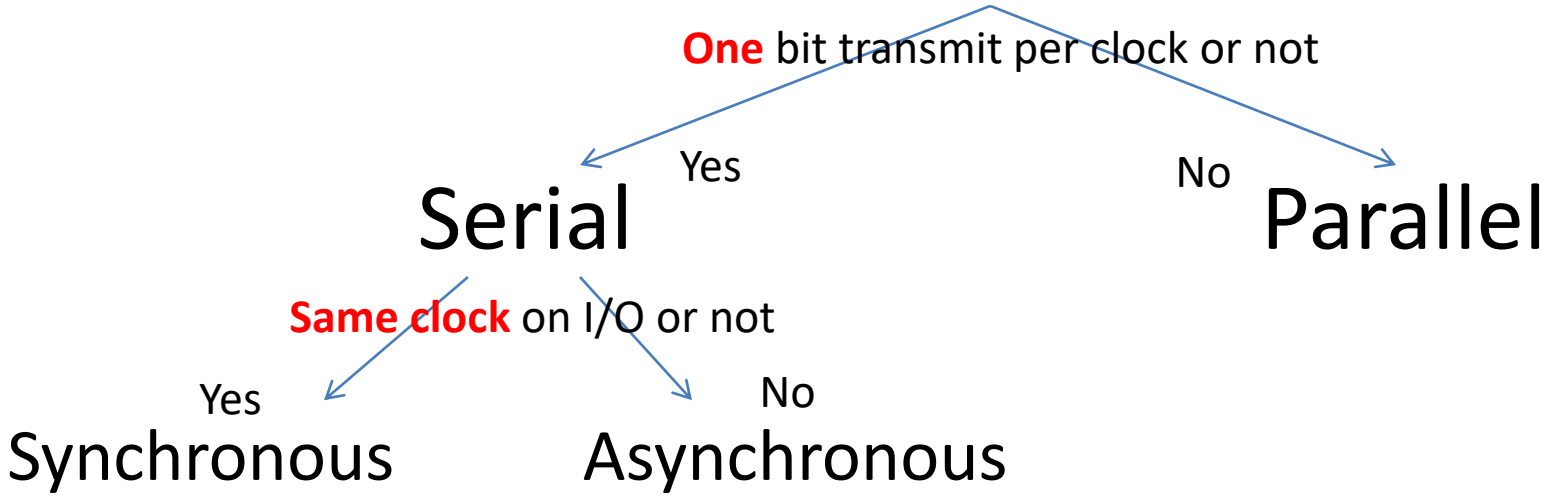
CET 241: Day 17
Serial Interfaces II

Dr. Noori Kim

Agenda

- Introducing SPI
- SPI working with TFT LCD
- Introducing I2C
- I2C working with TMP102 I2C temperature sensor

I/O Interfaces



i.e., SPI

- Same clock (need to share clock signal)
- 4 signal lines (miso, mosi, clk,cs (ss, or fs))
- Common configuration (baud rate, spo sph for freescale, data frame)

