ELECTRONICS FOR EMBEDDED SYSTEMS PROJECT by DANIELE CASTRO s253244

The present document shows the other two complementing mini-projects I have made to let you know my knowledge in:

- Programmable Logic Devices (PLD) and VHDL final state machine

- Communication protocols (SPI)

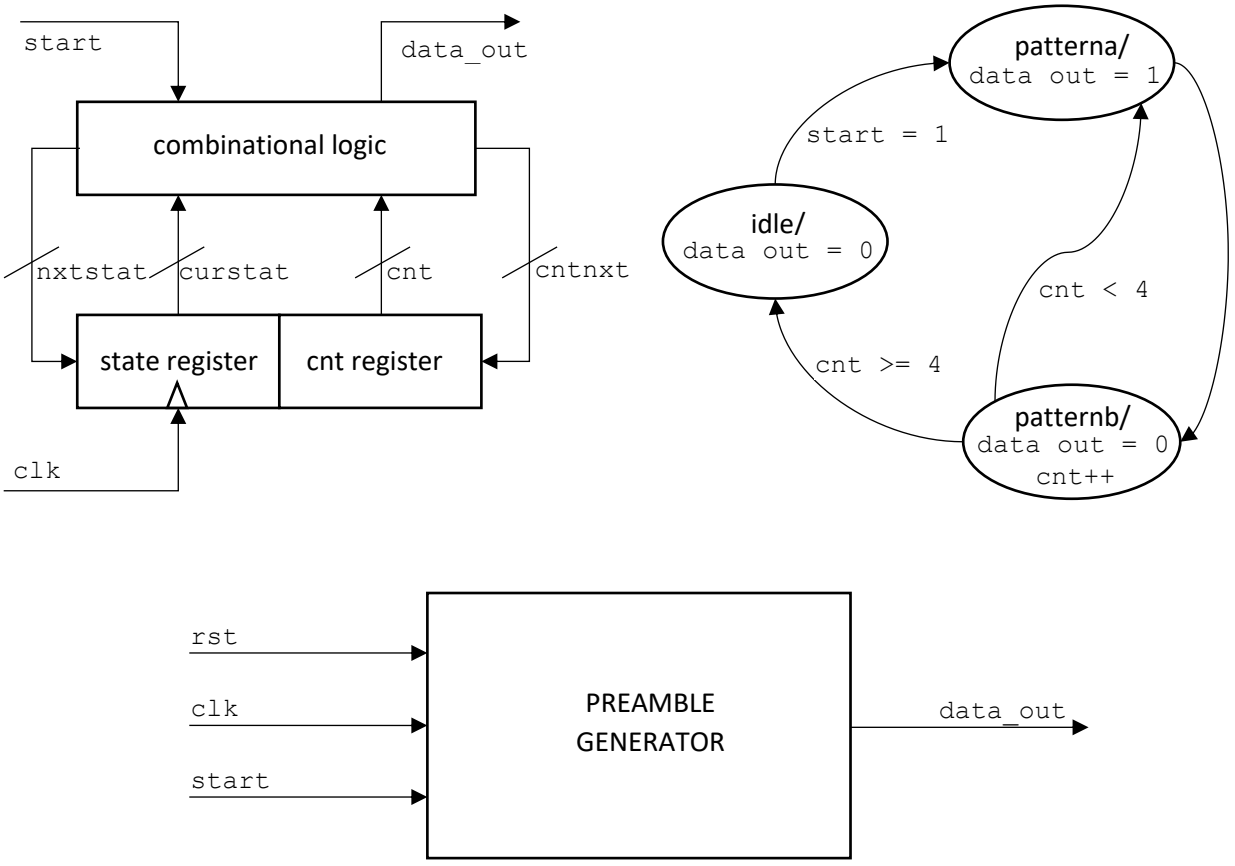
- Power management (PWM)

Programmable Logic Device:

I have developed a High Level State Machine (HLSM) that implements a preamble generator for ethernet II protocol. I have used a Moore machine model for representing it.

This machine will generate "10101010" every time `start` signal is asserted. Reset is asynchronous.

Here is the RTL design with the Final State Machine:



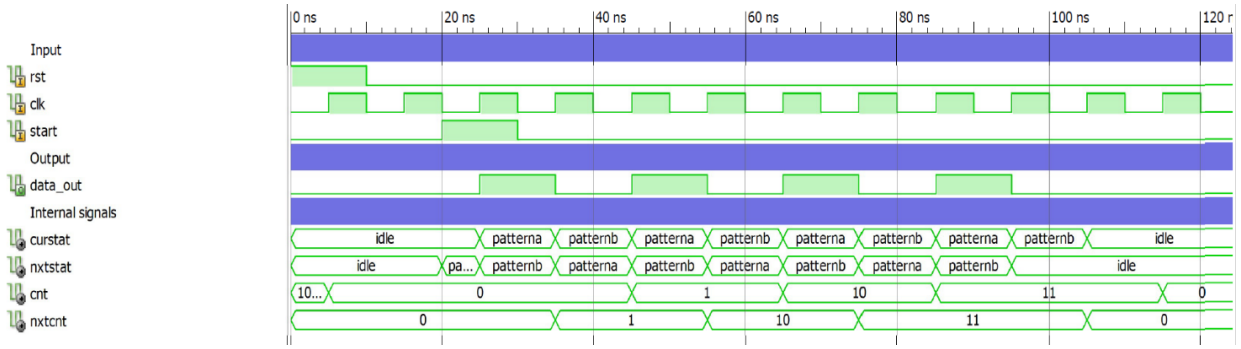
States explanation: the initial machine state is idle and the output `data_out` is low. Then, when start signal goes high, current state (`curstat`) moves to `patterna` where the output `data_out` is high. At this point we move unconditionally to `patternb` where the output `data_out` is again low. Now we check if `cnt` register is less then 4. If it is, we increment `cnt` and go back to `patterna` and repeat the cycle above described. Otherwise we go back to `idle`.

