LAB3

Peripherals using I2C and SPI bus

European Regional Development Fund - Investing in your Future - PN: 7-022-BE_i-MOCCA
"INTERREG IVA 2 Mers Seas Zeeën Cross-border Cooperation Programme 2007-2013"

Author: B. STEFANELLI - july 2012 / revised -august 2015

Table of contents

Aims

This lab aims at acquiring a practical understanding on I2C and SPI protocols. A practical application will be to create a new library to control the MCP4725 12-bit D/A converter through the I2C bus.

In this lab, you will:

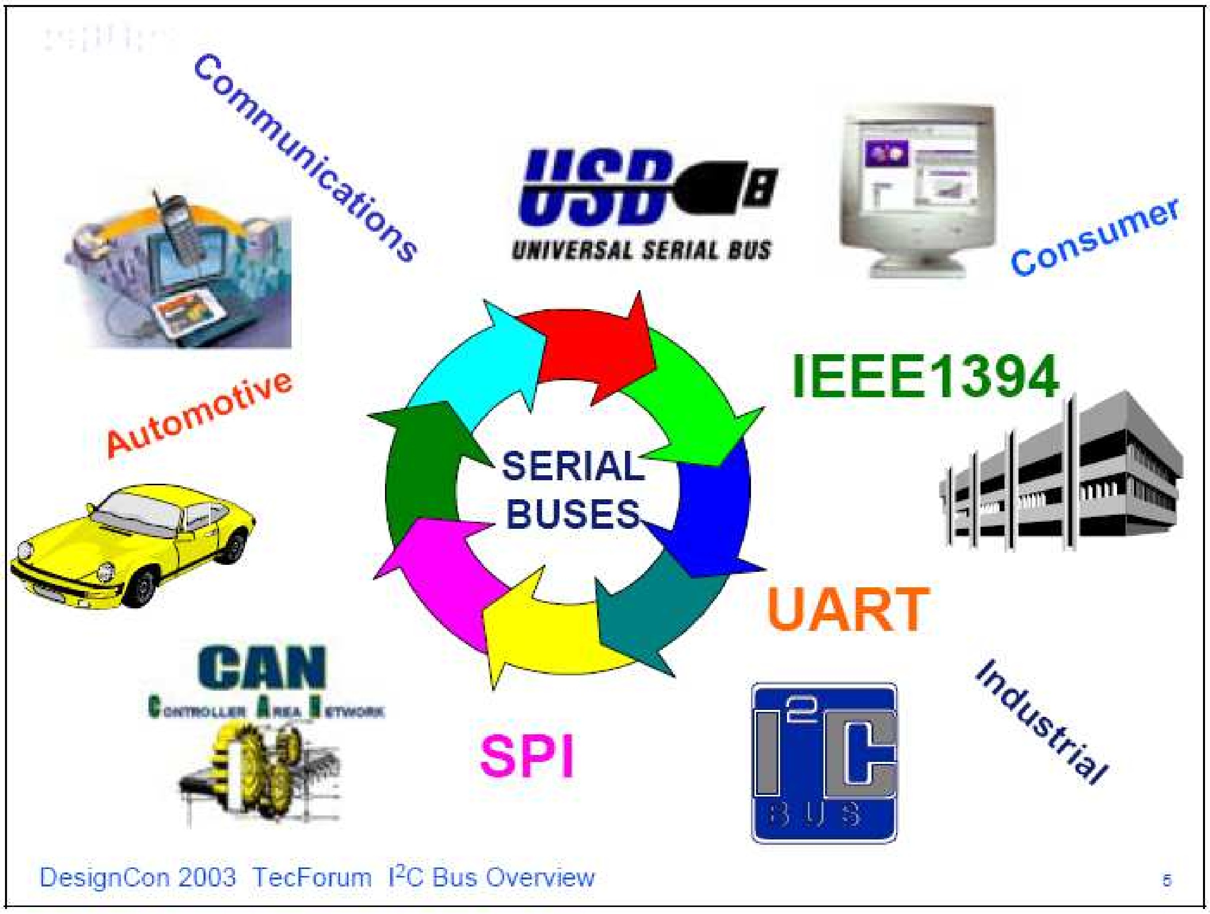
- ✓ Learn (review) the basics of I2C and SPI protocol
- ✓ Learn the steps to create a new library
- ✓ Measure signals on I2C and SPI buses

Prerequisite

Arduino IDE properly installed and previous labs successfully completed.

1. Introduction to SPI and I2C (from AN10216-01 I2C manual NXP)

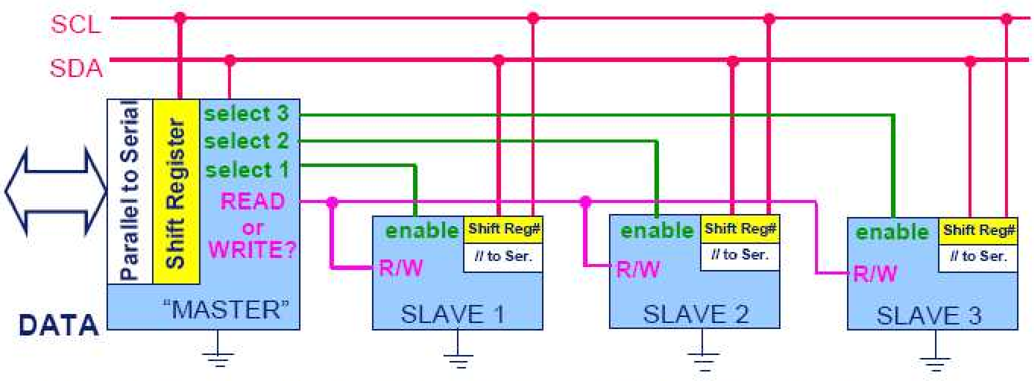
1.1. Buses at a glance



Buses come in two forms, serial and parallel. The data and/or addresses can be sent over 1 wire, bit after bit, or over 8 or 32 wires at once. Always there has to be some way to share the common wiring, some rules, and some synchronization. Slide below shows a serial data bus with three shared signal lines, for bit timing, data, and R/W. The selection of communicating partners is made with one separate wire for each chip. As the number of chips grows, so do the selection wires. The next stage is to use multiplexing of the selection wires and call them an address bus.