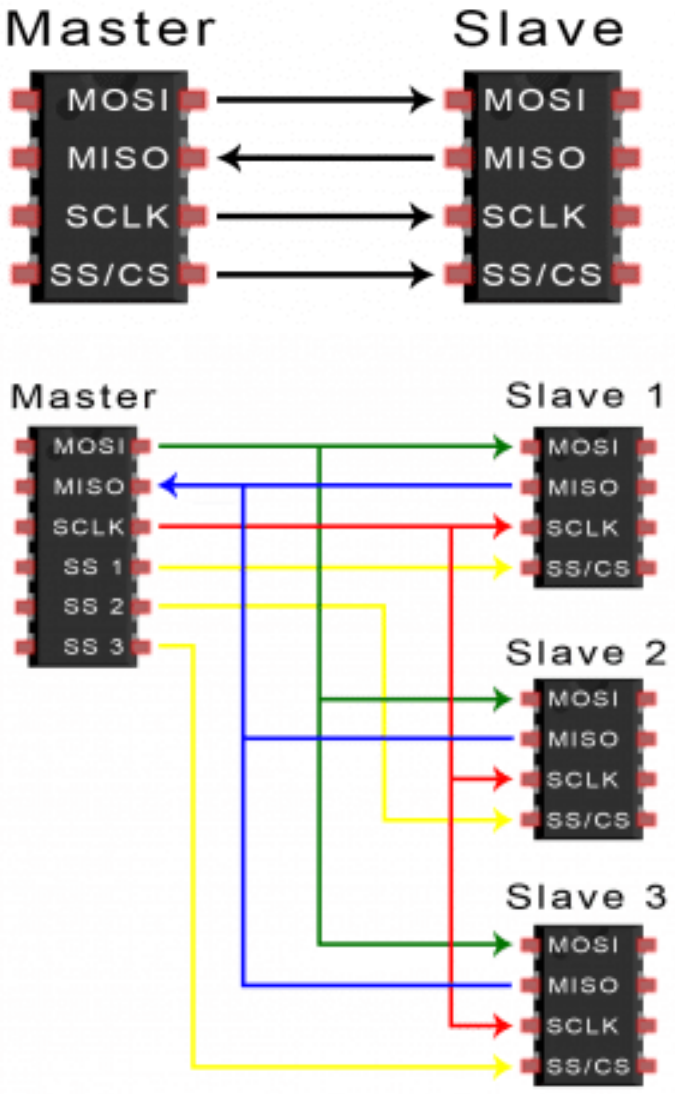
start, stop are configured by cs

i.e., UART

- Sender receiver clocks are separated (no need to send clock signal)
- 3 signal lines (rx,tx,sg)
- Common configuration (Baudrates, frame)
- +- 5% baud rate tolerance
- Old UART no FIFO, most UART nowadays with FIFO

start, stop, data, parity

SPI

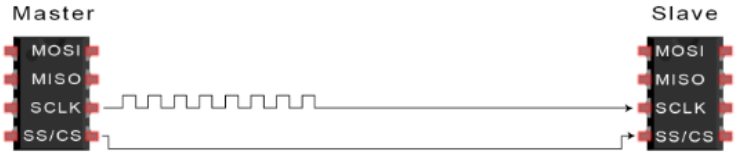


- **MOSI (Master Output/Slave Input)** – Line for the master to send data to the slave.
- **MISO (Master Input/Slave Output)** – Line for the slave to send data to the master.
- **~SS/CS(Slave Select/Chip Select)** –Line for the master to select which slave to send data to. Theoretically this is extendable as many as slaves you want. Mostly active low
- **SCLK (clock)** – mostly from Master

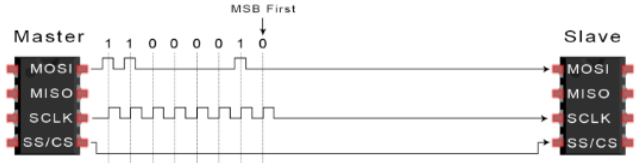
STEPS OF SPI DATA TRANSMISSION

1. The master outputs the clock signal:

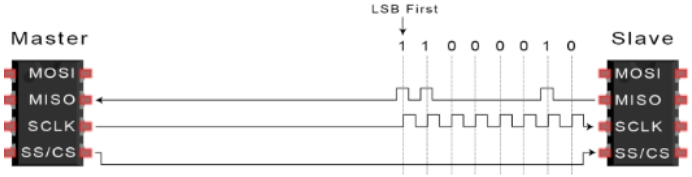
2. The master switches the SS/CS pin to a low voltage state, which activates the slave:



3. The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:



4. If a response is needed, the slave returns data one bit at a time to the master along the MISO line. The master reads the bits as they are received:



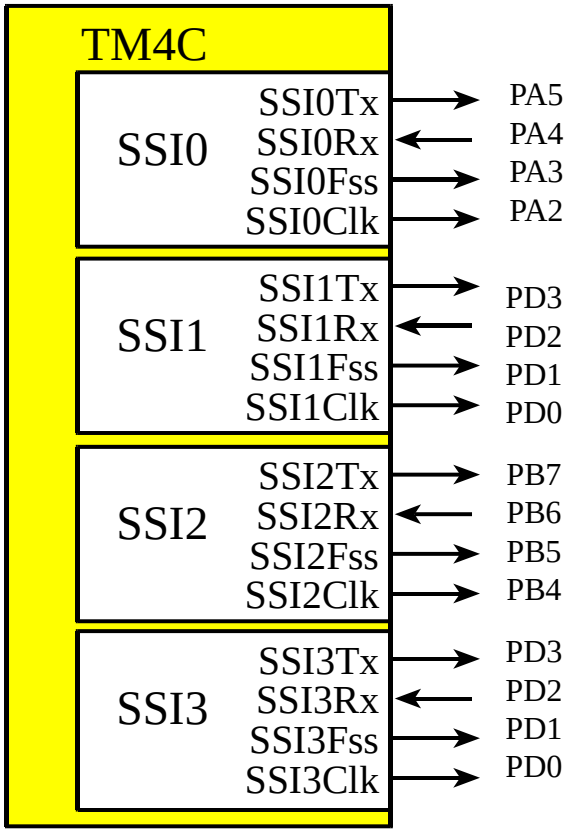
ADVANTAGES

- No start and stop bits, frame start and stop is controlled by SS signal
- No complicated slave addressing system like I2C
- Higher data transfer rate than I2C (almost twice as fast, up to 10Mbps)
- Separate MISO and MOSI lines, so data can be sent and received at the same time

