Section-7 SPI Port driven Serial Communication

7.1 Introduction

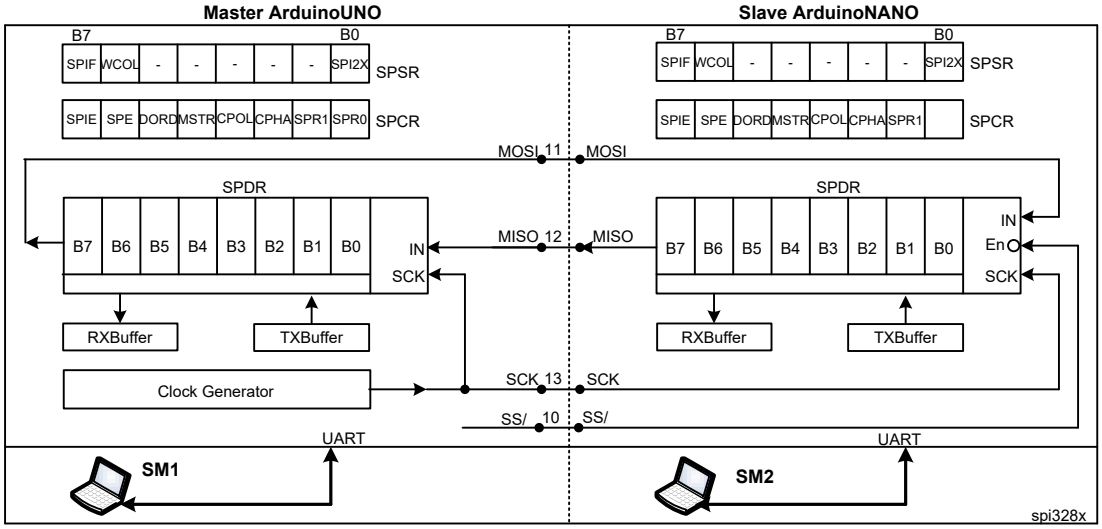

Figure-7.1: *SPI Port based connection between Arduino UNO and Arduino NANO*

(1) SPI Stands for Serial Peripheral Interface. It is a synchronous serial data communication system; because, data shift (sampling) coincides exactly with the rising edge (falling edge) of the clock signal. Therefore, variation in the clock frequency does not affect the data accuracy. In SPI, data exchange can take place at a bit rate from 125 Kbits/sec to 8 Mbits/sec.

(2) There are 4 SPI Transfer Formats one of which must agree with the transfer format of Slave for proper data exchange.

- **(a)** Mode 0 [CPOL, CPHA = 0, 0] : SCK at LOW state, data sampling at rising edge
- **(b)** Mode 1 [CPOL, CPHA = 0, 1] : SCK at LOW state, data sampling at falling edge
- **(c)** Mode 2 [CPOL, CPHA = 1, 0] : SCK at HIGH state, data sampling at falling edge
- **(d)** Mode 3 [CPOL, CPHA = 1, 1] : SCK at HIGH state, data sampling at rising edge

(3) In SPI (as we can see in Fig-7.1), the data exchange is simultaneous which means that data transmission and reception happen at the same time. Under the same 8 clock pulses of the Clock Generator, the present content of the SPDR Register of the Master is shifted out bit-by-bit with MSBit first and enters into the SPDR register of the Slave; at the same time, the present content of the SPDR Register of the Slave enters into the SPDR Register of the Master.

(4) When the Master loads something into SPDR Register by executing this instruction: SPDR = 0x12, the 8 SCK pulses are automatically generated by the Clock Generator to complete data exchange between Master and Slave. Assuming 125 Kbits/sec transfer rate, the data exchange should happen within 64 μs (1/125000*8). It is the Master that always initiates the SPI transfer by generating the SCK pulses.

(5) To be sure that the data exchange has taken place, the Master should check that the SPIF-bit of its SPSR register has assumed HIGH state and then transfers the content of SPDR Register (that has come from Slave) into RXBuffer by executing this code: byte x = SPDR. The value of x is unknown as we don't know what was there in the SPDR of the Slave at the time of SPI transfer. The Master should read the data from SPDR whether it is needed or not, and then the next SPI transfer may take place. The same procedure is also applicable for the Slave.

```
SPDR = 0x12;
while(bitRead(SPSR, SPIF) != HIGH)  //wait until data has arrived from Slave
```

```
{
     ;
}
byte RXBuffer = SPDR;
Serial.println(RXBuffer, HEX);    //will show unpredictable value; it will be 0x67 if Slave
sketch writes 0x67 into SPDR in the setup and then polls the SPIF-bit in the loop() function.
```