If there are 8 address wires we can select any one of 256 devices by using a ‘one of 256' decoder IC. In a parallel bus system there could be 8 or 16 (or more) data wires. Taken to the next step, we can share the function of the wires between addresses and data but it starts to take quite a bit of hardware and worst is, we still have lots of wires. We can take a different approach and try to eliminate all except the data wiring itself. Then we need to multiplex the data, the selection (address), and the direction info - read/write. We need to develop relatively complex rules for that, but we save on those wires. This presentation covers buses that use only one or two data lines so that they are still attractive for sending data over reasonable distances - at least a few meters, but perhaps even km.

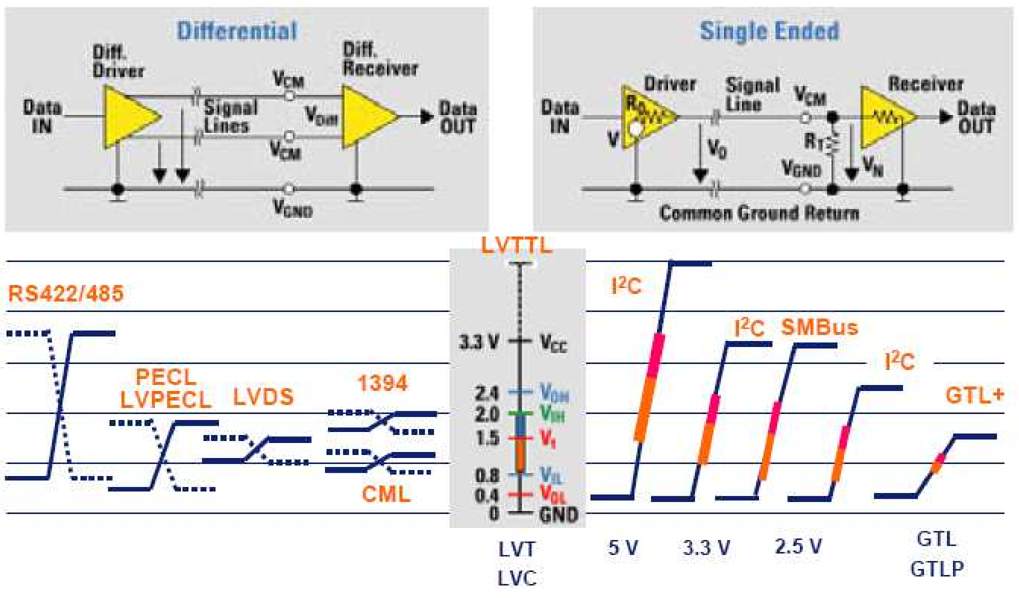
General concept for Serial communications



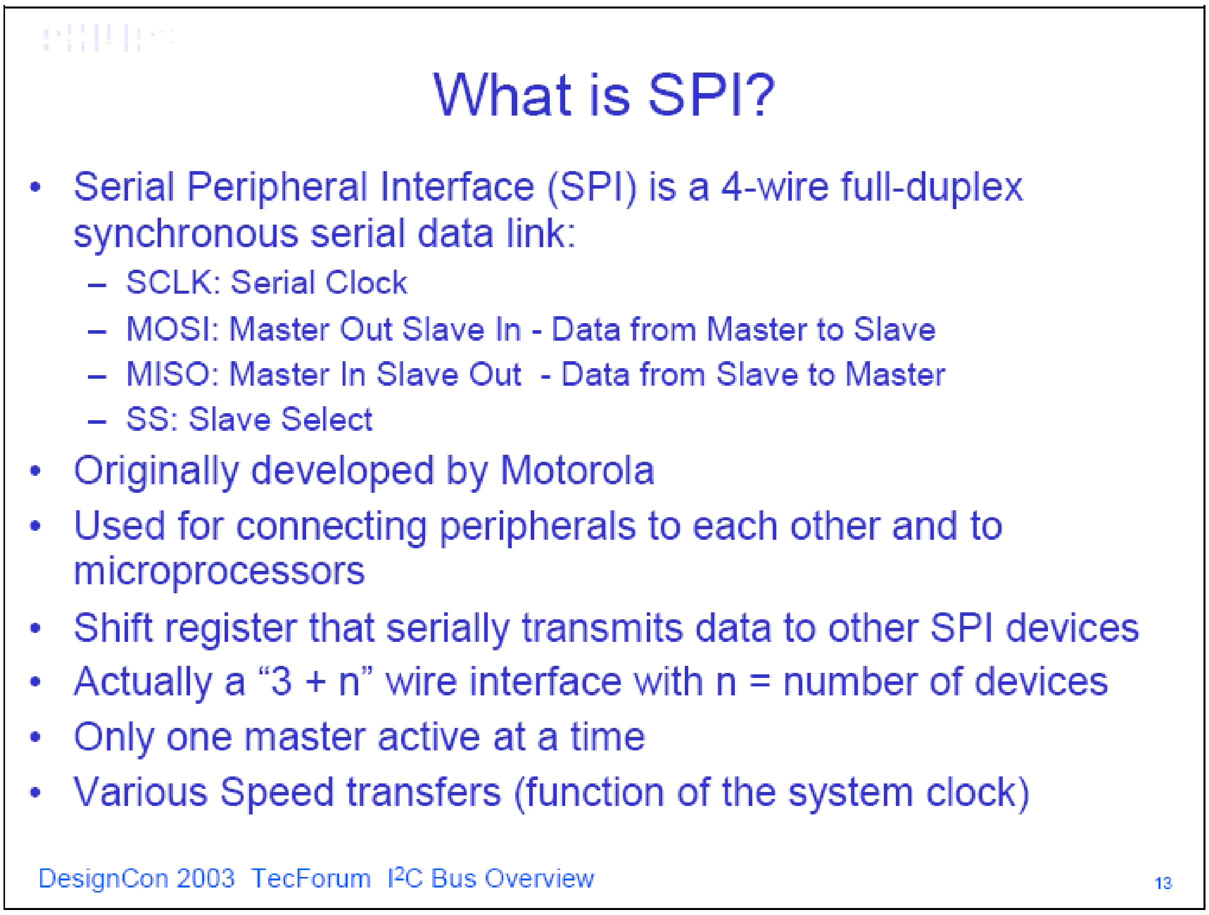
- A point to point communication does not require a Select control signal
- An asynchronous communication does not have a Clock signal
- Data, Select and R/W signals can share the same line, depending on the protocol
- Notice that Slave 1 cannot communicate with Slave 2 or 3 (except via the 'master')
  Only the 'master' can start communicating. Slaves can 'only speak when spoken to'

Devices can communicate differentially or single ended with various signal characteristics as shown below.