This automaton is represented in VHDL as a two processes machine. The first process represents respectively the "state register" that moves the machine through the various states and the "cnt register"

that increments the counter. The second and last process represents the “combinational logic” that defines the machine behavior based on the above listed input signals.

```
library ieee;
use ieee.std_logic_1164.all;

entity preamble is
	port(
		rst, clk, start : in std logic;
		data_out, rst_o, clk_o, start_o : out std_logic
	);
end preamble;
architecture beh of preamble is
type states is (idle, patterna, patternb);
signal curstat, nxtstat : states;
signal cnt, nxtcnt : integer;
begin
rst_o <= rst;
clk_o <= clk;
start_o <= start;
	process(rst, clk)
	begin
		if(clk'event and clk = '1') then
			if(rst = '1') then
				curstat <= idle;
				cnt <= 0;
			else
				curstat <= nxtstat;
				cnt <= nxtcnt;
			end if;
		end if;
	end process;
	process(curstat, cnt, start)
	begin
		nxtcnt <= cnt;
		case curstat is
		when idle =>
			if(start = '1') then
				nxtstat <= patterna;
			else
				nxtstat <= idle;
			end if;
			data_out <= '0';
			nxtcnt <= 0;
		when patterna =>
			nxtstat <= patternb;
			data_out <= '1';
		when patternb =>
			if(cnt >= 3) then
				nxtstat <= idle;
			else
				nxtstat <= patterna;
				nxtcnt <= cnt + 1;
			end if;
			data_out <= '0';
		end case;
	end process;
end beh;
```

For testing the machine I have designed a dsPIC33FJ128GP802 program that generates the same testbench’s input signals pattern.



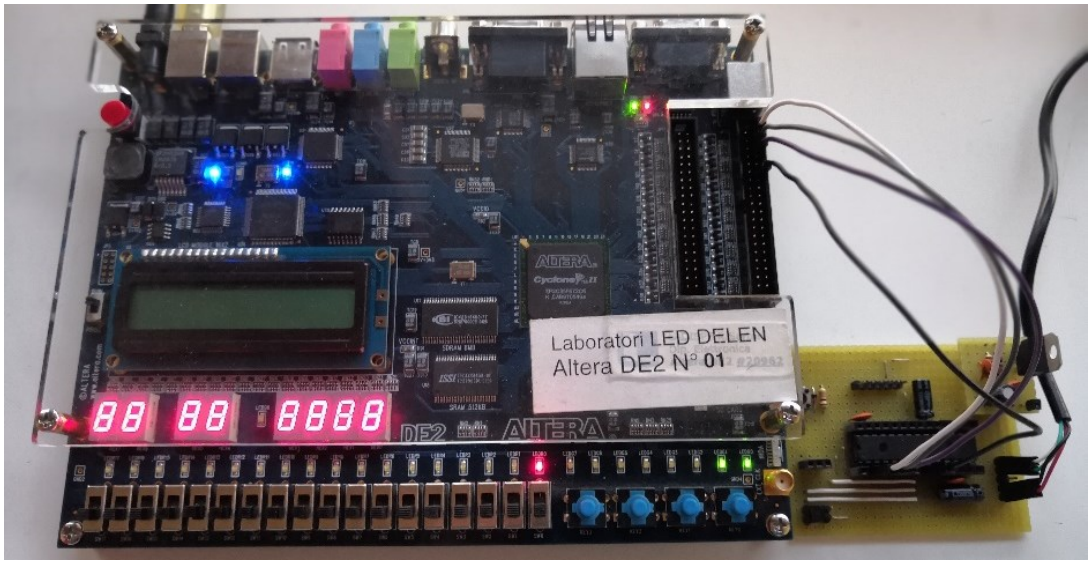
The dsPIC program uses the serial port for displaying the simulation state. The user has only to press “return” every time he wants to toggle `clk` and the other input signals in the DE2 board where the VHDL code above explained is fisically implemented.

COM9 - PuTTY

Session Special Command Window Logging Files Transfer Hangup ?

```
RESET
Digitare man mano un tasto per far progredire la simulazione.
rst     clk     start
1       0       0
1       1       0
0       0       0
0       1       0
0       0       1
0       1       1
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
0       0       0
0       1       0
Fine simulazione.
Digitare man mano un tasto per far progredire la simulazione.
```

Results and input signals will be shown through four DE2 board’s LEDs.