(6) Task of Step-4 is to be done also by the Slave. This time the value of x is known, and it has to be 0x12 which the Master has loaded and transferred in Step-3.

```
while(bitRead9SPSR, SPIF) != HIGH)  //wait until data has arrived from Master
{
     ;
}
byte RXBuffer = SPDR;
Serial.println(RXBuffer, HEX);    //will show 0x12 that has just come from Master. See Step-4
```

(7) The arrival of data at the SPDR Register of the Master can also be known through interrupt. When data is ready, the SPI Interrupt Logic generates an interrupt signal. The Master goes to the ISR(SPI_STC_vect) interrupt subroutine, reads the data byte from SPDR, places data byte into TXBuffer for next SPI transfer, and then goes back to loop() function. The same procedure is also applicable for the Slave. When the MCU vectors at the ISR, the SPIF flag bit is automatically cleared.

```
ISR(SPI_STC_vect)
{
     byte RXBuffer = SPDR;    //data moves into RXBuffer
     byte TXBuffer = 0x23;    //23 will be moved to slave in the next SPI transfer
     SPDR = TXBuffer;         //the above two lines could be written this way: SPDR = 0x23
}
```

(8) The inclusion of the following lines in the Master sketch configures the Arduino as Master SPI Device. The default settings are:

(a) MSBit is transferred first;
(b) CPOL (clock polarity) is LOW – initially SCK is at LOW state;
(c) CPHA (clock phase) is the rising edge – data bit is sampled at the rising edge of SCK;
(d) Bit transfer speed is 125 Kbits/sec;
(e) DPin-10 is SS (Slave Select) output line with HIGH state;
(f) DPin-11 is MOSI (Master Out Slave In) output line wih LOW state;
(g) DPin-12 is MISO (Master In Slave Out) input line;
(h) DPin-13 is SCK (Serial Clock) output line with LOW state.

```
#include<SPI.h>
SPI.begin();
```

(9) The inclusion of the following lines in the Slave sketch configures the Arduino as Slave SPI Device. The default settings are:

(a) MSBit is transferred first;
(b) CPOL (clock polarity) is LOW – initially SCK is at LOW state;
(c) CPHA (clock phase) is the rising edge – data bit is sampled at the rising edge of SCK;
(d) Bit transfer speed is same as the Master (125 Kbits/sec);
(e) DPin-10 is SS (Slave Select), and its direction must be set as input to enable SPI interface;
(f) DPin-11 is MOSI (Master Out Slave In) input line;
(g) DPin-12 is MISO (Master In Slave Out), and its direction must be set as output;
(h) DPin-13 is SCK (Serial Clock) input line.

```
#include<SPI.h>          //allows the use of the symbolic names: SS, MISO, SPI related methods
pinMode(SS, INPUT_PULLUP); //must receive LOW for the SPI interface to get enabled
pinMODE(MISO, OUTPUT);
SPCR |= _BV(SPE);        //SPI interface is enabled
biClear(SPCR, 6);        //Arduino is Slave
SPI.attachInterrupt();   //TWI/I2C Interrupt logic is enable; SRCR |= _BV(SPIE)
```

7.2 Arduino Functions for SPI Communication

Sn.	Function/Method	Description
1	#include<SPI.h>	Library File
2	SPI.begin()	Creates SPI Interface with attributes described in Section-7.1(8), 7.1(9).
3	SPI.end()	Disables SPI Port; SPI signals go back to Digital IO pins
4	SPI.setBitOrder(arg1)	arg1 = MSBFIRST, LSBFIRST; which bit will be transferred first
5	SPI.setDataMode(arg1)	arg1 = SPI_MODEX (X=0 to 3); see Section – 7.1(2)
6	SPI.setClockDivider(arg1)	arg1 = SPI_CLK_DIVX (x = 2, 4, 8, 16, 32, 64, 128
7	SPISettings(arg1, arg2, arg3)	arg1 = speedMaximum, arg2 = dataOrder, arg3 = dataMode
8	SPI.beginTransaction(SPISettings)	SPISettings mySetting(speedMaximum, dataOrder, dataMode)
9	SPI.endTransaction()	Stop using the SPI bus. Normally this is called after de-asserting the chip select, to allow other libraries to use the SPI bus.
10	SPI.attachInterrupt()	Enables: ISR(TWI_STC_vect){} interrupt subroutine
11	byte x = SPI.transfer(0xnn)	0xnn is shifted out, check if data from slave has arrived, and then puts into x.
12	int x = SPI.transfer16(0x1234);	12 is transferred first and then 0x34; received bytes are packed in x.
13	SPI.transfer(buffer, sizeof(buffer))	bytes of buffer are transferred one after another; received bytes are stored back in buffer