Typical Signaling Characteristics



1.2. SPI bus overview



What is SPI?

- Serial Peripheral Interface (SPI) is a 4-wire full-duplex synchronous serial data link:
  - SCLK: Serial Clock
  - MOSI: Master Out Slave In - Data from Master to Slave
  - MISO: Master In Slave Out - Data from Slave to Master
  - SS: Slave Select
- Originally developed by Motorola
- Used for connecting peripherals to each other and to microprocessors
- Shift register that serially transmits data to other SPI devices
- Actually a "3 + n" wire interface with n = number of devices
- Only one master active at a time
- Various Speed transfers (function of the system clock)

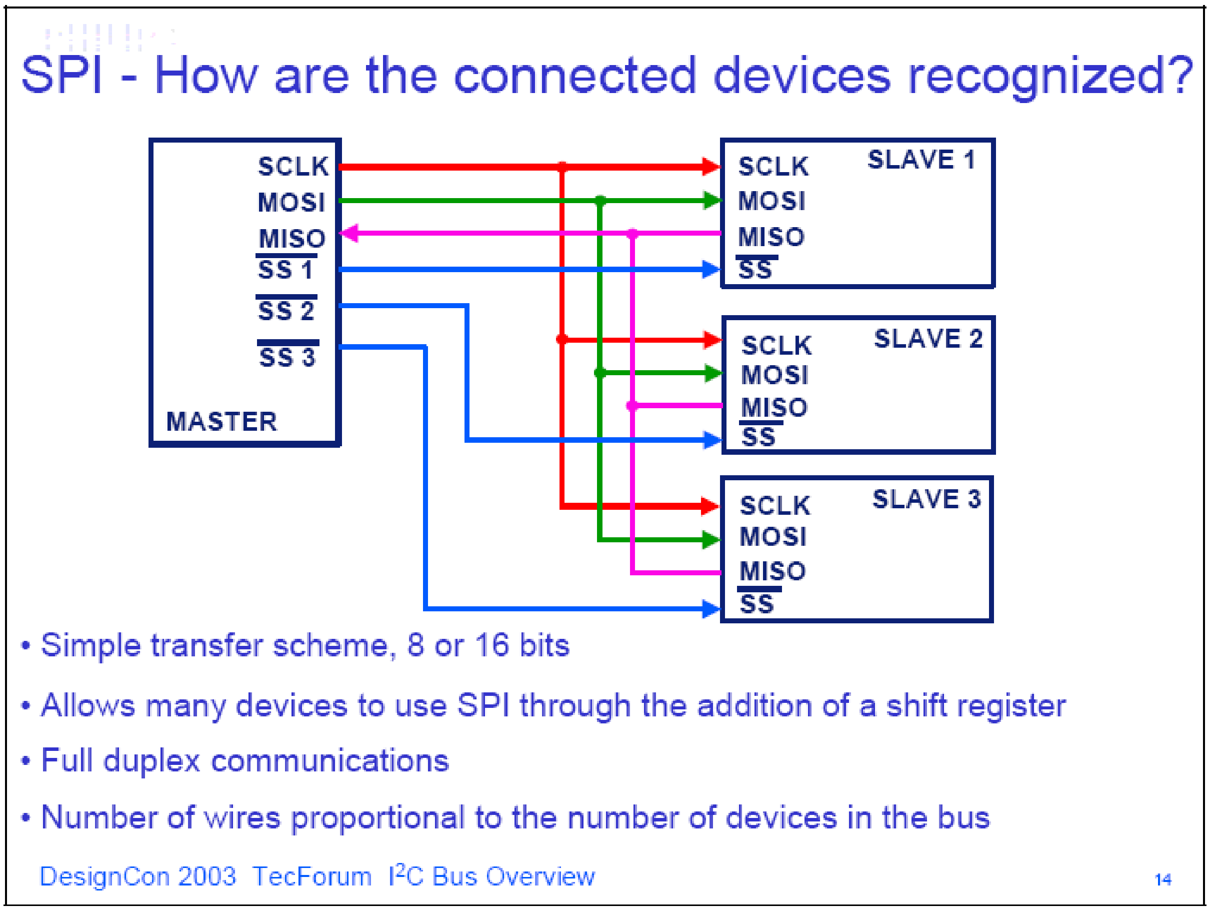
DesignCon 2003 TecForum I^{2}C Bus Overview 13

The Serial Peripheral Interface (SPI) circuit is a synchronous serial data link that is standard across many Motorola microprocessors and other peripheral chips. It provides support for a high bandwidth (1 mega baud) network connection amongst CPUs and other devices supporting the SPI. The SPI is essentially a "three-wire plus slave selects" serial bus for eight or sixteen bit data transfer applications. The three wires carry information between devices connected to the bus. Each device on the bus acts simultaneously as a transmitter and receiver. Two of the three lines transfer data (one line for each direction) and the third is a serial clock. Some devices may be only transmitters while others only receivers. Generally, a device that transmits usually possesses the capability to receive data also. An SPI display is an example of a receive-only device while EEPROM is a receiver and transmit device.

The devices connected to the SPI bus may be classified as Master or Slave devices. A master device initiates an information transfer on the bus and generates clock and control signals. A slave device is controlled by the master through a slave select (chip enable) line and is active only when selected. Generally, a dedicated select line is required for each slave device. The same device can possess the functionality of a master and a slave but at any point of time, only one master can control the bus in a multi-master mode configuration. Any slave device that is not selected must release (make it high impedance) the slave output line. The SPI bus employs a simple shift register data transfer scheme: Data is clocked out of and into the active devices in a first-in, first-out fashion. It is in this manner that SPI devices transmit and receive in full duplex mode.

All lines on the SPI bus are unidirectional: The signal on the clock line (SCLK) is generated by the master and is primarily used to synchronize data transfer. The master-out, slave-in (MOSI) line carries data from the master to the slave and the master-in, slave-out (MISO) line carries data from the slave to the master. Each slave device is selected by the master via individual select lines (although another selection scheme is sometimes implemented, as in the MCP23S08 8-bit I/O used on the ARDUINO-ISEN board).