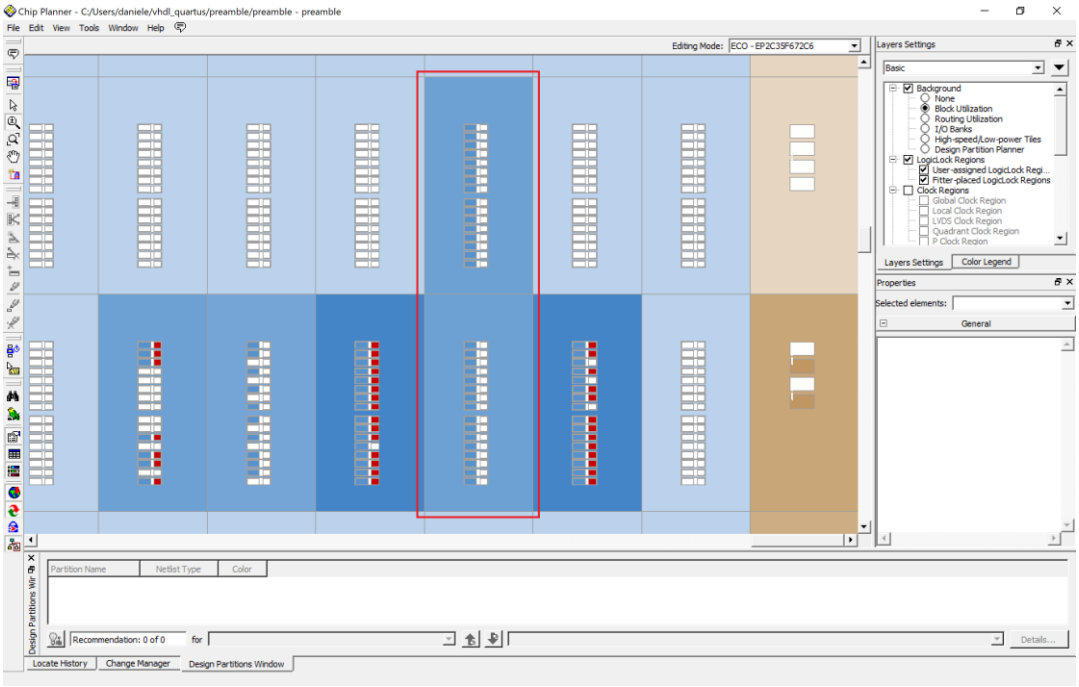
Pin mapping on the DE2 board is:

```
clk, Input, PIN_M23
clk_o, Output, PIN_AF22
data_out, Output, PIN_AE23
rst, Input, PIN_K26
rst_o, Output, PIN_W19
start, Input, PIN_M20
start_o, Output, PIN_AE22
```

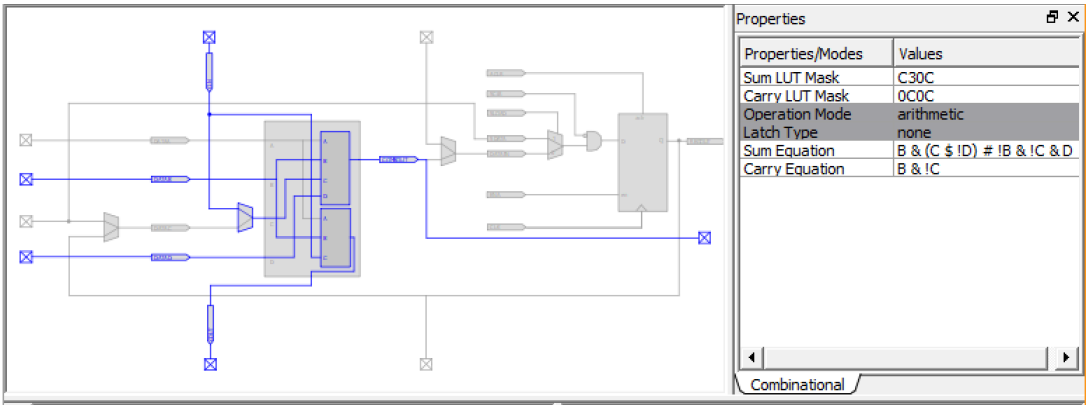
On dsPIC33FJ128GP802:

```
rst, Output, RA0
clk, Output, RA1
start, Output, RB0
```

Concerning the hardware implementation automatically made by the tool Quartus II on the Cyclon II FPGA, we can observe this internal structure:



In particular we can observe the highlighted part in which we can notice a big use of logic cells in arithmetic mode instead of the normal one:



This increases the working speed and this was possible thanks to the FSM implementation. The automatic tool can translate FSM designs more easily than the other ones.

The component's maximum working speed resulted in 131.84 MHz. Assigning pins in a more efficient way will surely increase this clock speed even more because of the PAD distance from the internal logic core.

Communication protocol:

For the last two parts I have made a single PIC18F26K22 program that implements both SPI protocol reading and writing strings to three 23LCV1024 in one SPI bus and a PWM signal with selectable duty cycle used for driving a simple DC motor using the L293D full H bridge. This device will also use serial port for interfacing with the user.

This time, to set quickly all the necessary registers, I have used the MPLAB Code Configurator tool (MCC) by Microchip. In this way I could concentrate exclusively to the SPI and PWM protocol.