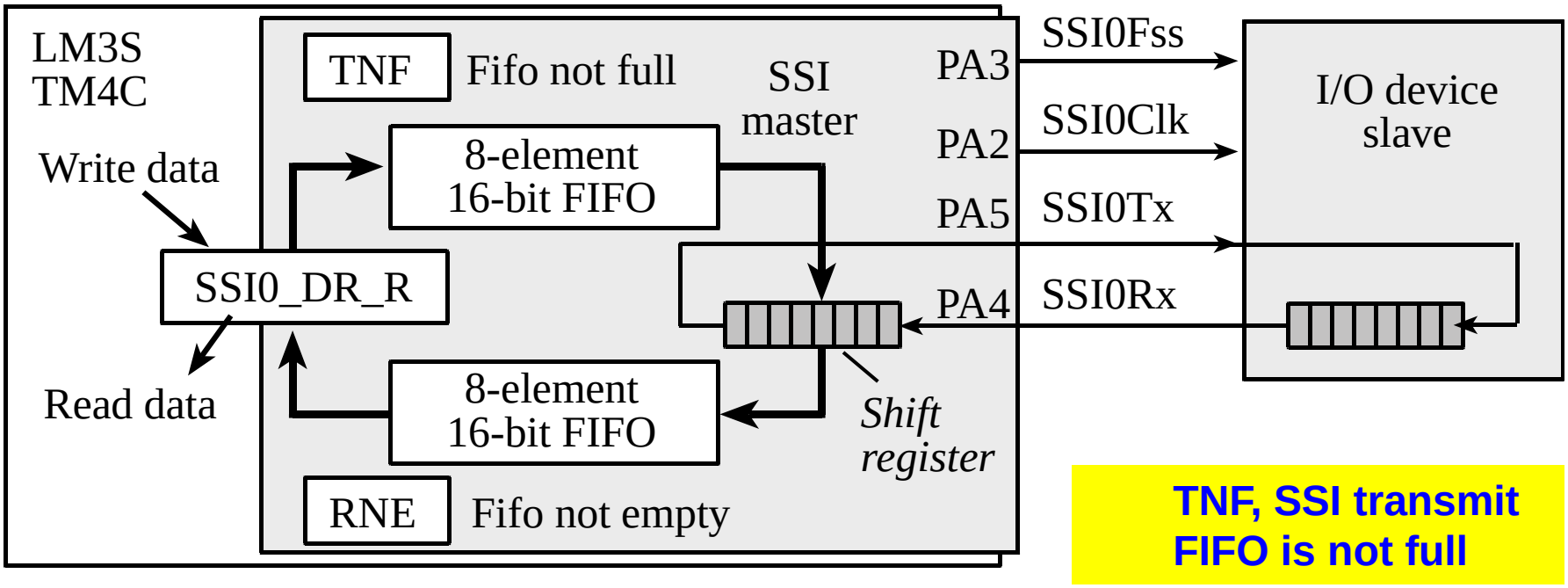
DISADVANTAGES

- Uses four wires (I2C and UARTs use two)
- No acknowledgement that the data has been successfully received (I2C has this)
- No form of error checking like the parity bit in UART
- Only allows for a single master (I2C multi master)