7.3 Exercises

1 Assume that SPDR of Slave of Fig-7.2 contains 0x67 before the Slave enters into loop() function. The Master has executed SPDR = 0x12 instruction at 500 Kbits/sec SPI speed. After SPI transaction, what values are there in the SPDR registers of both Master and Slave? What is the SPI transfer time? Create sketch to establish your answers. Use polling method (reading the SPIF bit again and again) to be sure that SPI transaction is complete.

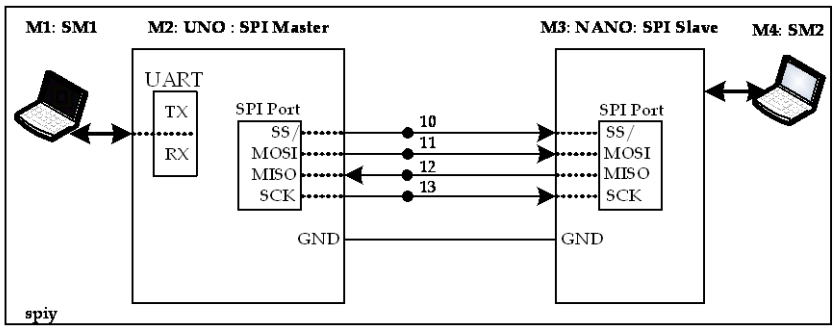

Figure-7.2: *SPI Port connection between UNO and NANO*

Ans:
SPDR Register of Master will contain 0x67; SPDR Register of Slave will contain 0x12.
SPI transfer time: 16 µs (1/125000*8).

Master Sketch:

```
#include <SPI.h>

void setup (void)
{
  Serial.begin(115200);
  SPI.begin ();
  digitalWrite(SS, LOW);    //Slave is selected
  SPI.setClockDivider(SPI_CLOCK_DIV32);  //500 Kbits/sec
}

void loop()
{
  SPDR = 0x12;
  while(bitRead(SPSR, SPIF) != HIGH)
    ;
  byte x = SPDR;
```

```
  Serial.println(x, HEX); //should show: 67 which has come from Slave
  while(1);
}
```

Slave Sketch:

```
#include <SPI.h>

void setup ()
{
  Serial.begin(115200);
  pinMode(SS, INPUT_PULLUP);  // ensure SS stays high for now
  pinMode(MISO, OUTPUT);
  SPCR |= _BV(SPE);
  SPCR |= !(_BV(MSTR)); //Arduino is Slave
  SPDR = 0x67;  //initial value of SPDR
}

void loop()
{
  while(bitRead(SPSR, SPIF) != HIGH)
    ;
  byte x = SPDR;
  Serial.println(x, HEX); //should show: 12 which has come from Master
}
```

2 Repeat Q1 using *byte x = SPItransfer(arg1)* instruction.

Practically, `byte x = SPI.transfer(x12)` instruction is equivalent to the following codes:

```
SPDR = 0x12;
while(bitRead(SPSR, SPIF) != HIGH)
  ;
byte x = SPDR;
```

3 Repeat Q1 using *interrupt* strategy rather than polling the SPIF bit of SPSR Register.

Master Codes:

```
#include <SPI.h>
byte x = 0x12;

void setup (void)
{
  Serial.begin(115200);
  SPI.begin ();
  delay(100);
  digitalWrite(SS, LOW);    //Slave is selected
  SPI.setClockDivider(SPI_CLOCK_DIV128);  //500 Kbits/sec
}

void loop()
{
 SPDR = x;
 while(bitRead(SPSR, SPIF) != HIGH)
  ;
 byte y = SPDR;
 Serial.println(y, HEX);
 x++;
 delay(1000);
}
```

Slave Codes:

```
#include <SPI.h>
volatile bool flag1 = false;
byte x, y;

void setup ()
{
  Serial.begin(115200);
```

```
  pinMode(SS, INPUT_PULLUP);  // ensure SS stays high for now
  pinMode(MISO, OUTPUT);
  SPCR |= _BV(SPE);
  SPCR |= !(_BV(MSTR)); //Arduino is Slave
  SPDR = 0x67;
  SPCR |= _BV(SPIE);    //interrupt logic is enabled
//  SPI.attachInterrupt();
}

void loop()
{
  if (flag1 == true)
  {
    Serial.println(x, HEX); //
    flag1 = false;
  }
}

ISR(SPI_STC_vect)  //SPI SerialTransferComplete
{
  x = SPDR;
  y = x + 2;
  SPDR = y;
  flag1 = true;
}
```