As I said above I'll use three 1Mb NVRAM. Those RAMs can automatically switch using V_{cc} or V_{bat} if no V_{cc} is applied. They can also work both with 5 or 3.3 volts. They have three different working modes:

- "Byte Operation" for reading or writing a single byte

- "Page Operation" for reading or writing page blocks

- "Sequential Operation" for reading or writing the entire memory array starting from a specified address.

We are interested only in the "Sequential Operation" mode.

The device has also the possibility to use SDI (Serial Dual Interface) protocol to transfer data instead of SPI.

The 23LCV1024 has some binary commands for selecting what said above.

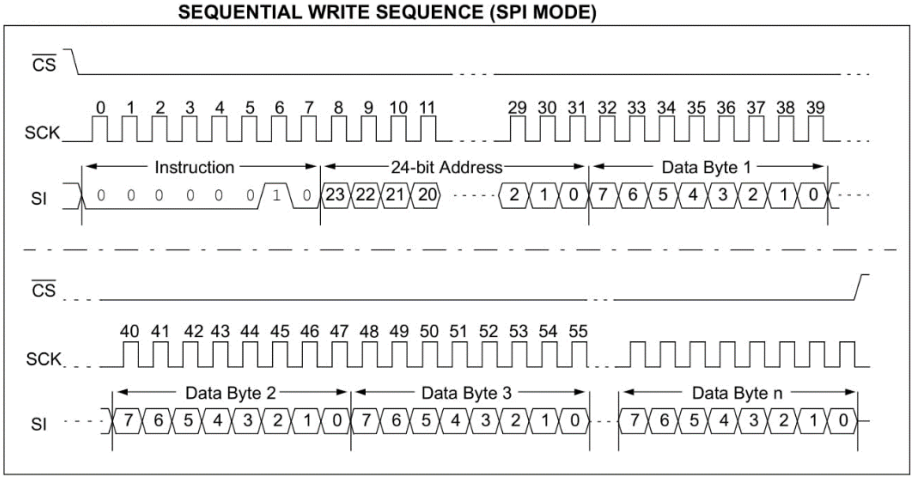
INSTRUCTION SET

Instruction Name	Instruction Format	Hex Code	Description
READ	0000 0011	0x03	Read data from memory array beginning at selected address
WRITE	0000 0010	0x02	Write data to memory array beginning at selected address
EDIO	0011 1011	0x3B	Enter Dual I/O access
RSTIO	1111 1111	0xFF	Reset Dual I/O access
RDMR	0000 0101	0x05	Read Mode Register
WRMR	0000 0001	0x01	Write Mode Register

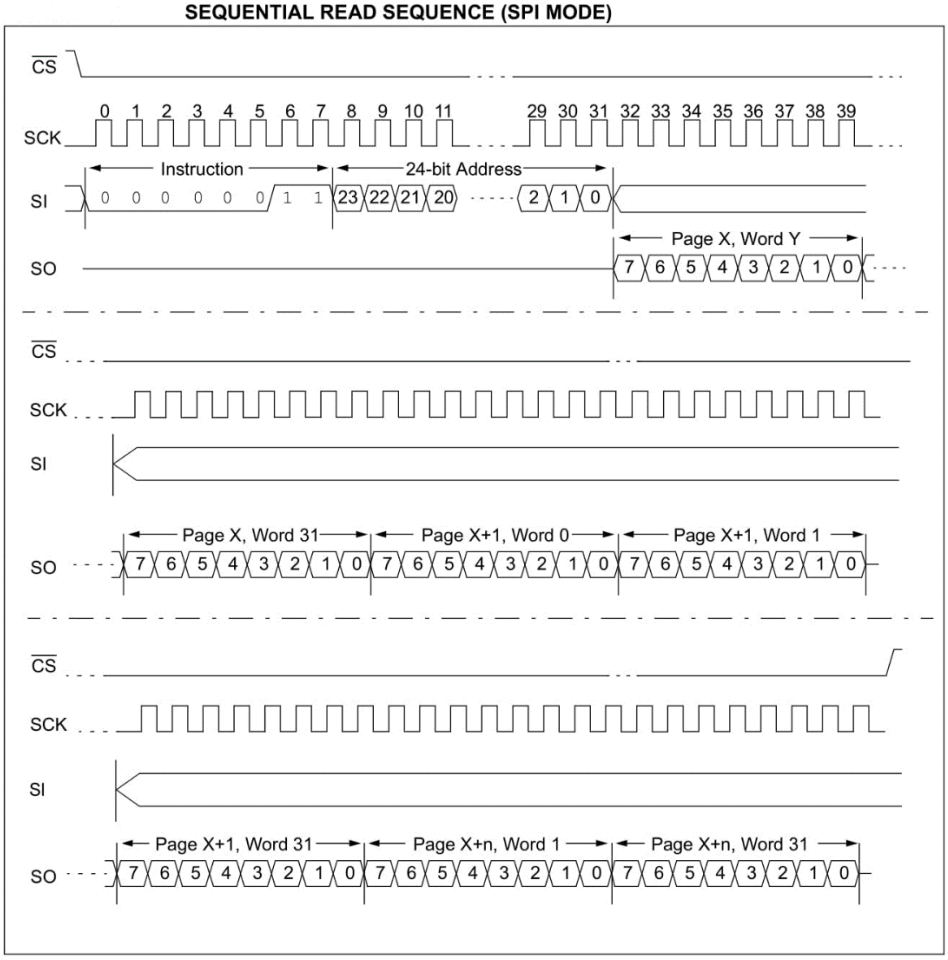
We are interested only in READ, WRITE and RSTIO for resetting into SPI mode.