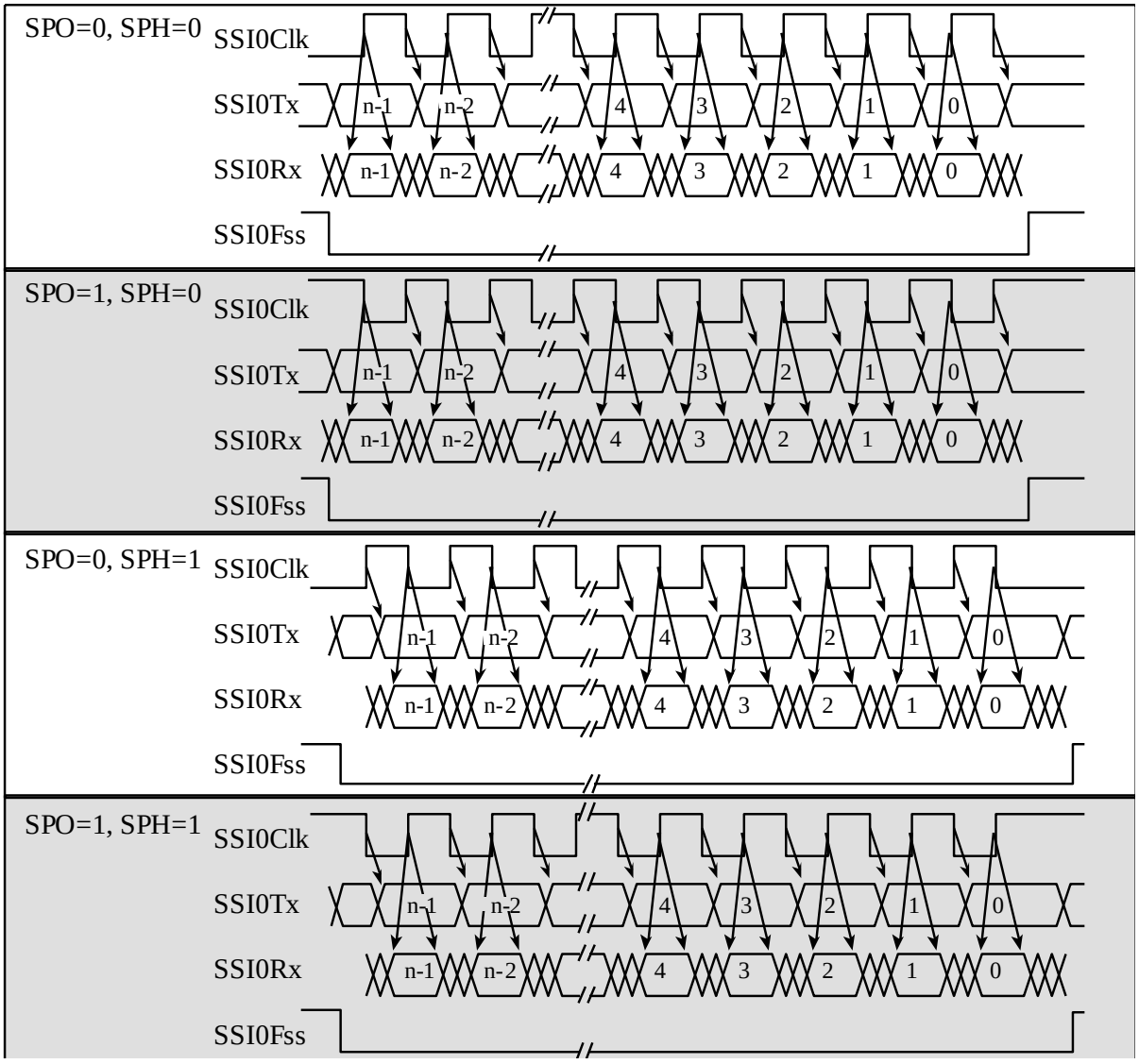
TM4C123 SPI


TM4C
SSI0
SSI0Tx
SSI0Rx
SSI0Fss
SSI0Clk
PA5
PA4
PA3
PA2
SSI1
SSI1Tx
SSI1Rx
SSI1Fss
SSI1Clk
PD3
PD2
PD1
PD0
SSI2
SSI2Tx
SSI2Rx
SSI2Fss
SSI2Clk
PB7
PB6
PB5
PB4
SSI3
SSI3Tx
SSI3Rx
SSI3Fss
SSI3Clk
PD3
PD2
PD1
PD0

- When software writes to the data register, this 4 to 16-bit register is serially shifted 4 to 16 bit positions by the SSIClk clock



Synchronous serial modes of the Freescale SPI interface



Data flow is controlled by

1. Base clock
2. Transmission at 1st or 2nd clock edge

You input this parameter on SSICR register

SPI on TM4C

1. GPIO CONFIGURATION: System Control Registers(SYSCTL) & GPIO Registers (GPIOx)
   - Enable the clock for GPIOx (RCGCGPIO)
   - Wait until GPIOx is ready (PRGPIO)
   - Configure the CLK, CS (FSS), MOSI (Tx), and MISO(Rx) pins as a digital pin (DEN)
   - Configure the CLK, CS (FSS), MOSI (Tx), and MISO(Rx) pins for their alternate function (AFSEL)
   - Configure the CLK, CS (FSS), MOSI (Tx), and MISO(Rx) port control pins to route the SSI interface to the pins (PCTL)

2. SPI CONFIGURATION (POLLED): System Control Registers(SYSCTL)& SSI Registers (SSIx)
   - Enable the clock for SSIx (RCGCSSI)
   - Wait for the SSI peripheral to be ready (PRSSI)
   - Disable the SPI interface (CR1)
   - Set the clock rate of the SPI Clock (CPSR, CR0)
   - Set the data size to be 8-bits and Freescale mode (CR0)
   - Set the SPI mode (CR0)
   - Re-enable the SPI interface (CR1)

