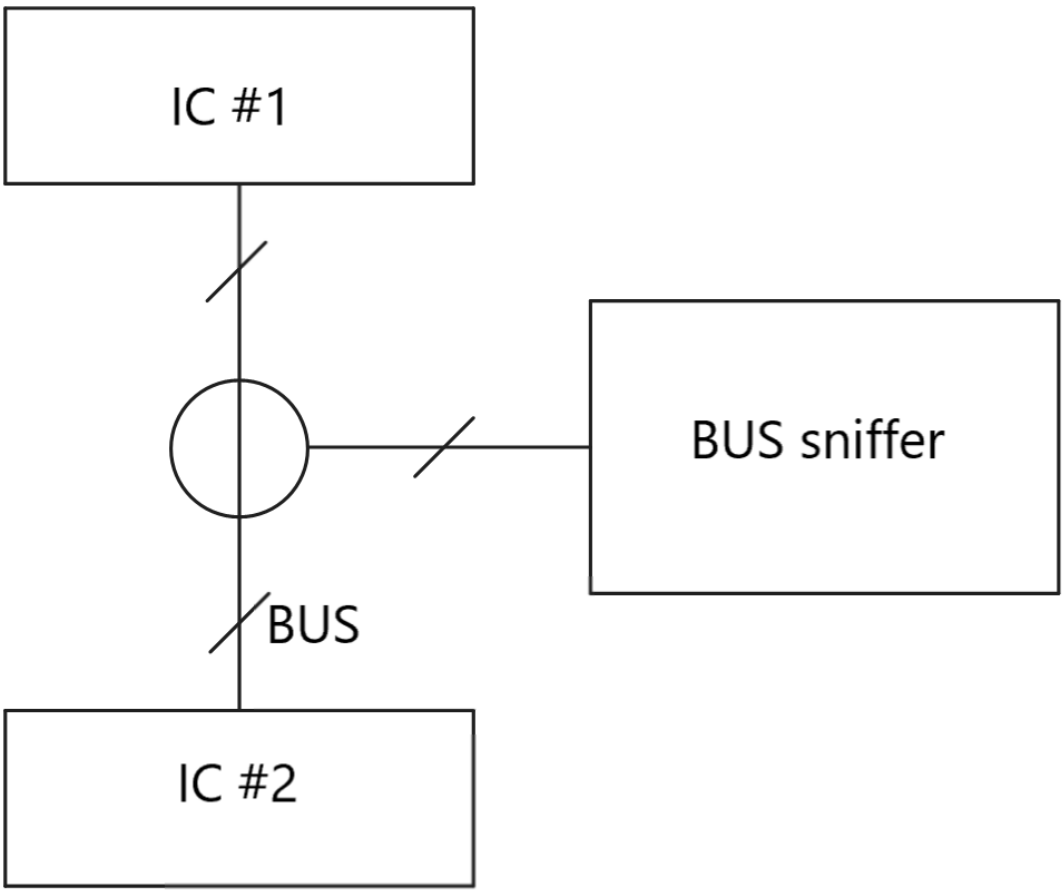
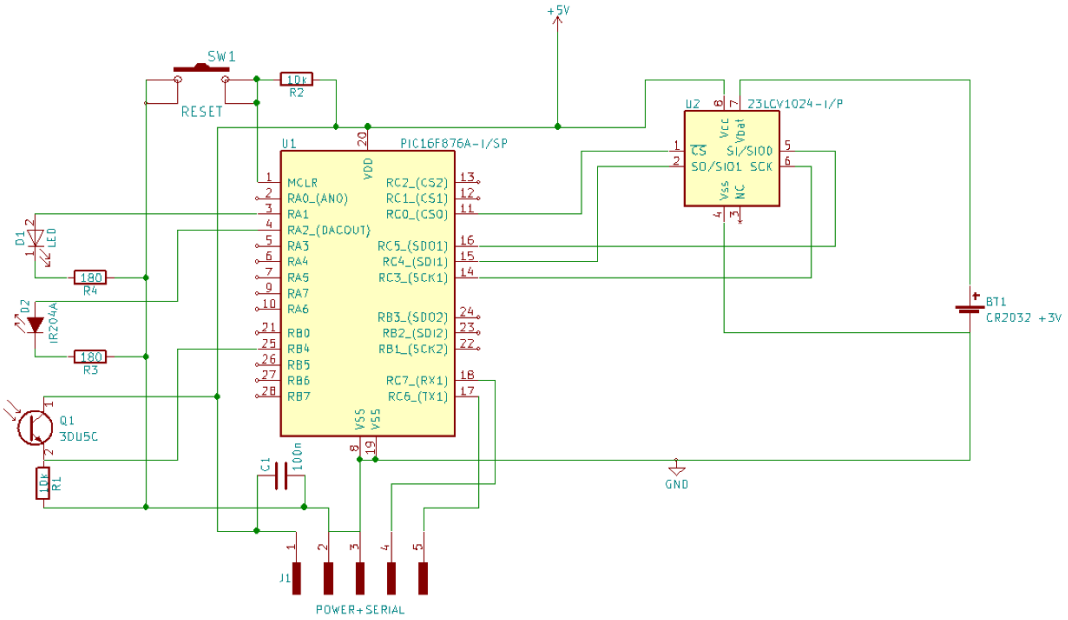
Microproject number 1