Information on the SPI bus can be transferred at a rate of near zero bits per second to 1 Mbits per second. Data transfer is usually performed in eight/sixteen bit blocks. All data transfer is synchronized by the serial clock (SCLK). One bit of data is transferred for each clock cycle. Four clock modes are defined for the SPI bus by the value of the clock polarity and the clock phase bits. The clock polarity determines the level of the clock idle state and the clock phase determines which clock edge places new data on the bus. Any hardware device capable of operation in more than one mode will have some method of selecting the value of these bits.



SPI - How are the connected devices recognized?

- Simple transfer scheme, 8 or 16 bits
- Allows many devices to use SPI through the addition of a shift register
- Full duplex communications
- Number of wires proportional to the number of devices in the bus

DesignCon 2003 TecForum I²C Bus Overview 14

1.3. I2C bus overview

Originally, the I2C bus was designed to link a small number of devices on a single card, such as to manage the tuning of a car radio or TV. The maximum allowable capacitance was set at 400 pF to allow proper rise and fall times for optimum clock and data signal integrity with a top speed of 100 kbps. In 1992 the standard bus speed was increased to 400 kbps, to keep up with the ever-increasing performance requirements of new ICs. The 1998 I2C specification increased top speed to 3.4 Mbits/sec. All I2C devices are designed to be able to communicate together on the same two-wire bus and system functional architecture is limited only by the imagination of the designer.

But while its application to bus lengths within the confines of consumer products such as PCs, cellular phones, car radios or TV sets grew quickly, only a few system integrators were using it to span a room or a building. The I2C bus is now being increasingly used in multiple card systems, such as a blade servers, where the I2C bus to each card needs to be isolatable to allow for card insertion and removal while the rest of the system is in operation, or in systems where many more devices need to be located onto the same card, where the total device and trace capacitance would have exceeded 400 pF.

New bus extension & control devices help expand the I2C bus beyond the 400 pF limit of about 20 devices and allow control of more devices, even those with the same address. These new devices are popular with designers as they continue to expand and increase the range of use of I2C devices in maintenance and control applications.

1.3.1. I2C Features:

- ✓ Only two bus lines are required: a serial data line (SDA) and a serial clock line (SCL).Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers. DesignCon2003TecForumI2C Bus Overview24What is I2C ? (Inter-IC)
- ✓ It’s a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.
- ✓ Serial, 8-bit oriented, bi-directional data transfers can be made at up to 100 kbit/s in the standard mode, up to 400 kbit/s in the fast mode, or up to 3.4 Mbit/s in the High-speed mode.
- ✓ On-chip filtering (50 ns) rejects spikes on the bus data line to preserve data integrity.
- ✓ The number of ICs that can be connected to the same bus segment is limited only by the maximum bus capacitive loading of 400 pF.

1.3.2. I2C Communication Procedure:

One IC that wants to talk to another must:

- ✓ Wait until it sees no activity on the I2C bus. SDA and SCL are both high. The bus is 'free'.
- ✓ Put a message on the bus that says 'it’s mine' - I have STARTED to use the bus. All other ICs then LISTEN to the bus data to see whether they might be the one who will be called up (addressed).
- ✓ Provide on the CLOCK (SCL) wire a clock signal. It will be used by all the ICs as the reference time at which each bit of DATA on the data (SDA) wire will be correct (valid) and can be used. The data on the data wire (SDA) must be valid at the time the clock wire (SCL) switches from 'low' to 'high' voltage.
- ✓ Put out in serial form the unique binary 'address' (name) of the IC that it wants to communicate with.

- ✓ Put a message (one bit) on the bus telling whether it wants to SEND or RECEIVE data from the other chip. (The read/write wire is gone!).
- ✓ Ask the other IC to ACKNOWLEDGE (using one bit) that it recognized its address and is ready to communicate.
- ✓ After the other IC acknowledges all is OK, data can be transferred.
- ✓ The first IC sends or receives as many 8-bit words of data as it wants. After every 8-bit data word the sending IC expects the receiving IC to acknowledge the transfer is going OK.
- ✓ When all the data is finished the first chip must free up the bus and it does that by a special message called 'STOP'. It is just one bit of information transferred by a special 'wiggling' of the SDA/SCL wires of the bus.

The bus rules say that when data or addresses are being sent, the DATA wire is only allowed to be changed in voltage (so, '1', '0') when the voltage on the clock line is LOW. The 'start' and 'stop' special messages BREAK that rule, and that is how they are recognized as special.

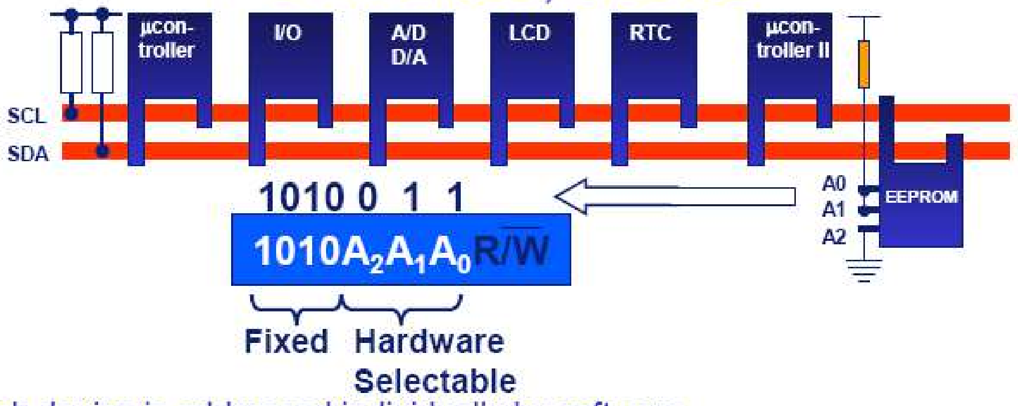
1.3.3. How are the connected devices recognized?

- ✓ Master device ‘polls' using a specific unique identification or "addresses" that the designer has included in the system
- ✓ Devices with Master capability can identify themselves to other specific Master devices and advise their own specific address and functionality. This allows designers to build ‘plug and play' systems and bus speed can be different for each device (there is only a maximum limit)
- ✓ Only two devices exchange data during one ‘conversation'

Any device with the ability to initiate messages is called a ‘master'. It might know exactly what other chips are connected, in which case it simply addresses the one it wants, or there might be optional chips and it then checks what's there by sending each address and seeing whether it gets any response (acknowledge).

