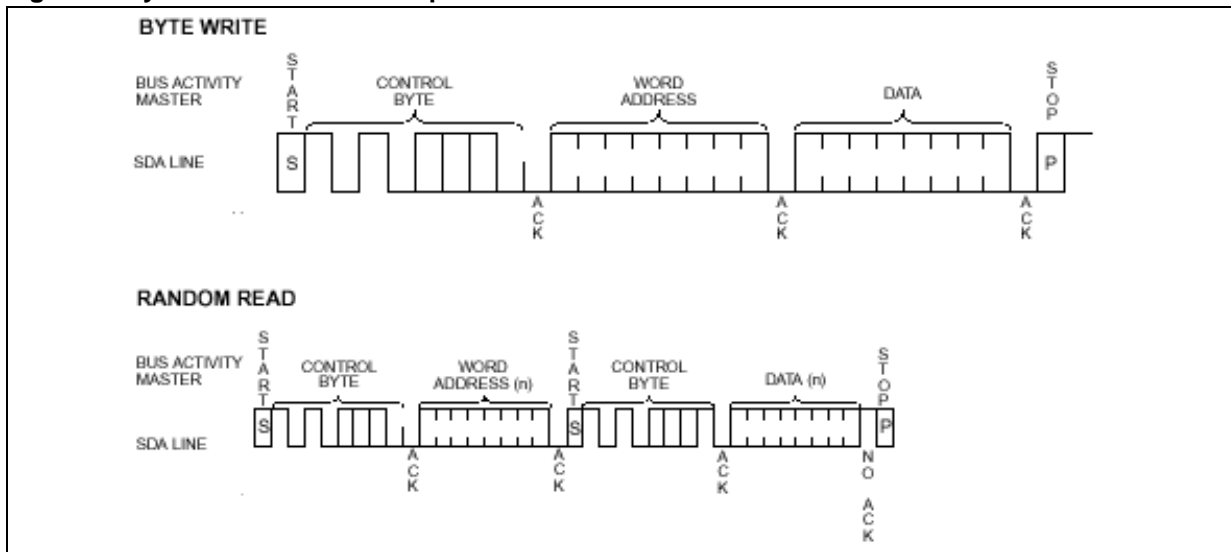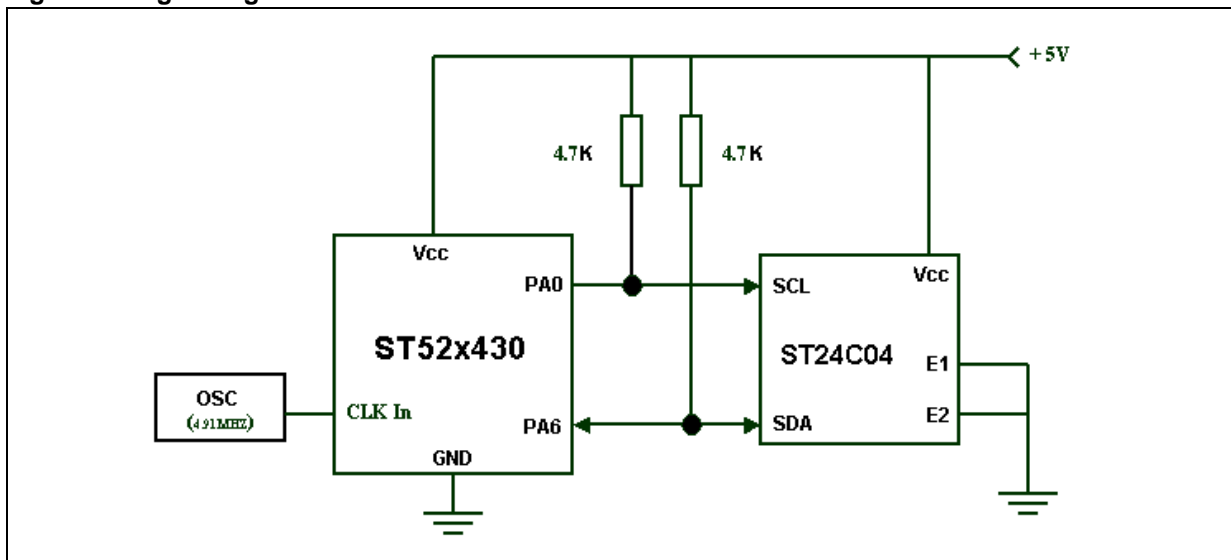

4. HARDWARE DESCRIPTION

In Figure 7 there is a scheme that shows the simple connections between the micro ST52x430 and ST24C04 memory. To reduce consumption, the two pull-up resistances suggest to configure the two pins of port A as input when the memory is not in use.