This project is a bus sniffer: it records bits transmitted trough a serial or parallel bus and retransmits back the recorded pattern.



Can be helpful in reverse engineering situations where we don't know anything about the communication protocol of the ICs in the same circuitry and we simply want to copy and repeat a chosen pattern. In this case the project is implemented using a PIC16F876A. Because of the low number of pins (IC package is a PDIP with 28 pin) we will use only one GPIO and, consequently, a single bit of a 8-bit GPIO register. Clearly the code is made for being as scalable as possible. In this way moving to a bigger IC package or to another circuit will be easy and there will be the possibility to use more pins of the GPIO register. In this situation one of the best communication BUS we can sniff can be that one between a phototransistor and the rest of the circuit of a common TV. That's because of IR protocols uses relatively low transmitting frequencies. So we can test the sniffer even if it's not working at full speed. Consequently to these decisions the testing hardware of the sniffer will be a phototransistor connected to one input of the microcontroller, an infrared LED connected to an output of the IC and a normal LED for visually verify that the circuit is working. In addition I will use an NVRAM 23LCV1024 for being able to store more than 48 byte of samples in memory (the maximum PIC16F876A contiguous allocable memory uint8_t or uint16_t).

Here we can see the full circuit:

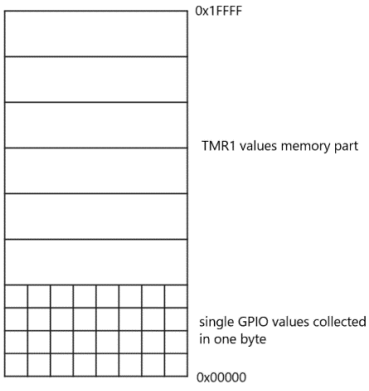


Let's now take a look at the code:

Time distance between a change of edge in the input line and the other is measured with a 16 bit timer (TMR1). In the 23LCV1024 NVRAMS only one byte at time can be sent. I decided to organize the sample frame for the memories using this simple structure:

GPIO_high_or_low_bit (remaining 7 bits are wasted)	TMR1 register high part	TMR1 register low part

Surely for every sample, 7 bits will be wasted out but 23LCV1024 are cheap and we have really a lot of space (0x20000 = 131072 that means 0x1FFFF = 131071 of address space). I originally thought to organize memory in such a way that those bits will be used. I also begin to write code and I immediately discovered that for saving those bits I have to complicate very a lot code and, in this way, I also have to introduce more calculations delay because of readdressing of memory and bit ORing for storing multiple bit in a single byte with each bit referring to a different TMR1 value stored in different part of the memory array. The final result was a low performance circuit with a non scalable code: those 7 unused bits can be used in future for storing an entire 8 bit GPIO register. Here the image representing the memory struct with the approach described above.



Change of edge in the RB4 pin is detected through a special function of PORTB GPIO register that rises an interrupt on every change. In this way the code is interrupt driven and precise. Unfortunately the structure of the interrupt routine that stores in memory the sample frame is a kind of unrolled. That's because of performance. We can notice in the code the first write call in the main function, when recording, and the others in the interrupt service routine because of this reason.

PIC16F876A does not have any timer period register so I have to subtract 0xFFFF to each recorded value in the memory array. In this way on play time I can start the TMR1 register from the subtraction result and timer will rise an interrupt on normal overflow at 0xFFFF instead of at a value in an eventual period register. I decide to make the above subtraction at the and of recording time in a separate loop so recording time will not be affected by the computation of the subtraction in terms of speed.

Let's now take a look at the 23LCV1024 structure and how I configured the SPI (MSSP) peripheral: 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.

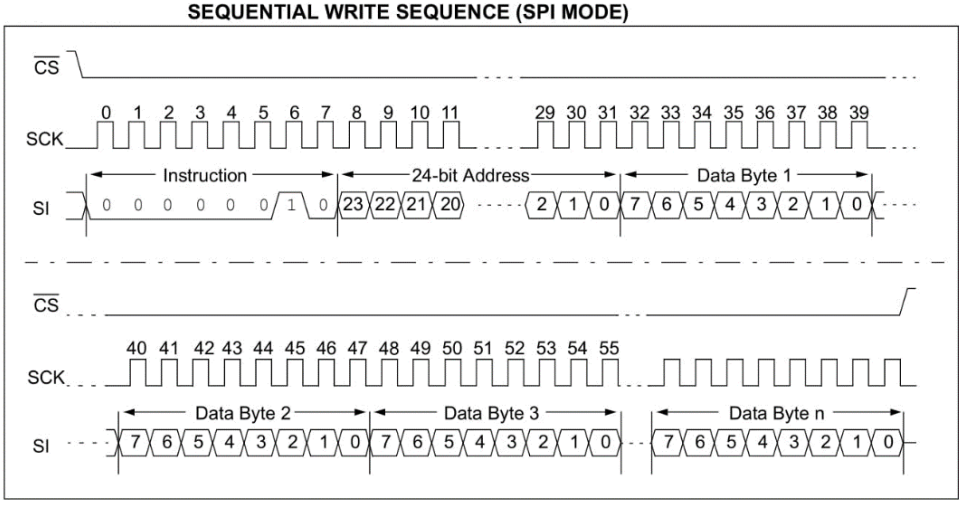
INSTRUCTION SET

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

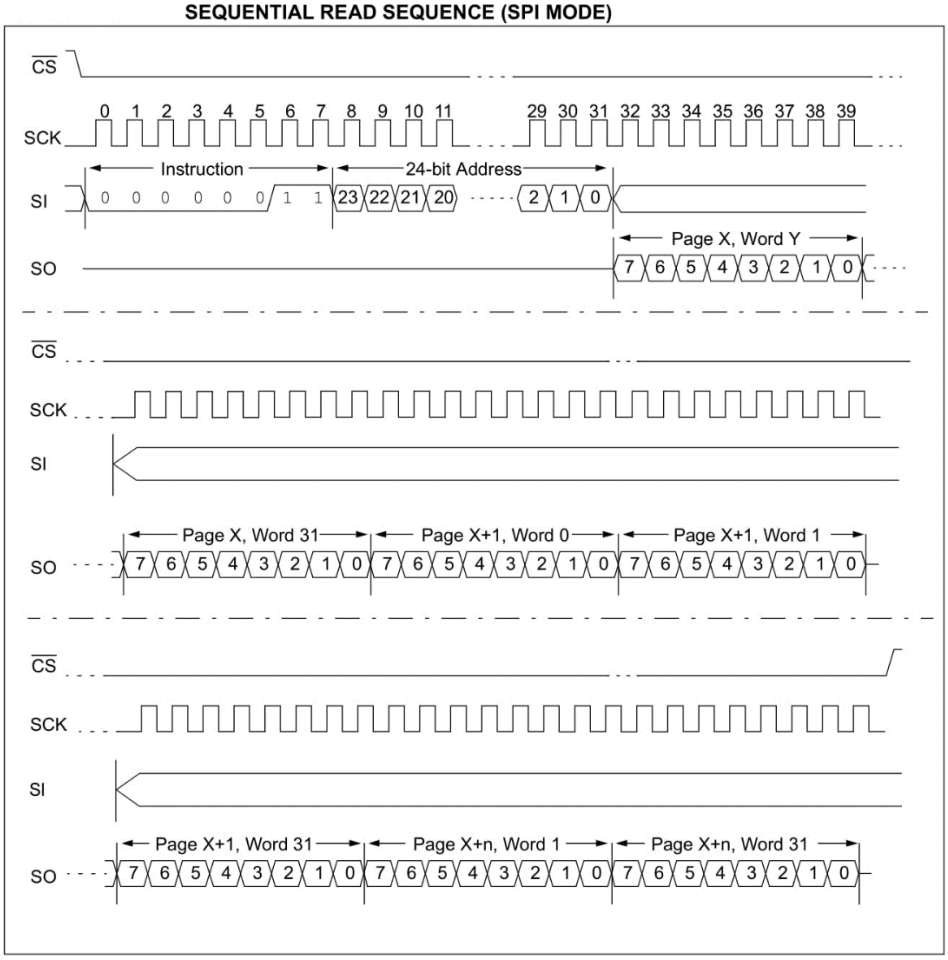
We are interested only in READ, WRITE and RSTIO for resetting into SPI mode.

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)

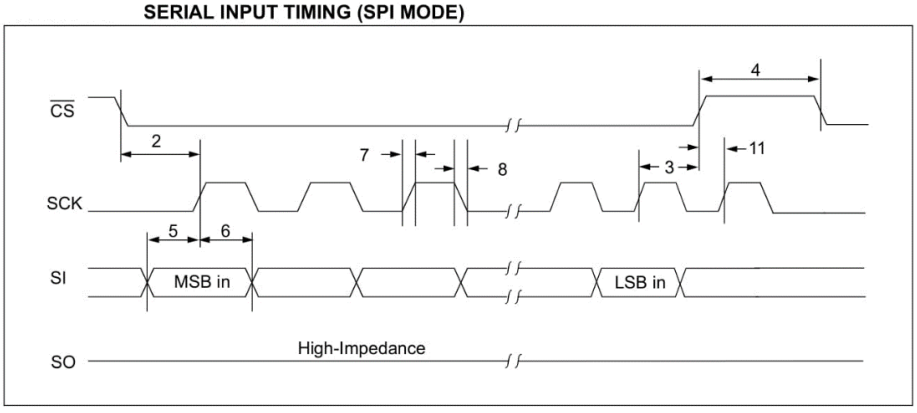
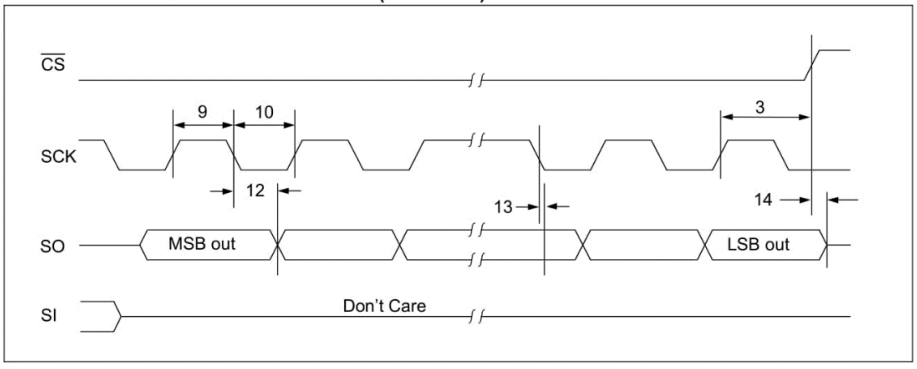


FIGURE 1-2: SERIAL OUTPUT TIMING (SPI MODE)



I originally used MCC automatically generated routines for reading/writing in those memories, but I then decided to compact the code and remove many abstraction layers to create more performant routines. I'll firstly explain the MCC automatically generated ones and then I show the new ones I made:

```
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 previously 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`.

The new routines I made do the same thing but avoiding a lot of control flow commands (if/else) and unrolls some statically dimensioned loops:

```
void RAM_set_SPI_mode(volatile unsigned char* latch, uint8_t pin_number) {
    uint8_t dummy = 0;
    *latch &= ~(1 << pin_number); //set SS low for enabling transfers
    //send RSTIO command
    SSPCONbits.WCOL = 0;
    SSPBUF = 0b11111111;
    while (SSPSTATbits.BF == 0);
    dummy = SSPBUF;
    *latch |= (1 << pin_number); //set SS high for disabling transfers
}

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 i, dummy = 0;
    *latch &= ~(1 << pin_number); //set SS low for enabling transfers
    //send WRITE command
    SSPCONbits.WCOL = 0;
    SSPBUF = 0b00000010;
    while (SSPSTATbits.BF == 0);
    dummy = SSPBUF;
    //send address
    SSPCONbits.WCOL = 0;
    SSPBUF = address >> 16;
    while (SSPSTATbits.BF == 0);
    dummy = SSPBUF;
    SSPCONbits.WCOL = 0;
    SSPBUF = address >> 8;
    while (SSPSTATbits.BF == 0);
    dummy = SSPBUF;
    SSPCONbits.WCOL = 0;
    SSPBUF = address;
    while (SSPSTATbits.BF == 0);
    dummy = SSPBUF;
    //sed data
    for (i = 0; i < data_out_size; i++) {
        SSPCONbits.WCOL = 0;
        SSPBUF = data_out[i];
        while (SSPSTATbits.BF == 0) {}
        dummy = SSPBUF;
    }
    *latch |= (1 << pin_number); //set SS high for disabling transfers
}
```

```
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 i, dummy = 0;
    *latch &= ~(1 << pin_number); //set SS low for enabling transfers
    //send READ command
    SSPCONbits.WCOL = 0;
    SSPBUF = 0b00000011;
    while (SSPSTATbits.BF == 0);
    dummy = SSPBUF;
    //send address
    SSPCONbits.WCOL = 0;
    SSPBUF = address >> 16;
    while (SSPSTATbits.BF == 0);
    dummy = SSPBUF;
    SSPCONbits.WCOL = 0;
    SSPBUF = address >> 8;
    while (SSPSTATbits.BF == 0);
    dummy = SSPBUF;
    SSPCONbits.WCOL = 0;
    SSPBUF = address;
    while (SSPSTATbits.BF == 0);
    dummy = SSPBUF;
    //retrieve data
    for (i = 0; i < data_in_size; i++) {
        SSPCONbits.WCOL = 0;
        SSPBUF = dummy;
        while (SSPSTATbits.BF == 0) {}
        data_in[i] = SSPBUF;
    }
    *latch |= (1 << pin_number); //set SS high for disabling transfers
}
```

The only thing that was not optimizable are dummy read/writes (`dummy = SSPBUF; SSPBUF = dummy;`). They are needed by the peripheral for correctly transfer bytes through the bus.