An example might be a telephone with a micro in it. In some models, there could be EEPROM to guarantee memory data, in some models there might be an LCD display using an I2C driver. There can be software written to cover all possibilities. If the micro finds a display then it drives it, otherwise the program is arranged to skip that software code. I2C is the simplest of the buses in this presentation. Only two chips are involved in any one communication - the Master that initiates the signals and the one Slave that responded when addressed. But several Masters could control one Slave, at different times. Any ‘smart' communications must be via the transferred DATA, perhaps used as address info. The I2C bus protocol does not allow for very complex systems. It's a ‘keep it simple' bus. But of course system designers are free to innovate to provide the complex systems - based on the simple bus.

I^{2}C Address, Basics



- Each device is addressed individually by software
- Unique address per device: fully fixed or with a programmable part through hardware pin(s).
- Programmable pins mean that several same devices can share the same bus
- Address allocation coordinated by the I^{2}C-bus committee
- 112 different types of devices max with the 7-bit format (others reserved)

DesignCon 2003 TecForum I^{2}C Bus Overview 33

2. The Arduino SPI library

This library manages all the SPI signals but the device selection signal (usually labelled _CS) and therefore, the device selection signal must be managed by the user. In addition, since SPI bus is full-duplex[1], SPI.transfer() is used for both transmit and receive. A sample SPI transaction is shown in the example below:

```
void setup() {
  pinMode (slaveSelectPin, OUTPUT); // set the slaveSelectPin (_CS) as an output:
    SPI.begin();  // initialize SPI:
}
void loop() {
  digitalWrite(slaveSelectPin, LOW);     // take the SS pin low to select the chip:
  SPI.transfer(address);          // send in the address via SPI:
  Value = SPI.transfer(0);          //   send a null value via SPI and get the value returned
                                  //  by the slave device. The value sent can be any since it is
                                  // ignored by the slave device during a read operation.
  digitalWrite(slaveSelectPin, HIGH);    // take the SS pin high to de-select the chip:
}
```

The SPI library also provides functions to set the transmission speed (setClockDivider()), the bit order (setBitOrder()) and the transmission mode (setDataMode()). Please refer to the Arduino documentation for detail.

3. The Arduino Wire library

This library manages the signals SDA and SCL used by the I2C bus. Since these signals are open-drain, they require a pull-up to the supply voltage. In Arduino, these pull-up are internal by default (i.e. no external pull-up resistor is required) and therefore, SDA and SCL are pulled to VDD (5V for Arduino UNO or Duemilanove) which can cause problems if the slave circuit is powered with less than 5V.

The sample program below sends 2 bytes (val1 and val2) to a slave at address 44 (decimal). Note that address must range between 0 and 127. Some datasheets show the address value on 8 bits, that is with the LSB corresponding to the R/_W bit set to 0 or 1 depending on the operation considered. In this case, you must shift this byte one position to the right to get the correct address value.

[1] Full-duplex : a link is full-duplex when a data is sent and another received at the same time

```
void setup(){
  Wire.begin();   // join i2c bus (address optional for master)
}
void loop()
{
Wire.beginTransmission(44); // transmit to a device at address 44 (0x2c), device address is specified in datasheet
  Wire.write(val1);          // sends a first byte
  Wire.write(val2);          // sends another byte
  Wire.endTransmission();     // stop transmitting
}
```

In the above example, a stop message (bus released) is sent after transmission since *Wire.endTransmission()* has no argument. Some devices however require **<u>not</u>** to stop the transmission between write and read operation. In this case, *Wire.endTransmission(FALSE)* sends a restart message after transmission, keeping the connection active.

The example below shows how to get 6 bytes from a slave device at address 44 (the setup() part was omitted here for clarity):

```
void loop()
{
  Wire.requestFrom(44, 6);    // request 6 bytes from slave device at address 44
  while (Wire.available())   // slave may send less than requested
  {
    char c = Wire.read(); // receive a byte as character
    Serial.print(c);        // print the received character
  }
}
```

4. Using the MCP23x08 circuits

4.1. The MCP23X08 library

To drive the two I/O circuits available on the Arduino-ISEN board (MCP23008 for I2C and MCP23S08 for SPI), the MCP23X08 library is available. A device library contains a header (.h) file, a code (.cpp) file and an optional keyword text file (keyword.txt).

4.1.1. The MCP23X08.h file

```
#ifndef MCP23X08_h
#define MCP23X08_h

#include <inttypes.h>
```

The following lines associate the physical address of the internal registers to a symbolic name (and optionally to a particular value) which is the name that appears in the circuit's datasheet. It can be used later instead of the physical address in order to improve the readability of the code.

```
# define IODIR 0x00
# define IPOL 0x01
# define GPINTEN 0x02
# define DEFVAL 0x03
# define INTCON 0x04
# define IOCON 0x05
# define GPPU 0x06
# define INTF 0x07
# define INTCAP 0x08
# define GPIO 0x09
# define OLAT 0x0a
```

```
# define MCP23008_ADDR 0x20 // offset MCP23008 address
# define MCP23S08_ADDR 0x20 // offset MCP23S08 address
```

The following lines define the classes and prototype functions:

```
class MCP23X08 {

protected:

uint8_t _deviceAddr;
uint8_t _csPin;

  MCP23X08(uint8_t _deviceAddr, uint8_t _csPin);
public:
virtual void Write(uint8_t, uint8_t) = 0;
 virtual uint8_t Read(uint8_t) = 0;
};
```

```
class MCP23008 : public MCP23X08 {
public:
MCP23008(uint8_t);
void Write(uint8_t, uint8_t);
uint8_t Read(uint8_t);
};

class MCP23S08 : public MCP23X08 {
public:
MCP23S08(uint8_t, uint8_t);
 void Write(uint8_t, uint8_t);
uint8_t Read(uint8_t);
};

#endif
```

4.1.2. The MCP23X08.cpp file

#include "MCP23X08.h" defines function prototypes and some useful variable

#include <stdio.h>
#include <string.h>
#include <inttypes.h>
#include <Wire.h> defines functions used for I2C bus protocol (MCP23008)
#include <SPI.h> defines functions used for SPI bus protocol (MCP23S08)

#include <Arduino.h> defines functions and constants defined in Arduino IDE

The following function is used in the constructor. Since both circuits are identical (with the exception of the bus used for communication), the same function can be used. This function defines the address associated to a particular device deviceAddr: this is a mandatory feature. The second parameter csPin matters only for the MCP23S08, it may take any value for MCP23008.

```
//===================================================
// MCP23X08
//====================================================

MCP23X08::MCP23X08(uint8_t deviceAddr, uint8_t csPin) {
_deviceAddr = deviceAddr;
_csPin = csPin;
}
```

The constructor for I2C: used to create an instance of the device (csPin set to 0 for example)

```
//====================================================
// MCP23008 I2C I/O
//====================================================

        /*---------------------------------------------------------------------------------
        Set up MCP23008 device address
        deviceAddr : device address from 0 to 7
        ---------------------------------------------------------------------------------*/
        MCP23008::MCP23008(uint8_t deviceAddr) : MCP23X08(deviceAddr, 0) {
        Wire.begin();     // join i2c bus as master device
        _deviceAddr = _deviceAddr | MCP23008_ADDR; // add offset address for MCP23008
        }
```

The *Write* function: used to set a particular value to a given register

```
        /*---------------------------------------------------------------------------------
        Write data to MCP23008 register
        reg : MCP23008 register
        val : data value
        ---------------------------------------------------------------------------------*/
        void MCP23008::Write(uint8_t reg, uint8_t val ) {
        Wire.beginTransmission(_deviceAddr); // begin transmission with slave device
        #_deviceAddr
        Wire.write(reg);            // select register
        Wire.write(val);            // put 1 byte
        Wire.endTransmission(); // send all bytes
        }
```

The *Read* function: used to read the value from a given register

```
        /*---------------------------------------------------------------------------------
        Read data from MCP23008 register
        reg : MCP23008 register
        return : data value
        ---------------------------------------------------------------------------------*/
        uint8_t MCP23008::Read(uint8_t reg) {

        Wire.beginTransmission(_deviceAddr); // begin transmission with slave device
        #_deviceAddr

        Wire.write(reg);            // select register
        Wire.endTransmission(); // send all bytes

        Wire.requestFrom(_deviceAddr, (uint8_t)1); // request 1 byte from slave device
        #_deviceAddr

        if (Wire.available())       // slave may send less than requested
        {
        return Wire.read();       // get 1 byte
        }
```

```
else {
return 0;
}
}
```

The constructor for SPI: used to create an instance of the device (csPin value matters here)

```
//=====================================================
// MCP23S08 SPI I/O
//=====================================================

/*-----------------------------------------------------------------------------
Set up MCP23S08 device address
deviceAddr : device address from 0 to 3
csPin : chip select pin
-----------------------------------------------------------------------------*/
MCP23S08::MCP23S08(uint8_t deviceAddr) : MCP23X08(deviceAddr, csPin) {

  SPI.begin();            // initialises SPI: MSB first, mode 0, 4MHz, _CS pin = 10
  pinMode(_csPin, OUTPUT); // useful if _CS is not pin 10
   digitalWrite(_csPin, HIGH);          // Disable slave pin CS
  _deviceAddr = _deviceAddr | MCP23S08_ADDR;     // add offset address for
  MCP23S08 device
  }


  /*-----------------------------------------------------------------------------
  Write data to MCP23S08 register
  reg : MCP23S08 register
  val : data value
  -----------------------------------------------------------------------------*/
  void MCP23S08::Write(uint8_t reg, uint8_t val ) {

  digitalWrite(_csPin, LOW);                // Enable slave pin CS
   SPI.transfer(0 | (_deviceAddr << 1)); // begin transmission with slave device
  #_deviceAddr
  SPI.transfer(reg);                        // select register
  SPI.transfer(val);                        // put 1 byte
  digitalWrite(_csPin, HIGH);          // Disable slave pin CS
   }


  /*-----------------------------------------------------------------------------
  Read data from MCP23S08 register
  reg : MCP23S08 register
  return : data value
  -----------------------------------------------------------------------------*/
  uint8_t MCP23S08::Read(uint8_t reg) {

  digitalWrite(_csPin, LOW);                // Enable slave
  SPI.transfer(1 | (_deviceAddr << 1));  // begin transmission with slave device
  #_deviceAddr
  SPI.transfer(reg);                        // select register

   byte data;
   data = SPI.transfer(0xFF);              // SPI.transfer dummy byte to get response
  digitalWrite(_csPin, HIGH);             // Disable slave

    return data;
  }
```

4.1.3. The keyword.txt file

This optional file is very useful for code readability, since it defines reserved words that appear in a particular color when the source file is edited. Below is the keyword file associated to the MCP23X08 library.

```
########################################
# Syntax Coloring Map For MCP23X08
########################################

########################################
# Datatypes (KEYWORD1)
########################################

MCP23008 KEYWORD1
MCP23S08 KEYWORD1
MCP23X08 KEYWORD1

########################################
# Methods and Functions (KEYWORD2)
########################################
Write KEYWORD2
Read KEYWORD2

########################################
# Constants (LITERAL1)
########################################
IODIR LITERAL1
IPOL LITERAL1
GPINTEN LITERAL1
DEFVAL LITERAL1
INTCON LITERAL1
IOCON LITERAL1
GPPU LITERAL1
INTF LITERAL1
INTCAP LITERAL1
GPIO LITERAL1
OLAT LITERAL1
```

It is worth noting that a tab **must** be used to separate the label and the associated type (KEYWORD1, KEYWORD2, LITERAL1).

4.2. Using the MCP23X08 library

The code below illustrates the use of the two I/O circuits (file: MCP23X08_intro):

```
#include <MCP23X08.h>
#include <Wire.h>       //Mandatory, although they are included in MCP23XO8.h
#include <SPI.h>

# define MCP23008_ADDR 0x00         // this is the default address
# define MCP23S08_ADDR 0x00         // this is the default address

// creates instances of each IO circuit

MCP23008 i2c_io(MCP23008_ADDR);        // Init MCP23008
MCP23S08 spi_io(MCP23S08_ADDR, 10);       // Init MCP23S08
```

Remark: there is only one MCP23008 device on the board. If there were many MCP23008 circuits on the same I2C bus, for example three circuits, each with a different physical address like 0x00, 0x01, 0x02, we had to create three instances using the constructor:

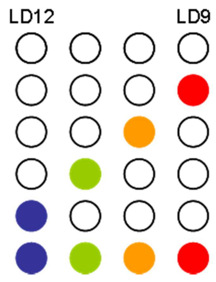
```
// MCP23008 i2c_io0(0x00, 10);

// MCP23008 i2c_io1(0x01, 10);

// MCP23008 i2c_io2(0x02, 10);
```

Later, the subsequent read and write operations will use the instance names to address the correct device.

```
uint8_t sw_state;    // uint_8 is the type returned by the Read function. Other types do not work

void setup() {

  spi_io.Write(IOCON, VAL23S08_IOCON);  // Sets defaults for MCP23S08, in particular
puts interrupt output open-drain

 i2c_io.Write(IODIR, 0x0F);   // sets port direction for individual bits
spi_io.Write(IODIR, 0x0F);

spi_io.Write(GPIO, 0x55);
i2c_io.Write(GPIO, 0x55);

 delay(2000);

}

void loop() {

 sw_state = i2c_io.Read(GPIO); // reads the state of I2C switches
sw_state = (sw_state & 0x0F) << 4;
spi_io.Write(GPIO, sw_state); // state of I2C switches is applied to SPI leds
 sw_state = spi_io.Read(GPIO); // reads the state of SPI switches
 sw_state = (sw_state & 0x0F) << 4;
 i2c_io.Write(GPIO, sw_state); // state of SPI switches is applied to I2C leds

  delay(100);
}
```

4.3. Applying the MCP23X08 library

Write a program that switches on leds LD9 to LD12 in a sequence described below, with a 500ms delay between each figure:



5. Building a library for the MCP4725

5.1. Case study: the MCP4725

In the MCP4725 datasheet, find out the necessary information to be able to use the circuit, particularly:

- ✓ The relationship between the data code and the output voltage
- ✓ The slave address of the device (how many devices can be used on the same bus?)
- ✓ The format for sending data in fast mode
- ✓ The format for sending data for DAC input register
- ✓ The format for sending data for DAC input register <u>and</u> EEPROM
- ✓ The format for reading data from DAC register

5.2. Library for the MCP4725 (V1.0)

Create a library called MCP4725 with the following functions:

- ✓ A constructor to instantiate the device(s)
- ✓ A write function named dacWrite(data), where *data* corresponds to the D11 ... D0 data bits in hexadecimal format. This function must operate in "fast" mode and must set the power-down mode to "normal mode". For example, to write to the DAC the data value 0x0F81, the function call will be: *dacWrite(0x0F81);*

Reminder :

When accessing a I2C device, the address (deviceAddr for example) is assumed to be a 7-bit value which forms the seven MSB of the effective device address, the LSB being automatically set to 0 or 1 depending on whether a read or write operation is performed. For example, the MCP4725 uses 110000Ao for address (where Ao is externally set to 0 or 1 using the Ao pin). Let's assume Ao=1, then the address is 1100001 at the end of which 0 or 1 will be appended, that is 11000011 for read and 11000010 for write.

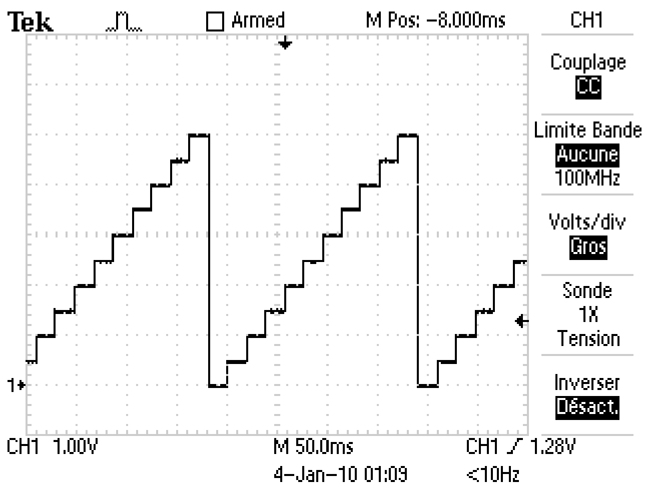
When using the "wire" library of arduino IDE, you must pay attention to the fact that a command like:

Wire.beginTransmission(deviceAddr);

first left-shifts the provided deviceAddr before appending the LSB. Therefore, if the device address is 1100001 (the seven MSB's), you must provide the value 61, which corresponds to 01100001 because this value will be shifted to form the effective device address 1100001x where x will be set to 0 or 1.

5.3. Test program for the MCP4725 (V1.0)

Write a test program to show the capabilities of MCP4725 library (version 1.0). This program will generate an eleven-step staircase signal. The signal will be between 0 and Vcc, with equal height steps and a repetition rate around 5Hz.



Sample staircase signal

5.4. Library for the MCP4725 (V2.0)

Improve library version 1.0 with the following functions:

✓ An additional dacWrite function which format is dacWrite(mode,pdval,data), where:
- *mode* is an integer which value can be 0, 1 or 2 corresponding to FAST, DAC and DAC&EEPROM write modes respectively.
- *pdval* can take an integer value between 0 and 3 corresponding to the PD1,PD0 power-down control bits.
- *data* corresponds to the D11 ... D0 data bits in hexadecimal format

(This new function overloads the function defined in library version 1.0)

For example, to write to the DAC the data value 0x0F81 in fast mode and power-down mode normal, the function call will be: *dacWrite(0,0,0x0F81);*

✓ A read function named dacRead() that ONLY returns the value of the 2nd byte (status byte of the DAC)

5.5. Test program for the MCP4725 (V2.0)

Write a test protocol to show the capabilities of MCP4725 library (version 2.0). This protocol will comprise the following programs:

Program 1: only stores in the built-in EEPROM the data corresponding to an output voltage of VDD/2

Program 2: waits upon an action on SW12 to generate a eleven-step staircase signal. The signal will be between 0 and VDD, with equal height steps and a repetition rate around 5Hz.