4 Write codes to acquire 16-bit data from a SPI Slave using *SPI.transfer16(0xnnnn)* instruction.

5 Write codes to acquire 32-bit data from a SPI Slave using *SPI.transfer(buffer, sizeof(buffer))* instruction.

6 Write codes to acquire 32.67 float number from a SPI Slave.

Master SPI Codes:

```
#include <SPI.h>
byte myData[4] = {0};

union
{
  long x;
  byte myArray[4];
} data;

void setup (void)
{
  Serial.begin(115200);
  SPI.begin ();
  delay(100);
  digitalWrite(SS, LOW);    //Slave is selected
  SPI.setClockDivider(SPI_CLOCK_DIV128);  //500 Kbits/sec
}

void loop()
{
  SPI.transfer(myData, sizeof(myData));
  data.myArray[3] = myData[0]; //12345678
  data.myArray[0] = myData[1];
  data.myArray[1] = myData[2];
  data.myArray[2] = myData[3];

  Serial.print(data.x, HEX);

  Serial.println();
  delay(1000);
}
```

Slave SPI Codes:

```
#include <SPI.h>
```

```
volatile bool flag1 = false;
volatile int i = 0;
volatile byte myData[4] = {0x78, 0x56, 0x34, 0x12};  //data to be sent to Master : 12345678

void setup ()
{
  Serial.begin(115200);
  pinMode(SS, INPUT_PULLUP);  // ensure SS stays high for now
  pinMode(MISO, OUTPUT);
  SPCR |= _BV(SPE);
  SPCR |= !(_BV(MSTR)); //Arduino is Slave
  SPDR = 0x67;
  // SPCR |= _BV(SPIE);
  SPI.attachInterrupt();
}

void loop()
{

}

ISR(SPI_STC_vect)
{
  SPDR = myData[i];
  i++;
  if (i == 4)
  {
    i = 0;
  }
}
```

7 Based on the following SPI network, collect temperature signal from the Slave and show it on SM1 at 2-sec interval.

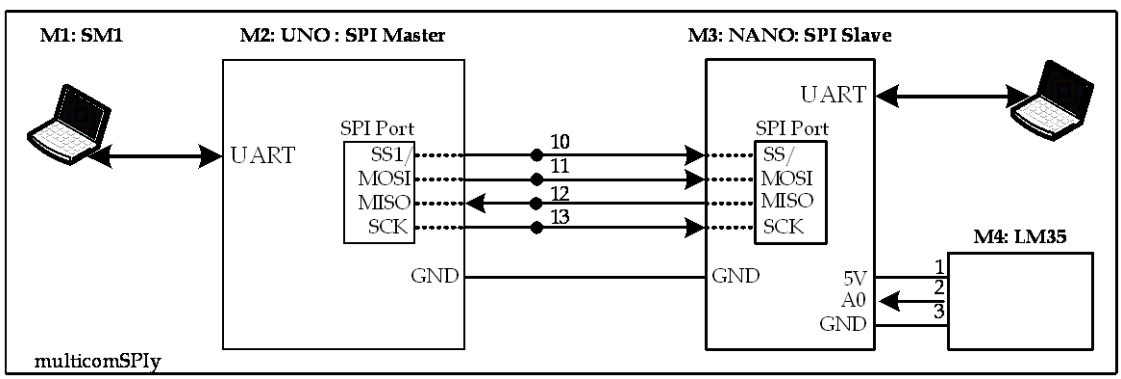

Figure-7.3: *SPI Port connection between UNO and NANO yo operate LM35 temperature sensor*

Master SPI Codes:

```
#include <SPI.h>
byte myData[4] = {0x12, 0xEF, 0xCD, 0xAB};

union
{
  float myTemp;
  long x;
  byte myArray[4];
} data;

void setup (void)
{
  Serial.begin(115200);
```

```
  SPI.begin ();
  delay(100);
  digitalWrite(SS, LOW);    //Slave is selected
  SPI.setClockDivider(SPI_CLOCK_DIV128);  //500 Kbits/sec
}

void loop()
{
  SPI.transfer(myData, sizeof(myData));
  data.myArray[3] = myData[0]; //12345678
  data.myArray[0] = myData[1];
  data.myArray[1] = myData[2];
  data.myArray[2] = myData[3];

  Serial.println(data.x, HEX);    //binary32 formatted 32-bit receibed from Slave
  Serial.println(data.myTemp,2);
  delay(2000);
}
```