The datasheet shows timing both for writing and reading in "Sequential Operation" mode:

SEQUENTIAL WRITE SEQUENCE (SPI MODE)



SEQUENTIAL READ SEQUENCE (SPI MODE)



It also shows the SPI timing that will result in SPI mode 0:

SERIAL INPUT TIMING (SPI MODE)

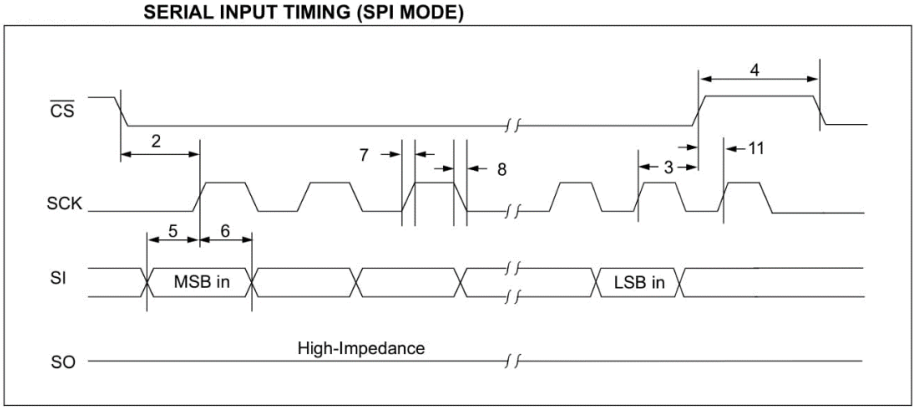
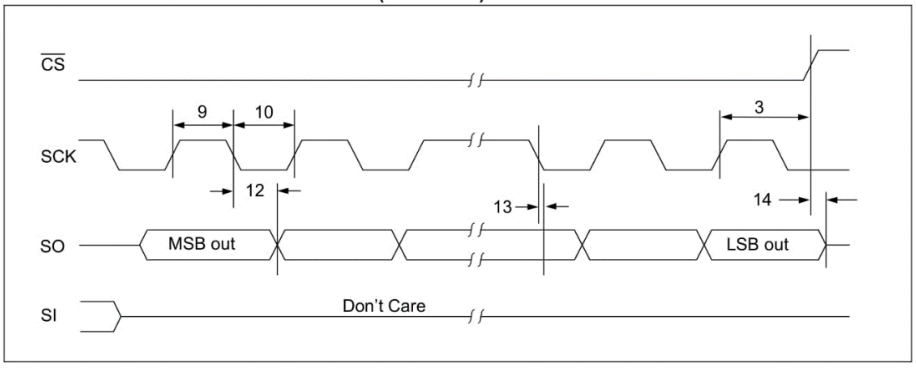


FIGURE 1-2: SERIAL OUTPUT TIMING (SPI MODE)



Once peripherals have been correctly configured by MCC I have written the routines for reading and writing from these RAMs. MCC has generated these two functions automatically:

```
uint8_t SPI1_Exchange8bit(uint8_t data) {

    // Clear the Write Collision flag, to allow writing
    SSP1CON1bits.WCOL = 0;

    SSP1BUF = data;

    while (SSP1STATbits.BF == SPI_RX_IN_PROGRESS) {
    }

    return (SSP1BUF);
}
```

- It retrieves and sends one byte (exchanges a byte)

```
uint8_t SPI1_Exchange8bitBuffer(uint8_t *dataIn, uint8_t bufLen, uint8_t *dataOut) {
    uint8_t bytesWritten = 0;

    if (bufLen != 0) {
        if (dataIn != NULL) {
            while (bytesWritten < bufLen) {
                if (dataOut == NULL) {
                    SPI1_Exchange8bit(dataIn[bytesWritten]);
                } else {
                    dataOut[bytesWritten] = SPI1_Exchange8bit(dataIn[bytesWritten]);
                }

                bytesWritten++;
            }
        } else {
            if (dataOut != NULL) {
                while (bytesWritten < bufLen) {
                    dataOut[bytesWritten] = SPI1_Exchange8bit(DUMMY_DATA);

                    bytesWritten++;
                }
            }
        }
    }

    return bytesWritten;
}
```

- It simply bufferizes the above `uint8_t SPI1_Exchange8bit(uint8_t data)` function.

So, the following three functions I have written will:

```
void RAM_set_SPI_mode(volatile unsigned char* latch, uint8_t pin_number) {
    *latch &= ~(1 << pin_number); //metto CS in low per avviare i trasferimenti
    SPI1_Exchange8bit(0b11111111); //comando RSTIO
    *latch |= (1 << pin_number); //metto CS in high per bloccare i trasferimenti
}
```

- send the RSTIO (`0b11111111`) command for resetting the SPI mode instead of the SDI one putting CS in low during transfers that single byte transfer.