3. SPI CONFIGURATION (POLLED): System Control Registers(SYSCTL) & SSI Registers (SSIx)
   - Enable the clock for SSIx (RCGCSSI)
   - Wait for the SSI peripheral to be ready (PRSSI)
   - Disable the SPI interface (CR1)
   - Set the clock rate of the SPI Clock (CPSR, CR0)
   - Set the data size to be 8-bits and Freescale mode (CR0)
   - Set the SPI mode (CR0)
   - Re-enable the SPI interface (CR1)

We can practice “SPI interface” using Adafruit 1.8" TFT screen which comes with ST7735 driver

LCD connection	Launchpad	Description
LITE	+3.3V	Provides power supply to backlight of LCD
MISO	NOT used	We do not read data from LCD, so this pin is not used
SCK	PA2 (SSI0Clk)	Provides clock from launch pad to LCD
MOSI	PA5 (SSI0Tx)	Input data from Launch pad to LCD
TFT_CS	PA3 (SSI0Fss)	Chip select line to select LCD (active low)
CARD_CS	NOT used	Chip select line to select CARD
D/C	PA6 (GPIO)	LCD takes data in MOSI line as DATA or COMMAND by sensing this line. D/C=0 > Command, D/C=1 > Data
RESET	PA7 (GPIO)	Resets LCD for initialization (active low)
VCC	+3.3V	Provides operating voltage to LCD
GND	GND	Provides ground return for power supply

Note that TFT screen concept itself is nothing to do with SPI
https://www.youtube.com/watch?v=5y4NMzjLXus

TM4C123 TRANSMITTING/RECEIVING DATA

- Transmitting: write the data to the transmit FIFO.
  - Chip select should be low during whole data transfer
- Receiving: due to the nature of the SPI interface, we will receive the same number of bytes that we transmit.
  - After we have filled the transmit FIFO, we will busy wait until the same number of bytes have been received by the receive FIFO.

```
const uint16_t Logo[] = {
int main(void){
  int x, y;
  PLL_Init(4);                        // set system clock to 80 MHz ==> 4
  ST7735_InitR();
  x = 20;  //initial position x and y of my DigiPen ECE300 logo
  y = 135;
  ST7735_FillRect(0, 0, ST7735_TFTWIDTH, ST7735_TFTHEIGHT, 0xFFFF);
  while(1){
    ST7735_DrawBitmap(x, y, Logo, 79, 108);
  }
}
```

```
void ST7735_InitR(void) {
  //commonInit(Rcmd1, Rcmd2red, Rcmd3);
  commonInit(Rcmd1, Rcmd2green, Rcmd3);
}
```

commonInit

```
volatile uint32_t delay;
ColStart  = RowStart = 0; // May be overridden in init func

SYSCTL_RCGCSSI_R |= 0x01;  // activate SSI0
SYSCTL_RCGCGPIO_R |= 0x01; // activate port A
while((SYSCTL_PRGPIO_R&0x01)==0){}; // allow time for clock to start
```

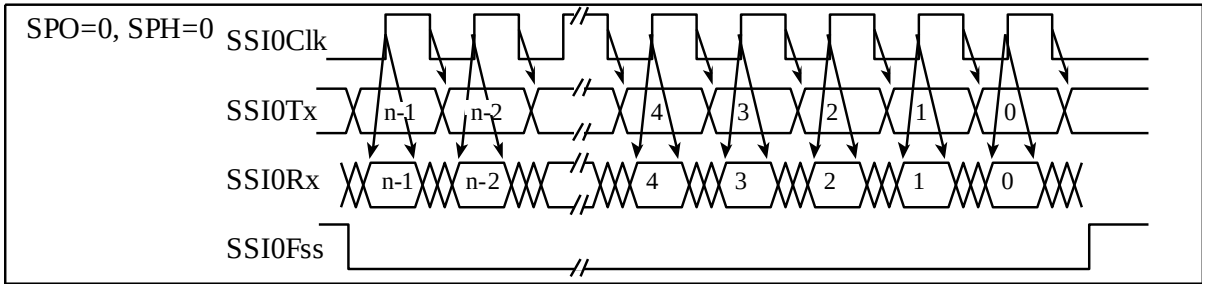
```
//*********************************For TFT GPIO pins*****************************************//
  // toggle RST low to reset;
  GPIO_PORTA_DIR_R |= 0xC0;             // make PA6,7 out (Reset and Data/Command pins)
  GPIO_PORTA_AFSEL_R &= ~0xC0;          // disable alt funct on PA6,7
  GPIO_PORTA_DEN_R |= 0xC0;             // enable digital I/O on PA6,7
                                        // configure PA6,7 as GPIO
  GPIO_PORTA_PCTL_R = (GPIO_PORTA_PCTL_R&0x00FFFFFF)+0x00000000;
  GPIO_PORTA_AMSEL_R &= ~0xC0;          // disable analog functionality on PA 6,7

  RESET = 0x00;       // RESET_LOW              0x00 ; active low, this will reset tft
                      //RESET addr.-->PA7 data register
                      //page 37 of ST7735R_V0.2-TFT.pdf
  Delay1ms(500);    //page 74 of ST7735R_V0.2-TFT.pdf
  RESET = 0x80;       // RESET_HIGH              0x80
                      //RESET addr.-->PA7 data register
                      //page 37 of ST7735R_V0.2-TFT.pdf
  Delay1ms(500);    //page 74 of ST7735R_V0.2-TFT.pdf (wait more than 10 ms)
```