Slave SPI Codes:

```
#include <SPI.h>
volatile bool flag1 = false;
volatile int i = 0;
volatile byte myData[4] = {0x78, 0x56, 0x34, 0x12};
union
{
  float T;
  byte myArray[4];
}data;

void setup ()
{
  Serial.begin(115200);
  analogReference(INTERNAL);
  pinMode(8, OUTPUT);
  pinMode(SS, INPUT_PULLUP);  // ensure SS stays high for now
  pinMode(MISO, OUTPUT);
  SPCR |= _BV(SPE);
  SPCR |= !(_BV(MSTR)); //Arduino is Slave
  SPDR = 0x67;
  SPI.attachInterrupt();  // SPCR |= _BV(SPIE);
}

void loop()
{

 // digitalWrite(8, !digitalRead(8));
  //delay(1000);

  float myTemp = (float)100*(1.1/1023.0)*analogRead(A0);
  data.T = myTemp;
  Serial.println(myTemp, 2);
  delay(5000);
}

ISR(SPI_STC_vect)
{
  SPDR = data.myArray[i];
  i++;
  if (i == 4)
  {
    i = 0;
  }
}
```

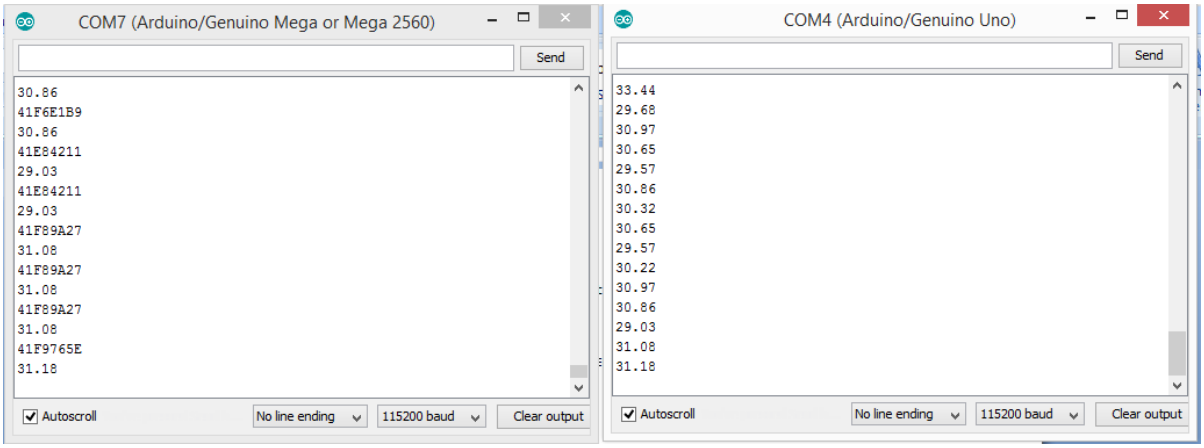

Figure-7.4: *Screen showing data sent and receive*

8 Connect DHT22 and LM35 sensors with UNO as per following figure. Create sketch for UNO to acquire temperatures from both sensors at 1-sec interval and show them on SM2. The UNO will also put the temperature signals on the SPI Port when data read request will come from NANO. Create sketch for NANO to acquire temperature signals from Uno at 2-sec interval and show them on SM1.

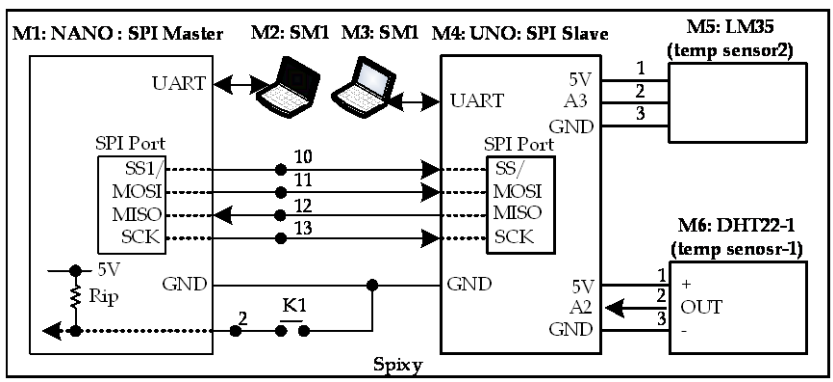

Figure-7.5: *SPI Port connection between NANO, UNO, DHt22, and LM35 temperature sensors*