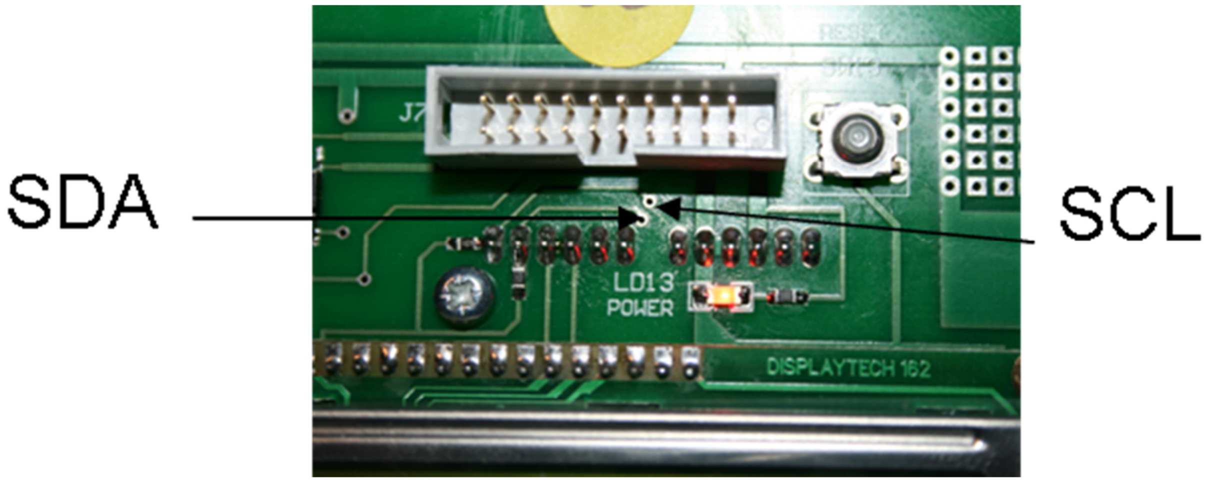
NOTE: make sure that program 2 tests **all** the available write function in the library, that is the coverage of your tests is sufficient

6. Measuring I2C signals

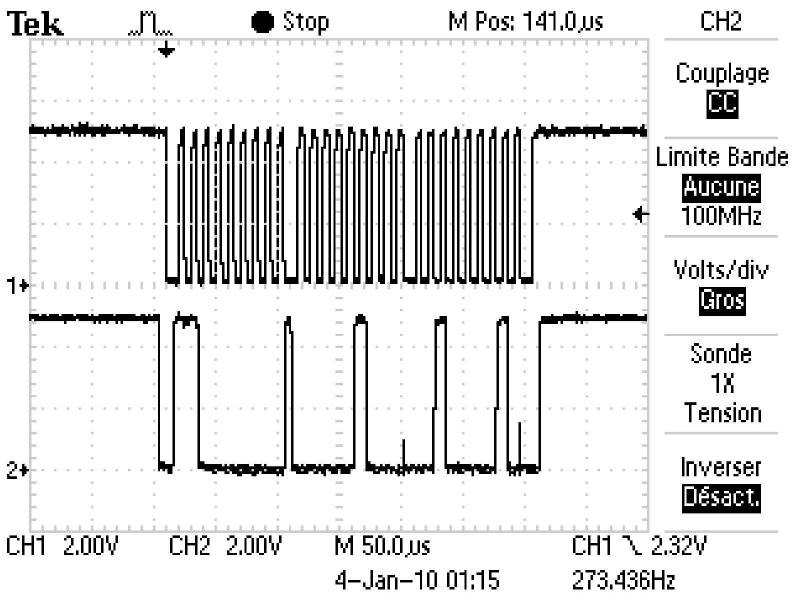
Using the documentation, find out the waveform corresponding to the following write sequence (don't care about the timing however).

```
#include <MCP4725.h>
#include <Wire.h>
#include <Spi.h>

# define MCP4725_ADDR 0x00 // this is the default address
// creates instance of DAC circuit
MCP4725 dac(MCP4725_ADDR); // Init MCP4725

void setup(){
pinMode(9,OUTPUT);
digitalWrite(9,LOW);
}


void loop(){

digitalWrite(9,HIGH);
dac.dacWrite2(0,0,0x0421);
delay(50);
digitalWrite(9,LOW);
delay(50);
}
```

Copy and paste the code, upload it and capture SCL and SDA signals on the board.



You normally should obtain a display similar to the one below:

7. 

Sample I2C transaction

Compare the acquired waveform with your original waveform and find the data actually sent along with the START, STOP and ACK signals. Repeat the measurements with a wrong address and comment.

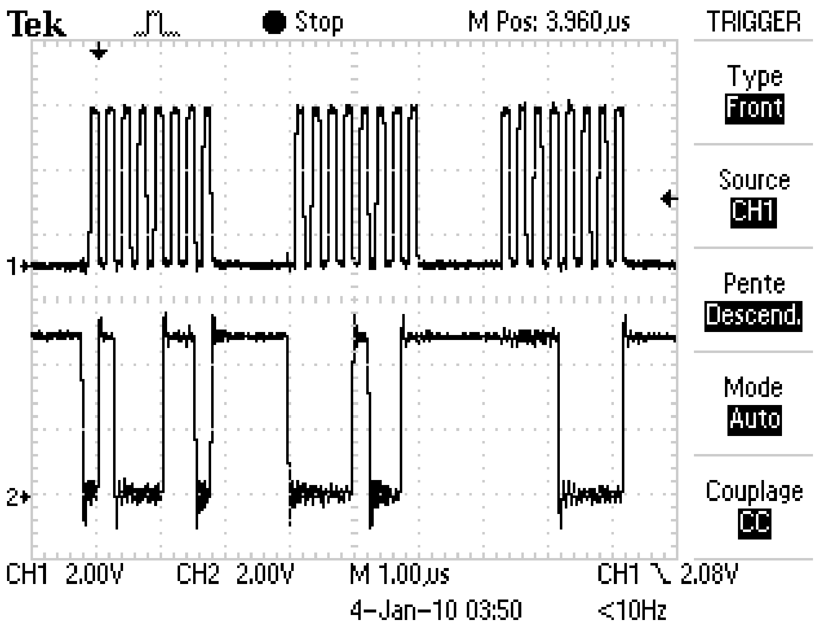
8. Measuring SPI signals

Using the documentation, find out the waveform corresponding to the following write sequence (don't care about the timing however).

```
#include <MCP23X08.h>
#include <Wire.h>
#include <Spi.h>

# define MCP23S08_ADDR 0x03 // this is the default address
```

```
// creates instance of DAC circuit
MCP23S08 spi_io(MCP23S08_ADDR); // Init MCP23S08

void setup(){
spi_io.Write(REG23x08_IODIR,0x0F);
}

void loop(){

spi_io.Write(REG23x08_GPIO,0x00);
delay(50);
spi_io.Write(REG23x08_GPIO,0xF0);
delay(50);
}
```

Copy and paste the code, upload it and capture SCK and SI signals on the board. You normally should obtain a display similar to the one below:



Sample SPI transaction

Compare the acquired waveform with your original waveform and find the data actually sent. Repeat the measurements with a wrong address and comment.