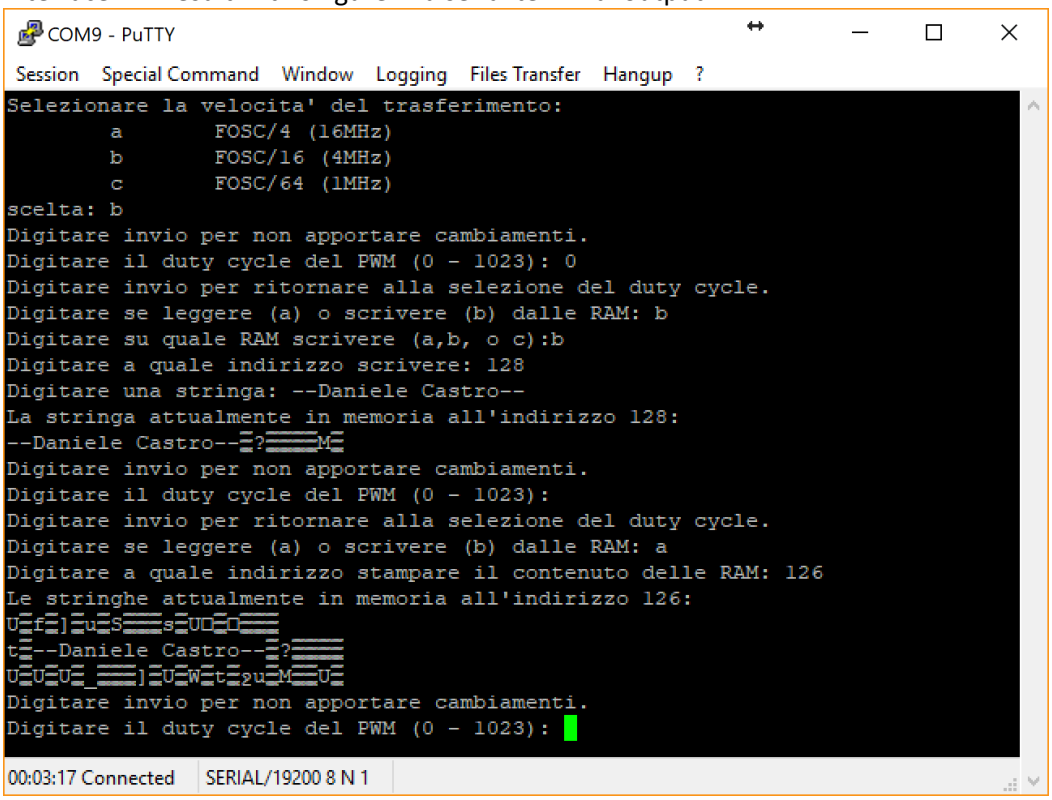
```
void RAM_sequential_SPI_write(volatile unsigned char* latch, uint8_t pin_number, uint24_t address,
uint8_t *data_out, uint8_t data_out_size) {
    uint8_t address_array[3];
    address_array[2] = address;
    address_array[1] = address >> 8;
    address_array[0] = address >> 16;
    *latch &= ~(1 << pin_number); //metto CS in low per avviare i trasferimenti
    SPI1_Exchange8bit(0b00000010); //comando WRITE
    SPI1_Exchange8bitBuffer(address_array, 3, NULL);
    SPI1_Exchange8bitBuffer(data_out, data_out_size, NULL);
    *latch |= (1 << pin_number); //metto CS in high per bloccare i trasferimenti
}
```

- split the 24-bit `address` into three 8-bit long `address_array[…]` variables, so they can be sent through SPI using `SPI1_Exchange8bitBuffer(address_array, 3, NULL);` after WRITE command has sent through `SPI1_Exchange8bit(0b00000010);` clearly CS is low during all the transfers. Data to be sent must be in `data_out` with specified length in `data_out_size`.

```
void RAM_sequential_SPI_read(volatile unsigned char* latch, uint8_t pin_number, uint24_t address,
uint8_t *data_in, uint8_t data_in_size) {
    uint8 t address array[3];
    address_array[2] = address;
    address_array[1] = address >> 8;
    address_array[0] = address >> 16;
    *latch &= ~(1 << pin_number); //metto CS in low per avviare i trasferimenti
    SPI1_Exchange8bit(0b00000011); //comando READ
    SPI1 Exchange8bitBuffer(address array, 3, NULL);
    SPI1_Exchange8bitBuffer(NULL, data_in_size, data_in);
    *latch |= (1 << pin_number); //metto CS in high per bloccare i trasferimenti
}
```

- split the 24-bit `address` into three 8-bit long `address_array[…]` variables, so they can be sent through SPI using `SPI1_Exchange8bitBuffer(address_array, 3, NULL);` after WRITE command has sent through `SPI1_Exchange8bit(0b00000010);` clearly CS is low during all the transfers. Retrieved data will be in `data_in` with the specified length in `data_in_size`.