NANO-SPIMaster Codes:

```
#include <SPI.h>
byte myData[] = {0x12, 0xEF, 0xCD, 0xAB, 0x12, 0xEF, 0xCD, 0xAB};

union
{
  float rxmyTemp;
  byte rxmyTempArray[4];
} data1, data2;

void setup (void)
{
  Serial.begin(9600);
  SPI.begin ();
  digitalWrite(SS, LOW);    //Slave is selected
  delay(100);
  SPI.setClockDivider(SPI_CLOCK_DIV128);  //500 Kbits/sec
}

void loop()
```

```
{
  // SPI.transfer(0x01);   //dta request command
  //  byte x = SPI.transfer(0x01);
  //  if (x == 0x13)
  //  {
  // delay(100);
  SPI.transfer(myData, sizeof(myData));
  data1.rxmyTempArray[3] = myData[0];
  data1.rxmyTempArray[0] = myData[1];
  data1.rxmyTempArray[1] = myData[2];
  data1.rxmyTempArray[2] = myData[3];

  data2.rxmyTempArray[3] = myData[4];
  data2.rxmyTempArray[0] = myData[5];
  data2.rxmyTempArray[1] = myData[6];
  data2.rxmyTempArray[2] = myData[7];

  // Serial.println(data.x, HEX);    //binary32 formatted 32-bit receibed from Slave
  Serial.print("Temperature1 from DHT22: ");
  Serial.print(data1.rxmyTemp, 2);
  Serial.println(" *C");
  Serial.print("Temperature2 from LM35: ");
  Serial.print(data2.rxmyTemp, 2);
  Serial.println(" *C");
  Serial.println("==================================");
  delay(2000);
}
```

UNO-SPISlave Codes:

```
#include "DHT.h"
#include <SPI.h>
volatile bool flag1 = false;
volatile int i = 0, j = 0, k=0;
//volatile byte myData[4] = {0x78, 0x56, 0x34, 0x12};
#define DHTPIN1 A2     // what pin we're connected to
#define DHTTYPE1 DHT22   // DHT 22  (AM2302)
float myTemp1;
float myTemp2;
byte dataArray[8];
union
{
  float txmyTemp;
  byte txmyTempArray[4];
} data1, data2;

DHT dht1(DHTPIN1, DHTTYPE1);

void setup()
{
  Serial.begin(9600);
  dht1.begin();
  analogReference(INTERNAL);
  //-------------------------
  pinMode(SS, INPUT_PULLUP);  // ensure SS stays high for now
  pinMode(MISO, OUTPUT);
  SPCR |= _BV(SPE);
  SPCR |= !(_BV(MSTR)); //Arduino is Slave
  SPCR |= _BV(SPIE);
  //  SPDR = 0x67;
  //SPI.attachInterrupt();  // SPCR |= _BV(SPIE);
}

void loop()
{
```

```
  tempDht1();
  tempLM35();
  for (int i = 0; i < 4; i++)
  {
    dataArray[i] = data1.txmyTempArray[i];
  }

  for (int i = 4, j = 0; i < 8, j < 4; i++, j++)
  {
    dataArray[i] = data2.txmyTempArray[j];
  }
  Serial.println("====================");
  delay(1000);
  //Serial.println(data1.txmyTempArray[2], HEX); debugging
  //Serial.println(data2.txmyTempArray[2], HEX);
}

void tempDht1()
{
  myTemp1 = dht1.readTemperature();
  Serial.print("Temperature1: ");
  Serial.print(myTemp1, 2);
  Serial.println(" *C");
  //-----------------------
  data1.txmyTemp = myTemp1;
}

void tempLM35()
{
  myTemp2 = (float)100 * (1.1 / 1023.0) * analogRead(A3);
  Serial.print("Temperature2: ");
  Serial.print(myTemp2, 2);
  Serial.println(" *C");
  //-----------------------
  data2.txmyTemp = myTemp2;
}

ISR(SPI_STC_vect)
{

  SPDR = dataArray[k];
  k++;
  if (k == 8)
  {
    k = 0;
  }
}
```

9 Add K1 with the diagram of Fig-7.5. Write codes for both NANO and UNO so that data acquisition an exchange occurs only for once and when K1 is pressed.