Using these procedures, the user has to reserve the two pins PA0 and PA6 to allow the communication link. Since the procedures are generalized, the user program can configure the other pins of port A according to its preferences. The configuration will be maintained at the end of execution of each routine.

The resistance values are strictly connected with the bus capacitance, according to the graphic shown in Figure 2. This influences also the maximum speed reachable in the serial communication. For further information about this, consult the memory datasheet.

The two chip enables are statically connected to ground. This means that, for addressing this memory chip, we have to put the value '0' in bits *b3* and *b2* of *device selector*.

Figure 7. Logic Diagram

5. SOFTWARE DESCRIPTION

We analyze each procedure for reading and writing a byte, separately, although they use some subroutines that are commons. In fact, as every operation in the memory begins with a write command (to specify the device selector byte and the address), the *Read Procedure* must contain also the code that allows writing into the memory.

Another important point to discuss is that the two routines are done in such a way that any user program can import them directly with few modifications. The price to pay for this end is to reserve a user-variable to keep track of the configuration register relative to the I/O PORT_A (Configuration Register #4) of the micro (this is an hardware register that is only writable from the user but not readable). Acting in this manner we will be able to modify the I/O configuration only for the involved pins (PA0 and PA6) without changing the configuration of other six pins. Because inside the routine the direction of the PA6 is continually changed (input or output), what the user has to do is to initialize the reserved variable (named REG_CONF4_IM) with the wanted configuration of the PORT_A. At the end of each routine execution, the PORT_A will restore its original configuration.

Finally, as the procedures were conceived to work inside a host program, it would be better that these routines could be interrupted during their execution when an interrupt signal (internal or external) is generated by the micro. At this end, the FUZZYSTUDIO^{TM}4.0 provides different ways to do this. See the documentation for major details about this.

5.1 Program WRITE

The main program sets the value for the auxiliary register REG_CONF4_IM (in our case we suppose that the PORT_A is all configured as input) and, then, what has just to do is to provide the value for the variable *Address* (from 0 to 255) and the value for the *Data*.

Then it calls the subroutine *Write* (Figure 8).

Figure 8. Data specification and call to "Write" subroutine
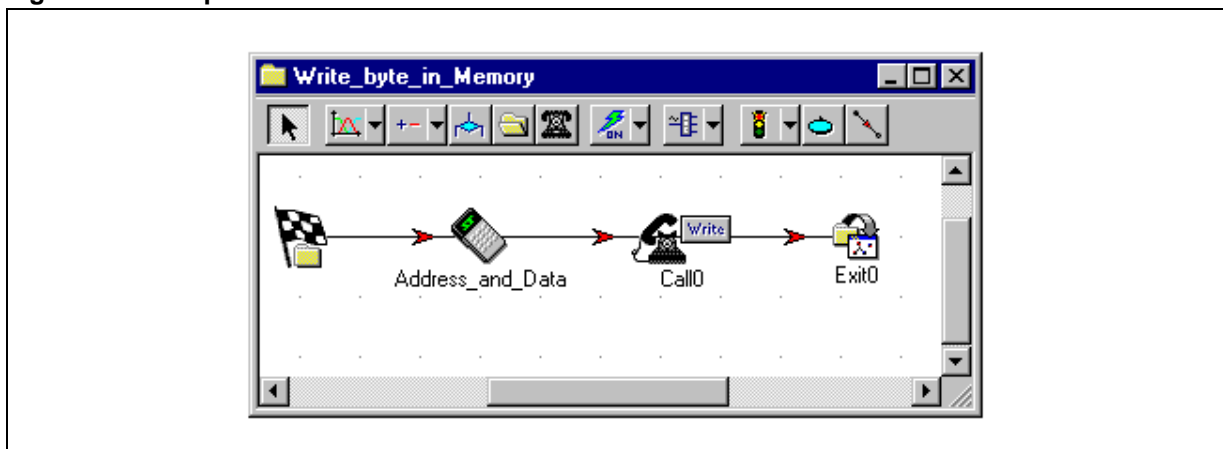
From the user point of view, the operation for writing into the memory is over.