Serial interface will result in this figure in a serial terminal output:

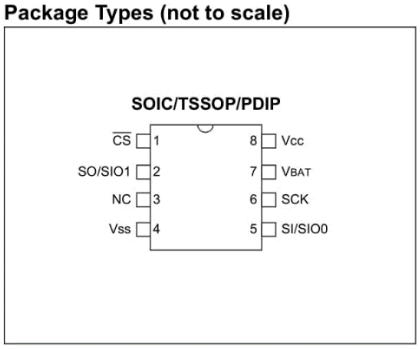


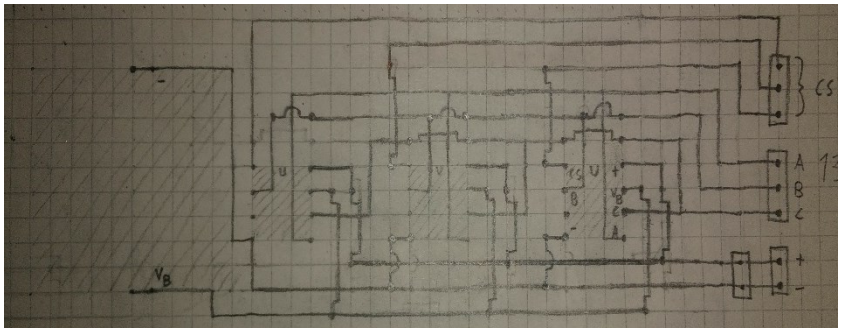
The datasheet says that these memories cannot go faster than 20MHz. This is true in a SPI bus with multiple SPI slaves. But in a single slave bus I was able to achieve 64MHz.

Strange characters are random values coming from the yet unprogrammed memory.

Below the 23LCV1024 footprint and my 3Mb SPI NVRAM bank for perfored boards:

Package Types (not to scale)

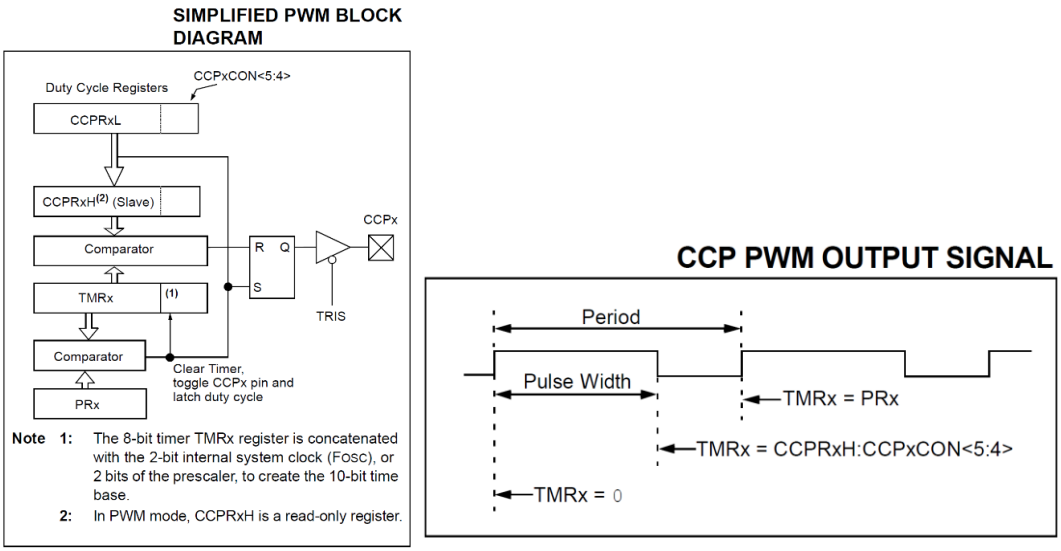




Power management:

As I said before the PWM (Pulse With Modulation) signal will be generate from the same PIC18F26K22 circuit board discussed above and I will also configure CCP4 (CAPTURE/COMPARE/PWM MODULE 4) again using MCC.

Below the simplified block diagram of the CCP4 module in PWM working mode from the datasheet and a simple explanation of the registers involved in the control of the PWM period and duty cycle:



As shown in the figures the CCP4 module will use timer 2 (TMR2) module for controlling the PWM period/frequency. TMR2 register is the timer value and PR2 register will contain the value, in the range of a 16-bit value, at which the timer must go back to 0. CCPR4H and CCPR4CON are MSB and LSB registers in which will be stored the value at with the signal will go from high to low. In other words the duty cycle. This value must fit in 10-bit.

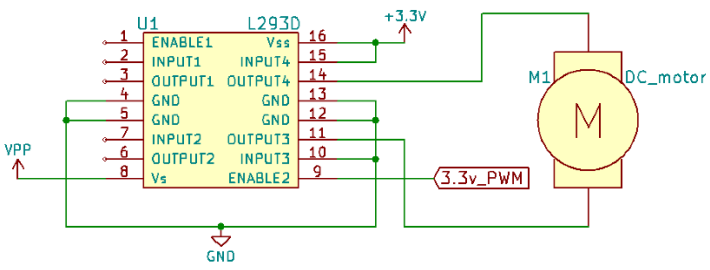
PR2 was set to 0xFFFF so the timer period will be 4.096 ms because, starting from 64MHz of system clock, I have set both timer prescaler and postscaler at 1:16 divisor value.