Datasheet 965

```
  // initialize SSI0
  GPIO_PORTA_AFSEL_R |= 0x2C;           // enable alt funct on PA2,3,5
  GPIO_PORTA_DEN_R |= 0x2C;             // enable digital I/O on PA2,3,5
                                        // configure PA2,3,5 as SSI
  GPIO_PORTA_PCTL_R = (GPIO_PORTA_PCTL_R&0xFF0F00FF)+0x00202200;
  GPIO_PORTA_AMSEL_R &= ~0x2C;          // disable analog functionality on PA2,3,5
  SSI0_CR1_R &= ~0x00000002;           // disable SSI
                                        // SSI_CR1_SSE   0x00000002  // SSI Synchronous Serial Port
  SSI0_CR1_R &= ~0x00000004;            // master mode
                                        //SSI_CR1_MS  0x00000004  // SSI Master/Slave Select
                                        // configure for system clock/PLL baud clock source
  SSI0_CC_R = (SSI0_CC_R&~0x0000000F)+0x00000000;
//                                        // clock divider for 3.125 MHz SSIClk (50 MHz PIOSC/16)
                                        // SSI_CC_CS_M       0x0000000F  // SSI Baud Clock Source
                                        //SSI_CC_CS_SYSPLL     0x00000000  // Either the system clock (if the
                                        // PLL bypass is in effect) or the
                                         // PLL output (default)

//  SSI0_CPSR_R = (SSI0_CPSR_R&~SSI_CPSR_CPSDVSR_M)+16;
                                        // clock divider for 8 MHz SSIClk (80 MHz PLL/24)
                                        // SysClk/(CPSDVSR*(1+SCR))
                                        // 80/(10*(1+0)) = 8 MHz (slower than 4 MHz)
  SSI0_CPSR_R = (SSI0_CPSR_R&~0x000000FF)+10; // must be even number
                                        //SSI_CPSR_CPSDVSR_M      0x000000FF  // SSI Clock Prescale Divisor
```




```
SSI0_CR0_R &= ~(0x0000FF00 |        // SCR = 0 (8 Mbps data rate) SSI_CR0_SCR_M    0x0000FF00  // SSI Serial Clock
                0x00000080 |          // SPH = 0 SSI_CR0_SPH  0x00000080  // SSI Serial Clock Phase
                0x00000040);          // SPO = 0 SSI_CR0_SPO  0x00000040  // SSI Serial Clock Polarity
                                       // FRF = Freescale format

SSI0_CR0_R = (SSI0_CR0_R&~0x00000030)+0x00000000;
                                       // DSS = 8-bit data
                                       //SSI_CR0_FRF_M     0x00000030  // SSI Frame Format Select
                                       //SSI_CR0_FRF_MOTO   0x00000000  // Freescale SPI Frame Format
SSI0_CR0_R = (SSI0_CR0_R&~0x0000000F )+0x00000007;
                                       //SSI_CR0_DSS_M  0x0000000F  // SSI Data Size Select
                                       //SSI_CR0_DSS_8  0x00000007  // 8-bit data
SSI0_CR1_R |= 0x00000002;             // enable SSI
                                       //SSI_CR1_SSE             0x00000002  // SSI Synchronous Serial Port
```

+ command list init. Based on driver's datasheet

31-16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rsv	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1	1

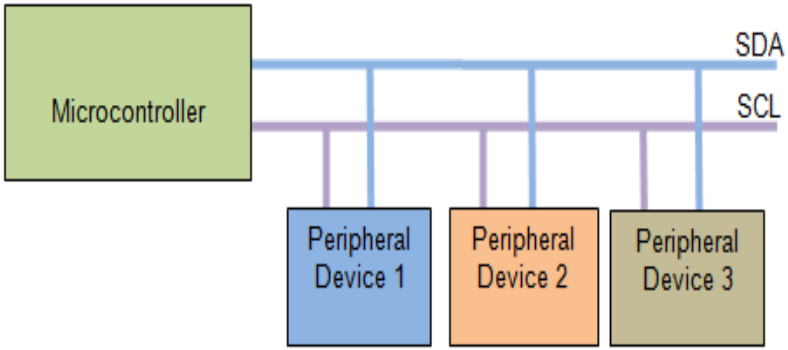
References

- http://www.circuitbasics.com/basics-of-the-spi-communication-protocol/
- Textbook example 7.2-7.4

I2C

- The name I2C is shorthand for Standard Inter-Integrated Circuit bus
- I2C is a serial data protocol which operates with a master/slave relationship
- I2C only uses two physical wires, this means that data only travels in one direction at a time.

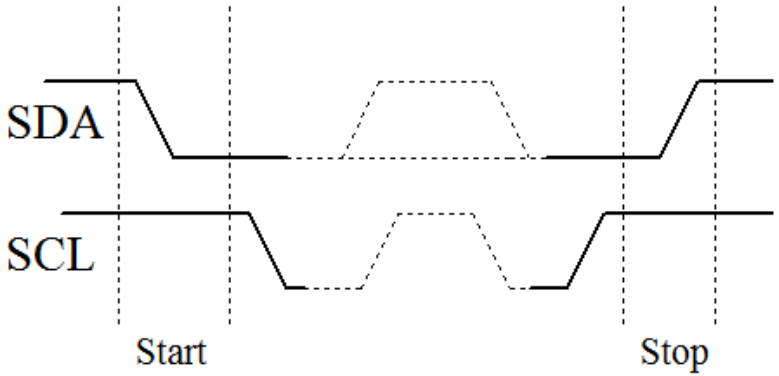
- The I2C protocol is a two-wire serial bus:

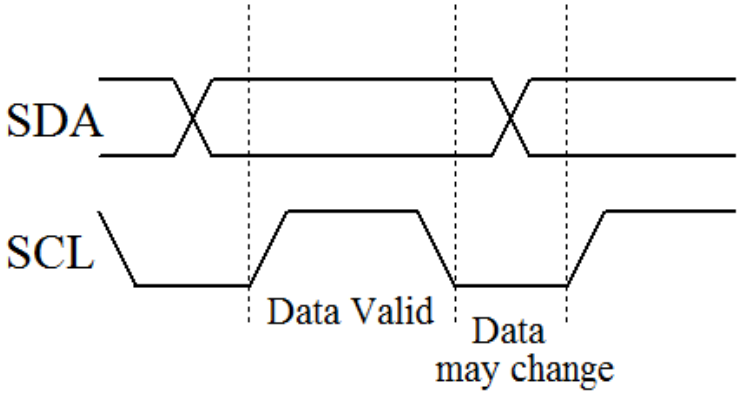


- The I2C communication signals are serial data (SDA) and serial clock (SCL)
  - These two signals make it possible to support serial communication of 8-bit data bytes, 7-bit device addresses as well as control bits
  - Using just two wires makes it cheap and simple to implement in hardware

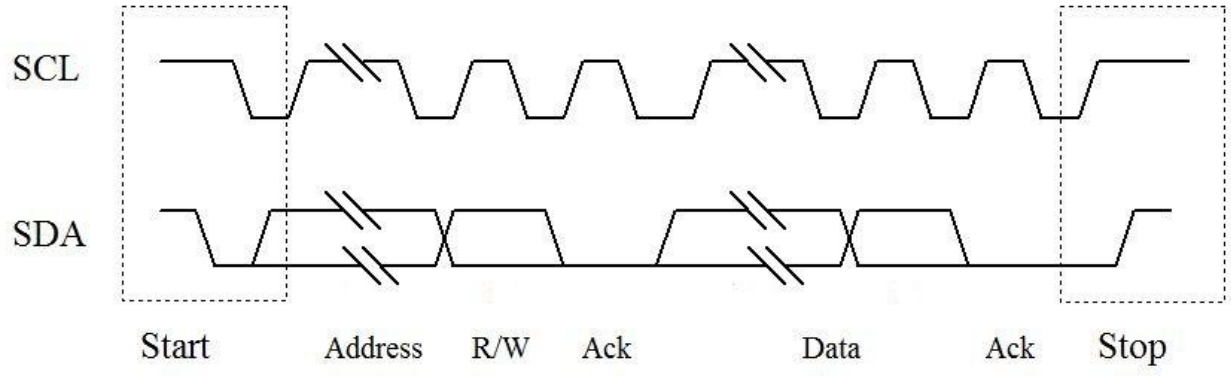
- I2C has a built-in addressing scheme, which simplifies the task of linking multiple devices together.
  - In general, the device that initiates communication is termed the ‘master’. A device being addressed by the master is called a ‘slave’.
  - Each I2C-compatible slave device has a predefined device address. The slaves are therefore responsible for monitoring the bus and responding only to data and commands associate with their own address.
  - This addressing method, however, limits the number of identical slave devices that can exist on a single I2C bus, as each device must have a unique address. For some devices, only a few of the address bits are user configurable.

- A data transfer is made up of the Master signalling a Start Condition, followed by one or two bytes containing address and control information.
- The Start condition is defined by a high to low transition of SDA when SCL is high.
- A low to high transition of SDA while SCL is high defines a Stop condition
- One SCL clock pulse is generated for each SDA data bit, and data may only change when the clock is low.





- The byte following the Start condition is made up of seven address bits, and one data direction bit (Read/Write)
- All data transferred is in units of one byte, with no limit on the number of bytes transferred in one message.
- Each byte must be followed by a 1-bit acknowledge from the receiver, during which time the transmitter relinquishes SDA control.



Evaluating the TMP102 I2C temperature sensor

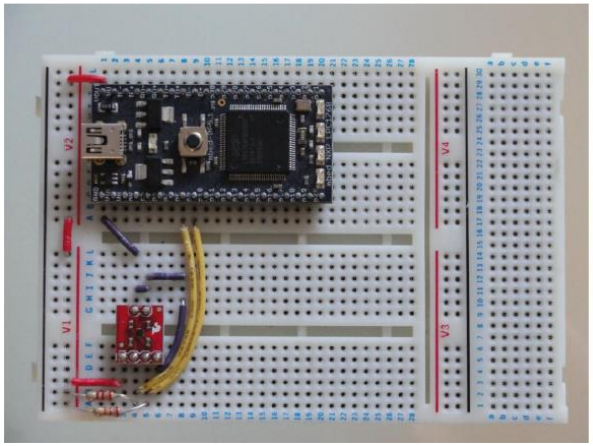
- Configuration and data register details are given in the TMP102 data sheet
  - http://focus.ti.com/lit/ds/sbos397b/sbos397b.pdf
- To configure the temperature sensor we need to:
  - Use arrays of 8-bit values for the data variables, because the I2C bus can only communicate data in one bytes.
  - Set the configuration register; we first need to send a data byte of 0x01 to specify that the Pointer Register is set to ‘Configuration Register’.
  - Send two bytes of data to perform the configuration. A simple configuration value to initialize the TMP102 to normal mode operation is 0x60A0.

- To read the data register we need to:
  - To read the data register we need to set the pointer value to 0x00.
  - To print the data we need to convert the data from a 16-bit data reading to an actual temperature value. The conversion required is to shift the data right by 4 bits (its actually only 12-bit data held in two 8-bit registers) and to multiply by the 1-bit resolution which is 0.0625 degrees C per LSB.

Exercise: Connect the temperature sensor to an I_2C bus. Verify that the correct data can be read by continuously updates of temperature (to screen or Putty).
–Test that the temperature increases when you press your finger against the sensor. You can even try placing the sensor on something warm, for example a pocket hand warmer, in order to check that it reads temperature correctly.

- The TMP102 can be connected to the mbed as shown:

Signal	TMP102 Pin	Mbed Pin	Notes
Vcc (3.3V)	1	40	
Gnd (0V)	4	1	
SDA	2	9	2.2kΩ pull-up to 3.3V
SCL	3	10	2.2kΩ pull-up to 3.3V



The following program will configure the TMP102 sensor, read data, convert data to degrees Celsius and then display values to the screen every second:

```
#include "mbed.h"
I2C tempsensor(p9, p10); //sda, scl
Serial pc(USBTX, USBRX); //tx, rx
const int addr = 0x90;
char config_t[3];
char temp_read[2];
float temp;
int main() {
    config_t[0] = 0x01;                    //set pointer reg to 'config register'
    config_t[1] = 0x60;                    // config data byte1
    config_t[2] = 0xA0;                    // config data byte2
    tempsensor.write(addr, config_t, 3);
    config_t[0] = 0x00;                    //set pointer reg to 'data register'
    tempsensor.write(addr, config_t, 1);   //send to pointer 'read temp'
    while(1) {
        wait(1);
        tempsensor.read(addr, temp_read, 2);       //read the two-byte temp data
        temp = 0.0625 * (((temp_read[0] << 8) + temp_read[1]) >> 4);  //convert data
        pc.printf("Temp = %.2f degC\n\r", temp);
    }
}
```

http://mbed.org