Now, let us have a look as the *Write* subroutine is organized (Figure 9). First of all, the subroutine *Start1* is called to begin the communication between the two devices. After that, the "*device selector*" is loaded into the variable *Byte* (acting like a data source for the following subroutine) and then the subroutine *Write_Byte* is called. To this part of code is demanded the actual transmission of each bit constituting the data, generating the appropriate clock (SCL) and data bus (SDA) signals.

Figure 9. The "Write" subroutine
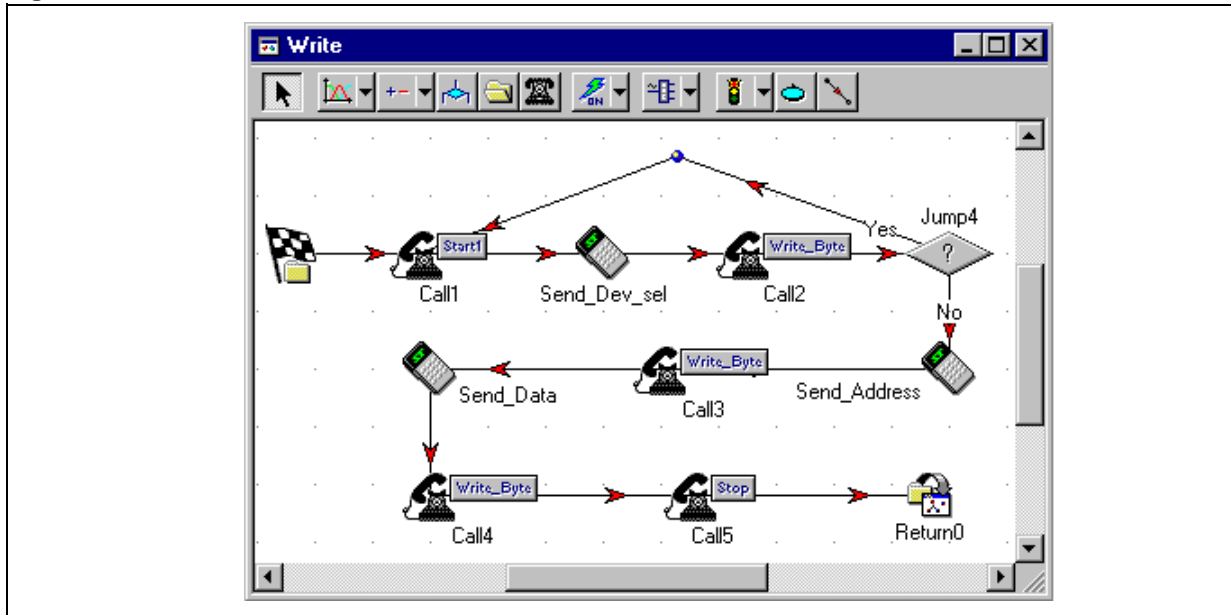
Now, the program waits for an acknowledge from the memory. In this phase of communication (that is, following a *Start* command), the memory could be busy, as it could be completing a previous operation of write (recall that during an internal write cycle the memory is unusable). As the memory datasheet specifies, if there is no acknowledge, the master has to restart and re-transmit the device selector. If acknowledged, then the master sends the address specified by *Address* and, then, the *Data*. For the two successive *Write* subroutines, the program doesn't check for an acknowledge although the 9th clock cycle has to be generated in any case. At the end, the *Stop* subroutine ends the communication process.

The *Write_Byte*, *Start1* and *Stop* subroutines are described and commented in the in the FuzzyStudio™4.0 project called "*ByteWrite.fs4*" you can download from the web.

5.2 Program READ

The main program sets the value for the auxiliary register REG_CONF4_IM (in our case we suppose that the PORT_A is configured as input) and, then, what the user has to do is just provide the value for the variable *Address* (from 0 to 255) whose content has to be read and loaded in the variable *Data*.

As only the *Random Read mode* was implemented, the *Read* procedure is very similar to that of *Write*. Hence, as shown in Figure 10, we note that the first part of the program is like the one analyzed above. In fact, after the device selector byte (with the bit R/W set to 0) has been acknowledged, we have to send to the memory a dummy byte specifying the address to be read. After that, we send a device selector with the R/W set to 1 and then the micro configures PA6 in input to read the content of the selected address.

Figure 10. The "Read" subroutine
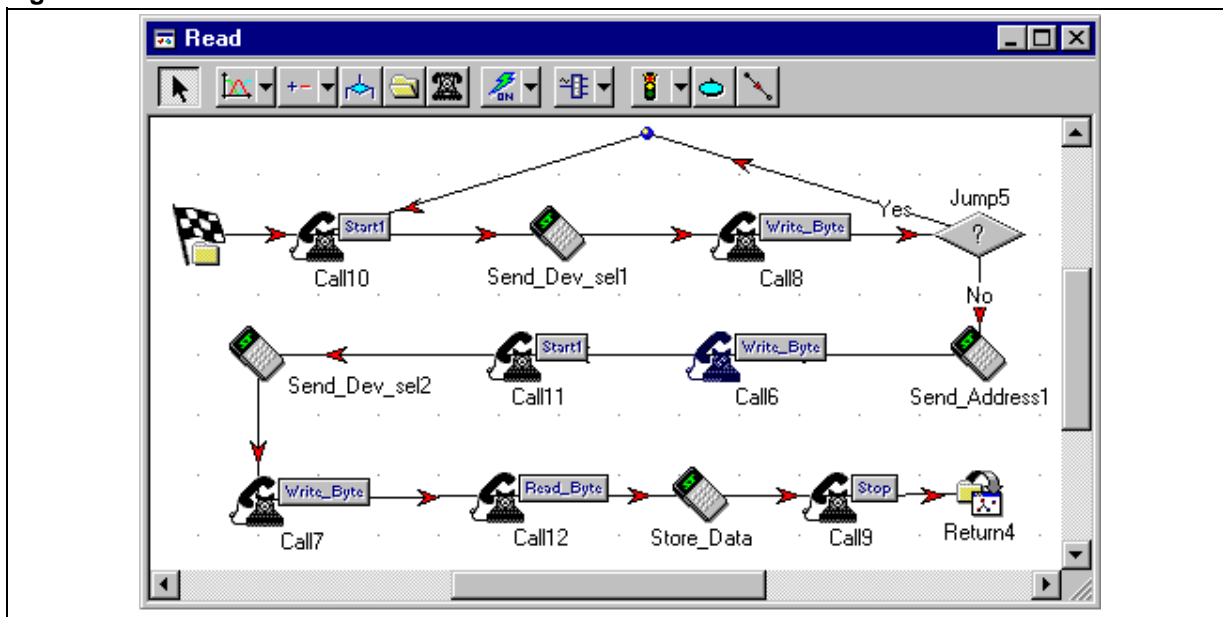
The *Read_Byte* subroutine is described and commented in the FUZZYSTUDIO™4.0 project "*ByteRead.fs4*" you can download from our web site.

6. TIMES

Here are reported the times need to perform a write and read cycle.

T_write = 2.24 ms

and

T_read = 3.02 ms

These times are intended from the START to STOP commands. They has been measured using a logic data analyzer with the micro working at **4.91MHz** frequency (oscillator frequency).

Note that the write cycle time does not include the t_W (**internal write cycle**) time needed to the memory for doing its internal operation after receiving a STOP command. This time depends only by the memory device and it can be 10 ms in the worst case.

7. CONCLUSION

In this application note the basics to realize an I^2C communication protocol are provided.

The structure of the two routines to write and read the byte to/from the memory was planned to be recalled easily in a user program without doing significant changes on it. As the procedures have been written in FUZZYSTUDIO^{TM}4.0, they result very easy to be understood and any extension implementing the other features of the I^2C protocol is easy to be developed starting from the described code.