MCC has generated this function automatically and this let me easily control the duty cycle:

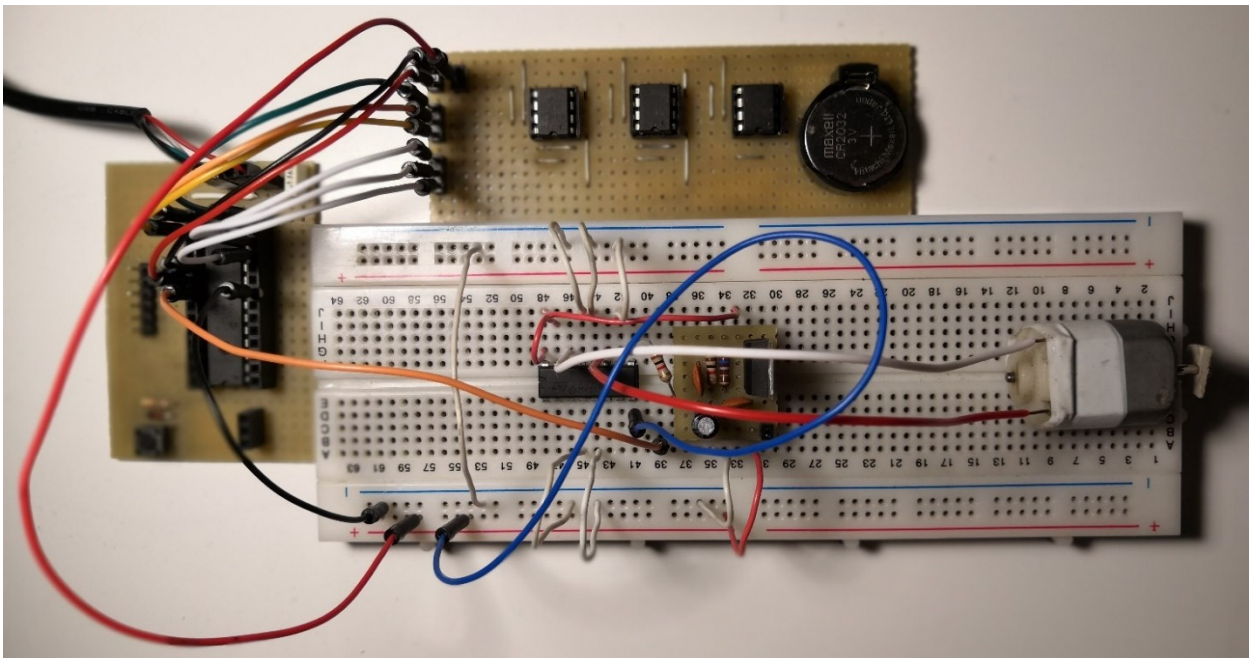
```
void PWM4_LoadDutyValue(uint16_t dutyValue) {
    // Writing to 8 MSBs of pwm duty cycle in CCPRL register
    CCPR4L = ((dutyValue & 0x03FC) >> 2);

    // Writing to 2 LSBs of pwm duty cycle in CCPCON register
    CCP4CON = (CCP4CON & 0xCF) | ((dutyValue & 0x0003) << 4);
}
```

In the above figure it is possible to see the serial interface asking to input the desired duty cycle.

For controlling the DC motor I have used the L293D full H bridge like in this schematic:

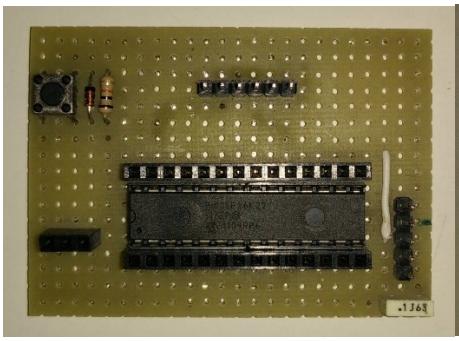


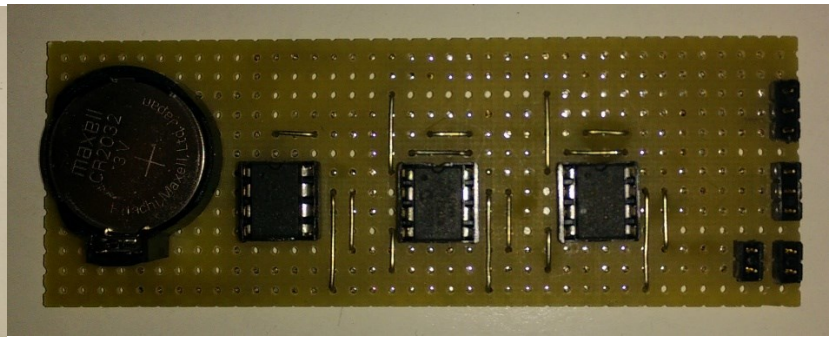
Here is shown the complete circuitry I have made:



In the left the PIC18F26K22 board. On the right the breadboard with the L293D, the motor and my 3,3 volt power supply module also used in my dsPIC and PIC32 boards. At the top of the picture the 3 Mb SPI NVRAM module.

Here are the two modules of the above picture:





The PIC18F26K22 board project for perfored board: