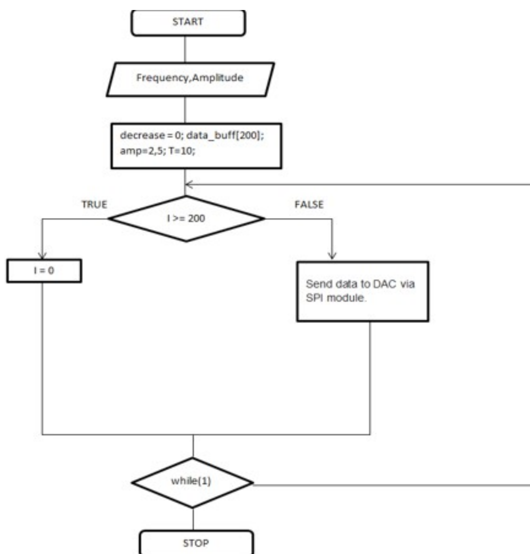
PROJECT: *Sine wave generation*

Students:
Gaciu (Morari) Maria Manuela   and Morari Eugeniu

E-mail: manuela.gaciu@gmail.com   eu.morari1993@gmail.com

Description:

This project aims on understanding the wave forms, especially the sinus wave.
We have started with a diagram to have a clear picture of the project.



The main idea is to control the frequency and the amplitude for the signal to be via USART from PC to microcontroller. The connection between XMC1100 and DAC (MCP4921) module is made through the SPI communication interface. The output of DAC, Vout, will be watched on the oscilloscope.

Some details about the communication interfaces can be found below:

The universal asynchronous receiver-transmitter (UART) takes bytes of data and transmits

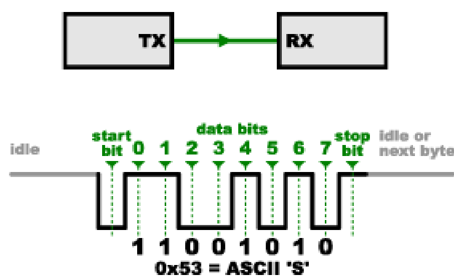
the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes. Each UART contains a shift register, which is the fundamental method of conversion between serial and parallel forms. Serial transmission of digital information (bits) through a single wire or other medium is less costly than parallel transmission through multiple wires.

The UART usually does not directly generate or receive the external signals used between different items of equipment. Separate interface devices are used to convert the logic level signals of the UART to and from the external signalling levels, which may be standardized voltage levels, current levels, or other signals.

Communication may be *simplex* (in one direction only, with no provision for the receiving device to send information back to the transmitting device), *full duplex* (both devices send and receive at the same time) or *half duplex* (devices take turns transmitting and receiving).

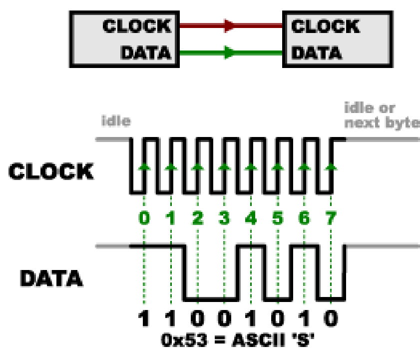
A common serial port, the kind with TX and RX lines, is called “asynchronous” (not synchronous) because there is no control over when data is sent or any guarantee that both sides are running at precisely the same rate. Since computers normally rely on everything being synchronized to a single “clock” (the main crystal attached to a computer that drives everything), this can be a problem when two systems with slightly different clocks try to communicate with each other.

To work around this problem, asynchronous serial connections add extra start and stop bits to each byte help the receiver sync up to data as it arrives. Both sides must also agree on the transmission speed (such as 9600 bits per second) in advance. Slight differences in the transmission rate aren’t a problem because the receiver re-syncs at the start of each byte.



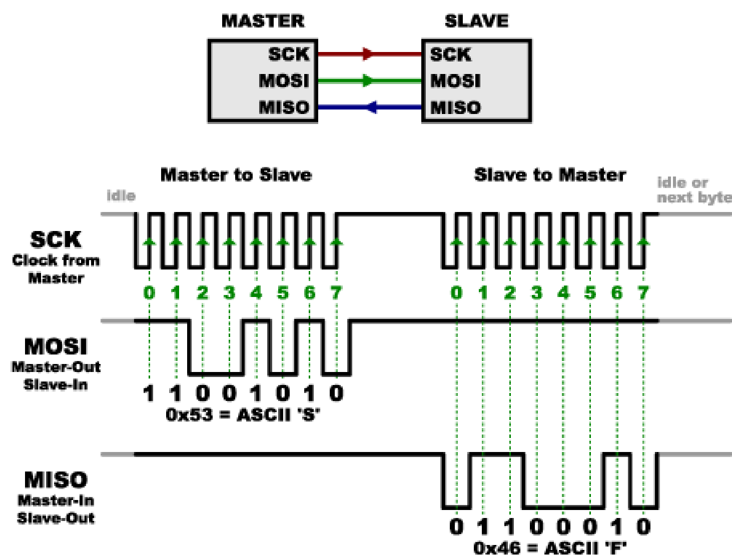
SPI works in a slightly different manner. It’s a “synchronous” data bus, which means that it

uses separate lines for data and a “clock” that keeps both sides in perfect sync. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line. This could be the rising (low to high) or falling (high to low) edge of the clock signal; the datasheet will specify which one to use. When the receiver detects that edge, it will immediately look at the data line to read the next bit (see the arrows in the below diagram). Because the clock is sent along with the data, specifying the speed isn’t important, although devices will have a top speed at which they can operate (We’ll discuss choosing the proper clock edge and speed in a bit).

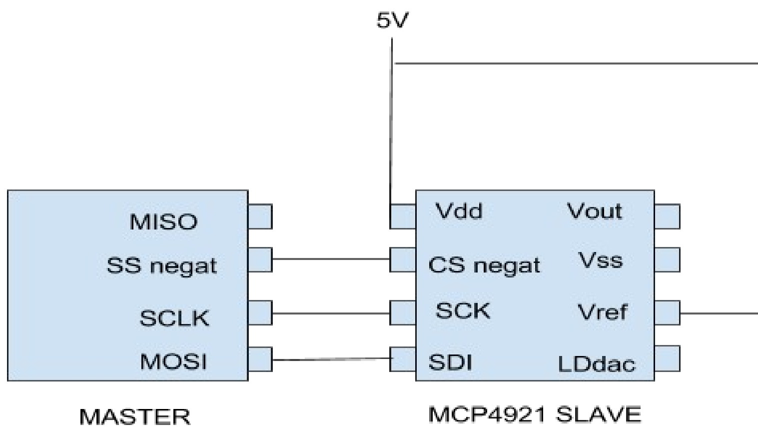


In SPI, only one side generates the clock signal (usually called CLK or SCK for Serial ClocK). The side that generates the clock is called the “master”, and the other side is called the “slave”. There is always only one master (which is almost always your microcontroller), but there can be multiple slaves (more on this in a bit).

When data is sent from the master to a slave, it’s sent on a data line called MOSI, for “Master Out / Slave In”. If the slave needs to send a response back to the master, the master will continue to generate a prearranged number of clock cycles, and the slave will put the data onto a third data line called MISO, for “Master In / Slave Out”.



In our project, we have configured a master and only a slave - the DAC module.
Data is sent to the microcontroller as a string, is decoded and transmitted to the digital - analog converter; in this way, an increasing or decreasing V out value will result in a sine wave.



As the focus was on the communication interfaces, the code will clarify the implementation:

```
/* Private variables ---------------------------------------------------------*/

 uint8_t mess_buff[20] = " ";
 uint16_t decrease = 0;
 uint16_t send_data;
 // Vector cu 50 elemente pentru generarea sinusului / val max = 1600
 uint16_t data_buff[200] = {819, 844, 870, 896, 921, 947, 972, 997, 1022, 1047, 1072,
1096, 1120, 1144, 1167, 1191, 1213, 1236, 1258, 1279, 1300, 1321, 1341, 1360, 1379,
1398, 1416, 1433, 1450, 1466, 1481, 1496, 1510, 1524, 1537, 1549, 1560, 1571, 1580,
1589, 1598, 1605, 1612, 1618, 1623, 1628, 1631, 1634, 1636, 1637, 1638, 1637, 1636,
1634, 1631, 1628, 1623, 1618, 1612, 1605, 1598, 1589, 1580, 1571, 1560, 1549, 1537,
1524, 1510, 1496, 1481, 1466, 1450, 1433, 1416, 1398, 1379, 1360, 1341, 1321, 1300,
1279, 1258, 1236, 1213, 1191, 1167, 1144, 1120, 1096, 1072, 1047, 1022, 997, 972, 947,
921, 896, 870, 844, 819, 793, 767, 742, 716, 691, 665, 640, 615, 590, 566, 541, 517, 493,
470, 447, 424, 402, 380, 358, 337, 317, 297, 277, 258, 239, 222, 204, 187, 171, 156, 141,
127, 114, 101, 89, 77, 67, 57, 48, 40, 32, 25, 19, 14, 10, 6, 3, 1, 0, 0, 0, 1, 3, 6, 10, 14, 19,
25, 32, 40, 48, 57, 67, 77, 89, 101, 114, 127, 141, 156, 171, 187, 204, 222, 239, 258, 277,
297, 317, 337, 358, 380, 402, 424, 447, 470, 493, 517, 541, 566, 590, 615, 640, 665, 691,
716, 742, 767, 793};
```

```
 float amp=2.5; // amp max = 2457

 uint16_t delay1 = 2500;
 float freq = 1000;
 uint8_t aRxBuffer[10];

void init_cs_ldac(void)
{
 CS_disable;
 LDAC_disable;
}
/* USER CODE END 0 */
// Functia de conversie string to int
float stof(const uint8_t* s){
 float rez = 0, fact = 1;
 if (*s == '-'){
   s++;
   fact = -1;
 };
 for (int point_seen = 0; *s; s++){
   if (*s == '.'){
     point_seen = 1;
     continue;
   };
   int d = *s - '0';
   if (d >= 0 && d <= 9){
     if (point_seen) fact /= 10.0f;
     rez = rez * 10.0f + (float)d;
   };
 };
 return rez * fact;
};

uint16_t T=10;

//Functia de generare sinus

/*void gen_vector(float freq){
```

```
 float freq_step=5.0/4096.0; // Pasul pentru frecventa
 float aux;
 for (int i=0;i<200;i++)
 {
   double x= (double)2*3.14*freq*i;
   x=x/1.0;
   aux = sin(x) +1; // functia sinus
   data_buff[i] = (uint16_t) (aux/freq_step);
 }
}*/
```